Outline

Quiz announcement

Secret Message

- feedback

Text Manipulations

- string functions

Group work

Prof. Angela Chang

Lecture 8: Text & ~~Regex~~

Fall 2017. Oct 2, 2017

CODE, CULTURE, AND PRACTICE

Text challenges

Counting Spaces

Write count_spaces(), a function that accepts a string as an argument and returns the number of spaces in the string. Use iteration to determine this.

If you can think of more than one way to accomplish this, write count_spaces2() and go on to write count_spaces_3() and beyond if you like, showing the alternatives. To accomplish the basic, initial count_spaces() function, no special knowledge of Python is needed beyond what has already been covered.

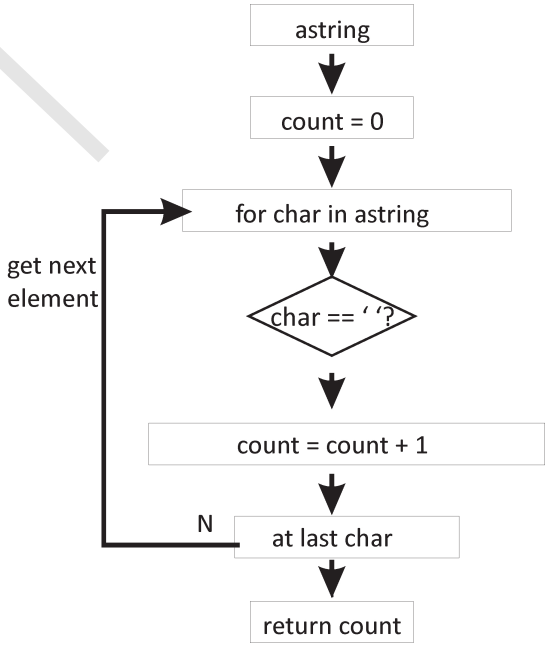
Counting Non-spaces

Write a function count_nonspaces() that returns the number of characters in a string that are not spaces. Try figuring this out using iteration, with reference to the problem just solved. Once you have solved the problem this way, see if you can you determine how do this in a single line (not counting the line beginning with def) by having count_nonspaces() call count_spaces() from before.

*ps. use stackoverflow or google to find syntax and sample code
e.g != inequality in python

Read and break it down:

1. function that accepts a string
2. returns a number
3. iterate on each character
4. test if each character is a space/or nonspace
5. counts the spaces/nonspaces



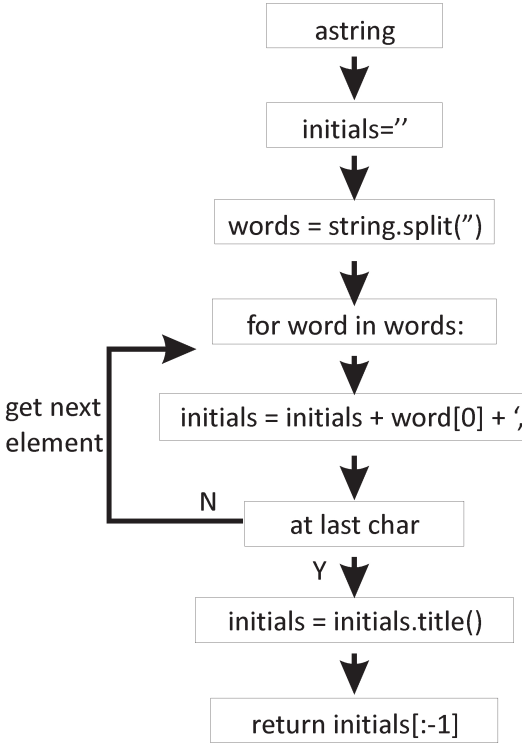
Determining Initials

Write a function initials() that takes a string containing any name (a personal or business name, for instance) and returns the initials. For instance, the values returned by the following function calls will be:

Initials(“International Business Machines”) → IBM

Initials(“M. Lee Pelton”) → MLP

You should be able to tell what type the return value (that is, your result: the initials) should be. The function should work properly on names with any length string. Do not worry about special handling for cases where punctuation makes up its own “word,” or where a word begins with a punctuation mark, or where you know that a compound word. Just return the first character of each part of the string separated by whitespace.



