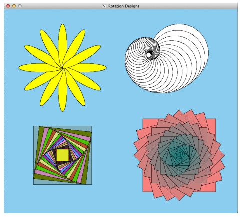
Iteration:
Sequences and `for` Loops



CS111 Computer Programming

Department of Computer Science
Wellesley College

Overview pt. 1

- **Primitive** types in Python: int, float, Boolean, NoneType.
  - Values of such types cannot be decomposed in smaller parts.
  - This is why in the memory diagram model for variables we depict these values **within the variable box**. They cannot be broken into smaller units.
- **Composite** types in Python: str, list, range, tuple, dict.
  - Values of such types can be decomposed further.
  - This is why in the memory diagram model for variables we depict these values **outside of the variable box**.

Overview pt. 2

- Strings are known as **sequences** in Python, because they are **ordered**.
- Lists, ranges, and tuples (new data types we will learn about), are also ordered sequences.
- To represent order in a sequence, we use **indices**. Python has an indexing operator **[ ]** (square brackets) that allows us to **access** an element at a certain position in the ordered sequence.
- We always start counting indices in a sequence at the value 0.

Recall:

```
word = 'Boston’
```

`word[0]` has the value `'B'`, `word[3]` has the value `'t'` ,
`word[-1]` and `word[5]` have the value `'n'`, and so on.

Overview pt. 3

- Most of the time, we will access sequence items through a **`for` loop**.
- This lecture discusses sequences and `for` loops together.
- We present two types of `for` loops: **value loops** and **index loops**.

Motivation Example:
How many 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 countVowels(word):
    # body here

         ?
```

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

         ?
```

Slides 4 to 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')

 # A more concise version
 def isVowel(char):
     return (len(char) == 1
             and char.lower() in 'aeiou')
```

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

Review: Accessing characters in a string through indices

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 pronounce `word[0]` as "word sub 0".
- `[]` is the **indexing operator**.

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

Possible solution: which side is correct?

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

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

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

Does our solution work for all words?

- Do you think the right-side solution from the previous slide 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?

Approach 1:
Using a **while** loop to visit all string indices

```
word = 'Boston'
index = 0
while index < len(word):
    print('word[' + str(index) + '] => ' + word[index])
    index += 1
```

```
word[0] => B
word[1] => o
word[2] => s
word[3] => t
word[4] => o
word[5] => n
```

while loops to the rescue!

Concepts in this slide:
An example of a **while** loop over a string that accumulates a value

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

```
word = 'Boston'
vowelCount = 0
index = 0
while index < len(word):
    if isVowel(word[index]):
        vowelCount += 1
    index += 1
print(vowelCount)
```

Approach 2:
Iterating over sequence elements with **for** loops

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

A common way 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

A `for` loop executes the statements in the body of the loop for each element in the sequence. In each execution of the body, the **iteration variable** *`var`* holds the current element.

Value `for` loop for vowel counting

```
word = 'Boston'

vowelCount = 0

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

print(vowelCount)
```

What does this code print?

Guess before going to the next slide!

To notice:
- There are three variables in the code: `word`, `vowelCount`, `char`.
- The variables `char` and `vowelCount` can change values from one iteration to the next.
- `char` takes as values the elements of the string sequence.
- There's no need for an explicit `index` variable! This is the main advantage of a **`for`** loop over a **`while`** loop

`for` loop model example for vowel counting

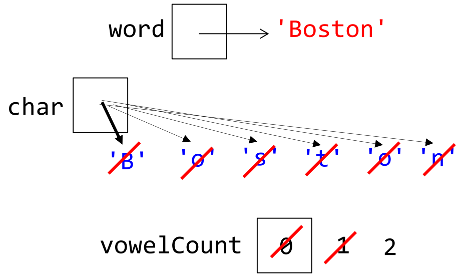
```
word = 'Boston'

vowelCount = 0

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

print(vowelCount)
```

2



New ordered, composite data type: Lists

While strings are sequences of characters, a **list** is a sequence of elements that are **any type of value**.

Lists are written as values that are separated by commas and wrapped in a pair of square brackets.

Examples:

`[17, 8, 12, 5]` is a list of four integers.

`['Boston', 'Paris’, 'Seoul']` is a list of three strings.

`['New York', 13, True]` is a list with a string, int, and bool.

New ordered, composite data type: Lists

The type of a list value is `list`.

E.g. `type([17, 8])` is `list`.

There's also a built-in function `list` that converts a string to a list of characters.

Example:

`list('cat')` evaluates to `['c', 'a', 't']`

for loops can iterate over lists of values

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

What does this code print?

```
def sumList(nums):
   '''Returns the sum of the elements in nums'''
    sumSoFar = 0
    for num in nums:
        sumSoFar += num
    return sumSoFar
```

What does `sumList([17,8,12,5])` return?

for loops can iterate over lists of values

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

```
an!
autumn!
day!
```

```
def sumList(nums):
   '''Returns the sum of the elements in nums'''
    sumSoFar = 0
    for num in nums:
        sumSoFar += num
    return sumSoFar
```

```
In [1]: sumList([17,8,12,5])
Out [1]: 42
```

for loops are disguised while loops!

Digging Deeper

```
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
```

Creating a sequence of numbers 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.

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

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

Creating a sequence of numbers with range

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

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 the defined range.

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

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

In [5]: list(range(3, 3))
Out[5]: []

In [6]: list(range(3))# missing 1st argument defaults to 0
Out[6]: [0, 1, 2]
```

Properties of the range function

An optional third argument to **range** controls the step size between elements. Without the third argument, default step is 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, the start value has to be larger than the end value.

