Chapter_8_Strings

March 6, 2024

A string is a sequence, which means it is an ordered collection of other values

A segment of a string is called a slice. Selecting a slice is similar to selecting a character.

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

[2]:
```
if 'Ankit' == 'ankit':
    print('All right!')
else:
    print("not equal")
```

```
not equal
```

[1]:
```
fruit = 'banana'
```

[2]:
```
len(fruit)
```

[2]:
```
6
```

[3]:
```
fruit[-2]
```

[3]:
```
'n'
```

[4]:
```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

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

[5]:
```
fruit = 'banana'
fruit[:3]
```

[5]:
```
'ban'
```

[6]:
```
fruit[3:]
```

[6]:
```
'ana'
```

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:
greeting = 'Hello, world!'
greeting[0] = 'J'
TypeError: 'str' object does not support item assignment The "object" in this case is the string and the "item" is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later (Section 10.10). 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:

[7]:
```
greeting = 'Hello, world!'
greeting[0] = 'J'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-217c78420f62> in <module>
      1 greeting = 'Hello, world!'
----> 2 greeting[0] = 'J'

TypeError: 'str' object does not support item assignment
```

[8]:
```
greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
```

[8]:
```
'Jello, world!'
```

[9]:
```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

0.1 8.8 String methods

Strings provide methods that perform a variety of useful operations. 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.

Instead of the function syntax upper(word), it uses the method syntax word.upper().

[10]:
```
word = 'banana'
new_word = word.upper()
new_word
```

[10]:
```
'BANANA'
```

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 arguments

A method call is called an invocation; in this case, we would say that we are invoking upper on word.

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

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

[1]:
```
1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

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

[2]:
```
word.find('na')
```

[2]:
```
2
```

By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start:

[3]:
```
word.find('na', 3)
```

[3]:
```
4
```

This is an example of an optional argument; find can also take a third argument, the index where it should stop:

[4]:
```
name = 'bob'
>>> name.find('b', 1, 2)
```

[4]:
```
-1
```

This search fails because b does not appear in the index range from 1 to 2, not including 2. Searching up to, but not including, the second index makes find consistent with the slice operator.

0.2 8.9 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:

```
[5]: 'a' in 'banana'
```

```
[5]: True
```

```
[6]: >>> 'seed' in 'banana'
```

```
[6]: False
```

```
[ ]:
```