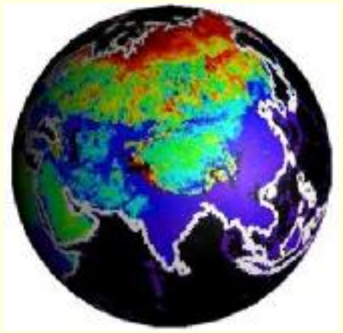
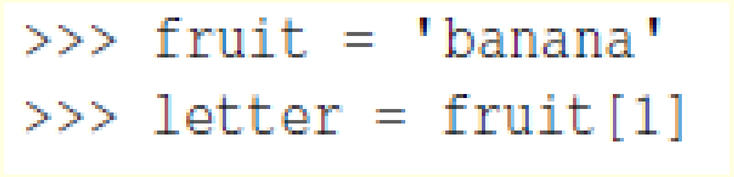


Chapter 8 Strings

Jihn-Fa Jan, Ph.D.
Associate Professor
Department of Land Economics
National Chengchi University

A string is a sequence

- A string is a **sequence (序列)** of characters. You can access the characters one at a time with the **bracket operator (中括號運算子)**:

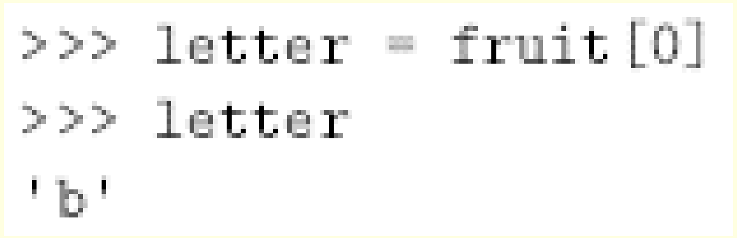


```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

  - The second statement selects character number 1 from fruit and assigns it to letter.
- The expression in brackets is called an **index (索引)**. The index indicates which character in the sequence you want (hence the name).

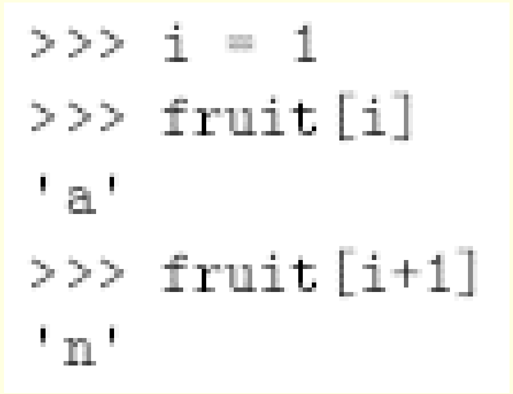
A string is a sequence

- For most people, the first letter of ‘banana’ is b, not a. But for computer scientists, **the index is an offset (偏移量) from the beginning of the string, and the offset of the first letter is zero**.
  - So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th (“two-eth”) letter.



```
>>> letter = fruit[0]
>>> letter
'b'
```

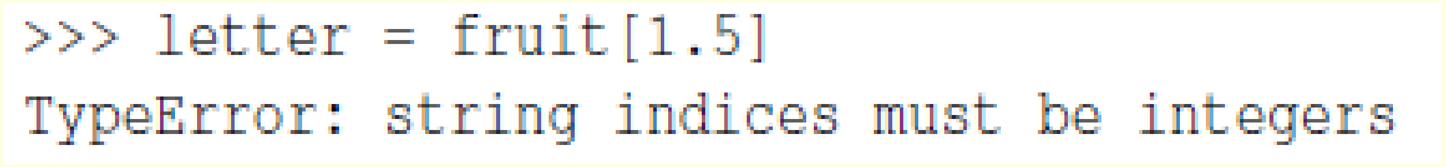
- You can use any expression, including variables and operators, as an index.



