@kyoumoh_07

PYTHON SEQUENCE :→ STRING, LIST, TUPLE

[MODULE - 3]

- **PYTHON STRINGS :→**
  - Python strings is a sequence of characters. can also be called an array of characters.

    "hello" → 'h', 'e', 'l', 'l', 'o'
  - They are represented using single quotes '', or double quotes " ".

ex :→ name = "kyouma"

#variable name stores a string value "kyouma"

- print(name).

- **ACCESSING CHARACTERS OF STRING :**

This can be achieved in different ways :→

- Indexing :→ treating strings as an array / list and using its indexing values.

ex :

```
name = "kyouma"
print(name[0])

#output :→ k
```

index starts from 0, as it is treated as an array / list first element of an array/list is indexed at 0.

ex :→

k	y	o	u	m	a
name[0]	[1]	[2]	[3]	[4]	[5]

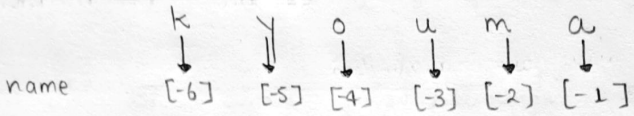
- **NEGATIVE INDEXING :→**

Similar to an array / list, python have concept of negative indexing for its strings.

ex:
```
name = "kyouma"
print( name[-1])

#output a
```

[-1] → indicates the last element of the string.



- **SLICING :→**

This helps us to get a sub-string from an existing string using some given parameter (indexes).

syntax:

string [start[index] : ending[index-1])

example:→
```
name = "kyouma"
print( name[1:4])

#output: you
```

k y o u m a
0 1 2 3 4 5

☺

Note:→ [if we try to access an index out of the range or use numbers other than an integer, it throws an error.].

PYTHON strings are immutable :→ meaning that once a string is created it cannot be changed and its content cannot be modified directly. Any operation that appears to modify a string actually creates a new string object.

- String operations do not actually modify the original string. Instead they return a new string with the requested changes.
- Memory can be optimized by using string literals, which is possible because strings are immutable.

ex:→

```
name = "kyouma"
name[0] = H
print(name)
```

{ cannot make direct changes to the string.

#output TypeError: 'str' object does not support assignment

Creating modified strings actually creates a new string

ex:

```
name = "kyouma"
name = "Hououin"
print(name)
#output → Hououin
```

{ #new string "Hououin" is assigned to name variable.

- This can also lead to inefficient processes operations where it involves building up strings and hence other methods are also introduced

- Multi-lined string

```
name = """
       Kyouma
       Hououin
       """
print(name)
```

- **PYTHON STRING OPERATIONS :**

⇒ A lot of operations can be performed with strings and on strings and is one of the most used python data type.

1. **[STRING COMPARISON :]**

→ the comparison | relational operator == can be used to compare two strings. If two strings are equal, the operator returns True, else false.

- The equality operator compare objects based on their values.
- equality operator is/are employed. It calls left `__eq__()` class method.
- Python uses "string interning" for small strings literals.
- Interned strings are stored in a special memory area, and identical strings share the same memory location.
- python checks for memory addresses of strings, if they share same memory addresses they are considered equals, [comparing pointers]
- Unterned strings have their lengths compared first.

ex :->

Interned strings

```
x = "kyouma"
y = "kyouma"
print(x is y)      # True, same memory location.
print(x == y)      # True, same values.

a = "hello" * 1000
b = "hello" * 1000
print(a is b)      # false, different memory location
print(a == b)      # True, character by character comparison

c = "hello"
d = "help"
print(c < d)       # True, 'o' comes before 'p' in Unicode ordering
```

2. STRING CONCATENATION :

String concatenation is the process of combining two or more string to create a new string.

- **USING THE '+' operator :**

```
s1 = "Hello"
s2 = "world"
result = s1 + " " + s2      # "Hello world"
```

- straightforward but inefficient for large strings.
- Creates a new string each time its used.
- High Memory usage cause it creates many intermediate strings.
- ex: result = s1 + s2 + s3 creates 2 new strings.

- **Using the '+=' operator:**

```
s = "Hello"
s += " world"      # s is now "Hello world"
```

  - Modifies the string in place (creates a new string and rebinds the name).
  - Better memory impact than '+' operator, but still creates new strings each time
  - `s+= " world"` creates a new string and reassigns **s**.

- **Using THE str.join() method:**

```
words = ["Hello", "world"]
result = " ".join(words)    # "Hello world"
```

→ generally the most efficient method for concatenating multiple strings, especially in loops.

  - calculate total length first, then allocates memory once.
  - `" ".join(["a", "b", "c"])` allocates memory only once.

- **Using f-strings (formatted string literals):**

```
s1 = "Hello"
s2 = "world"
result = f"{s1} {s2}"    # Hello world
```

  - Readable and efficient way to embed expression in string literals.
  - Efficient similar to join() for simple cases.
  - `f"{a}{b}"` creates only 1 final string.

- USING THE str.format() method:

```
S1 = "Hello"
S2 = "world"
result = "{} {}".format(S1,S2)
```

  - Similar to f-strings but was more commonly used in earlier python versions
  - slightly less efficient.
  - Moderate memory utilization, creates the format string and then the result.

- USING % operator (old-style formatting):

```
S1 = "Hello"
S2 = "World"
result = "%s %s" %(S1,S2)
```

→ older method, used in previous version of pythons.
→ Similar to str.format() method in memory usage.
→ "%s %s" %(S1,S2) creates format string and final result.

example:

```
import time
start = time.time()                    # time with + operator
result = ""
for i in range(100000):
        result += str(i)
print(f"Time with +: {time.time()-start}")

start = time.time()                    # time with .join()
result = ''.join(str(i) for i in range(100000))
print(f"Time with join: {time.tim()-start}")
```

* [• [String MULTIPLICATION | REPETETION] :

• → the * operator is used to repeat or multiply a string.

example :→

```
s1 = "Hello World"
print(s1 * 3)
```

→ prints Hello world three times

Note: f-string, str.format(), % operator are a part of string formatting.

3. [STRING CREATION AND CONVERSION: →

```
s1 = 'single quotes'
s2 = " Double quotes"
s3 = ''' Triple Quotes '''
s4 = """ for multiple strings """

num_str = str(42)      # convert int to string
char_str = chr(65)     # convert int to character : "A"
num = ord('A')         # convert char to int: 65
```

4. [STRING TRAVERSAL AND SLICING] : →

- Strings can be accessed using indexs
- string traversal can be done through a loop as well as index.

[example:

```
name = 'kyouma'
for character in name:
    print(character)
```

- string slicing are done in order to obtain sub-strings from original string.

- Reverse sequence slicing such as strings, lists, tuples. S[::-1]
  Take the entire sequence and step through it backwards.

- Efficient and concise compared to using a loop or other methods to reverse a sequence.

ex:-> 1

```
def palindrome(s):
    return s == s[::-1]

print(is_palindrame("naman"))    #output → True
```

ex: 2->

```
s = "Hello World"
first_char = s[0]
last_char = s[-1]
substring = s[1:4]
reversed_string = s[::-1]
```

5: STRING LENGTH AND COMPARISON: →

- The length of a string can be calculated using the len() function.

```
s1 = "Hello"
s2 = "VIT Vellore"
print(len(s1))
print(len(s2))
```

- we can also use a counter in for loop to calculate the same

```
name = "kyuma"
count = 0
for i in name:
    count += 1
print(count)
```

- **[COMPARING STRINGS]**

  - Can be acheived using the equality operator.
  - we can also use comparison operators to compare strings called lexicographical comparison.

ex:-

```
s1 = "hello"
s2 = "hello"
s3 = "hellp"

print(s1 == s2)                    # True
print(s1 < s3 and s2 < s3)         # True
```

6: [CASE CONVERSION]:

6.1 Upper() method: → converts a string to all UPPERCASE and returns it.

syntax

→ string.upper()

does not take any parameters

example :-

```
name = "kyouma"
s1 = name.upper()
print(name)        #kyouma
print(s1)          # KYOUMA
```

6.2 Lower() method: converts a string to all lowercase and returns it.

Syntax:

→ string.lower()

→ does not take any parameters

```
name= "KYOUMA"
print(name.lower())
```

6.3 **capitalize() method:**

→ converts the first character of a string to an uppercase letters and all other alphabets to lowercase.

syntax → string.capitalize()

- → takes no parameter.
- → returns the string (modified)

example:

```
s1 = "hello world"
print(s1.capitalize())   # output → Hello world
```

6.4 **title() method:**

- → returns a title cased version of the string, → first character of each word is capitalized.
- → syntax str.title()
- → takes no parameters.

```
s1 = "hello world"
print(s1.title())
#output  Hello World
```

→ capitalizes the first letter after an apostrophes as well

6.5 **swapcase() method:**

→ returns the string by converting all the characters to their opposite letter case().

