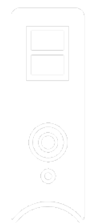
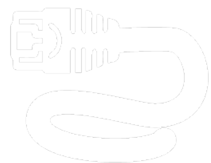
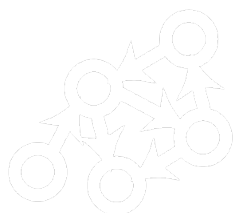
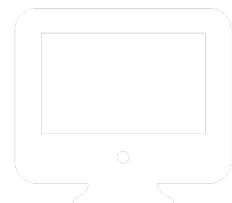
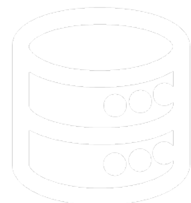
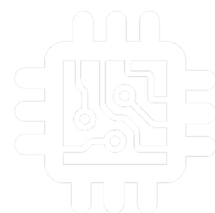
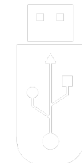
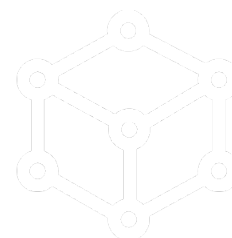
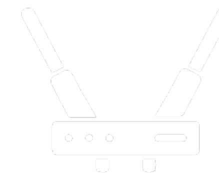
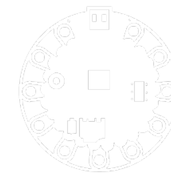
CS 134 Lecture:
Sequences and Loops

Announcements & Logistics

- **Homework 3** will be posted to GLOW, due next Monday @ 10 pm
- **Lab 1** graded feedback will be released today
  - Instructions on how to view feedback on course webpage
    - It may seem like an odd procedure, but we're using real-world software development practices
- **Lab 2** due today 10pm / tomorrow 10pm
- No class on Friday: **Winter Carnival**
- Lab 3 (with a **prelab**) will be released on Friday

Do You Have Any Questions?

Last Time

- Looked at more complex decisions in Python
    - Used Boolean expressions with **and**, **or**, **not**
- Chose between many different options in our code
    - `if elif else` chained conditionals

Today's Plan

- Introduce *iteration* using **for loops** to iterate over **sequences**
- Introduce a new data type which is also a sequence:
  - the 'List'
- Revisit an old type in the context of sequences:
  - the 'string'
- We will discuss sequences more on Monday to fill in any remaining gaps for Lab 3

Sequences in Python: Strings

- **Sequences** in Python represent **ordered collections of elements**: e.g., strings, lists, ranges, etc.
- **Strings** (type `str`) are ordered sequences of individual characters
    - Example: `word = "Hello"`
    - `'H'` is the first character of word, `'e'` is the second character, and so on
    - Each sequence element has a position, known as its index
    - In CS, we often **zero-index**, so we say that `'H'` is at **index** 0, `'e'` is at **index** 1, and so on
- We can access each character of a string using these **indices**

How Do Indices Work?

- Can access elements of a sequence (such as a list) using its **index**
- Indices in Python are both positive and negative
- Everything outside of these values will cause an `IndexError`.

0	1	2	3	4	5	6	7
"W	i	l	l	i	a	m	s"
-8	-7	-6	-5	-4	-3	-2	-1

Note: Most other languages do not support negative indexing!

Accessing Elements of Sequences

0	1	2	3	4	5	6	7
'W	i	l	l	i	a	m	s'
-8	-7	-6	-5	-4	-3	-2	-1

```
>>> word = "Williams"
>>> word[0] # character at 0th index?
'W'
>>> word[3] # character at 3rd index?
'l'
>>> word[7] # character at 7th index?
's'
>>> word[8] # will this work?

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sequence Length

- The **len(seq)** function returns the length of the sequence **seq**
- Even though we zero-index, we still include the total number of elements in the length

0	1	2	3	4	5	6	7
'W	i	l	l	i	a	m	s'
-8	-7	-6	-5	-4	-3	-2	-1

```
>>> word = "Williams"
>>> len(word) # total number of characters
8
```

```
>>> word[len(word)] # will this work?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

```
>>> word[len(word)-1] # what about this?
's'
```

Iteration Motivation: Counting Vowels

- **Problem:** Write a function `count_vowels(word)` that takes a **string** `word` as input and returns the number of vowels in the string (an **int**)
- We'll create a function `is_vowel()` to help us:

```
def count_vowels(word):
    '''Returns number of vowels in the word'''
    pass

>>> countVowels("Williamstown")
4
>>> countVowels("Ephelia")
4
```

is_vowel(char)

```
def is_vowel(ch) :
    return ch == 'a' or ch == 'e' or ch == 'i' or ch == 'o' or ch == 'u' \
        or ch == 'A' or ch == 'E' or ch == 'I' or ch == 'O' or ch == 'U'
```

First Attempt with Conditionals

- **Note**: `val += 1` is shorthand for

  `val = val + 1`

- Any downsides to this approach?
- What if I change `word` to `"Williamstown"`?

```
word = "Williams"
counter = 0
if isVowel(word[0]):
    counter += 1
if isVowel(word[1]):
    counter += 1
if isVowel(word[2]):
    counter += 1
if isVowel(word[3]):
    counter += 1
if isVowel(word[4]):
    counter += 1
if isVowel(word[5]):
    counter += 1
if isVowel(word[6]):
    counter += 1
if isVowel(word[7]):
    counter += 1
print(counter)
3
```