```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

A string is a sequence

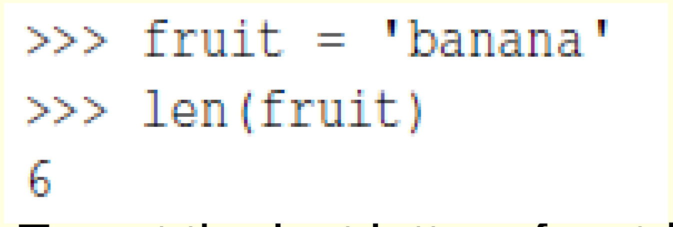
- But the value of the index has to be **an integer**, otherwise you get:



```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

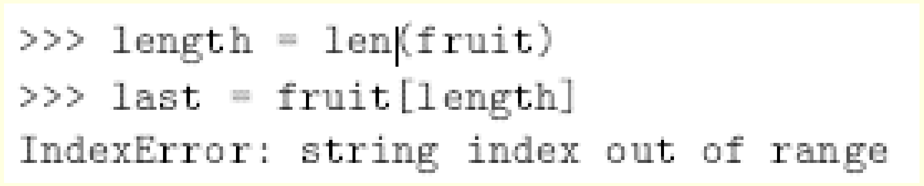
len

- len is a built-in function that returns the number of characters in a string:



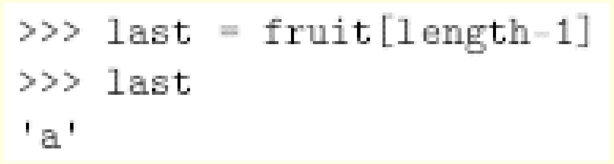
```
>>> fruit = 'banana'
>>> len(fruit)
6
```

- To get the last letter of a string, you might be tempted to try something like this:



```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

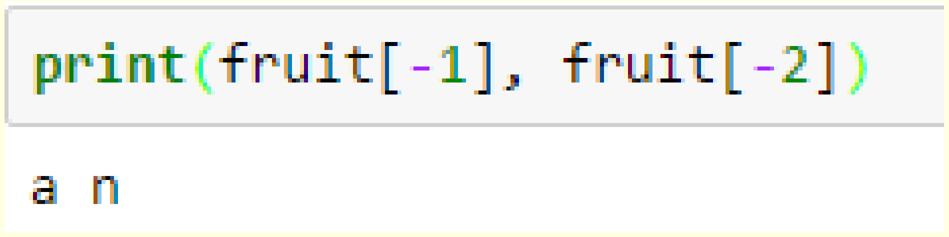
- The reason for the **IndexError** is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered **0 to 5**. To get the last character, you have to subtract 1 from length:



```
>>> last = fruit[length-1]
>>> last
'a'
```

len

- Alternatively, you can use **negative indices**, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.



```
print(fruit[-1], fruit[-2])
```

```
a n
```

Traversal with a for loop

- A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal (巡訪)**. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

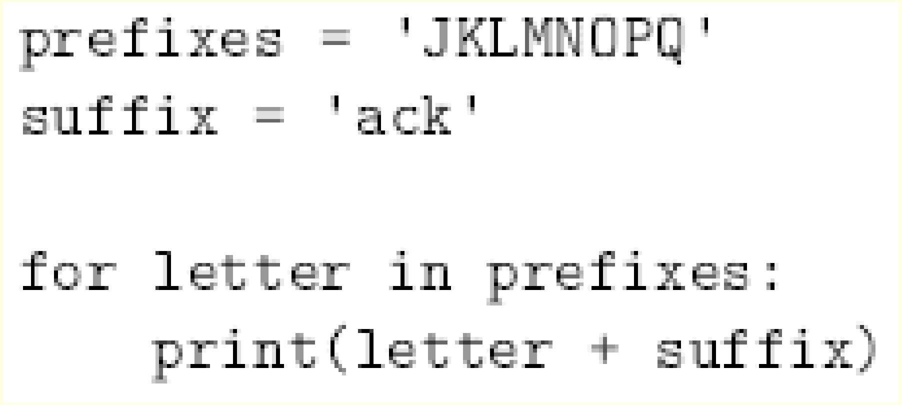
- Another way to write a traversal is with a for loop:

```
for letter in fruit:
    print(letter)
```

  - Each time through the loop, the next character in the string is assigned to the variable letter. The loop continues until no characters are left.

