Lecture – Strings

Python Strings

- Access a single character in a string using its index position.
- Find the length of a string.
- Use in / not in operator.
- Traverse a string with a loop.
- Accessing a substring (slice) of a string (**pre-recorded video**)
- String methods.
- Use string comparison (>, <, >=, <=, ==, !=).

Adapted from Chapter 8 of Think Python: How to Think Like a Computer Scientist (online reading list)

String Data Type – we already know:

1. A string is a sequence of characters
2. A string literal uses quotes ‘Hello’ or “Hello”
3. For strings, + means “concatenate”
4. When a string contains numbers, it is still a string
5. We can convert numbers in a string into a number using int()

Indexing

- You can access a single character in a string with the bracket operator []. The number in the brackets is called an **index**. Note, the index of the first letter is zero.

```
fruit = 'banana'
print(fruit[0])   # b (first character)
print(fruit[1])   # a (second character)
```

- The value of the index has to be an integer. As an index, you can use an expression that contains variables and operators:

```
i = 1
fruit[i]
fruit[i+1]
```

Length of a string - len()

len() - function returns the number of characters in string:

```
fruit = 'banana'
length = len(fruit)                # 6
```

- Remember, there is no letter in the sting with the index 6 (only 0 to 5). To get the last character, you have to subtract 1 from length:

```
last = fruit[length-1]
# a
```

0	1	2	3	4	5
b	a	n	a	n	a
-6	-5	-4	-3	-2	-1

- Or use negative indices, which count backward from the end of the string. E.g., `fruit[-1]` yields the last letter, `fruit[-2]` yields second to last.

Using in operator

- The **in** keyword can be used to check to see if one string is "in" another string
- The **in** expression is a logical expression and returns True or False and can be used in an if statement
- There is also a **not** in operator.

```
fruit = 'banana’
'n' in fruit        # True
'm' in fruit        #  False
'nan' in fruit      # True

if 'a' in fruit :
    print ('Found it!‘)     # Found it!

'x' not in 'apple’
```

Traversal with a for loop: By Item

- Python has a version of the for loop that operates exclusively on strings. We can replace the range() function as shown in Example 1:

```
fruit = "banana"
for letter in fruit:
    print(letter)
```

- Each time through the loop, the next character in the string is assigned to variable **letter** until no characters are left. Example 2:

```
s = "python rocks"
for ch in s:
    print(ch)
```

Traversal with a for loop

- How many times is the word HELLO printed by the following statements?

```
s = "python rocks"
for ch in s:
    print("HELLO")
```

a) 10
b) 11
c) 12
d) Error, for loop must use range()

String Slices (pre-recorded video)

- A string segment is called a **slice** - similar to selecting a character:

```
s = 'WestMinster!'
print(s[0:4])          # West
print(s[4:11])         # Minster
```

0	1	2	3	4	5	6	7	8	9	10	11
W	e	s	t	M	i	n	s	t	e	r	!

- The first index number is where the slice starts, and the second index number is 1 before the slice ends. E.g.,

```
[0:4]      index range from 0 to 4 (not including 4)
[4:11]     index range from 4 to 11 (not 11)
```

String Slices (pre-recorded video)

- 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'
print(fruit[:3]) # ban
print(fruit[3:]) # ana
```

Self-Check Question 1 **(pre-recorded video)**

```
singers = "Peter, Paul, and Mary"
```

- What will print?

```
print(singers[2])      _____________

print(singers[0:5])    _____________

print(singers[7:11])   _____________

print(singers[17:21])  _____________
```

Strings Are Immutable

- You can’t change an existing string so create a new string as a variation of original:

```
greeting = 'Hello, world!'
new_greeting = 'J' + greeting[1:]
print(new_greeting)        # Jello, world!
print(greeting)            # Hello, world!
```

- Example concatenates a new first letter onto a slice of greeting (original string remains the same).

String Method Syntax

- Python string method **syntax example:** `word.upper()`
- This dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word.

```
word = 'banana'
new_word = word.upper()
print(new_word)                    # BANANA
```

- lower()takes a string and returns a new string with all lowercase letters.

String methods

- Lots of string methods to experiment with:

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

- Note that methods that return strings do not change the original.

Method	Parameters	Description
capitalize	none	Returns a string with first character capitalized, rest lower
strip	none	Returns a string with the leading and trailing whitespace removed
count	item	Returns the number of occurrences of item
replace	old, new	Replaces all occurrences of old substring with new
find	item	Returns the leftmost index where substring found, or -1 if not found

String Comparison

- Comparison operators also work on strings. To check if two strings are equal:

```
word = 'banana'
if word == 'banana':
    print('They match.')
```

- But what about this example?

```
print("apple" == "Apple")
```

- Uppercase and lowercase letters are different from one another.
- A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

String Comparison

- Other relational operators (<, >, !=, <=, >=) work on strings.
- Python compares string lexicographically. Similar to the alphabetical order you would use with a dictionary, except that all the uppercase letters come before all the lowercase letters.

'apple' < 'banana' evaluates to True.

E.g., *apple* would be less than (**come before**) the word banana. After all, *a* is before *b* in the alphabet.

' banana’ < 'cherry’ evaluates to True.

- However, all the uppercase letters come before all the lowercase letters.

‘apple’ < ‘Apple’ evaluates to False

‘Apple’ < ‘apple’ evaluates to True

'Zeta' < 'Apricot' evaluates to False.

'Zebra' <= 'aardvark' evaluates to True

Summary Self-Check – Lecture Part 1

1. indexing - Access a character in a string using its index position.

   'Was'[2] evaluates to? _____

2. len() function returns the number of characters in a string.

   len('Monday PM') evaluates to? _____

3. slicing ([:]) A *slice* is a substring of a string.

   'bananas and cream'[3:6] evaluates to? _____

   'bananas and cream'[1:4] evaluates to? _____

Summary Self-Check – Lecture Part 1

4. What is printed by the following statements:

```
s = "Ball"
s[0] = "C"
print(s)
```

a) Ball
b) Call
c) Error

Summary Self-Check – Lecture Part 2

5. The comparison operators **(**>, <, >=, <=, ==, !=**)** work with strings.

'cat' < 'dog'	evaluates to True or False?
'Wasp' < 'Bear'	evaluates to True or False?
'Zebra' <= 'alligator'	evaluates True or False?

6. The in operator tests whether one string is inside another string.

'heck' in "I'll be checking you."	evaluates True or False ?
'cheese' in "I'll be checking you."	evaluates True or False?
'cheese' not in "I'll be checking you."	evaluates True or False?

Lecture Summary

Python Strings

- indexing ([]) **-** 'Was'[2] evaluates to **'s'.**
- len() function returns the number of characters in a string.
- Use in / not in operator - `'b' in 'banana'`
- Traverse a string with a loop.
- slicing ([:]) A *slice* is a substring of a string (**pre-recorded video**)
- String methods.
- String comparison (>, <, >=, <=, ==, !=).