First Attempt with Conditionals

- Using conditionals as shown is repetitive and does not generalize to arbitrarily long words
  - shorter word would "index out of bounds"
  - longer word would stop too soon
- We need something else that allows us to "loop" over the characters in an arbitrary input string
  - "For each character word, add 1 if that character is a vowel"

```
word = "Williams"
counter = 0
if isVowel(word[0]):
    counter += 1
if isVowel(word[1]):
    counter += 1
if isVowel(word[2]):
    counter += 1
if isVowel(word[3]):
    counter += 1
if isVowel(word[4]):
    counter += 1
if isVowel(word[5]):
    counter += 1
if isVowel(word[6]):
    counter += 1
if isVowel(word[7]):
    counter += 1
print(counter)
3
```

For Loops

Iterating with `for` Loops

- One of the most common ways to traverse or manipulate a sequence is to perform some action **for each element** in the sequence
- This is called **looping** or **iterating** over the elements of a sequence
- Syntax of a for loop:

var is called the loop variable

```
for var in seq:
    # body of loop
    # body of loop
```

seq is any type of sequence (for example, a string or a list)

Iterating with `for` Loops

- As the loop executes, the loop variable (`char` in this example) takes on the value of successive sequence elements, one by one

```
>>> # small example of for loop
>>> word = "Williams"

>>> for char in word:
...     print(char)

W
i
l
l
i
a
m
s
```

Note. Python for loops are meant *specifically* for iterating over *sequences* and are also called a "for each" loop.

Why might we call it that?

Counting Vowels

- Let us use a for loop to implement `count_vowels()` function
- What do we need to keep track of as we iterate over `word`?

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    pass
```

Counting Vowels

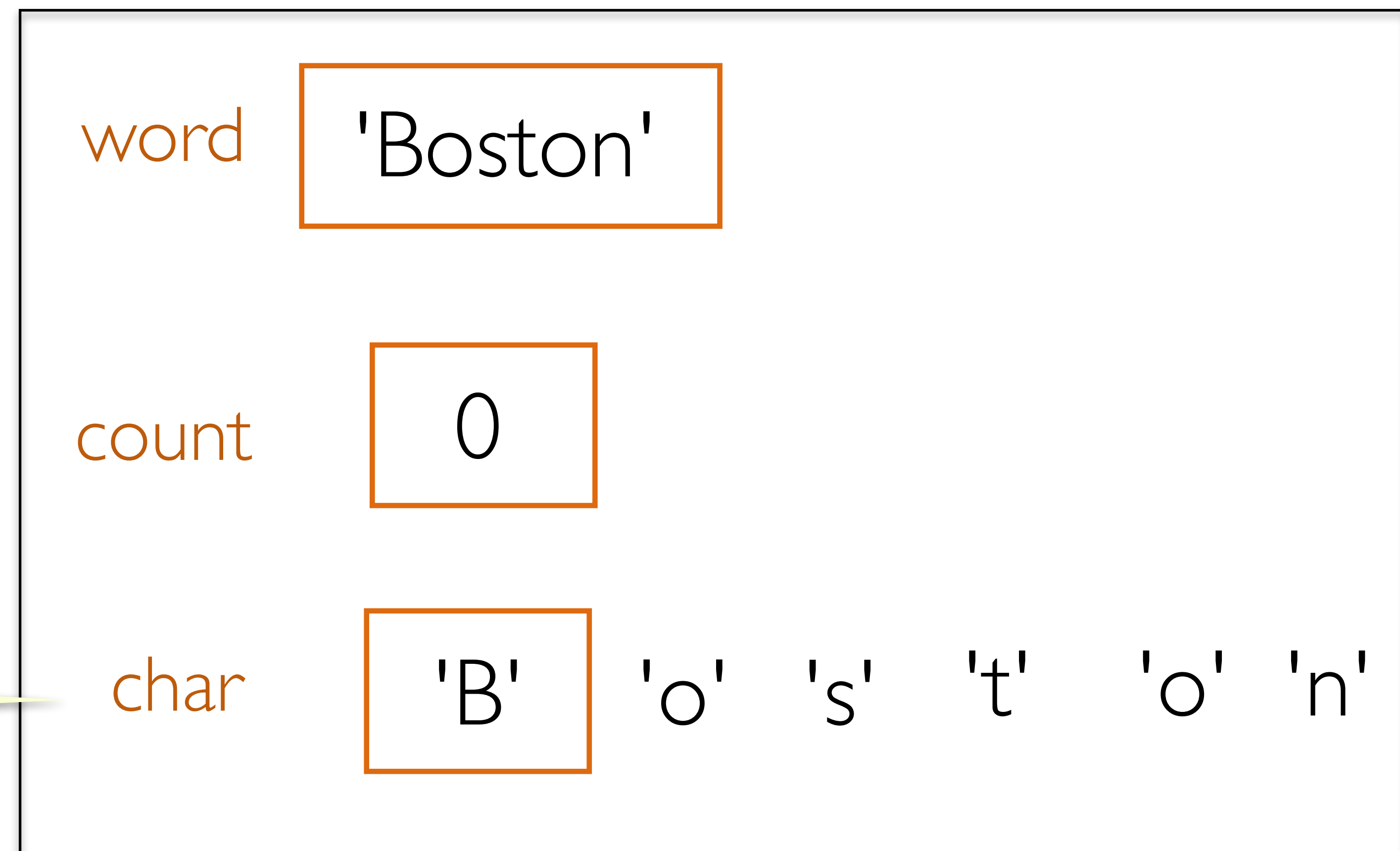
- Notice how `count` "accumulates" values in the loop
- We call `count` an **accumulation variable**

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0 # initialize counter

    # iterate over word one character at a time
    for char in word:
        if is_vowel(char):
            count += 1  # increment counter
    return count
```

