QUIZ • 30 MIN

Week 1 Exercise

Submit your assignment

DUE DATE Aug 16, 2:59 PM CST **ATTEMPTS** 3 every 8 hours

Receive grade

TO PASS 80% or higher

Week 1 Exercise

TOTAL POINTS 12

1. To determine whether a string is a palindrome, the third algorithm we explored was: 1 point

   1. Compare the first character to the last character, the second to the second last, and so on.
   2. Stop when the middle of the string is reached. (That means that the middle character is not compared with anything.)

   We implemented this algorithm using a `while` loop, but we could have used a `for` loop. The Python code is posted as a Reading, but here is the function header:

Try again

Grade 100%

View Feedback

We keep your highest score

```
def is_palindrome_v3(s):
    """ (str) -> bool

    Return True if and only if s is a palindrome.

    >>> is_palindrome_v3('noon')
    True
    >>> is_palindrome_v3('racecar')
    True
    >>> is_palindrome_v3('dented')
    False
    """
```

The function bodies below all use `for` loops to try to solve the palindrome problem. Select the one(s) that correctly implement the algorithm.

Hint: try tracing the code on a string of length 1, and then on a string of length 2.

- [ ]
```
for i in range(len(s) // 2):
    if s[i] != s[len(s) - i]:
        return False

return True
```

- [x]
```
for i in range(len(s) // 2):
    if s[i] != s[len(s) - i - 1]:
        return False

return True
```

- [ ]
```
for i in range(len(s) // 2 + 1):
    if s[i] != s[len(s) - i - 1]:
        return False

return True
```

- [x]
```
j = len(s) - 1
for i in range(len(s) // 2):
    if s[i] != s[j - i]:
        return False

return True
```

2. A string `s1` is an *anagram* of string `s2` if its letters can be rearranged to form `s2`. For example, `'listen'` is an anagram of `'silent'`, and `'admirer'` is an anagram of `'married'`. For this question, a word is considered to be an anagram of itself. 1 point

   Consider this code:

```
def is_anagram(s1, s2):
    """ (str, str) -> bool

    Return True if and only if s1 is an anagram of s2.

    >>> is_anagram("silent", "listen")
    True
    >>> is_anagram("bear", "breach")
    False
    """
```

Select the algorithm(s) that can be used to implement `is_anagram`.

- [ ] 1. Create a list of the characters in `s1`.
  2. Create a list of the characters in `s2`.
  3. For each item in the list of characters from `s1`, remove one occurrence of that item from the list of characters from `s2` (if it exists).
  4. If the list of characters from `s2` becomes empty, `s1` is an anagram of `s2`.

- [x] 1. Create a dictionary `d1` in which each key is a letter from `s1` and each value is the number of occurrences of that letter in `s1`.
  2. Create a dictionary `d2` in which each key is a letter from `s2` and each value is the number of occurrences of that letter in `s2`.
  3. If `d1 == d2`, then `s1` is an anagram of `d2`.

- [x] 1. Create a list `L1` of the characters in `s1`.
  2. Create a list `L2` of the characters in `s2`.
  3. Sort both lists.
  4. If `L1 == L2`, `s1` is an anagram of `s2`.

- [ ] For each letter in `s1`, count the number of occurrences of the letter in `s1` and count the number of occurrences of the letter in `s2`. If each letter in `s1` occurs the same number of times in `s1` and `s2`, then `s1` is an anagram of `s2`.

3. Consider this code: 1 point

```
def count_startswith(L, ch):
    """ (list of str, str) -> int

    Precondition: the length of each item in L is >= 1, and len(ch) == 1

    Return the number of strings in L that begin with ch.

    >>> count_startswith(['rumba', 'salsa', 'samba'], 's')
    2
    """

    ch_strings = []

    for item in L:
        if item[0] == ch:
            ch_strings.append(item)

    return len(ch_strings)
```

Select the algorithm that *best* describes the approach taken in the function defined above.

- (•) 1. Use a list accumulator.
  2. For each item in `L`, if the item begins with `ch`, add it to the accumulator.
  3. Return the length of the accumulator.

- ( ) 1. Create a new list that contains the same values as `L`.
  2. For each item in `L`, if the item *does not* begin with `ch`, remove it from the new list.
  3. Return the length of the new list.

- ( ) 1. Use an integer accumulator.
  2. For each item in `L`, if the item begins with `ch`, add 1 to the accumulator.
  3. Return the accumulator.

4. Consider this function header: 1 point

```
def count_startswith(L, ch):
    """ (list of str, str) -> int

    Precondition: the length of each item in L is >= 1, and len(ch) == 1

    Return the number of strings in L that begin with ch.

    >>> count_startswith(['rumba', 'salsa', 'samba'], 's')
    2
    """
```

Select the code fragment(s) that correctly implement the function according to the header above.

- [ ]
```
    startswith = L[:]

    for item in L:
        if item.startswith(ch):
            startswith.remove(item)

    return len(startswith)
```

- [x]
```
    startswith = L[:]

    for item in L:
        if item.startswith(ch):
            startswith.remove(item)

    return len(L) - len(startswith)
```

- [x]
```
    startswith = L[:]

    for item in L:
        if not item.startswith(ch):
            startswith.remove(item)

    return len(startswith)
```

- [ ]
```
    count = 0

    for item in L:
```