Week 1 Exercise

测验, 12 个问题

1
point

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

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:

```
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.

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

return True
```

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

return True
```

[✓]
```
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):
  if s[i] != s[len(s) - i]:
    return False

return True
```

1
point

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.

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` .

☐ 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` .

[✓]
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` .

☐
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` .

[✓]
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**.

1
point

3。
Consider this code:

```
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 an integer accumulator.
2. For each item in **L**, if the item begins with **ch**, add 1 to the accumulator.
3. Return 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 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
point

4。

Consider this function header:

```
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.

[ ]

```
count = 0

for item in L:
    if item.startswith(ch):
      count = count + 1
      return count
    else:
      return count
```

[ ]

```
startswith = L[:]

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

return len(startswith)
```

[✓]

```
startswith = L[:]

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

return len(startswith)
```

[✓]

```
startswith = L[:]

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

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

1
point

5。
Consider this code, in which **s** refers to a string:

```
digits = ""

for ch in s:
    if ch.isdigit():
        digits = digits + ch
```

Select the code fragment(s) that will produce the same value for `digits` .

[✓]

[ ]
```
digits = ''

for i in range(len(s)):
    if s[i].isdigit():
        digits = digits + s[i]
```

[✓]
```
indices = []
digits = ''

for i in range(len(s)):
    if s[i].isdigit():
        indices.append(i)

for index in indices:
    digits = digits + s[index]
```

[✓]
```
digits = ''

for ch in s:
    if ch in '0123456789':
        digits = digits + ch
```

[ ]
```
digits = ''

for ch in s:
    if ch == '0123456789':
        digits = digits + ch
```

1
point

6。
Consider this function header and docstring:

```
def is_one_to_one(d):
    """ (dict) -> bool

    Return True if and only if no two of d's keys map
        to the same value.

    >>> is_one_to_one({'a': 1, 'b': 2, 'c': 3})
    True
    >>> is_one_to_one({'a': 1, 'b': 2, 'c': 1})
    False
    >>> is_one_to_one({})
    True
    """
```

Select the algorithm(s) that can be used to implement This algorithm matches the description of **`is_one_to_one`** ..

[✓]
1. Put all the values from **`d`** into a list.
2. For each value in the list, count how many times it appears in the list. If a value appears more than once in the list, return **`False`** .
3. Once all the values in the list have been processed, return **`True`** because we didn't see a duplicate value.

[ ]
1. Use a list accumulator to keep track of the values we've seen so far.
2. For each key in **`d`** , if the value associated with that key has already been seen, return **`False`** ; otherwise, return **`True`** .

[✓]

1. Put all the values from **d** into a list.
2. Make a copy of that list.
3. Remove all the duplicate items from the second list.
4. Compare the lengths of the two lists. If they are equal, return **True** because that means that there were no duplicate items; otherwise, return **False** .

[✓]
1. Use a list accumulator to keep track of the values we've seen so far.
2. For each key in **d** , if the value associated with that key has already been seen, return **False** ; otherwise, append it to the list of values that we've seen so far.
3. Once all the keys have been processed, return **True** because we didn't see a duplicate value.

1
point

7。
Consider this code:

```
def is_one_to_one(d):
    """ (dict) -> bool

    Return True if and only if no two of d's keys map
        to the same value.

    >>> is_one_to_one({'a': 1, 'b': 2, 'c': 3})
    True
    >>> is_one_to_one({'a': 1, 'b': 2, 'c': 1})
    False
    >>> is_one_to_one({})
    True
    """

    seen = []  # The values that have been seen so
        far.
    for k in d:
        if d[k] in seen:
             return False
        else:
            seen.append(d[k])
    return True
```

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

1. Put all the values from **d** into a list.
2. For each value in the list, count how many times it appears in the list. If a value appears more than once in the list, return **False** .
3. Once all the values in the list have been processed, return **True** because we didn't see a duplicate value.

1. Use a list accumulator to keep track of the values we've seen so far.

2. For each key in `d`, if the value associated with that key has already been seen, return `False`. Otherwise, return `True`.