Counting Vowels: Tracing the Loop

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0
    for char in word:
        if is_vowel(char):
            count += 1
    return count
```

count_vowels('Boston')

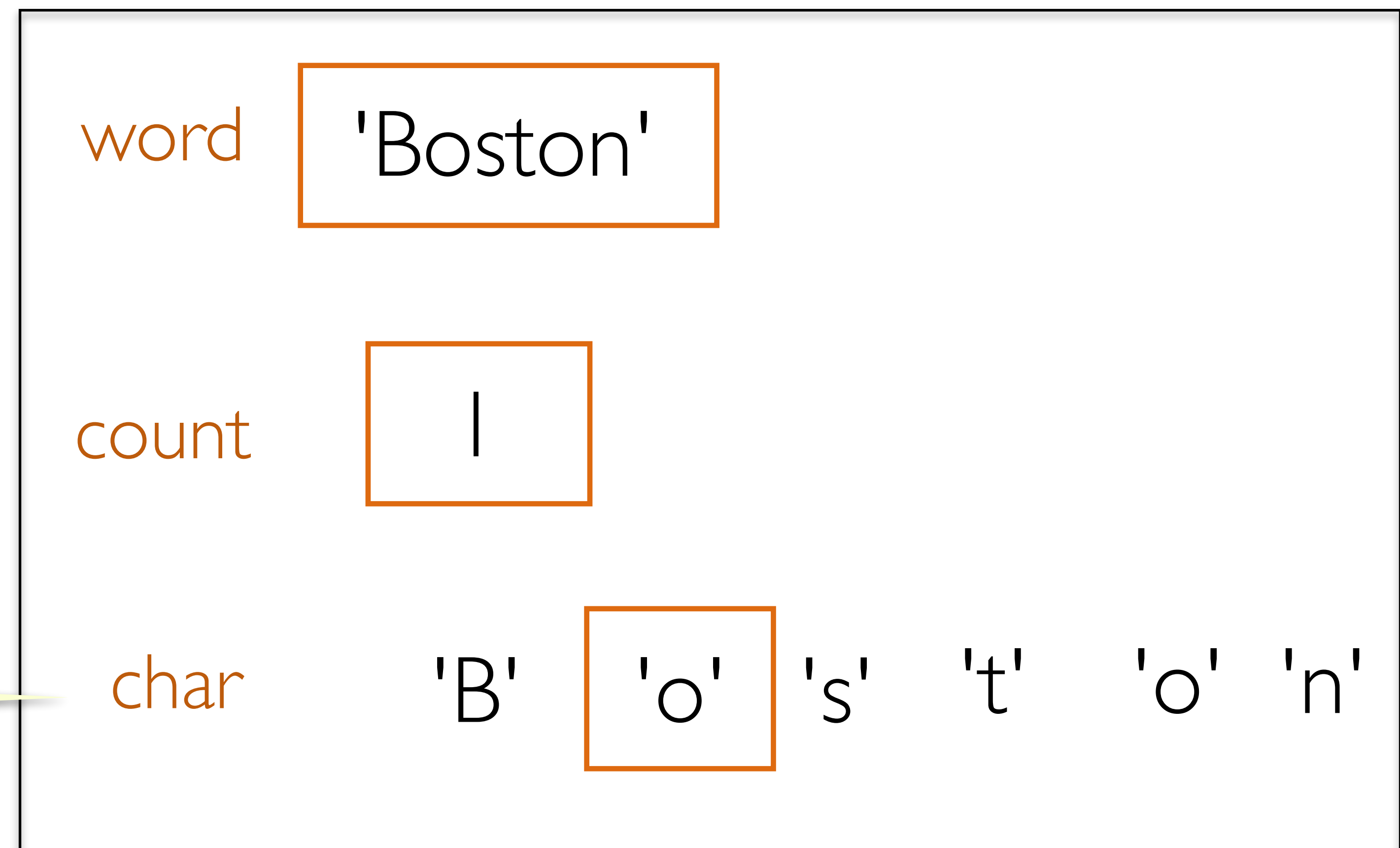


Loop variable

Counting Vowels: Tracing the Loop

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0
    for char in word:
        if is_vowel(char):
            count += 1
    return count
```

countVowels('Boston')

