ChandonnetModule07Lab01

December 11, 2022

1 Assignment 7

1.0.1 Ray Chandonnet

1.0.2 December 8, 2022

1.1 Question 1

A palindrome is a word, phrase, or sequence that is the same spelled forward as it is backwards. Write a function using a for-loop to determine if a string is a palindrome. Your function should only have one argument.

[10]:
```
# My basic logic is:
#   - I set a palindrome Boolean to True;
#   - I determine the midpoint of the string (truncated as an integer for␣
 ↪strings of odd length
#   - I use one counter in a for loop that iterates through the first half of␣
 ↪the string
#   - I use a secound counter that starts at the end of the string and is␣
 ↪decremented for each loop run
#   - In each loop run I compare the character at position of counter 1 to␣
 ↪character at position counter 2
#     so for example, if string is 7 characters long, I compare characters 1 and␣
 ↪7, 2 and 6, & 3 and 5;
#     if string is 8 characters long, I compare characters 1 and 8, 2 and 7, 3␣
 ↪and 6, & 4 and 5;
#   - As soon as I encounter a case where the compared characters don't match, I␣
 ↪know it's not a palindrome,
#     so I set the palindrome Boolean to False and "break" the for loop
#   - If I get all the way through the for loop without breaking, it is a␣
 ↪palindrome and the palindrome
#     Boolean has remained TRUE
#
# A couple of comments:
#
# 1) The reason I only iterate through half the string is that it saves work -␣
 ↪comparing characters 7 and 1
#    is the same as comparing characters 1 and 7 so I'd be repeating evaluations␣
 ↪already done if I looped
```

```
#    through every character
#
# 2) I personally HATE "breaking" a for loop as a coding construct - I did it in␣
 ↪order to save work (no need to
#    continue comparing characters once we have found a character mismatch) but␣
 ↪I feel that is bad use of a For
#    loop;  A better construct I think would be to use a while loop, as in␣
 ↪Question 2
#
def palindrome_for(eval_str):
    #
    # This function determines whether a string is a palindrome
    # Input:  eval_str = The string to be evaluated
    # Output:  The function returns a Boolean value True if eval_str is a␣
 ↪palindrome, False if not
    #
    if len(eval_str)==0:  # check for null string
        is_palindrome=False   #return False if no string provided; Though I␣
 ↪supposed you could argue null string
                              # IS in fact a palindrome :)
    else:
        is_palindrome=True  # Set palindrome Boolean to true
        mid_point=int(len(eval_str)/2) # find midpoint of string to iterate over␣
 ↪(truncated to deal with odd length)
        back_counter = len(eval_str)-1 # initialize the counter that works␣
 ↪backward
        for counter in range(mid_point) : # Loop through the front half of the␣
 ↪string
            if eval_str[counter] != eval_str[back_counter] :  # Not a palindrome!
                is_palindrome=False
                break
            back_counter -= 1 # Decrement the backwards counter
    return is_palindrome
#
# Here are some tests to prove it works for odd and even length strings, the␣
 ↪null string, and regardless of length
#
print("'level':",palindrome_for("level"))
print("'leveel':",palindrome_for("leveel"))
print("'hannah':",palindrome_for("hannah"))
print("'hanaah':",palindrome_for("hanaah"))
print("If no string provided: ",palindrome_for(""))
print("My favorite - 'amanaplanacanalpanama':
 ↪",palindrome_for("amanaplanacanalpanama"))
print("Misspelled - 'amanaplanaacanalpanama':
 ↪",palindrome_for("amanaplanaacanalpanama"))
```

```
'level': True
'leveel': False
'hannah': True
'hanaah': False
If no string provided:  False
My favorite - 'amanaplanacanalpanama': True
Misspelled - 'amanaplanaacanalpanama': False
```

1.2 Question 2

Write a function using a while-loop to determine if a string is a palindrome. Your function should only have one argument.