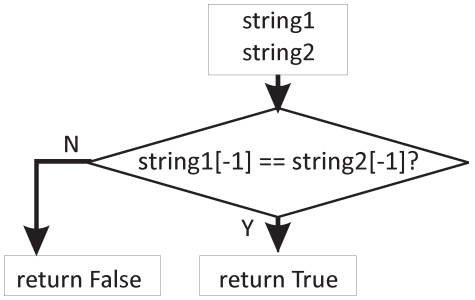
1. function that accepts a string
2. returns a string
3. iterate on each word
4. finds each first initial
5. joins together all the initials
6. returns initials

Same Last Character

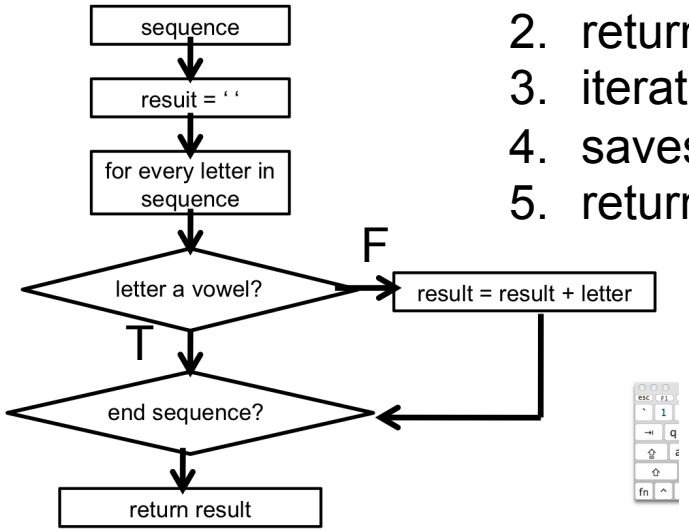
Write same_last(), a function that accepts two strings as arguments and returns True if they have the same last letter, False otherwise. For this exercise, you can assume that both of the strings are at least one letter long—it does not matter what happens (the program could crash, etc.) if one or both of the strings is the null string.

After you write a function that works, see if you have more than one line in the function body—that is, if you have any code besides the def line and one line after it. If your function is more than two lines long, refactor it so that it is only two lines long.

1. function that accepts 2 strings
2. returns True or False
3. finds each last character
4. tests if they are the same

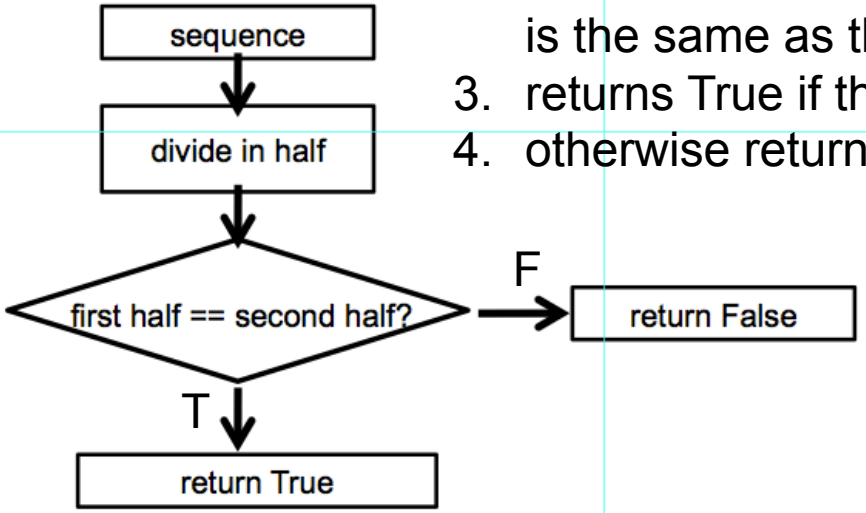


Remove Vowels
Write **devowel()**, a function that accepts a string as an argument and returns the string without the vowels. For instance, given ‘hello world’ it will return ‘hll wrld’. Just consider the five standard, full vowels for this exercise, neglecting poor y and w.



1. function that accepts a string
2. returns a string
3. iterate on each letter
4. saves only consonants
5. returns string without vowels

Tautonyms
Write a function **tautonym()** that accepts a string and returns **True** if the string consists of some sequence of characters (call it A) followed by the same sequence of characters, A, The function should return **False** otherwise. For instance, given ”hello world” it should return **False** but given ’worldworld’ it should returns **True**. Of course, for ‘worldworldbaby’ the answer **False**.