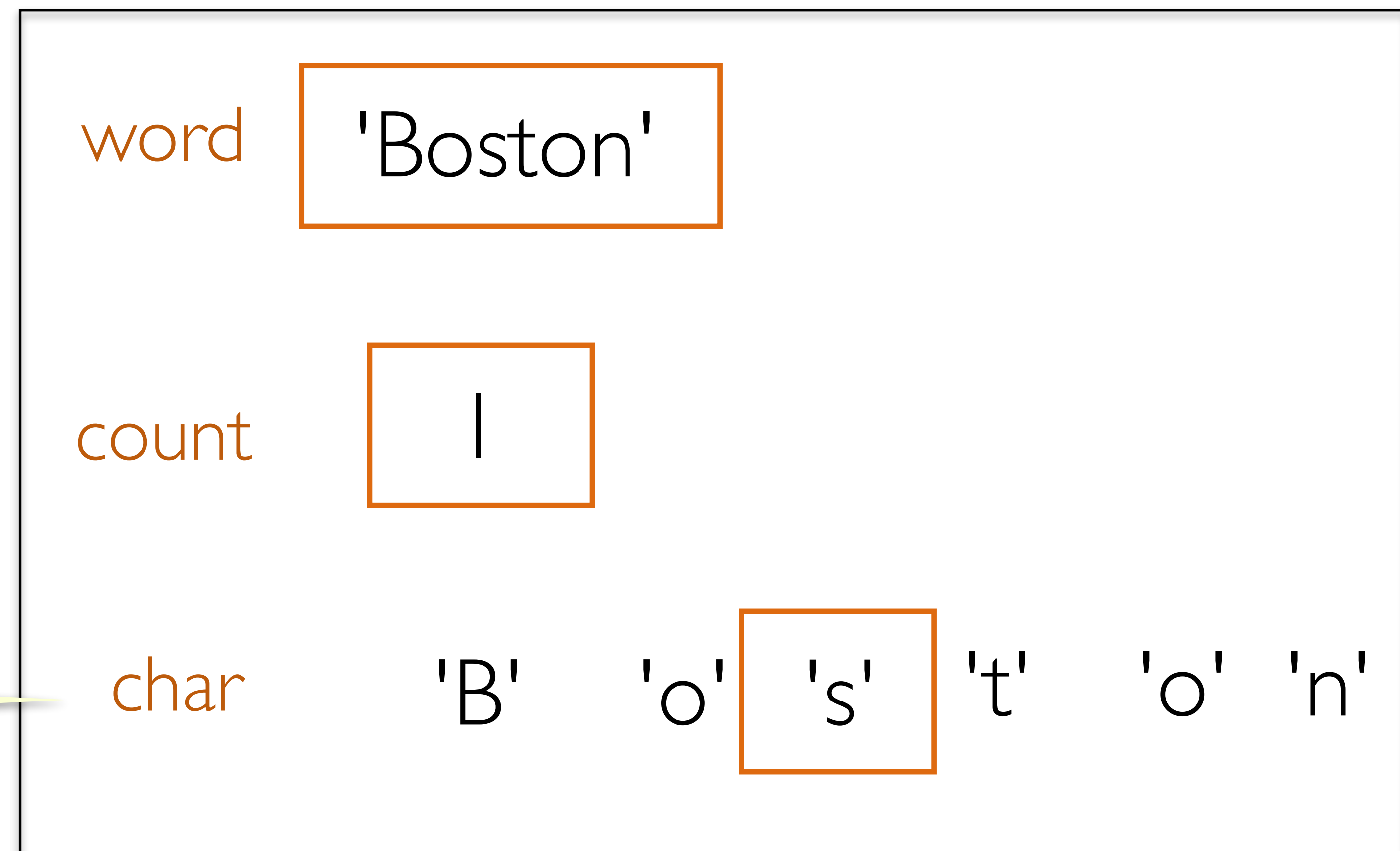


Loop variable

Counting Vowels: Tracing the Loop

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0
    for char in word:
        if is_vowel(char):
            count += 1
    return count
```

countVowels('Boston')

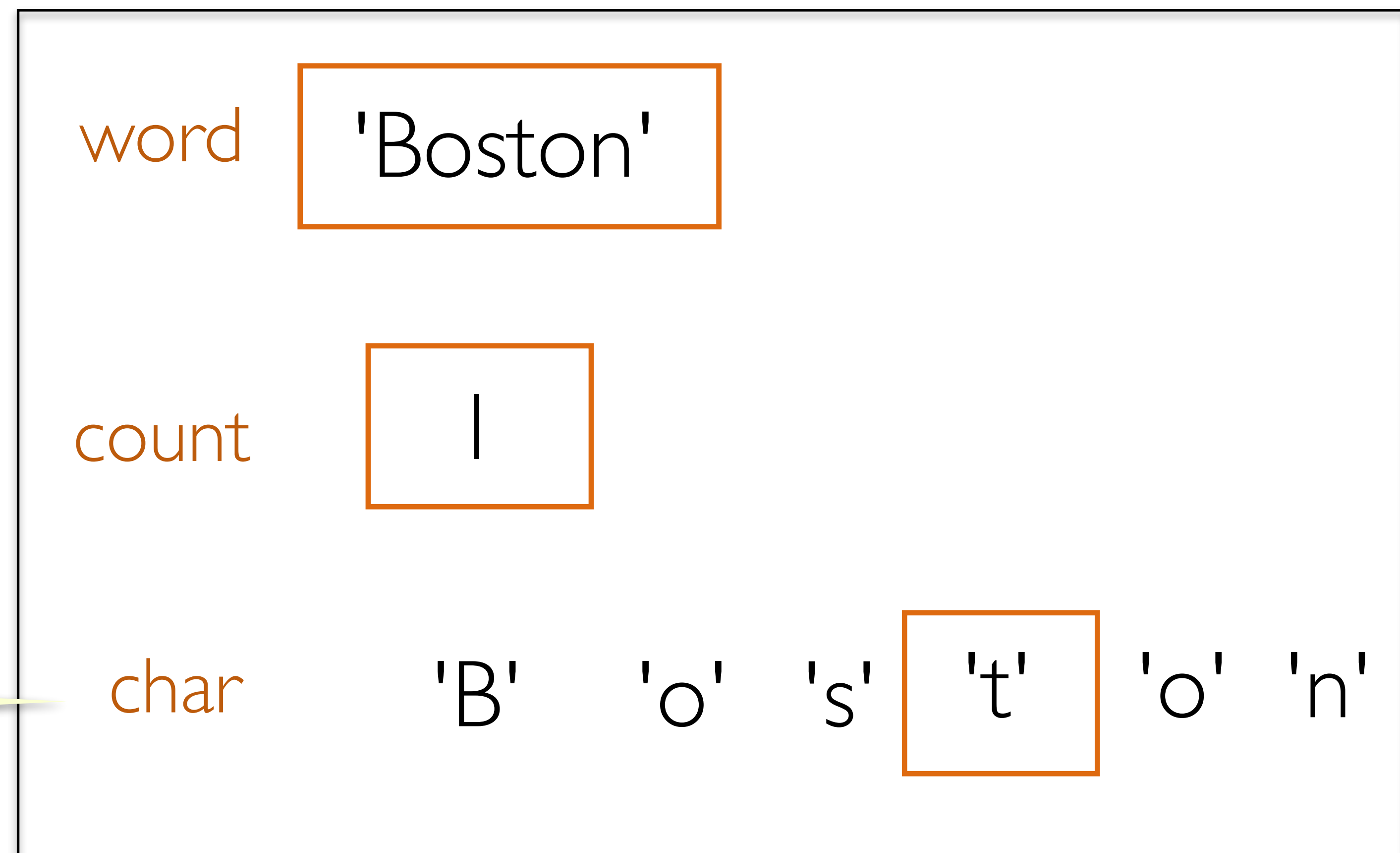


Loop variable

Counting Vowels: Tracing the Loop

```python