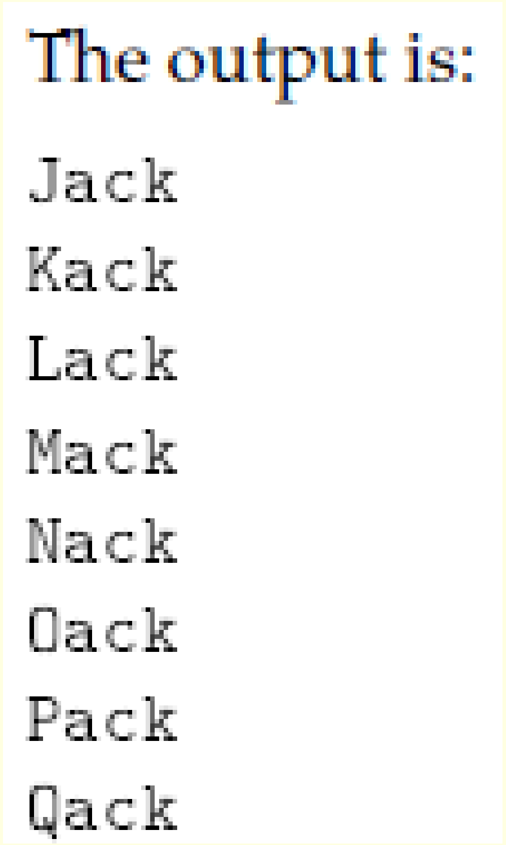
Traversal with a for loop

- The following example shows how to use concatenation (string addition) and a for loop to generate an abecedarian series (that is, in alphabetical order).



```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

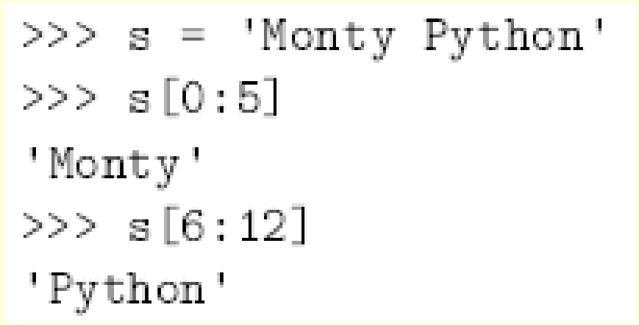


The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

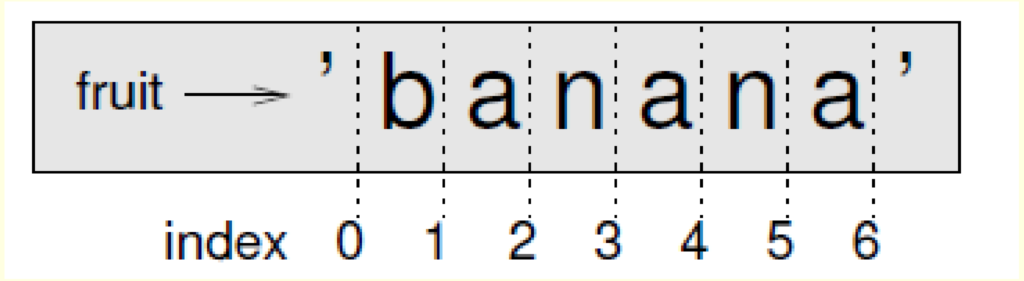
String slices

- A segment of a string is called a **slice (片段)**. Selecting a slice is similar to selecting a character:



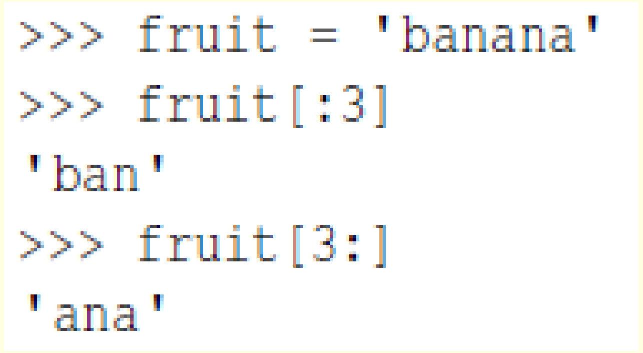
```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