1. function that accepts a string
2. tests if the first half of the string is the same as the second half
3. returns True if they’re equal
4. otherwise returns False

Palindromes

‘civic’

‘kayak’

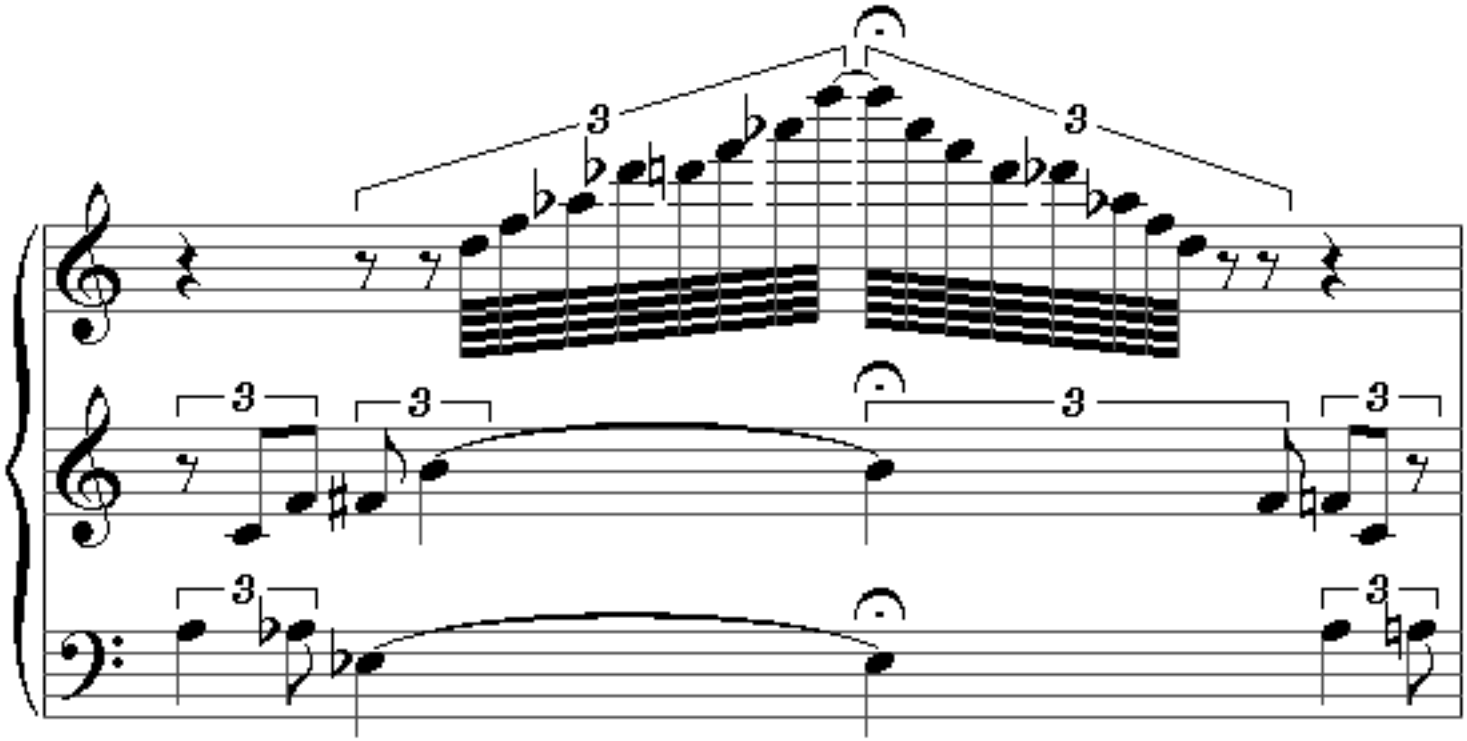
‘racecar’

palindrome | ˈpalinˌdrōm | noun
a word, phrase, or sequence that reads the same backward as forward, e.g., *madam* or *nurses run* .

Rise to vote, sir.