[11]:
```
# This code block / function uses the same logic as in Question 1.  The only␣
 ↪difference is in the use of a While
# loop.  Using a While loop feels like better code hygiene. as the loop only␣
 ↪runs while the palindrome Boolean
# is still True.  (The loop terminates once a mismatch has been found).  In␣
 ↪order for this to function, we have to
# manually increment the forward counter in each loop run, and terminate the␣
 ↪loop when EITHER it's not a palindrome OR
# we reached the midpoint of the string and so are done our work
#
def palindrome_while(eval_str):
    #
    # This function determines whether a string is a palindrome
    # Input:  eval_str = The string to be evaluated
    # Output:  The function returns a Boolean value True if eval_str is a␣
 ↪palindrome, False if not
    #
    if len(eval_str)==0: # Check for null string and if so return False
        is_palindrome=False
    else:
        is_palindrome=True  # Initialize palindrome Boolean
        mid_point=int(len(eval_str)/2) # Identify midpoint
        counter = 0 # Initialize forward counter
        opp_counter = len(eval_str)-1 # Initialize backward counter
        while is_palindrome and counter<=mid_point:  # Loop until either hbit␣
 ↪midpoint or fail palindrome test
            if eval_str[counter] != eval_str[opp_counter] : # Not a palindrome
                is_palindrome=False   # Will force the loop to end
            else:
                counter += 1 # Increment forward counter
                opp_counter -= 1  # Decrement backward counter
    return is_palindrome

print("'level':",palindrome_while("level"))
```

```
print("'leveel':",palindrome_while("leveel"))
print("'hannah':",palindrome_while("hannah"))
print("'hanaah':",palindrome_while("hanaah"))
print("If no string provided: ",palindrome_while(""))
print("My favorite - 'amanaplanacanalpanama':
 ↪",palindrome_while("amanaplanacanalpanama"))
print("Misspelled - 'amanaplanaacanalpanama':
 ↪",palindrome_while("amanaplanaacanalpanama"))
```

```
'level': True
'leveel': False
'hannah': True
'hanaah': False
If no string provided:  False
My favorite - 'amanaplanacanalpanama': True
Misspelled - 'amanaplanaacanalpanama': False
```

1.3 Question 3

Two Sum - Write a function named two_sum() Given a vector of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.Use defaultdict and hash maps/tables to complete this problem.

Example 1: Input: `nums = [2,7,11,15]`, target = 9 Output: `[0,1]` Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2: Input: `nums = [3,2,4]`, `target = 6 Output: [1,2]`

Example 3: Input: `nums = [3,3]`, `target = 6 Output: [0,1]`

Constraints: `2 <= nums.length <= 104 -109 <= nums[i] <= 109 -109 <= target <= 109` Only one valid answer exists.