→ syntax : string.swapcase()

→ does not take any parameters.

Note: in keyword is used to check substr ex: "substring" in string

7. WHITE SPACE HANDELING:

7.1 **strip() method**: removes any leading (starting) and trailing (ending) whitespaces from a given string.

- **syntax**: string.strip([chars])
- takes an optional parameter -chars: which specifys the set of a character to be removed from both the left and right parts of a string.

Note: if [chars] arguement is not provided, all leading and trailing whitespaces are removed from the string.

- Returns a string after removing both leading/ trailing spaces.

example: → **without arguement**:

```
string1 = "    Hello World    "
print(string.strip())
#output   Hello World.
```

→ **with arguement**:

```
string2 = 'xd xd   gg wp   xd xd'
print(string.strip('xd'))
#output   gg wp
```

7.2 **lstrip() method**: removes characters from left based on the arguement (a string specifying set of characters to be removed).

- syntax = string.lstrip([chars])
- takes in an optional parameter [chars] specifying set of characters to be removed.
- if arguement is not provided, all leading white spaces are removed from the string.
- Returns a copy of the string with leading characters stripped.

[example:]

[without arguements]

```
string1 =  '    hello world'
print( string1.lstrip())
#output   hello world
```

[with arguements]

```
string1 =  '    hello world'
print( string1.lstrip('he'))
#output   llo world
```

7.3 rstrip() method :→ returns a copy of the string with trailing characters removed (based on the string argument passed).

- syntax = string.rstrip() | string.rstrip([chars])
- takes [chars] as an optimal arguement.

example:

without arguements

```
string1 = 'hello world  '
print( string1.rstrip())
#output  'hello world'
```

with arguements

```
string1 =  'hello world  '
print( string1.rstrip('ld'))
#output  'hello wor'
```

8. SEARCHING AND COUNTING :→

8.1 find() method :

- The find() method returns the index of first occurrences of the substring (if found). If not found, it returns -1.

example :→

```
message = "My name is XYZ"
print( message.find('XYZ'))
#output: 12
```

Syntax: `str.find(sub[, start[, end]])`

- The find() method takes maximum of three parameters :
  - sub - it is the substring to be searched in the str string.
  - start and end : range str[start : end] within substring.
- find() method returns an integer value.

example :

```
quote = 'Do small things with great love'

#substring is searched in 'hings with great love'
print(quote.find('small things', 10))      #output -1

#substrings is searched in ' small things with great love'
print(quote.find('small things', 2))       3

#substrings is searched in 'hings with great lov'
print(quote.find('o small', 10, -1))       -1

#substrings is searched in 'll things with'
print(quote.find('things', 6, 20))         9
```

8.2 Index() method :

- index() method returns the index of a substring inside the string (if found). If the substring is not found, it raises an exception.
- syntax → str.index(sub [, start [, end]])
- index() parameters → takes 3 parameters
  - sub - substring to be searched
  - start and end (optional) → str[start:end]
- returns the lowest index where substring is found, if it doesn't exist inside the string, it raises a valueError exception.

example :

```
string1 = 'Python programming is fun.'
result = sentence.index('is fun')
print("Substring 'is fun': ", result)

result = sentence.index('is fun')
print("Substring 'Java': ", result)

#output  substring 'is fun': 19
         <value error>
```

8.3 rfind() method :

- syntax ⇒ rstring.find(sub [, start [, end]])
- rfind() parameters → takes a maximum of three parameters :
  - sub - It's the substring to be searched in the str string.
  - start and end optional.
- returns an integer value.
- if substring exists, returns the highest index where it is found else returns -1.