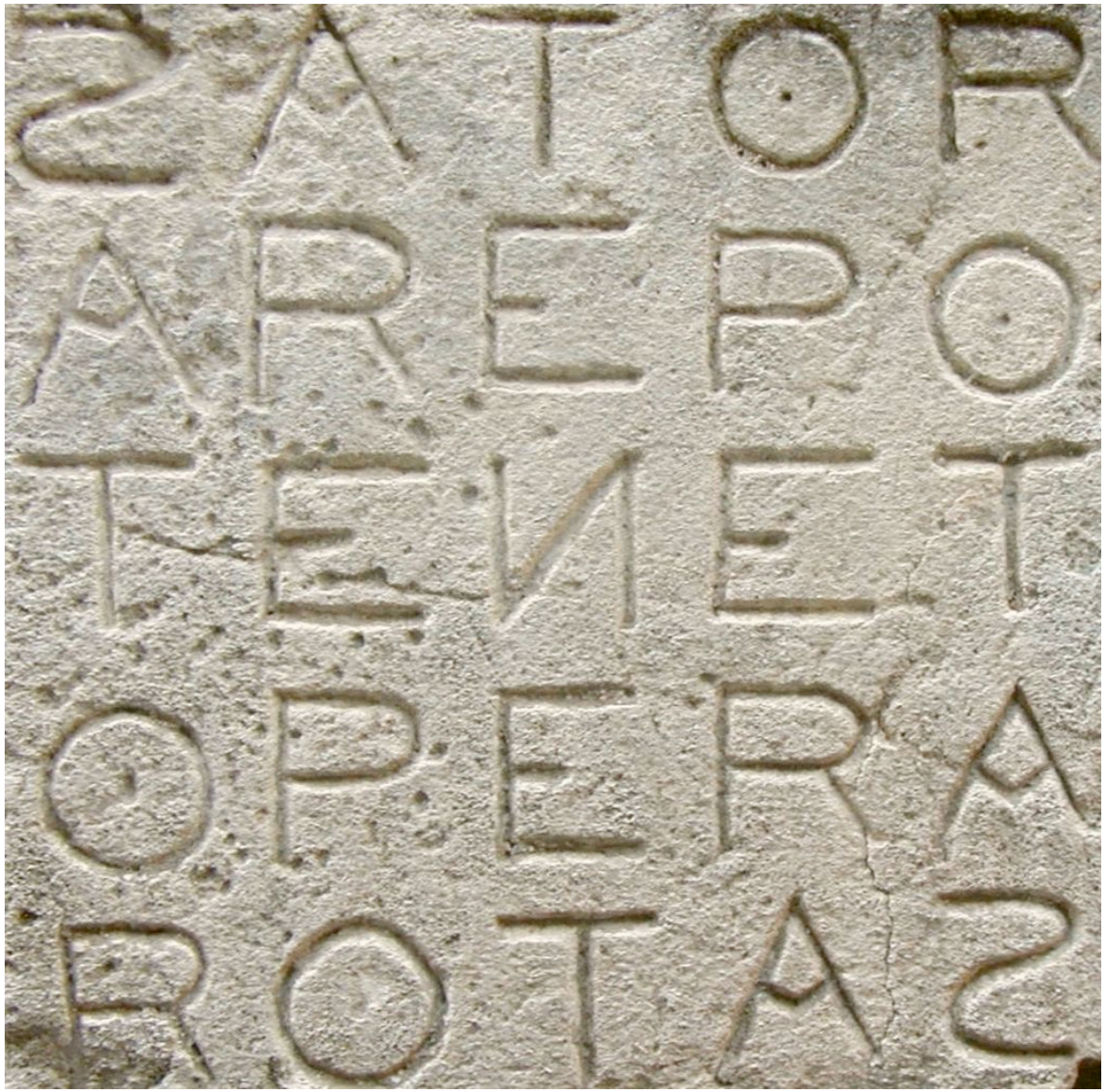
taco cat

Not so, Boston.



Center part of palindrom in *Lulu* by Magnus Manske on Wikimedia

Palindromic Graffiti



79 AD

Sator Square by M. Disdero on Wikimedia

Palindromic Puzzle

By Edgar Allen Poe 1827

First, find out a word that doth silence proclaim,
And that backwards and forwards is always the same;
Then next you must find out a feminine name
That backwards and forwards is always the same;
An act, or a writing on parchment whose name
Both backwards and forwards is always the same;
A fruit that is rare, whose botanical name
Read backwards and forwards is always the same;
A note used in music, which time doth proclaim,
And backwards and forwards is always the same;
Their initials connected, a title will frame,
That is justly the due of the fair married dame,
Which backwards and forwards is always the same.

Read more at Mathmagical site
Image by Magic Komlez from Deviant Art

also Nabokov was a big palindrome fan

Palindrome function

Write a function to detect whether a given sequence is a palindrome.

Tip # 1: Break it down:

1. Start with a test case
2. Convert to a **list**.
3. Reverse the list, make a copy.
4. Check if the two lists are equal.

```
#first step ignore casing
```

```
['c','i','v','i','c']  #test a list of characters
```

```
#build a list using the list() command
list('rotor')
```

```
#list function called reverse() can be helpful, so we use it
#strings don't have a list function
test = ['a','b','c']
test.reverse()
test
```

Palindromes.ipynb

Palindrome function

Write a function to detect whether a given sequence is a palindrome.

Tip # 1: Break it down:

1. Start with a test case
2. Convert to a list.
3. Reverse the list, make a copy.
4. Check if the two lists are equal.

```
#first step ignore casing
```

```
['c','i','v','i','c']  #test a list of characters
```

```
#build a list using the list() command
list('rotor')
```

```
#list function called reverse() can be helpful, so we use it
#strings don't have a list function
test = ['a','b','c']
test.reverse()
test
```

Tip #2 – use print statements

```
#note that reverse sorts in place-- alters the original list
```

```
word = 'hierarchy'
backlist = list(word)
backlist
backlist.reverse()
backlist #mote that the orginal is overwritten
```

```
print word
print backlist
```

Palindromes.ipynb

Palindrome function

Write a function to detect whether a given sequence is a palindrome.

Tip # 1: Break it down:

1. Start with a test case
2. Convert to a list.
3. Reverse the list, make a copy.
4. Check if the two lists are equal.

```
def pal(word): #just getting pieces in place
    print word
    backlist = list(word)
    backlist.reverse()
    print word, backlist
    for i in range(len(word)):
            print i, word[i], backlist[i]
```

```
#first step ignore casing
```

```
['c','i','v','i','c']  #test a list of characters
```

```
#build a list using the list() command
list('rotor')
```

```
#list function called reverse() can be helpful, so we use it
#strings don't have a list function
test = ['a','b','c']
test.reverse()
test
```

Tip #2 – use print statements

```
#note that reverse sorts in place-- alters the original list
```

```
word = 'hierarchy'
backlist = list(word)
backlist
backlist.reverse()
backlist #mote that the orginal is overwritten
```

```
print word
print backlist
```

Palindromes.ipynb

Palindrome function

Write a function to detect whether a given sequence is a palindrome.

Tip # 1: Break it down:

1. Start with a test case
2. Convert to a list.
3. Reverse the list, make a copy.
4. Check if the two lists are equal.

```
def pal(word): #just getting pieces in place
    print word
    backlist = list(word)
    backlist.reverse()
    print word, backlist
    for i in range(len(word)):
            print i, word[i], backlist[i]
```

```
def pal(word): #the final form
    backlist = list(word)
    backlist.reverse()
    for i in range(len(word)):
        if word[i]!=backlist[i]:
            return False
    return True
```

iterative solution 1

```
#first step ignore casing
```

```
['c','i','v','i','c']  #test a list of characters
```

```
#build a list using the list() command
list('rotor')
```

```
#list function called reverse() can be helpful, so we use it
#strings don't have a list function
test = ['a','b','c']
test.reverse()
test
```

Tip #2 – use print statements

```
#note that reverse sorts in place-- alters the original list
```

```
word = 'hierarchy'
backlist = list(word)
backlist
backlist.reverse()
backlist #mote that the orginal is overwritten
```

```
print word
print backlist
```

Palindromes.ipynb

Palindrome function

Write a function to detect whether a given sequence is a palindrome.

Approach:

1. Start with a test case
2. Use the list reverse function.
3. Check if the two lists are equal.

```
def pal(word): #the final form
    backlist = list(word)
    backlist.reverse()
    for i in range(len(word)):
        if word[i]!=backlist[i]:
            return False
    return True
```

Iterative solution by first creating a list.

```
def pal2(word): #refactored
    backlist = list(word)
    backlist.reverse()
    return list(word) == backlist
```

Use the list **reverse** function

Next, how to account for upper/lowercase?

…And then…..

Palindromes.ipynb

Palindrome function

```
def pal3(word):
    lowerword = word.lower()
    backlist = list(lowerword)
    backlist.reverse()
    return list(lowerword) == backlist
```

Solution #2: using list reverse

But there's another way….

…And then…..

Palindromes.ipynb

Palindrome function

```
def pal3(word):
    lowerword = word.lower()
    backlist = list(lowerword)
    backlist.reverse()
    return list(lowerword) == backlist
```

Solution #2: using list reverse

But there's another way….

```
def pal4(word):
    lowerword = word.lower()
    backlist = lowerword[::-1]  #SLICES!
    print backlist, lowerword
    return lowerword == backlist  #no need to convert to list
```

Solution 3: **String** slices*

…And then…..

*we'll refactor this later

Palindromes.ipynb

Palindrome function

Solution 4: Iteration #2

```
right = -1
pal = True
for letter in string:
    pal = (pal and (letter == string[right]))
    print letter, string[right], right
    right = right - 1
```

As the program iterates through the string, compare each forward character with each reverse character.

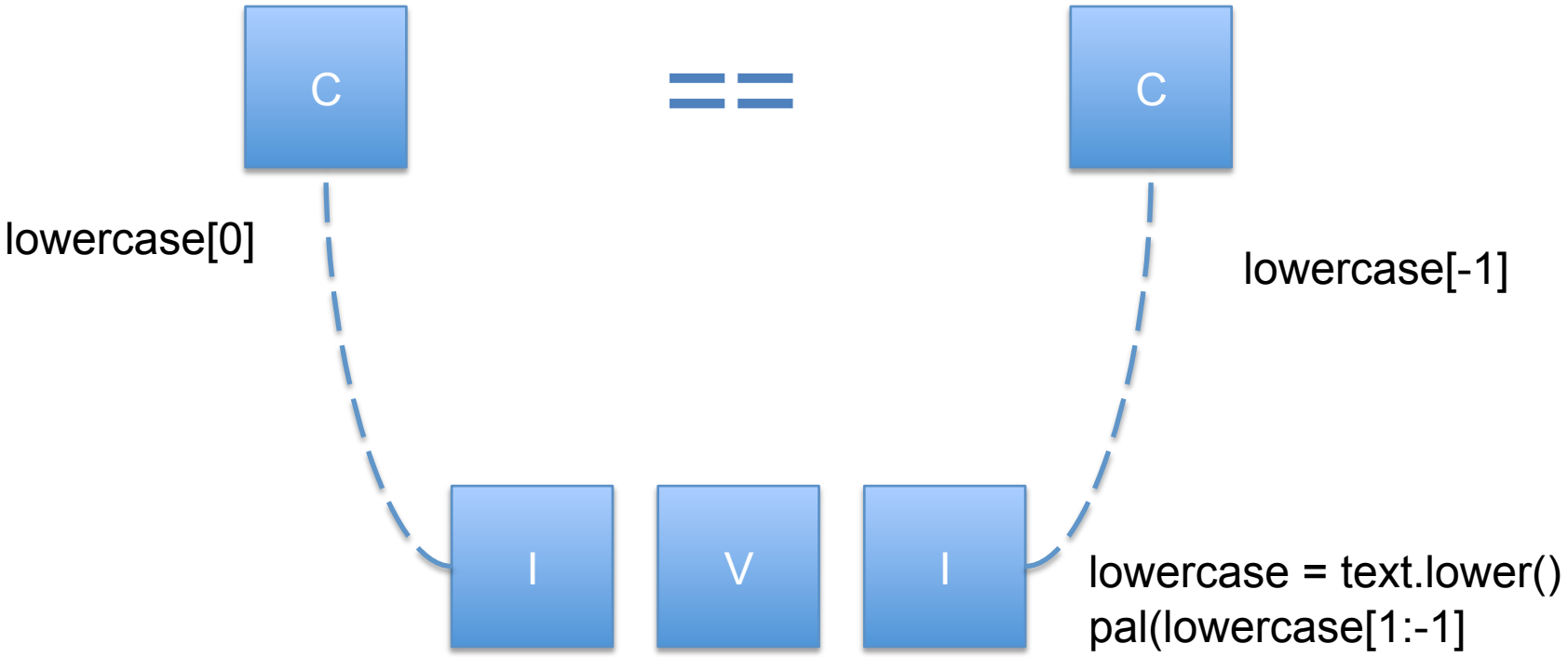
…And then…..