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0
    for char in word:
        if is_vowel(char):
            count += 1
    return count
```

`countVowels('Boston')`

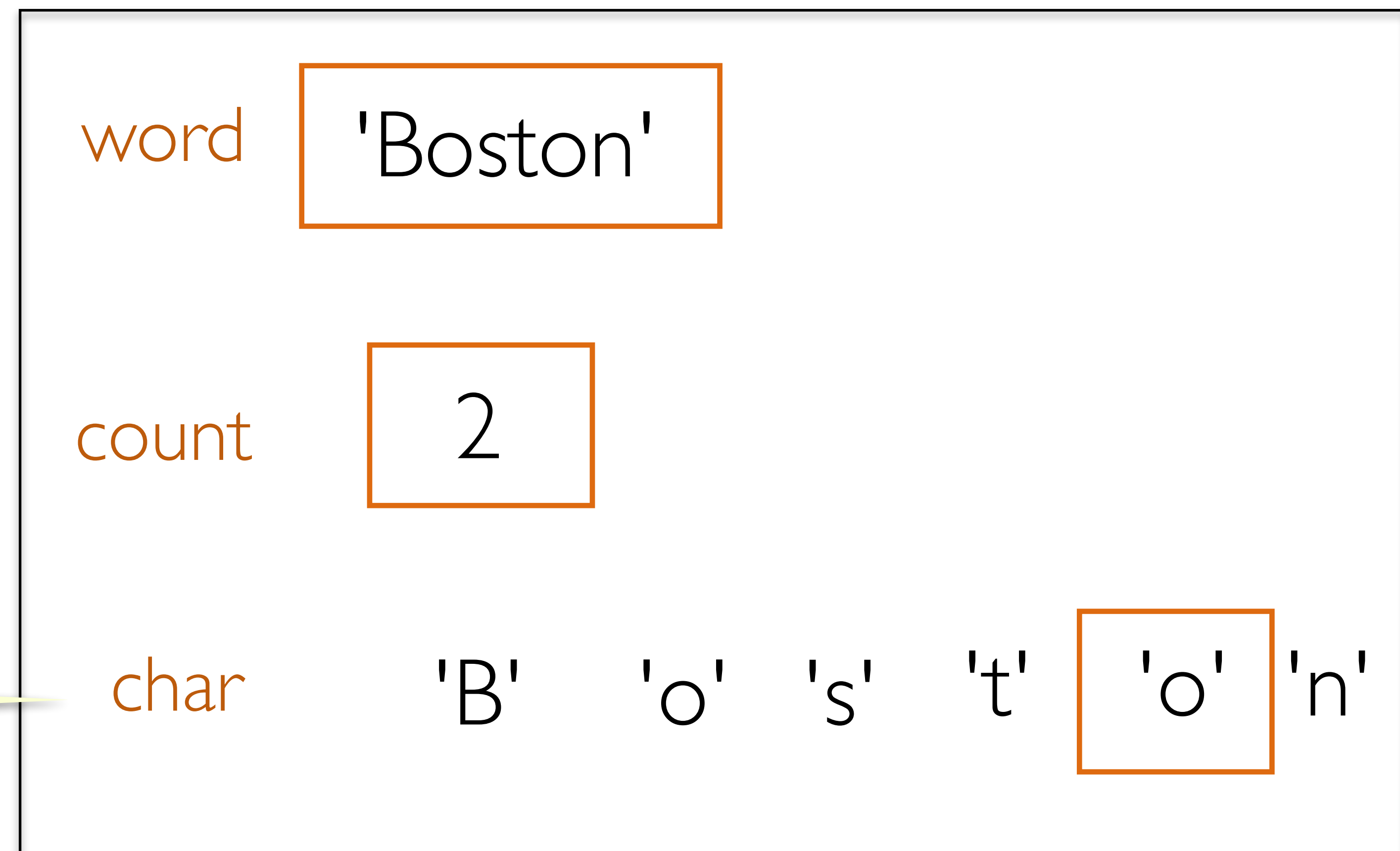


Loop variable

Counting Vowels: Tracing the Loop

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0
    for char in word:
        if is_vowel(char):
            count += 1
    return count
```