Example: [with no start and end argument]

```
song = "Let it go, Let it go, Let it go"
result = song.find('Let it')
print("substring 'Let it': ", result)

#output
substring, 'Let it': 22
```

[rfind() with start and end arguments]

```
message = 'Do small things with great love'
print(message.rfind('things', 10))          # -1
print(message.rfind('t', 2))                # 25
print(message.rfind('o small', 10, -1))     # -1
```

8.4 count() method:

- count() method returns the number of occurrences of a substring in the given string.
- Syntax → string.count(substring, start=...., end=...)
- requires only single parameter for execution but also has two optional parameters:

  substring, start (index), end (index)

- returns the number of occurrences of the substring in the given string.

```
string = " The speed of light in vaccume is fast, isn't it?"
substr = "is"
count = string.count(substr)
print(count)          # output => 2
```

[WITH all arguements]:

```
string = " The speed of light in vaccume is fast, isn't it?"
substr = 'i'
count = string.count(substr, 0, 22)
print(count)          # output => 2
```

(9) [CHECKING STRING PROPERTIES:

(9.1) startswith() :- method returns True if a string starts with the specified prefix (string). If not, it returns false.

- syntax :- startswith(prefix[, start[, end]])
- takes a maximum of three parameters:
  - prefix → string or tuples of string to be checked.
  - start(optional): Beginning position where prefix is to be checked within the string
  - end(optional): Ending position where prefix is to be checked within the strings.
- returns a boolean value

Example:

without optional parameters

```
text = "VIT Vellore is a reputed University"

result = text.startswith('is a')
print(result)      #output → false

result = text.startswith('VIT Vellore')
print(result)      #output → True

result = text.startswith('VIT Vellore is a')
print(result)      #output → True
```

with optional parameters (start and End)

```
text = "Python programming is easy"
result = text.startswith('programming is', 7)
#output → True

result = text.startswith('programming is', 7, 18)
print(result)
#output → false

result = text.startswith('program', 7, 18)
print(result)
# True
```

Sol ⇒ endswith() method :

- methods returns True if a string ends with the specified suffix. If not, it returns false.
- syntax → string.endswith(suffix [, start [, end]])
- takes three parameters:
  suffix, start, end.
- return a boolean.

example:

```
text = "VIT is a good university"
print(text.endswith('to learn')          # false
print(text.endswith('is', 7, 26)         # false
print(text.endswith('good', 7, -1)       # false
```

- passing Tuple to endswith()

if the string ends with any item of the tuple, endswith() returns True if not, it returns false.

```
print(text.endswith(('python', 'is', 'good', 'university'))
#output = True
```

9.3 isalnum() function:

- method returns True if all characters in the string are alphanumeric (either alphabets or numbers). If not, returns false
- syntax = string.isalnum()
- does not take any parameters.

ex:-

```
string1 = "@kyama"
string2 = "kyouma"
print(is.alnum(string1))
print(is.alnum(string2))
```

9.4 isalpha() methods:

- returns true if all the characters in the string are alphabets (both lowercase & uppercase) otherwise false.
- syntax string.isalpha()
- does not take any parameters.

9.5 **isdigit() method :** (0-9) only

- method returns True if all characters in the string are digits. If not returns false.
- syntax → string.isdigit()
- does not take any parameter and returns a boolean value

9.6 **isdecimal() method :**

- returns true if all characters in the string are decimal characters.

decimal → (decimal no system)

- syntax → string.isdecimal()
- does not take any parameter.
- includes other numeral forms.

Note: subscripts and superscripts are digit characters (numeric)

9.7 **isnumeric() method :**

- returns True if all characters ins the strings are numeric
- digits (0-9), numeric characters such as fractions, subscripts, superscripts and Roman numerals.
- syntax → string.isnumeric()

3.8 **islower() method :**

- returns true if all alphabets that exist in the string are lowercase alphabets.
- syntax: string.islower()
- does not take any parameter

9.9 **istitle() method:**

- method returns True if a string is title cased else False.
- does not take any parameters.
- syntax — string.istitle()

9.10 **isupper() method:**

- method returns True if a string is UPPER case, else False
- does not take any parameters.
- syntax — string.isupper()

10: SPLITTING AND JOINING:

10.1 **split() method:**

- method breaks down a string into a list of substrings using a chosen seperator.
- syntax — str.split(seperator, maxsplit)
- takes a maximum of two seperator
  - seperator → which delimiter to use
  - maxsplit → maximum number of splits
- returns a list of strings.

⊙ if maxsplit is defined we have a total of maxsplit+1 items.

10.2 **rsplit() method**

- syntax → str.rsplit([seperator, [,maxsplit]])
- takes 2 paramaters:
  - seperator → delimiter splits string starting from the right at specified seperator.
  - maxsplit → maximum no. of split, default = -1

example:

```
s = "Hello, world, Python"
rsplit_list = s.rsplit(",", 1)
#output -> "Hello, world", " Python"
```

10.3 splitlines() method :->

- method splits the string at line break and returns a list.
- syntax -> string.splitlines([keepends])
- parameters -> method takes a single parameters:

  (keepends :-> determines whether the line breaks are included in the resulting list or not. Can be True for any number.
- returns a list of lines in the string.

[splits on the given boundaries]:

- \n -> line feed
- \r -> Carriage return
- \r\n -> carriage + line feed
- \v or \x0b -> line tabulation
- \f -> form feed
- \x1c -> file seperator
- \x1d -> group seperator
- \x1e -> Record seperator

etc.

[example:]

```
grocery = 'Milk\n Bread\n Butter'
print(grocery.splitlines())
```

['Milk', 'Bread', 'Butter'].

10.4 partition() method

- → takes a string parameter [seperator] that seperates the string at the first occurrence of it.
- → returns a 3-tuple containing, the part before the seperator, seperator parameter, and the part after the seperator if the seperator parameter is found in the string.
- the string itself and two empty strings if the seperator parameter is not found.

(string.partition(sep))

[example:]

```
string = "Python is fun"
print(string.partition('is'))
print(string.partition('not'))
```

#output

```
('Python', 'is', 'fun')
('Python is fun', '', '')
```

10.5 rpartition() method

- takes a string parameter [seperator] that seperators the string at the last occurance of it.

11 REPLACING AND TRANSLATING:

11.1 replace() method

- replaces each matching occurence of a substring with another string.
- syntax str.replace(old, new [, count])
- arguments
  - old → substring we want to replace
  - new → new substring
  - count (optional) → no. of times you want to replace the old substring.

note: if count is specified, the replace() replaces all occurance of old substring.

- returns a copy of the string where old is replaced with new substring.

```
string1 =  "Hello World"
print( string1.replace('world', 'Python'))
#output ->   Hello Python
```

11.2 **maketrans() method :**

- static method that creates a one to one mapping of a character to its translation/ replacement.
- creates a Unicode representation of each character for translation.
- ~~syntax~~ → **~~string~~.maketrans(x [, y[, z]])**

  x → if only one arguement is supplied, it must be dict.

  y → if two are passed, it must be string of equal len.

  z → if three are passed, each character in the third arguement is mapped to None.

- returns a translation table with 1-1 mapping of a Unicode ordinal to its translation/ replacement.

example:

```
dict = { "a": "123", "b": 456, "c": "789" }
string = "abc"
print( string.maketrans(dict))

#output -   {97: '123', 98: '456', 99: '789'}
```

11.3 translate() method :

- method takes the translation table to replace / translate characters in the given string as per the mapping table.
- translation table is created by the static method maketrans().
- syntax → string.translate(table).
- takes a single parameter → table → a translation table containing the mapping between two characters usually created by [maketrans().

ex:→

```
s = "Hello, world"
transtable = str.maketrans("aeiou", "12345")
translated = s.translate(transtable)
#output → [H2ll4, w4rld!]
```

12 PADDING AND ALIGNMENT :→

12.1 Center() : methods returns a new centered string after padding it with the specified character.

[syntax() :→ string.center(width, [fillchar])

- width → length of the string with padded characters
- fillchar → padding character (optional)

[Note:→ if fillchar is not provided, whitespace is taken as the default argument.

```
s = "Python"
centered = s.center(10, '-') → "--Python--"
```

12.2 → rjust() method: right just

> method aligns the string up to a given width using a specified character.

- syntax → string.rjust(width, [fillchar])

```
s = "Python"
right_just = s.rjust(10, *)    #output "****Python"
```

12.3 ljust() method: → left just method

> method aligns the string up to a given width

syntax → string.ljust(width, [fillchar])

```
s = "Python"
right_just = s.ljust(10, *) → "Python****"
```

13 [ENCODING AND DECODING:]

```
s = "Hellō, world"
encoded = s.encode('utf-8')    #b 'Hell\xc3\xb3, world!'
decoded = encoded.decode('utf-8')    #"Hellō world!"
```