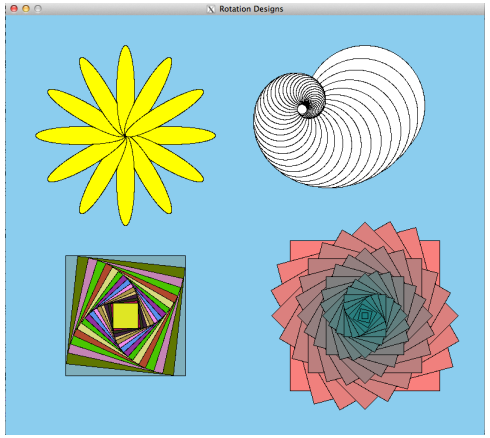
Iteration - Sequences and Loops



CS111 Computer Programming

Department of Computer Science
Wellesley College

Motivation: How to count the number of vowels in a word?

- You're given words like `'Boston'`, `'Wellesley'`, `'abracadabra'`, `'bureaucracies'`, etc.

- Tasks:
  - count the number of vowels in a word.
  - count the number of times a certain character appears in a word

```
def countAllVowels(word):
    # body here
```

?

```
def countChar(char, word):
    # body here
```

?

Slides 8-3 to 8-12 explain what we need to know/learn to solve these problems.

Old friend: `isVowel`

```
def isVowel(char):
    c = char.lower()
    return (c == 'a' or c == 'e' or c == 'i'
            or c == 'o' or c == 'u')
```

```
def isVowel(char):
     return (char.lower() in 'aeiou')
```

To think: How will the function **`isVowel`** be useful for solving our "counting vowels" problem?

Indices: accessing characters in a string

Concepts in this slide: Indices and their properties.

```
In [1]: word = 'Boston'
In [2]: word[0]
Out[2]: 'B'
In [3]: word[1]
Out[3]: 'o'
In [4]: word[2]
Out[4]: 's'
In [5]: word[3]
Out[5]: 't'
In [5]: word[4]
Out[5]: 'o'
In [5]: word[5]
Out[5]: 'n'
```

Notice

- 0, 1, 2, etc. are the **indices** (plural of **index**).
- Indices start at 0.
- Indices go from 0 to `len(word)-1`.
- We read **`word[0]`** as word sub 0.
- **`[]`** is known as the indexing operator.

To think: How will indices be useful for solving our "counting vowels" problem?

A possible solution: which is correct?

Concepts in this slide: Difference between independent vs. chained conditionals.
New operator**: +=**

```
word = 'Boston'
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
print(counter)
```

```
word = 'Boston'
counter = 0
if isVowel(word[0]):
    counter += 1
elif isVowel(word[1]):
    counter += 1
elif isVowel(word[2]):
    counter += 1
elif isVowel(word[3]):
    counter += 1
elif isVowel(word[4]):
    counter += 1
elif isVowel(word[5]):
    counter += 1
print(counter)
```

Does our solution work for all words?

- Do you think the right solution from 8-5 will work for all words: `'Wellesley'`, `'Needham'`, `'Lynn'`, etc.?
- What happens if we use an index that's greater than or equal to the length of the word?

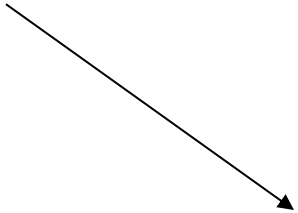
```
In [1]: word = 'Lynn'
In [2]: word[4]
IndexError: string index out of range
```

How to generate the correct indices of the string?

Creating a list of indices with `range`

Concepts in this slide: Built-in function **range** and new type list.

When the **range** function is given two integer arguments, it returns a range object of all integers starting at the first and up to, *but not including*, the second. However, when we give the interpreter `range(0, 10)`, for example, the output is not very helpful. To see all the numbers that are included in **range**, we pass that to the **list** function which returns to us a list of the numbers.



```
In [1]: range(0, 10)
Out[1]: range(0, 10)

In [2]: list(range(3, 7))
Out[2]: [3, 4, 5, 6]

In [3]: list(range(3, 2))
Out[3]: []

In [4]: list(range(3, 3))
Out[4]: []

In [5]: list(range(3))# missing 1st argument defaults to 0
Out[5]: [0, 1, 2]
```

A **list** is a new Python type. It stores a sequence of any values, delimited by square brackets, and separated by commas. More on slide 9.

Properties of the `range` function

An optional third argument to **`range`** controls the step size between elements (which defaults to 1).

```
In [1]: list(range(1, 10, 2))
Out[1]: [1, 3, 5, 7, 9]

In [2]: list(range(3, 70, 10))
Out[2]: [3, 13, 23, 33, 43, 53, 63]

In [3]: list(range(9, 0, -1))
Out[3]: [9, 8, 7, 6, 5, 4, 3, 2, 1]

In [4]: list(range(9, 0, -2))
Out[4]: [9, 7, 5, 3, 1]

In [5]: list(range(63, 0, -10))
Out[5]: [63, 53, 43, 33, 23, 13, 3]
```

To notice:

- With the help of the third argument of range, we can create different sequences of integers.
- Step can be positive or negative.
- When step is negative, start value has to be larger than end value.

Introducing a new value type: lists

Digging Deeper

`range()` returns values of type **`range`**

```
In [1]: type(range(0, 10))
Out[1]: range
```

`list()` returns values of type **`list`**

```
In [2]: type(list(range(0, 10)))
Out[2]: list
```

We'll return to lists again in a few lectures. Lists are an example of "mutable" objects in Python, different from the "immutable" strings.

`list()` can also convert a string into a list of characters

```
In [3]: list("Wendy Wellesley")
Out[3]: ['W', 'e', 'n', 'd', 'y', ' ', 'W', 'e', 'l',
'l', 'e', 's', 'l', 'e', 'y']
```

We can also specify a list directly as a comma separated list of values

```
In [4]: phrase = ["A", "lovely", "autumn", "day"]
In [5]: phrase
Out[5]: ['A', 'lovely', 'autumn', 'day']
```

Back to our vowel counting problem

Concepts in this slide: The combination of `range` and `len` to generate indices for a sequence.

```
word = 'Boston'
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
print counter
```

```
In [1]: word = 'Boston'
In [2]: list(range(len(word)))
Out[2]: [0, 1, 2, 3, 4, 5]

In [3]: word = 'Wellesley'
In [4]: list(range(len(word)))
Out[4]: [0,1,2,3,4,5,6,7,8]

In [5]: word = 'Lynn'
In [6]: list(range(len(word)))
Out[6]: [0, 1, 2, 3]
```

`range` solves our indexing problem, by generating the correct list of indices.

Concepts in this slide: An example of a `for` loop with a conditional.

Loops to the rescue!

```
word = 'Boston'
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
print counter
```

```
word = 'Boston'
counter = 0
for i in range(len(word)):
    if isVowel(word[i]):
        counter += 1
print(counter)
```

Important: we have been using **`list`** to display the numbers held in **`range`** but we do not need it to iterate! Note how we do not write **`list(range(len(word)))`**

Concepts in this slide: New execution kind: iteration done through loops.

Iterating Over Sequences with `for` Loops

One of the most common ways to 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. In Python, we use a **`for`** loop to iterate.

```
for var in sequence:
    # Body of the loop
    statements using var
```

Generic form of a **`for`** loop

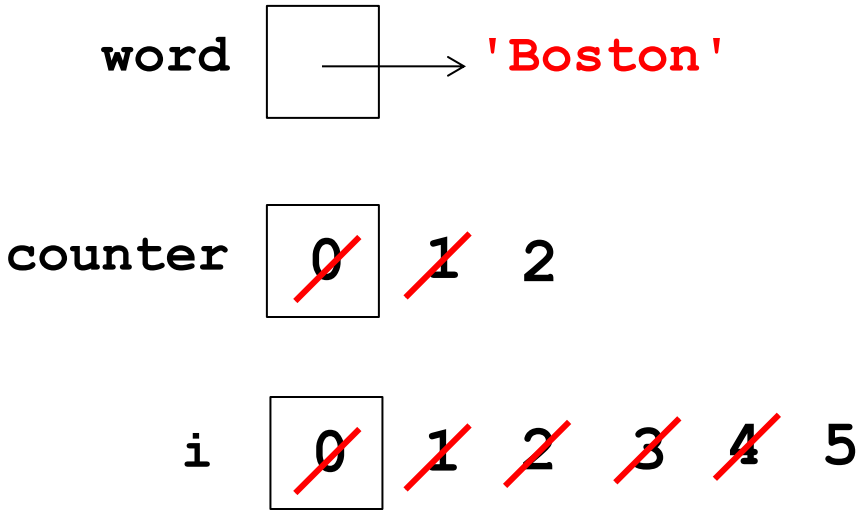
`for` loop model example

Concepts in this slide: Modeling how code in a loop is executed.

```
word = 'Boston'

counter = 0

for i in range(len(word)):
    if isVowel(word[i]):
        counter += 1

print(counter)
```

[0,1,2,3,4,5]

6



To notice:

- There are three variables in the code: **`i`**, **`word`**, **`counter`**.
- The variables **`i`** and **`counter`** change values from one iteration to the next.
- Because **`counter`** is within a conditional, it's only updated when the condition is true.

`for` loops without `range`

Concepts in this slide: Two different ways of looping over a sequence.

- The **`range`** function provides a sequence of indices

Index Loop

```
phrase = ["an", "autumn", "day"]
for i in range(len(phrase)):
        print(phrase[i] + '!')

an!
autumn!
day!
```

Unless there is a need for the index (see slide 8-15), we will prefer value loops over index loops.

- We can also loop directly over any list. The code below produces the **same output**.

Value Loop

```
phrase = ["an", "autumn", "day"]
for word in phrase:
        print(word + '!')
```

When is it better to use `range` instead of directly looping?

- Let's modify the previous example to print both the index and the item for each item in the list.

```
for i in range(len(phrase)):
    print(i, phrase[i], '!')
```

```
0 an!
1 autumn!
2 day!
```

- Notice this would NOT be possible if we directly looped over the list.

Strings, lists, and ranges are all sequences

Digging Deeper

```
In [1]: word =
'Boston'
In [2]: word[2]
Out[2]: 's'
In [3]: len(word)
Out[3]: 6
In [4]: word + 'Globe'
Out[4]: 'Boston Globe'
In [5]: 'o' in word
Out[5]: True
In [6]: 'b' in word
Out[6]: False
```

```
In [1]: digits =
[1, 2, 3, 4]
In [2]: digits[2]
Out[2]: 3
In [3]: len(digits)
Out[3]: 4
In [4]: digits + [4]
Out[4]: [1, 2, 3, 4, 4]
In [5]: 1 in digits
Out[5]: True
In [6]: 5 in digits
Out[5]: False
```

```
In [1]: digRange =
range(1, 5)
In [2]: digRange[2]
Out[2]: 3
In [3]: len(digRange)
Out[3]: 4
In [5]: 1 in digRange
Out[5]: True
In [6]: 5 in digRange
Out[5]: False
```

A sequence is an "abstract" type, which serves as template for "concrete" types such as string or list. Note that concatenation is not supported for range objects. Doing `range(3) + range(2)` will result in a TypeError.

Looping over a string


```
word = 'Boston'
counter = 0
for i in range(len(word)):
    if isVowel(word[i]):
        counter += 1
print(counter)
```

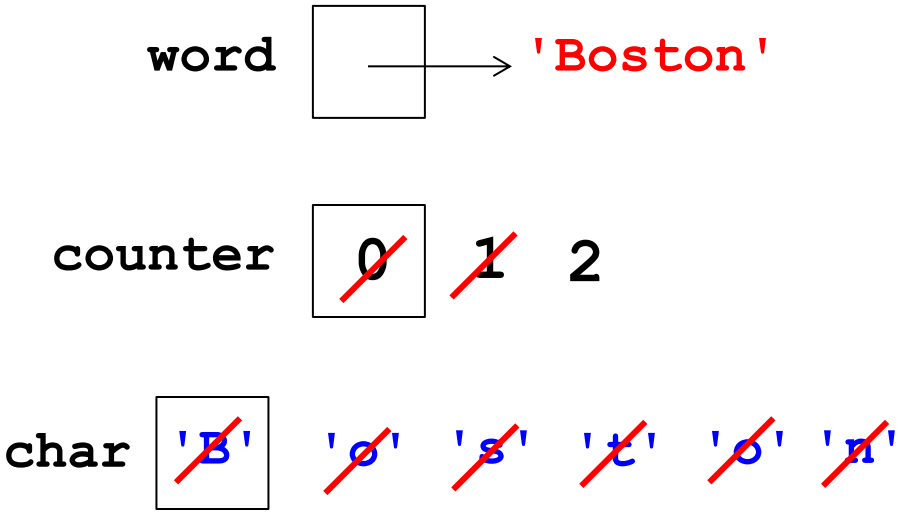
- Can we avoid **range** in this code as we did in 15? **It turns out yes, in the same way.**

```
word = 'Boston'
counter = 0
for char in word:
    if isVowel(char):
        counter += 1
print(counter)
```

for loop model example #2

Digging Deeper

```
word = 'Boston'

counter = 0


['B','o','s','t','o','n']

for char in word:
    if isVowel(char):
        counter += 1


print(counter)
```



To notice:

- There are again three variables in the code: **word**, **counter**, **char**.
- The variables **char** and **counter** change values from one iteration to the next.
- Notice that **char** takes as values the elements of the string sequence.

Operations in Sequences

Operation	Result
`x in seq`	True if an item of seq is equal to x
`x not in seq`	False if an item of seq is equal to x
`seq1 + seq2`	The concatenation of seq1 and seq2*
`seq*n, n*seq`	n copies of seq concatenated
`seq[i]`	i'th item of seq, where origin is 0
`seq[i:j]`	slice of seq from i to j
`seq[i:j:k]`	slice of seq from i to j with step k
`len(seq)`	length of seq
`min(seq)`	smallest item of seq
`max(seq)`	largest item of seq

*Recall that concatenation is not supported for range objects.

A simpler version of `isVowel` using `in`

Digging Deeper

```
# Old version
def isVowel(char):
    c = char.lower()
    return (c == 'a' or c == 'e' or c == 'i' or
            c == 'o' or c == 'u')
```

```
# Simpler version
def isVowel(char):
    return char.lower() in 'aeiou'
```

Digging Deeper

The Slicing operator [:]

```
In [1]: word =
'Boston'
In [2]: word[2]
Out[2]: 's'
In [3]: word[2:5]
Out[3]: 'sto'
In [4]: word[:3]
Out[4]: 'Bos'
In [5]: word[3:10]
Out[5]: 'ton'
In [6]: word[3:]
Out[6]: 'ton'
In [7]: word[0:6:2]
Out[7]: 'Bso'
In [8]: word[::-1]
Out[8]: 'notsoB'
```

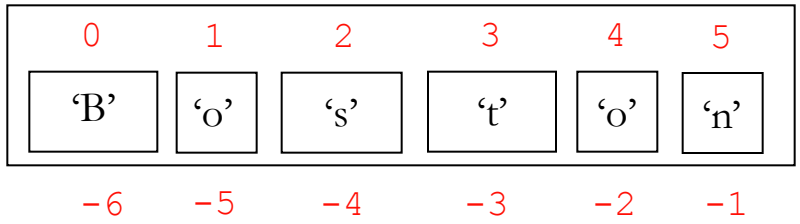
```
In [1]: digits =
[1, 2, 3, 4]
In [2]: digits[2]
Out[2]: 3
In [3]: digits[1:4]
Out[3]: [2, 3, 4]
In [4]: digits[:3]
Out[4]: [1, 2, 3]
In [5]: digits[3:10]
Out[5]: [4]
In [6]: digits[3:]
Out[6]: [4]
In [7]: digits[0:5:2]
Out[7]: [1, 3]
In [8]: digits[::-1]
Out[8]: [4, 3, 2, 1]
```

```
In [1]: digRange =
range(1, 5)
In [2]: digRange[2]
Out[2]: 3
In [3]: digRange[1:4]
Out[3]: range(2, 5)
In [4]: digRange[:3]
Out[4]: range(1, 4)
In [5]: digRange[3:10]
Out[5]: range(4, 5)
In [6]: digRange[3:]
Out[6]: range(4, 5)
In [7]: digRange[0:5:2]
Out[7]: range(1, 5, 2)
In [8]: digRange[::-1]
Out[8]: range(4, 0, -1)
```

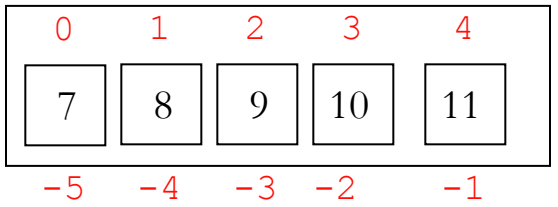
Digging Deeper

How do indices work?

```
word = 'Boston'
```




```
digits = range(7, 12)
```



Indices in Python are both positive and negative.
Everything outside these values will cause an IndexError.

```
In [7]: word[::-1]
Out[7]: 'notsoB'
```

This means: start at 0 until the end of sequence with step -1. And it works because of the negative indices.

`for` loops are disguised `while` loops!

Digging Deeper

```
sumList([17,8,5,12]) returns 42
sumList(range(1,11)) returns 55

def sumListFor(nums):
    '''Returns the sum of the elements in nums'''
    sumSoFar = 0
    for n in nums:
        sumSoFar += n # or sumSoFar = sumSoFar + n
    return sumSoFar


# If Python did not have a for loop, the above for loop
# could be automatically translated to the while loop below
def sumListWhile(nums):
    '''Returns the sum of the elements in nums'''
    sumSoFar = 0
    index = 0
    while index < len(nums):
        n = nums[index]
        sumSoFar += n # or sumSoFar = sumSoFar + n
        index += 1   # or index = index + 1
    return sumSoFar
```

Summary

1. Strings, lists, and ranges are examples of sequences, **ordered** items that are stored together. Because they are ordered, we can use indices to access each of them individually and sequentially.
2. How does a `for` loop differ from a `while` loop? How are they similar?
3. The indexing operator `[]` takes index values from 0 to len(sequence)-1. However, negative indices are possible too in Python.
4. If we can access each element of a sequence (string, list, range) one by one, then we can perform particular operations with them.
5. To access each element we need a **loop**, an execution mechanism that repeats a set of statements until a stopping condition is fulfilled.
6. When we loop over a sequence, the stopping mechanism is the arrival at the last element and not having anywhere to go further.
7. We use the built-in function `range` to generate indices for sequences.
8. Python makes it easy for us to iterate over a sequence's elements even without the use of indices. In fact we can write: `for item in sequence:` and that will access each item of the sequence.
9. In addition to accessing one element at a time with `[]`, one can use `[:]` (slicing) to get a substring, sublist, or subrange.