[✓ ●
1. Use a list accumulator to keep track of the values we've seen so far.
2. For each key in `d`, if the value associated with that key has already been seen, return `False`. Otherwise, append it to the list of values that we've seen so far.
3. Once all the keys have been processed, return `True` because we didn't see a duplicate value.]

---

1 point

8。
You are conducting a survey with an ordered list of questions to which people can answer `'Y'` or `'N'` ("yes" or "no"). You need to keep track of each person's responses so that you can find out which questions they answered `'Y'` to and which questions they answered `'N'` to. Which of the following data structures *could be used* to represent one person's responses to the questions?

- [✓] `dict of {str: list of int}`, where each key is a response (either `'Y'` or `'N'`) and each value is list of question numbers for which the person provided that response
- [ ] `dict of {str: int}`, where each key is a response (either `'Y'` or `'N'`) and each value is a question number
- [✓] `list of str`, where each character is either `'Y'` or `'N'`
- [ ] two `list of str`, where one list contains all the `'Y'` s and the other contains all the `'N'` s

---

1 point

9。
A cycling *time trial race* is a race in which each cyclist aims to finish in the fastest time. (All the cyclists start at different times, rather than everyone starting at the same time.) There may be ties.

Your job is to determine which data structure to use to keep track of the names and times of the cyclists. The data structure will initially be empty and when a cyclist crosses the finish line, their data will be added to the data structure.

Which of the following data structures *could be used* to represent all the cyclists and their times? You may assume that the names of the cyclists are unique.

- [✓]

A **`list of [str, float] lists`** , where each inner list represents **`[cyclist, time]`** . The outer list is ordered from fastest time to slowest time.

- [✓] A **`dict of {str: float}`** , where each key is a cyclist and each value is a time.
- [✓] Parallel lists, where one is a **`list of str`** and the other is a **`list of float`** : the list of cyclists, and the list of their times. The lists are sorted by the order in which the cyclists cross the finish line (which is **not** the same as how long they took).
- [ ] A **`dict of {float: str}`** , where each key is a time and each value is the cyclist who finished with that time.

1
point

10。
This question is a followup to the previous question about cycling time trials.

Now that the race is over, you need to determine the three fastest cyclists. (Assume there are no ties among the top three.)

Which data structure will make it easiest to look up the three fastest cyclists? You may assume that the names of the cyclists are unique.

- ○ A **`dict of {float: str}`** , where each key is a time and each value is the cyclist who finished with that time.
- ○ A **`dict of {str: float}`** , where each key is a cyclist and each value is a time.
- ● A **`list of [str, float] lists`** , where each inner list represents **`[cyclist, time]`** . The outer list is ordered from fastest time to slowest time.
- ○ Parallel lists, where one is a **`list of str`** and the other is a **`list of float`** : the list of cyclists, and the list of their times. The lists are sorted by the order in which the cyclists cross the finish line (which is **not** the same as how long they took).

1
point

11。

A *weather file* has the following format, where each city line contains a city name and the number of millimeters of precipitation for each day of that month.

Monthly data is separated by a single blank line.

```
Jan
Toronto: 3.5,0,1.8,0,...
Montreal: 1.5,0,0,0,...
Vancouver: 0,8.6,23.6,19.2,...

Feb
Toronto: 0,0,1.5,1.2,...
Montreal: 0.4,0,0.3,0.4,...
Vancouver: 14,0,0.2,0.2,...

...

Dec
Toronto: 1.3,13.7,0.6,3.8,...
Montreal: 0,7.7,0,6.9,
Vancouver: 15.2,21.4,11.4,14.6,...
```

This problem involves a *weather dictionary* in which the keys are month names and the values are dictionaries containing information about precipitation in cities for that month.

In each of the nested dictionaries, the keys are city names and the values are lists of millimetres of precipitation for each day that month, in order. We'll refer to these nested dictionaries as "city to precipitation" dictionaries.

Select the algorithm(s) that can be used to determine the city that had the maximum total precipitation in February. (You can break ties any way you like, or you can assume that there are no ties. Either is fine.)

