**Slicing Lists**

Slicing example: omitting start, end indices

my\_str[:5] reads indices 0-4.

my\_str[5:] yields all characters at and after index 5.

For literals:

Hello there. Nice to meet you!

first\_half = usr\_text[:len(usr\_text)//2]

last\_half = usr\_text[len(usr\_text)//2:]

* "Hello there. Ni"
* "ce to meet you!"

If a string's end is 5, then 1:7 or 1:99 are the same as 1:6.

Common string operations:

Graphical user interface, text, application

Description automatically generated

my\_str = 'http://reddit.com/r/python'

protocol = 'http://'

print(my\_str[len(protocol):])



len(protocol) is 7, thus the slicing operation is my\_str[7:], which reads the entire string excluding the first seven characters.



### The slice stride The ***stride*** determines how much to increment the index after reading each element.

my\_str[0:10:2] reads every other element between 0 and 10

{numbers[::2]}') Every even number: 02468

{numbers[1:9:3]} Every third number between 1 and 8: 147

my\_str = 'Agt2t3afc2kjMhagrds!'



print(my\_str[::2])

AttackMars

Assign sub\_lyric by slicing rhyme\_lyric from start\_index to end\_index, which are given as inputs.  
  
Sample output with inputs: 4 7

cow

start\_index = int(input())

end\_index = int(input())

rhyme\_lyric = 'The cow jumped over the moon.'

sub\_lyric = rhyme\_lyric[ start\_index:end\_index]

print(sub\_lyric)

**Aligning text in fields.**

name = 'Joe'

print(f'{name:9}')

=> Joe

\*The output includes at least 9 characters. If the name is less than 9 characters then empty spaces are added to the right of the name

f'{"Bob":<5}'

* Bob

"<" indicates that the inserted string should be left-aligned within the field. Left-aligned is the default for string values.

f'{"Bob":>5}'

* Bob

">" indicates that the inserted string should be right-aligned within the field.

f'{"Bob":^5}'

"^" indicates that the inserted string should be centered within the field.

f'{"Bob":<5}{1:<2}'

* 'Bob 1 '

"Bob" is left-aligned in its field, and two extra spaces are added after to meet the minimum field width requirement. "1" is also left-aligned, and one extra space is added.

f'{"Bob":<5}{1:>2}'

Bob 1

"Bob" is left-aligned in its field, and two extra spaces are added after to meet the minimum field width requirement. "1" is right-aligned, and one extra space is added before the value.

What's the fill character in the following format specification?

{score:\*>4}

\* is the fill character. \* will be used to pad the field to meet the minimum width of 4, if necessary.

If name = 'Sally', what is the result of: {name:@>8}?

* @@@Sally

The field is right-aligned, so @ is used to fill the remaining width on the left.

**String Methods**

### Finding and replacing

* ***replace(old, new)*** —Returns a copy of the string with all occurrences of the substring old replaced by the string new. The old and new arguments may be string variables or string literals.
* ***replace(old, new, count)*** —Same as above, except replace(old, new, count) only replaces the first count occurrences of old.

phrase = “One day I will have two goats”

phrase = phrase.replace('one', 'uno')

phrase = phrase.replace('two', 'dos')

print('Translation:', phrase)

find(x) Returns the index of the first occurrence of item x in the string

my\_str is 'Boo Hoo!':

my\_str.find('!') # Returns 7

***find(x, start)***  Same as find(x), but begins the search at index start:my\_str.find('oo', 2) # Returns 5

**find(x, start, end)** Same as find(x, start), but stops the search at index end

my\_str.find('oo', 2, 4) # Returns -1 (not found)

**rfind(x)** Same as find(x) but searches the string in reverse, returning the last occurrence in the string.

**count(x)** Returns the number of times x occurs in the string.

my\_str.count('oo') # Returns 2

### Comparing strings

relational operators (<, <=, >, >=)

'Yankee Sierra' > 'Amy Wise' True The first character of the left side 'Y' is "greater than" (in ASCII value)

the first character of the right side 'A'.

equality operators (==, !=)

'Hello' == 'Hello' True

'Hello' == 'Hello!' False

If one string is shorter than the other with all corresponding characters equal, then the shorter string is considered less than the longer string.

isalnum() -- Returns True if all characters in the string are lowercase or uppercase letters, or the numbers 0-9.

isdigit() -- Returns True if all characters are the numbers 0-9.

islower() -- Returns True if all cased characters are lowercase letters.

isupper() -- Returns True if all cased characters are uppercase letters.

isspace() -- Returns True if all characters are whitespace.

startswith(x) -- Returns True if the string starts with x.

endswith(x) -- Returns True if the string ends with x.

Note that the methods islower() and isupper() ignore non-cased characters. Ex: 'abc?'.islower() returns True, ignoring the question mark.

Creating new strings from a string

capitalize() -- Returns a copy of the string with the first character capitalized and the rest lowercased.

lower() -- Returns a copy of the string with all characters lowercased.

upper() -- Returns a copy of the string with all characters uppercased.

strip() -- Returns a copy of the string with leading and trailing whitespace removed.

title() -- Returns a copy of the string as a title, with first letters of words capitalized.

The statement name = input().strip().lower() reads in the user input, strips the leading and trailing whitespace, and changes all the characters to lowercase. Thus, user input of 'Bob', 'BOB ', or 'bob' would each result in name having just the value 'bob'.

Good practice when reading user-entered strings is to apply transformations when reading in data (such as input), as opposed to later in the program.

**Splitting and Joining**

***split()***  splits a string into a list of tokens. Each ***token*** is a substring that forms a part of a larger string. A ***separator*** is a character or sequence of characters that indicates where to split the string into tokens.

***join()***  string method performs the inverse operation of split() by joining a list of strings together to create a single string.

Using the split() and join() methods together

The split() and join() methods are commonly used together

For example: A programmer may want to change 'C:/Users/Brian/report.txt' to 'C:\\Users\\Brian\\report.txt', perhaps because a different operating system uses different separators to specify file locations.