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: `range(3) + range(2)` will result in a `TypeError`.

Solving the indexing problem

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

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

```
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.

Approach 3 of vowel counting: Index for loop

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

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

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

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

for loop model example with range

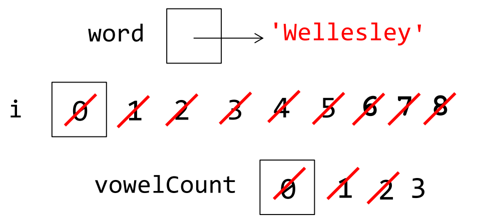
Concepts in this slide: Modeling how code in a loop is executed.

```
word = 'Wellesley'

vowelCount = 0

        [0,1,2,3,4,5,6,7,8]
                  9
for i in range(len(word)):
    if isVowel(word[i]):
    vowelCount += 1

print(vowelCount)
```



To notice:

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

Value loops vs. index loops

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

- We can loop directly over the elements in a list.

Value Loop

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

- The `range` function provides a sequence of indices that we can loop over to access the elements from a sequence. The code below produces the **same output** as the code above.

Index Loop

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

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

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

Digging Deeper

- 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!
```

- Tracking the order of elements would **not** be possible if we directly looped over the values in list.

Exiting loops early

Concepts in this slide:
A loop can be exited early via `return` or `break`.

- Sometimes we want to exit a loop early. Examples:
  - When we're iterating over the elements of a sequence, but have found the element or answer we're looking for. Then there's no need to look through the remaining elements.
  - When we're accumulating a value through a loop and reached some desired value, at which point the loop can end.
- There are two ways we can exit a loop early:
  - Within a function, via a `return` statement
  - Within a `while` or `for` loop, via a `break` statement, which exits the closest surrounding loop.
    We will not cover break in this lecture.

Returning early from a loop

In a function, **`return`** can be used to exit the loop early (e.g., before it visits all the elements in a list).

```
def isElementOf(val, elts):
    '''Returns True if val is found in elts; False otherwise'''
    for e in elts:
        if e == val:
            return True # return (and exit the function)
                        # as soon as val is encountered
    return False  # only get here if val is not in elts
```

```
In [1]: sentence = 'the cat that ate the mouse liked
                    the dog that played with the ball'

In [2]: isElementOf('cat', sentence.split())
Out[2]: True # returns as soon as 'cat' is encountered

In [3]: isElementOf('bunny', sentence.split())
Out[3]: False
```

Premature return done wrong (1)

GOTCHA!

```
def isElementOfBroken(val, elts):
   '''Faulty version that returns True if val is found
   in elts; False otherwise'''
    for e in elts:
        if e == val:
            return True
        else:
            return False
```

The function always returns after the 1st element without examining the rest of the list.

```
In[]: isElementOfBroken(2, [2, 6, 1])
Out[]: True
```

```
In[]: isElementOfBroken(6, [2, 6, 1])
Out[]: False
```

Exercises [in the notebook]

it's your turn

In the notebook we'll write the following functions that return early.

Hint
String objects have a method called **`isdigit`**, try it out.

containsDigits

	Returns
`containsDigit('The answer is 42')`	`True`
`containsDigit('76 trombones')`	`True`
`containsDigit('the cat ate the mouse')`	`False`
`containsDigit('one two three')`	`False`

areAllPositive

`areAllPositive([17, 5, 42, 16, 31])`	`True`
`areAllPositive([17, 5, -42, 16, 31])`	`False`
`areAllPositive([-17, 5, -42, -16, 31])`	`False`
`areAllPositive([])`	`True`

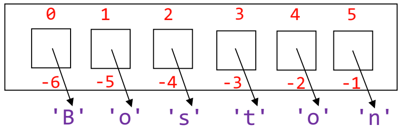
indexOf

`indexOf(8, [8,3,6,7,2,4])`	`0`
`indexOf(7, [8,3,6,7,2,4])`	`3`
`indexOf(5, [8,3,6,7,2,4])`	`-1`

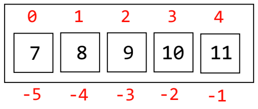
How do indices work?

Digging Deeper

```
word = 'Boston'
```



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



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

The Slicing operator [:]

Digging Deeper

```
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’
// This means: start at 0 until
// the end of sequence with
//  step -1. This reverses the
// string  b/c of negative indices.
```

```
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)
```

Operations in Sequences

Digging Deeper

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` up to, but not including, `j`
`seq[i:j:k]`	slice of `seq` from `i` up to, but not including, `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.

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 by writing `for item in sequence:` and this will access each item of the sequence. (A value loop!)
9. In addition to accessing one element at a time with `[]`, we can use `[:]` (slicing) to get a substring, sublist, or subrange.