[✓]

1. Build the weather dictionary.
2. Look up key **'Feb'** in the weather dictionary to get the "city to precipitation" dictionary for February.
3. Create a dictionary where the keys are cities and the values are the sum of the precipitation amounts for that city for February.
4. Find the maximum value in that dictionary of city maximums. The answer is the key associated with that maximum.

[✓]

1. Build the weather dictionary.
2. Look up key **'Feb'** in the weather dictionary to get the "city to precipitation" dictionary for February.
3. Iterate through the cities in that dictionary, calculating the sum of the precipitation amounts for that city. Keep track of the city that has the most precipitation so far.
4. Once the iteration is complete, whichever city had the most precipitation is the answer.

[ ]

1. Build the weather dictionary.
2. Look up key **'Feb'** in the weather dictionary to get the "city to precipitation" dictionary for February.

3. Create a list containing the sum of the precipitation amounts from each of the city precipitation lists for February. Also create a parallel list containing the city names.
4. Sort the list containing the sum of the precipitation amounts so that the largest value is last. The answer is the city in the parallel list at the last position.

☐

1. Build the weather dictionary.
2. Look up key `'Feb'` in the weather dictionary to get the "city to precipitation" dictionary for February.
3. Invert that dictionary so that the keys are the lists of precipitation amounts and the values are the cities.
4. For each key in this inverted dictionary, sum the precipitation amounts in that list, and keep track of the list that had the largest sum.
5. Once the iteration is complete, whichever list had the most precipitation is the key. To get the answer, look its value up in the inverted dictionary to get the corresponding city name.

1
point

12。
This question also involves a weather file and a weather dictionary. Please see the previous question for details.

Select the algorithm(s) that can be used to make a list of the days in which no city had precipitation — we'll call this the "zero-precipitation list". Each day should be a tuple of (month name, day number). For example, if all cities in the "city to precipitation" dictionary for February had no precipitation on the very first day, then `('Feb', 1)` would be in the resulting list.

Note: in the weather dictionary, the list of precipitation amounts starts at index 0; to get the corresponding day number for an index, just add 1.

Hint: You will probably find this question easier to do if you take notes about the problem on a piece of paper, including drawing a small example weather dictionary.

✓

1. Build the weather dictionary.
2. Create a "zero-precipitation" list containing all days of the year from `('Jan', 1)` through `('Dec', 31)` .
3. Iterate over the months to get each "city to precipitation" dictionary. For each of these dictionaries:

a) For each city in the current "city to precipitation" dictionary, iterate over the precipitation amounts. Because we know the current month and day number, we will remove from the "zero-precipitation" list any day that has a non-zero precipitation amount.

4. One this process is complete, the "zero-precipitation" list contains only the (month, day number) for days in which no city had precipitation.

[✓]

1. Build the weather dictionary.
2. Create an empty "zero-precipitation" list to accumulate the answer.
3. Iterate over the months to get each "city to precipitation" dictionary. For each of these dictionaries:

a) Build a dictionary where each key is a city from the current "city to precipitation" dictionary, and each value is a list of the day numbers on which the city had no precipitation.

b) Iterate over the values in that dictionary to build a list containing the day numbers that appear in all the lists of day numbers.

c) Iterate over the list of day numbers, appending tuples containing the current month name and the day number to the "zero-precipitation" list.

[ ]

1. Build the weather dictionary.
2. Iterate over the months to get each "city to precipitation" dictionary. Make a "day to precipitation" dictionary where the keys are all the days of the year from `('Jan', 1)` through `('Dec', 31)` and each value is a list of precipitation amounts for that day, one per city.
3. Iterate over the "day to precipitation" dictionary, making a list of the (month, day number) tuples where the **minimum** precipitation among the cities for that day is 0. This is the "zero-precipitation" list.

[✓] 我（**伟臣 沈**）了解提交不是我自己完成的作业 将永远不会通过此课程或导致我的 Coursera 帐号被关闭。 了解荣誉准则的更多信息

提交测试