[12]:
```
from collections import defaultdict
#
# This function takes in a vector of values and a target value, and finds the
# first combination of indices such that the values in the vector at those␣
 ↪indices
# adds up to the target value
#
# Inputs: nums_vector = The vector of values to be searched
# target = The target sum being sought
#
# Output: The function returns a vector (list) containing the two indexes whose␣
 ↪values add up to target
# The function will print an error message and return null if any of the␣
 ↪argument constraints are violated
def two_sum(nums_vector,target):
# First check for constraint violations in the arguments submitted
```

```
    argumentsOK=True
    if len(nums_vector) < 2:
        print("ERROR: Number vector must have at least two values")
        argumentsOK = False
    if len(nums_vector)>104 :
        print("ERROR: Number vector cannot exceed 104 values")
        argumentsOK = False
    if min(nums_vector) < -109 :
        print("ERROR: All values in vector must be >= -109")
        argumentsOK = False
    if max(nums_vector) > 109 :
        print("ERROR: All values in vector must be <= 109")
        argumentsOK = False
    if target < -109 :
        print("ERROR: Target sum must be >= -109")
        argumentsOK = False
    if target > 109 :
        print("ERROR: Target sum must be <= 109")
        argumentsOK = False
# If any error was triggered, abort the function
    if not argumentsOK:
        print("Function Aborted Due to bad argument(s) above")
        return
    else:
        def def_value():
            return "Not Present" # Set default value - this will be used to flag␣
 ↪whether a sum complement is founbd
        hash_table=defaultdict(def_value) # Create null hashtable
# Populate hashtable:  Keys are each value in the vector, values are the index␣
 ↪where that value occurs
# in the vector (the counter)
        for counter1 in range(len(nums_vector)):
            hash_table[nums_vector[counter1]] = counter1
# Now we go through the list again; For each element, search the hashtable for␣
 ↪its "complement" that when added to it,
# equates to target.  If found, and it isn't adding an element to itself, we␣
 ↪have successfully identified the pair
        counter2=0
        match_found=False
        while counter2 <= len(nums_vector) and not match_found:
            search_key = target-nums_vector[counter2] # this is the complement␣
 ↪we're looking for
            match_location = hash_table[search_key] # Retrieve the value␣
 ↪associated with that key, which will be
                                                    # equal to the default value␣
 ↪if it isn't found
```

```
            if match_location != def_value() and match_location != counter2:  #␣
↪Found a non-duplicative match!
                match = [counter2,match_location]
                match_found=True
            else:
                counter2 += 1
    return match

# Prove that error checking works
nums_vector=[1]
two_sum(nums_vector,2) #nums_vector too short
nums_vector=list(range(1,106))
two_sum(nums_vector,2) #nums_vector too long
nums_vector=list(range(-110,-10))
two_sum(nums_vector,2) #lowest nums_vector value not in bounds
nums_vector=list(range(100,111))
two_sum(nums_vector,2) #highest nums_vector value not in bounds
nums_vector=list(range(1,101))
two_sum(nums_vector,-110) #target below lower bound
two_sum(nums_vector,110) #target above higher bound
nums_vector=list(range(-120,120))
target=200
two_sum(nums_vector,target) #multiple constraint violations

# Now run function on test cases
nums_vector = [2,7,11,15]
target = 9
print(two_sum(nums_vector,target))
nums_vector = [3,2,4]
target = 6
print(two_sum(nums_vector,target))
nums_vector = [3,3]
target = 6
print(two_sum(nums_vector,target))
nums_vector = [1,3,6,10,15,21,28,36,45,55,66,78,91,105]
target=97
print(two_sum(nums_vector,target))
target=73
print(two_sum(nums_vector,target))
```

```
ERROR: Number vector must have at least two values
Function Aborted Due to bad argument(s) above
ERROR: Number vector cannot exceed 104 values
Function Aborted Due to bad argument(s) above
ERROR: All values in vector must be >= -109
Function Aborted Due to bad argument(s) above
ERROR: All values in vector must be <= 109
Function Aborted Due to bad argument(s) above
```

```
ERROR: Target sum must be >= -109
Function Aborted Due to bad argument(s) above
ERROR: Target sum must be <= 109
Function Aborted Due to bad argument(s) above
ERROR: Number vector cannot exceed 104 values
ERROR: All values in vector must be >= -109
ERROR: All values in vector must be <= 109
ERROR: Target sum must be <= 109
Function Aborted Due to bad argument(s) above
[0, 1]
[1, 2]
[0, 1]
[2, 12]
[6, 8]
```

1.4 Question 4

How is a negative index used in Python? Show an example

[14]:
```
# Wow this whole lab (well at least Questions 1 and 2) was a setup leading to␣
 ↪this!   Negative indexing lets you start
# from the end rather than the beginning of a list, array etc.  You could use␣
 ↪that for example to extract the right-
# most N characters from a string - for example, if a location is listed as␣
 ↪"City, ST" like "Boston, MA" you could
# isolate the state as the last two characters in the string using negative␣
 ↪indexing.
#
# Why was this a setup? Because we don't need to write a complicated for loop or␣
 ↪while loop to identify whether
# a string is a palindrome!  We can just use this negative indexing to return␣
 ↪the string in reverse order and
# compare that to the original.
#
# Some syntax:  slicing the string is string[start:stop:step];  If you leave␣
 ↪start and stop blank and use -1 for
# step it literally gives you the string in reverse:
test_str="Raymond"
backwards=test_str[::-1]
print(test_str,"backwards is",backwards)
#
# Knowing this, I can very easily test for palindromes!
def palindrome_easy(eval_str):
    if len(eval_str)==0:
        return False
    else:
        backwards = eval_str[::-1]
```

```
        is_palindrome= backwards == eval_str
    return is_palindrome

print("'level':",palindrome_easy("level"))
print("'leveel':",palindrome_easy("leveel"))
print("'hannah':",palindrome_easy("hannah"))
print("'hanaah':",palindrome_easy("hanaah"))
print("If no string provided: ",palindrome_easy(""))
print("My favorite - 'amanaplanacanalpanama':
 ↪",palindrome_easy("amanaplanacanalpanama"))
print("Misspelled - 'amanaplanaacanalpanama':
 ↪",palindrome_easy("amanaplanaacanalpanama"))
```

```
Raymond backwards is dnomyaR
'level': True
'leveel': False
'hannah': True
'hanaah': False
If no string provided:  False
My favorite - 'amanaplanacanalpanama': True
Misspelled - 'amanaplanaacanalpanama': False
```