- The operator [n:m] returns the part of the string from the “n-eth” character to the “m-eth” character, **including the first but excluding the last**. This behavior is counterintuitive, but it might help to imagine the indices pointing *between the characters, as in the following diagram:*



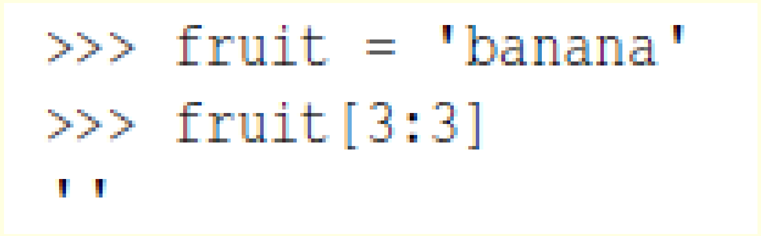
String slices

- If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:



```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

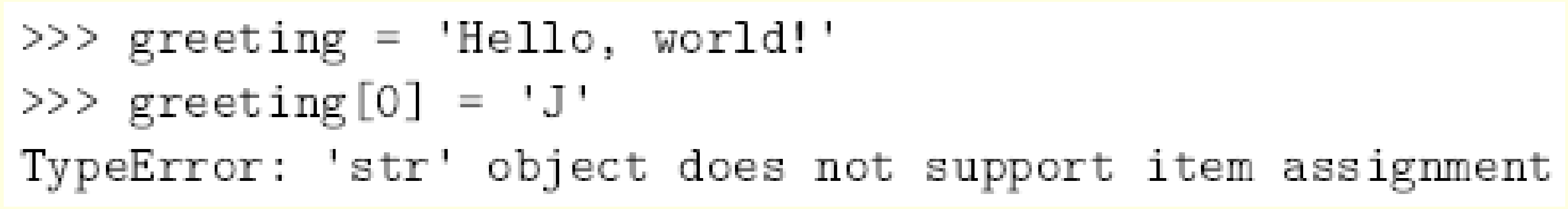
- If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:



```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

Strings are immutable

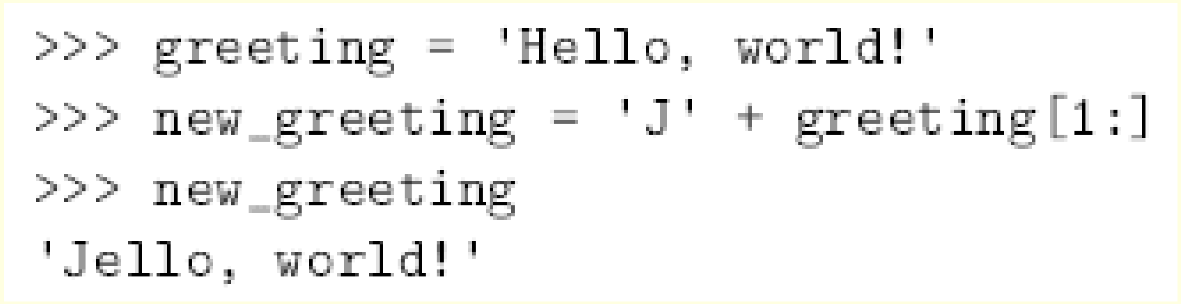
- Strings are **immutable (不可變的)**, which means you can’t change an existing string.



```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

  - The reason for the error is that strings are **immutable**, which means you can’t change an existing string

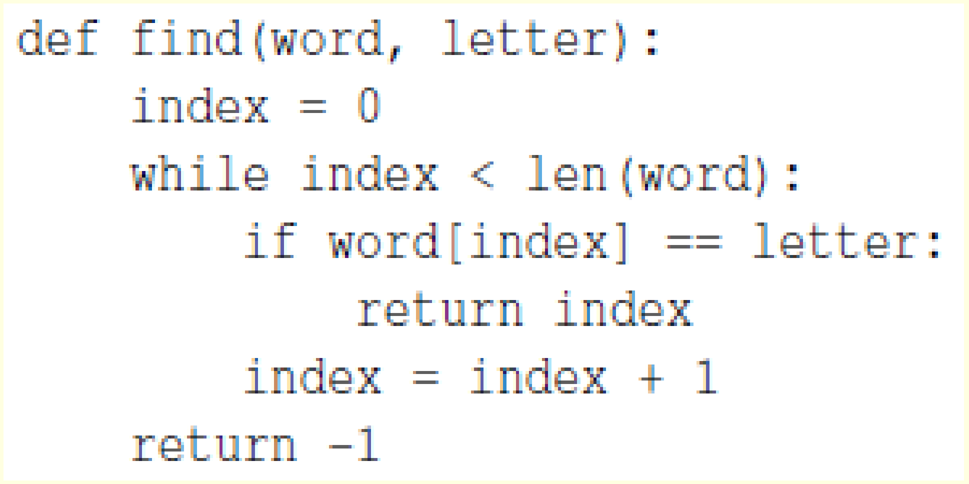
- The best you can do is create a new string that is a variation on the original:



```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

Searching

- The following function returns the index of a character in a string if it finds the character, otherwise, the function returns -1.

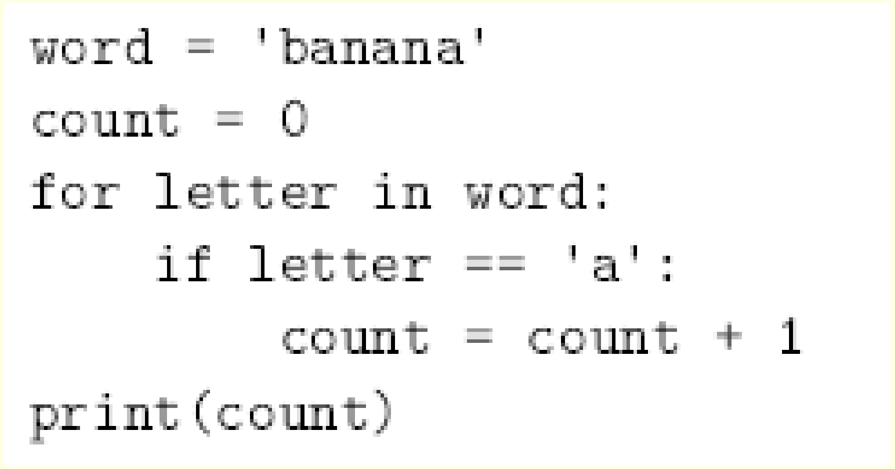


```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

  - If word[index] == letter, the function breaks out of the loop and returns immediately.
- This pattern of computation—traversing a sequence and returning when we find what we are looking for—is a called a **search (搜尋)**.

Looping and counting

- The following program demonstrates another pattern of computation called a counter (計數器).

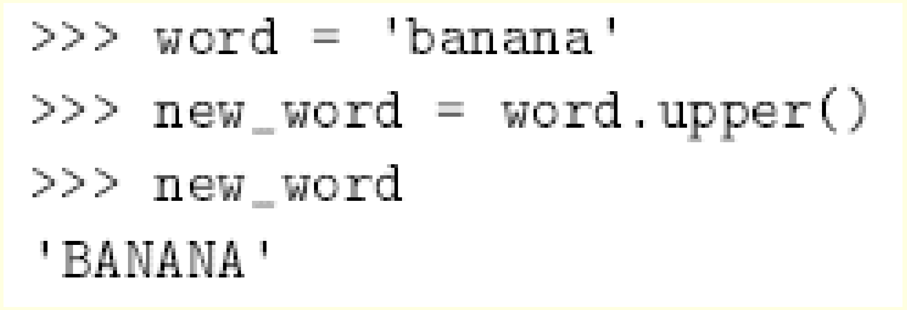


```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

  - The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result—the total number of a's.

String methods

- A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters:

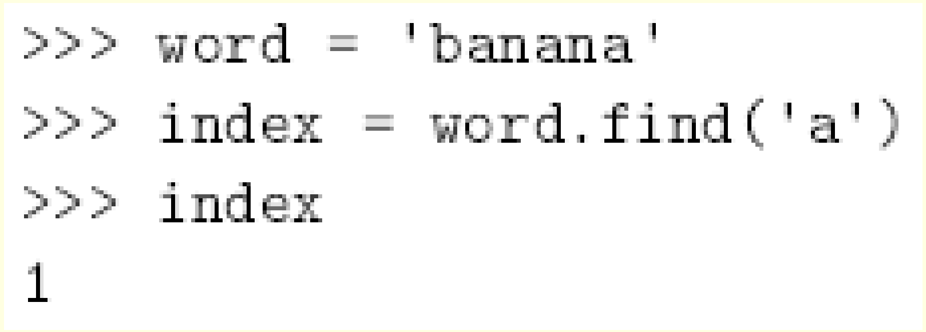


```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

  - Instead of the function syntax upper(word), it uses the method syntax word.upper().
  - This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word.
  - The empty parentheses indicate that this method takes no argument.

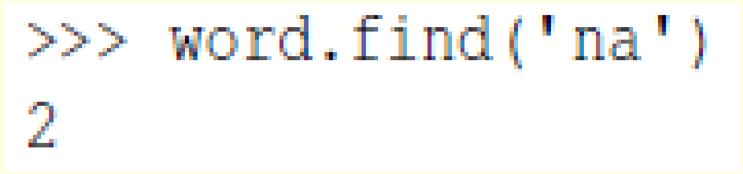
String methods

- A method call is called an **invocation (調用)**; in this case, we would say that we are invoking upper on the Word.
- As it turns out, there is a string method named find that is remarkably similar to the function we wrote:



```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

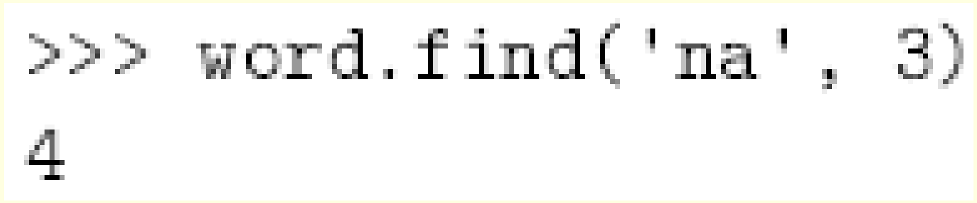
- Actually, the find method is more general than our function; it can find substrings, not just characters:



```
>>> word.find('na')
2
```

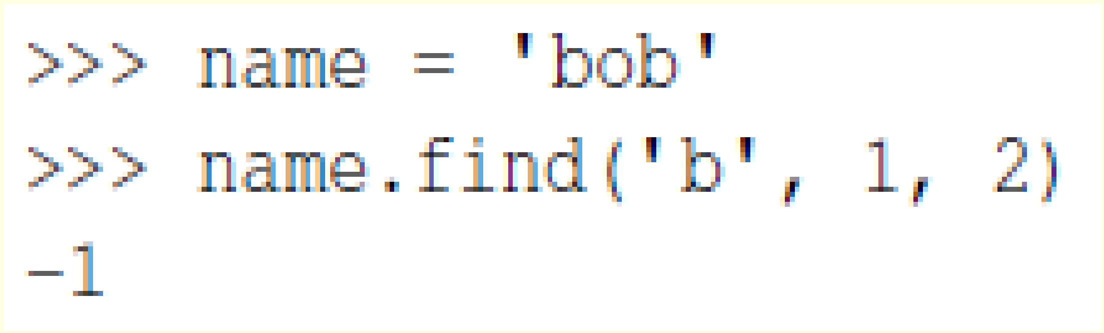
String methods

- The find method can take as a second argument the index where it should start, and as a third argument the index where it should stop:



```
>>> word.find('na', 3)
4
```

  - This is an example of an **optional argument (選擇性引數)**; find can also take a third argument, the index where it should stop.



```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

  - This search fails because b does not appear in the index range from 1 to 2 (not including 2).

The **in** operator

- The word **in** is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

- With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

String comparison

- The comparison operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

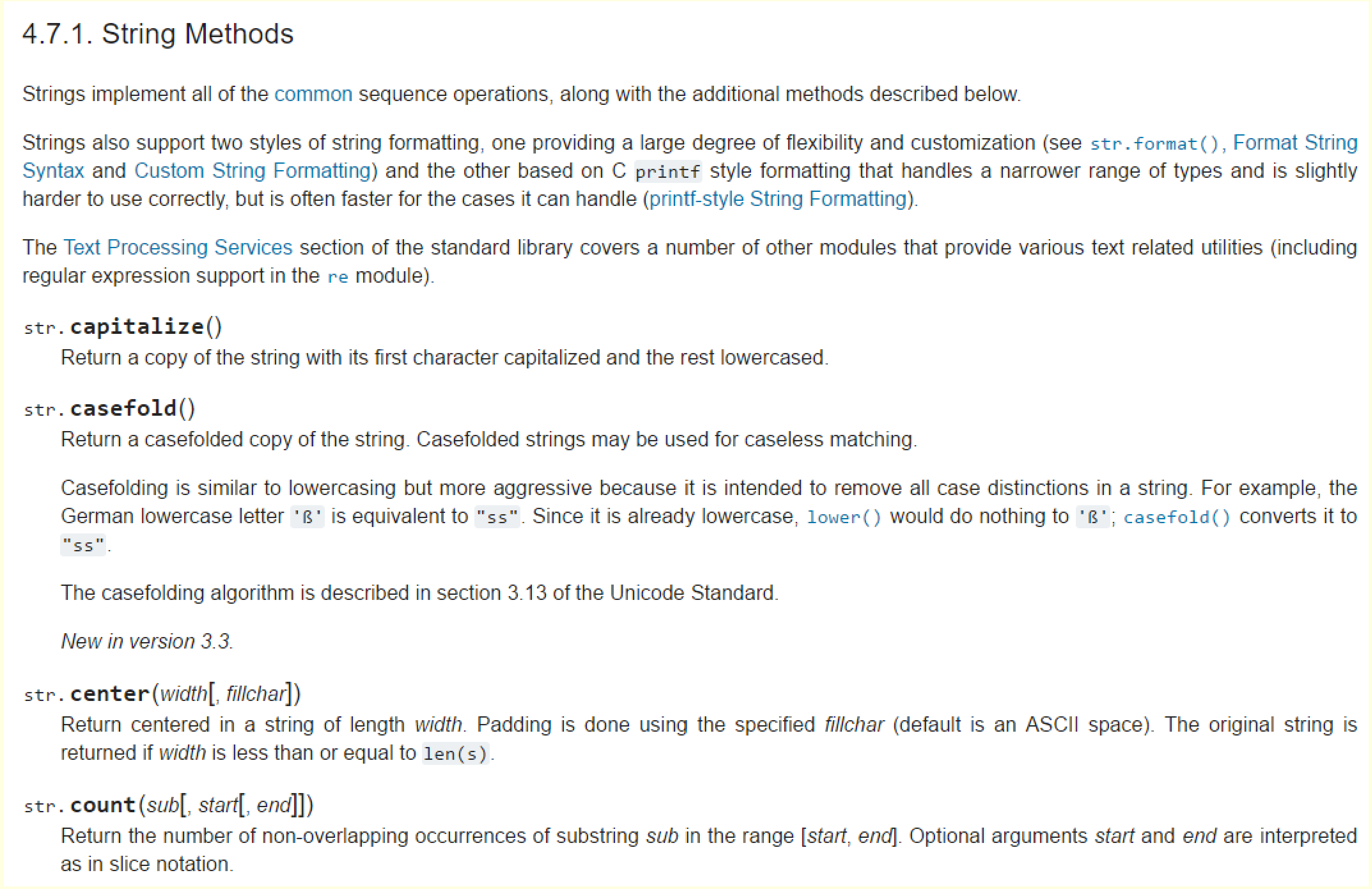
- Other comparison operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

- Python does not handle uppercase and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters.

Python String Methods

- 參考網址：https://docs.python.org/3/library/stdtypes.html#string-methods



4.7.1. String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see `str.format()`, Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the `re` module).

`str.`**`capitalize`**()

Return a copy of the string with its first character capitalized and the rest lowercased.

`str.`**`casefold`**()

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, `lower()` would do nothing to `'ß'`; `casefold()` converts it to `"ss"`.

The casefolding algorithm is described in section 3.13 of the Unicode Standard.

New in version 3.3.

`str.`**`center`**(*width*[, *fillchar*])

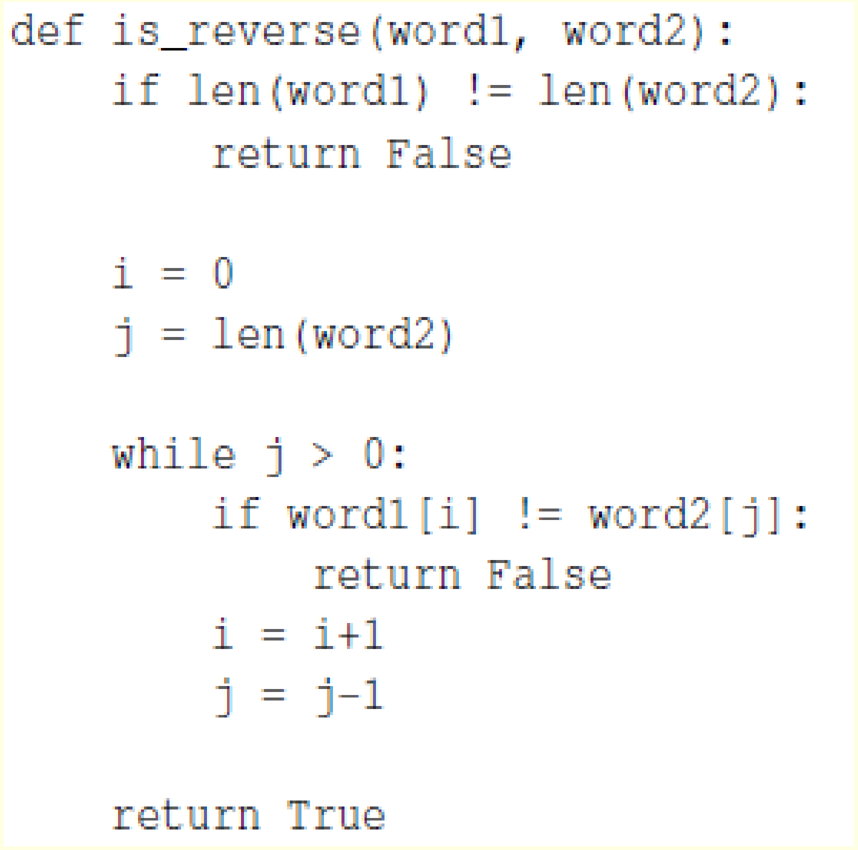
Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

`str.`**`count`**(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Debugging

- When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return True if one of the words is the reverse of the other, but it contains two errors:



```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```