Palindromes.ipynb

Palindrome function

Recursion

A palindrome is a string such that if you take off the first and last character, they are the same, and what's left is a palindrome.



Palindromes.ipynb

Palindrome function

Recursion

```
def palr(text):
    print text
    lowercase = text.lower()
    if len(lowercase) <=1:
        return True  #palindrome detected
    else:
        if lowercase[0] == lowercase[-1]:  #check ends
            lowercase = text.lower()
            return palr(lowercase[1:-1])  #keep recursion
        else:
            return False  #not a palindrome
```

Palindromes.ipynb

Solutions so far:

```
def pal(word): #the final form
    backlist = list(word)
    backlist.reverse()
    for i in range(len(word)):
        if word[i]!=backlist[i]:
            return False
    return True
```

Iterative list solution

```
def pal3(word):
    lowerword = word.lower()
    backlist = list(lowerword)
    backlist.reverse()
    return list(lowerword) == backlist
```

List reverse

```
def pal4(word):
    lowerword = word.lower()
    backlist = lowerword[::-1]
    print backlist, lowerword
    return lowerword == backlist
```

String slices*

```
def pali(string):
    right = -1
    pal = True
    for letter in string:
        pal = (pal and (letter == string[right]))
        print letter, string[right], right
        right = right - 1
    return pal
```

Iterative string solution

```
def palr(text):
    print text
    lowercase = text.lower()
    if len(lowercase) <=1:
        return True  #palindrome detected
    else:
        if lowercase[0] == lowercase[-1]:  #check ends
            lowercase = text.lower()
            return palr(lowercase[1:-1])  #keep recursion
        else:
            return False  #not a palindrome
```

Recursive string

Solutions so far:

```
def pal(word): #the final form
    backlist = list(word)
    backlist.reverse()
    for i in range(len(word)):
        if word[i]!=backlist[i]:
            return False
    return True
```

Iterative list solution

```
def pal3(word):
    lowerword = word.lower()
    backlist = list(lowerword)
    backlist.reverse()
    return list(lowerword) == backlist
```

List reverse

```
def pal4(word):
    lowerword = word.lower()
    backlist = lowerword[::-1]
    print backlist, lowerword
    return lowerword == backlist
```

String slices*

```
def pal(text):
    print text
    return text.lower() == text[::-1].lower()
```

String slices refactored

```
def pali(string):
    right = -1
    pal = True
    for letter in string:
        pal = (pal and (letter == string[right]))
        print letter, string[right], right
        right = right - 1
    return pal
```

Iterative string solution

```
def palr(text):
    print text
    lowercase = text.lower()
    if len(lowercase) <=1:
        return True  #palindrome detected
    else:
        if lowercase[0] == lowercase[-1]:  #check ends
            lowercase = text.lower()
            return palr(lowercase[1:-1])  #keep recursion
        else:
            return False  #not a palindrome
```

Recursive string

Many solutions so far:

```
def pal(word): #the final form
    backlist = list(word)
    backlist.reverse()
    for i in range(len(word)):
        if word[i]!=backlist[i]:
            return False
    return True
```

Iterative list solution

```
def pal3(word):
    lowerword = word.lower()
    backlist = list(lowerwor
    backlist.reverse()
    return list(lowerword) =
```

List reverse

A palindrome is a string that is the same as it's reverse. I

```
def pal4(word):
    lowerword = word.lower()
    backlist = lowerword[::-1]
    print backlist, lowerword
    return lowerword == backlist
```

String slices*

```
def pal(text):
    print text
    return text.lower() == text[::-1].lower()
```

String slices refactored

```
def pali(string):
    right = -1
    pal = True
    for letter in string:
        pal = (pal and (letter == string[right]))
        print letter, string[right], right
        right = right - 1
    return pal
```

Iterative string solution

```
def palr(text):
    print text
    lowercase = text.lower()
    if len(lowercase) <=1:
        return True  #palindrome
    else:
        if lowercase[0] == lowerca
            lowercase = text.lower
            return palr(lowercase[
        else:
            return False  #not a p
```

Recursive string

A palindrome is a string whose first character is equal to its last character, and so on.

Quiz review

Review the quiz review sheet and also take a look at the different ways for solving problems.

Look through the homework and class participation comments and see if you have any questions.

Creatively come up with your own functions and try to solve them.

Examine the different ways functions are implemented, and understand why they work.