```
>> help(str.replace)
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old
replaced by new.

s = 'Mississippi'
t = len(s.replace('ss', 'a'))
```

Miaiaippi

What is the value of t after the above code executes?

A. 11
B. "ss"
C. 9
D. "Miaaiaaippi"

Answer: C
Recall that len returns the length of its argument, so t must be an integer. Here all copies of "ss" "Mississippi" are replaced with "a" to create "Miaiaippi", which has length 9.

```
>> help(str.replace)
replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old
replaced by new.

s = 'Mississippi'
t = (s.upper()).replace('ss', 'a')
```

What is the value of `t` after the above code executes?

A. "Mississippi"
B. "Miaiaippi"
C. "MISSISSIPPI"
D. "MIAIAIPPI"

Answer: C
The upper method creates a new string "MISSISSIPPI" (all caps). When we invoke replace on that string (returned by upper), there are no instances
of lower case "ss" (Python distinguishes between upper and lower case letters), and so no replacements will be performed. Thus t is "MISSISSIPPI".

```
a = [2, 4, 6, 8]
a.remove(4)
a.pop(2)
```

pop(index)
Remove value at index in list

[2, 6, 8]
0 1 2

[2, 6]

What is the value of a after the above code executes

A. [2, 4]
B. [6, 8]
C. [2, 6]
D. [2, 8]

Answer: C
After a.remove(4), a is [2, 6, 8], after a.pop(2), the item at index 2 is removed, so a is [2,6]

```
a = [2, 4, 6, 8]
a.pop(2)
a.remove(4)
```

What is the value of a after the above code executes

A. [2, 4]
B. [6, 8]
C. [2, 6]
D. [2, 8]

Answer: D
After a.pop(2), the item at index 2 is removed, so a is [2,4,8] and a.remove(4), a is [2,8]

Assume `value` is defined and not empty. Which of the following is valid only if `value` is a string or a list, but not for both types?

A.
```
value.append("a")
```
B.
```
value[0]
```
C.
```
for c in value:
    print(c)
```
D. All are valid for both types

E. All are only valid for one of string or list

Answer: A

The lower method is only defined on strings. There is no notion of transforming a list of arbitrary values to lower case. B and C work on both types.

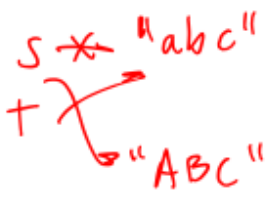
```
s = "CS"
t = "cs"
s.lower()
for i in range(len(s)):
    print(s[i] + t[i])
```

When run via the green arrow in Thonny, what will the following code print?

A. cc ss	B. Cc Ss
C. CSCS cscs	D. cC sS

Answer: B
Invoking the lower method does not change the string s, it remains “CS”. The loop executes two iterations, printing the concatenation of the characters in s and t at index 0, then index 1.

```
s = "abc"
t = s
s = s.upper()
```



What is the value of t after the above code executes?

A. "abc"
B. "ABC"
C. “s”
D. "S"

Answer: A
When s is assigned to t, both point to “abc”. When we invoke s.upper(), that doesn’t change that string, but instead creates a new string “ABC”. After assigning that value to s, it now points to “ABC”, but that statement does not change that t points to “abc”. Check out what happens at pythontutor.com:
https://pythontutor.com/visualize.html#code=s%20%3D%20%22abc%22%0At%20%3D%20s%0As%20%3D%20s.upper%28%29&cumulative=false&curInstr=0&heapPrimitives=true&mode=display&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false