`countVowels('Boston')`

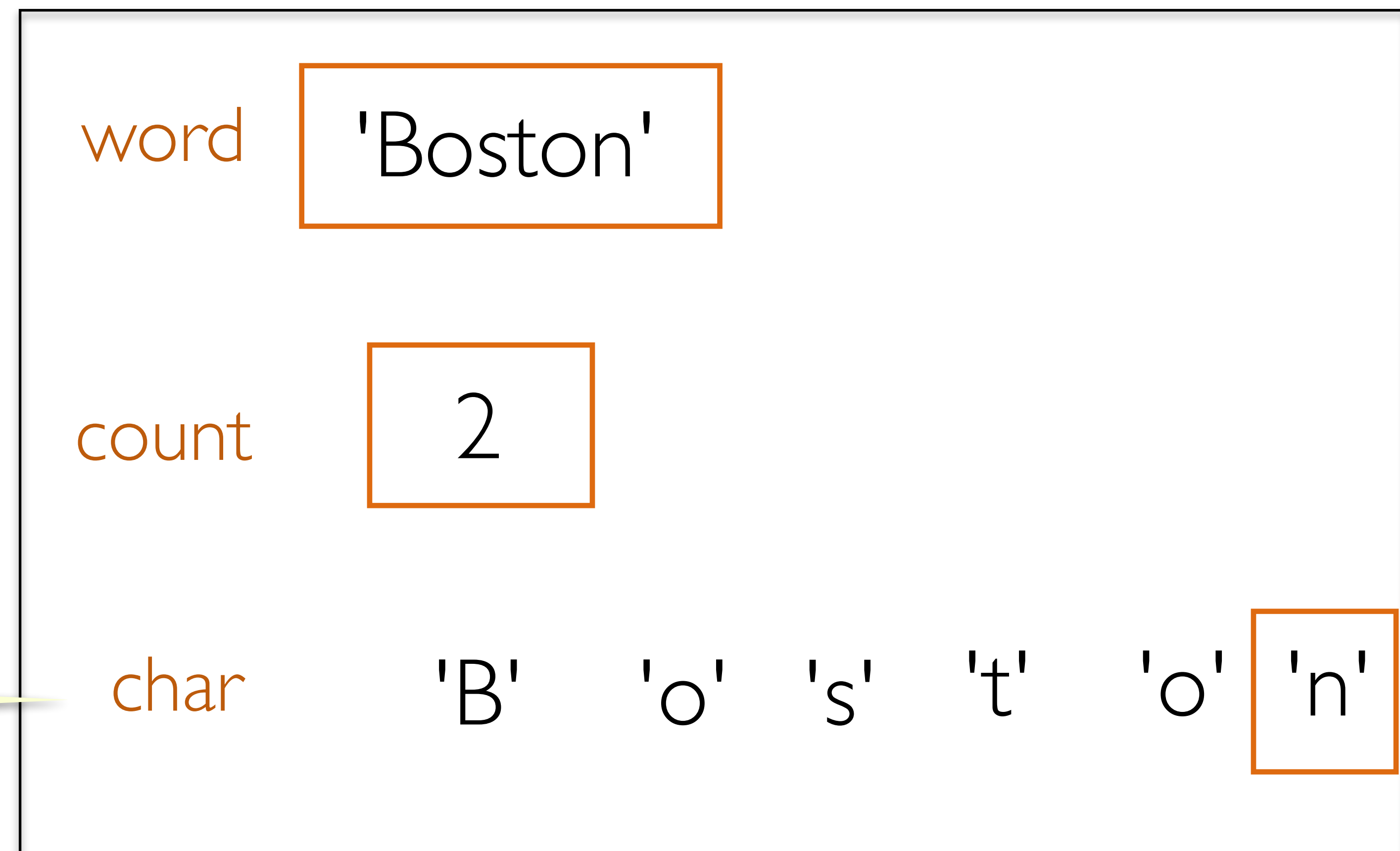


Loop variable

Counting Vowels: Tracing the Loop

```
def count_vowels(word):
    '''Takes word (str) as argument and returns
    the number of vowels in it (as int)'''

    count = 0
    for char in word:
        if is_vowel(char):
            count += 1
    return count
```

countVowels('Boston')



Loop variable

Exercise:
Vowel Sequences

Exercise: Vowel Sequences

- Define a function `vowel_seq(word)` that takes a string `word` and returns a string containing all the vowels in `word` in the order they appear

```
>>> vowel_seq("Chicago")
'iao'
>>> vowels_seq("protein")
'oei'
>>> vowel_seq("rhythm")
''
```

What might be other good values to test edge cases?

Exercise: Vowel Sequences

- Accumulation variables don't have to be counters!
- Can accumulate strings as well: initialize to '' instead of zero

```
def vowel_seq(word):
    '''Takes word (str) as input and returns
    the vowel subsequence in given word (str)'''
    vowels = ""  # initialize accumulation var
    for char in word:
        if is_vowel(char): # if vowel
            vowels += char # accumulate characters
    return vowels
```

Lists

A New Sequence: Lists

- A list is a comma separated, ordered sequence of values.
- These values can be **heterogenous** (strings, ints, floats, etc)
  - Example: `my_list = ['Hello', 42, 23.5, True]`
  - Remember, we **zero-index**! So we say that `'Hello'` is at **index** 0, `42` is at **index** 1, and so on
- Like strings, we can access each element of a list using these **indices**

How Do Indices Work?

- Can access elements of a sequence (such as a list) using its **index**
- Indices in Python are both positive and negative
- Everything outside of these values will cause an `IndexError`.

0	1	2	3	4
'a'	'e'	'i'	'o'	'u'
-5	-4	-3	-2	-1

```
['a', 'e', 'i', 'o', 'u']
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

Features of Lists

- Lists are:
  - **Comma separated, ordered sequences** of values
  - Can be **heterogenous**: multiple types can appear in the same list
  - **Mutable** (or "changeable") objects in Pythons. In contrast, strings are **immutable** (they cannot be changed).
    - We will discuss ***mutability*** in more detail soon!

```
# Examples of various lists:
>>> wordList = ["What", "a", "beautiful", "day"]
>>> numList = [1, 5, 8, 9, 15, 27]
>>> charList = ['a', 'e', 'i', 'o', 'u']
>>> mixedList = [3.14, 'e', 13, True]
>>> type(numList)
list
```

Lists can be heterogeneous (mixed)!

Accessing Elements of Sequences

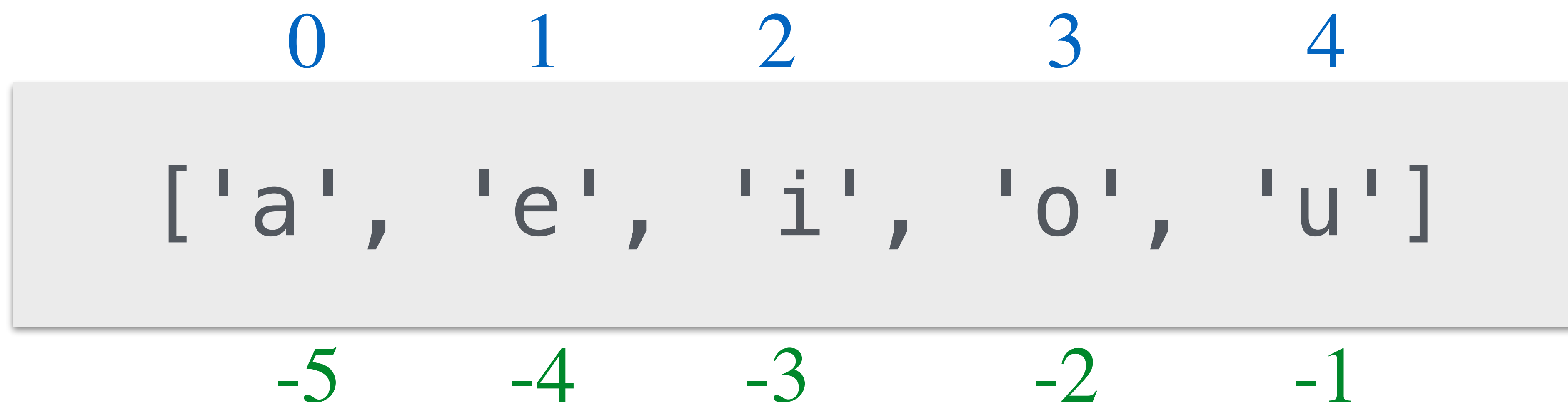
```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[0] # character at 0th index?
'a'
>>> vowels[3] # character at 3rd index?
'o'
>>> vowels[4] # character at 4th index?
'u'
>>> vowels[5] # will this work?
```

0	1	2	3	4
'a'	'e'	'i'	'o'	'u'
-5	-4	-3	-2	-1

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Negative Indexing

- Negative indexing starts from -1, and provides a handy way to access the last character of a non-empty sequence without knowing its length




```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels[-1]
'u'
```

Note: Most other languages do not support negative indexing!

Slicing Sequences

- We can extract **subsequences** of a sequence using the **slicing** operator `[:]`
- For a given sequence `var`, `var[start:end]` returns a new sequence starting at index '`start`' (inclusive), ending at index '`end`' (exclusive)
- Example: Suppose we want to extract the sublist `['a','e']` from `vowels` using slicing operator `[:]`

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> # return the sequence from 0th index up to 1st
>>> # (not including 2nd)
>>> vowels[0:2]
['a','e']
```

Slicing Sequences: Using Step

- The (optional) third **`step`** parameter to the slicing operator determines in what direction to traverse, and whether to skip any elements while traversing and creating the subsequence
- By default, **`start = 0, end = len(), step = +1`** (which means move left to right in increments of one)
- If we omit any of the three parameters, slice uses the default values

```
>>> evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> evens[0:5] # start is 0, end is 5, step is +1
[2, 4, 6, 8, 10]
>>> evens[:8:2] # start is 0, end is 8, step is +2
[2, 6, 10, 14]
>>> evens[::2] # start is 0, end is 10, step is +2
[2, 6, 10, 14, 18]
```

Slicing Sequences: Optional Step

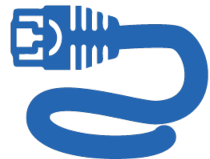
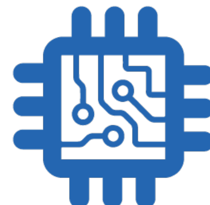
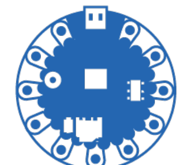
- When the step parameter is set to a negative value it gives a nifty way to reverse sequences
- Note: `start` and `end` are interpreted "backwards" when using a negative step!

```
>>> evens = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> evens[::-1] # reverse the sequence
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

```
>>> evens[::-2]
[20, 16, 12, 8, 4]
```

```
>>> evens[8:0:-1]
[18, 16, 14, 12, 10, 8, 6, 4]
```

Other List Operators

Length of a Sequence

- Python has a built-in `len()` function that computes the length of a sequence such as a list (or any other sequence like a string)
- For a list, `len()` returns the number of elements in the list
- Thus, any list called `words` has the following (positive) indices
  `0, 1, 2, ..., len(words)-1`

```
>>> len(['a', 'e', 'i', 'o', 'u'])
5

>>> len(["Chels", "Artie", "Pixel", "Linus"])
4
```

Testing Membership: `in` Operator

- The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns `True` or `False`

```
>>> "i" in ['a', 'e', 'i', 'o', 'u']
True

>>> "a" in ['a', 'e', 'i', 'o', 'u']
True
>>> "A" in ['a', 'e', 'i', 'o', 'u'] # caps matter
False
```

Membership in Sequences

- The `in` operator in Python is used to test if a given sequence is a subsequence of another sequence; returns True or False

```
>>> dogList = ["Chels", "Artie", "Pixel", "Linus"]
>>> "Linus" in dogList
True
>>> "Dizzy" in dogList
False
```

`not in` sequence operator

- The **`not in`** operator in Python returns True if and only if the given element is **not** in the sequence

```
>>> dogList = ["Chels", "Artie", "Pixel", "Linus"]
>>> "Linus" in dogList
True
>>> "Dizzy" in dogList
False
```

```
>>> "Dizzy" not in dogList
True
```

```
>>> "z" not in "Linus"
True
```

Note that `not in` also works for strings

List Concatenation

- We can use the **+** operator to ***concatenate*** lists together
- Creates a **new list** with the combined elements of the sublists
  - *Does not modify original lists!*

```
>>> aList = ["the", "quick", "brown", "fox"]
>>> bList = ["jumped", "over", "the", "dogs"]
>>> aList + bList # concatenate lists
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'dogs']
>>> aList
['the', 'quick', 'brown', 'fox']
>>> bList = bList + ["back"] # add "back" to bList
>>> bList # since we reassign result to bList, bList has changed
['jumped', 'over', 'the', 'dogs', 'back']
```

returns a **new list** with elements from aList and bList

aList is unchanged!

To change bList, we have to reassign bList to the new list

Review: Basic Operations on Sequences

```
>>> wordList = ["What", "a", "beautiful", "day"]
>>> wordList[3]
'day'
```

Indexing lists using [ ]

```
>>> wordList[-1]
'day'
>>> len(wordList)
4
```

Finding length of list using len( )

```
>>> dogList = ["Chels", "Artie", "Pixel", "Linus"]
>>> dogList[2:4]
['Pixel', 'Linus']
```

Slicing lists using [:] (can also use optional step)

Sequence Operations

Operation	Result
`seq[i]`	The `i`'th item of `seq`, when starting with 0
`seq[si:ee]`	slice of `seq` from `si` to `ee`
`seq[si:ee:s]`	slice of `seq` from `si` to `ee` with step `s`
`len(seq)`	length of `seq`
`seq1 + seq2`	The concatenation of `seq1` and `seq2`
`x in seq`	True if `x` is contained within `seq`
`x not in seq`	False if `x` is contained within `seq`

All of these operators work on both **strings** and **lists**!