1.5 Question 5

Check if two given strings are isomorphic to each other. Two strings str1 and str2 are called isomorphic if there is a one-to-one mapping possible for every character of str1 to every character of str2. And all occurrences of every character in 'str1' map to the same character in 'str2'.

```
Input:  str1 = "aab", str2 = "xxy"
```

```
Output: True
```

```
'a' is mapped to 'x' and 'b' is mapped to 'y'.
```

```
Input:  str1 = "aab", str2 = "xyz"
```

```
Output: False
```

```
One occurrence of 'a' in str1 has 'x' in str2 and other occurrence of 'a' has 'y'.
```

A Simple Solution is to consider every character of 'str1' and check if all occurrences of it map to the same character in 'str2'. The time complexity of this solution is O(n*n).

An Efficient Solution can solve this problem in O(n) time. The idea is to create an array to store mappings of processed characters.

[18]:
```
# This lends itself well to using a hashtable, with the key being character in␣
 ↪str1 and the value being the character
# it maps to in str2.  This is an easy construct:
# - Loop through the chracters in str1.
# - For each character, check to see if it is already in the hashtable
```

```
# - If it is, and the matching character value (from str2) doesn't match what we␣
 ↪have, then it's a fail!
# - Otherwise, add this character and its matching character from str2 into the␣
 ↪hashtable
#   (If the character pair is already in the hashtable, it won't be duplicated -␣
 ↪maybe it's lazy programming
#    to take advantage of this, but it works just fine) - I could test for this␣
 ↪first if necessary but result is same
#
# So the idea is we build a hashtable of unique (char1,char2) combos but as soon␣
 ↪as we find an entry with same char1
# but different char2, we know we have failed the isomorphic test
# From an algorithm performance standpoint, this will take at MOST O(n) time,␣
 ↪and then only if we have to evaluate
# every character in str1.
def isomorphic(str1, str2):
    def def_value():
        return "Not Present"
    if len(str1) != len(str2):  #if strings aren't same length it's an automatic␣
 ↪fail
        return False
    else:
        char_maps = defaultdict(def_value)  # set null hashmap
        counter=0
        is_isomorphic = True
        while counter < len(str1) and is_isomorphic: # loop through all␣
 ↪characters in str1 until failure or done
            if char_maps[str1[counter]] != def_value() and␣
 ↪char_maps[str1[counter]] != str2[counter]: # Check to see
                #if in the list already, with a different matching char2
                is_isomorphic = False
            else:
                char_maps[str1[counter]]=str2[counter] # if not, add it to␣
 ↪hashmap (knowing dupes are discarded)
                counter +=1
    return is_isomorphic
# Let's test it shall we?
str1="aab"
str2="xxy"
print(isomorphic(str1, str2))
str1="aab"
str2="xyz"
print(isomorphic(str1, str2))
str1="aabbccddabcddcba"
str2="wwxxyyzzwxyzzyxw"
print(isomorphic(str1, str2))
```

```
str1="aabbccddabcddcba"
str2="wwxxyyzzwxyzayxw"
print(isomorphic(str1, str2))
```

```
True
False
True
False
```

[ ]:

[ ]: