

PYTHON LISTS

LECTURE 04-1

JIM FIX, REED COLLEGE CSC112

A NEED FOR DATA STRUCTURES

- ▶ We're limited in our coding if we can store values *only using individual variables*.

A NEED FOR DATA STRUCTURES

- ▶ We're limited in our coding if we can store values *only using individual variables*.
- ▶ What if we want to process...
...a file full of data? ...a web site full of statistics? ...a collection of items?
- ▶ Suppose for example, a user enters in some arbitrary number of values...
 - With single variables, we can't name all of them.

A NEED FOR DATA STRUCTURES

- ▶ We're limited in our coding if we can store values *only using individual variables*.
- ▶ What if we want to process...
...a file full of data? ...a web site full of statistics? ...a collection of items?
- ▶ Suppose for example, a user enters in some arbitrary number of values...
 - With single variables, we can't name all of them.
- ▶ Languages provide ***data structures*** to hold collections of values.
 - Python has two built into the language:
 - Python ***lists*** and Python ***dictionaries***.

OUR FIRST DATA STRUCTURE: PYTHON LISTS

- ▶ Python lets you represent sequences of data values:

```
>>> xs = [2, 3, 7, 15, 100]
```

```
>>> xs
```

```
[2, 3, 7, 15, 100]
```

```
>>> xs[3]
```

```
15
```

```
>>> xs[0]
```

```
2
```

```
>>> len(xs)
```

```
5
```

OUR FIRST DATA STRUCTURE: PYTHON LISTS

- ▶ Python lets you represent sequences of data values:

```
>>> xs = [2, 3, 7, 15, 100]
```

```
>>> xs
```

```
[2, 3, 7, 15, 100]
```

```
>>> xs[3]
```

```
15
```

```
>>> xs[0]
```

```
2
```

```
>>> len(xs)
```

```
5
```

- ▶ This is a built-in data structure called a Python *list*.
 - A list is a *sequence* of numbered data slots, starting at 0.
 - Each slot stores a value.
 - A slot can be accessed by its *index*.
 - A list has a *length*.

MODIFYING A LIST'S CONTENTS

- ▶ A Python list is *mutable*. This means that its contents can be changed.

```
>>> xs
[2, 3, 7, 15, 100]
>>> xs[3]
15
>>> xs[3] = 200
>>> xs[3]
200
>>> xs
[2, 3, 7, 200, 100]
>>> xs[0] = xs[2] + xs[4]
>>> xs
[107, 3, 7, 200, 100]
>>> xs[4] = 1000
>>> xs
[107, 3, 7, 200, 1000]
```

LIST INDEXING

- ▶ You have to be careful when accessing a list; need to be mindful of its length.

```
>>> xs = [2, 3, 7, 15, 100]
>>> xs
[2, 3, 7, 15, 100]
>>> xs[5]
error!
```

- ▶ Using a negative index allows you to access backward from the end of the list:

```
>>> xs[-1]
100
>>> xs[-2]
15
>>> xs[-5]
2
>>> xs[-6]
error!
```

AN EXAMPLE LIST FUNCTION

- ▶ A function that averages the values in a list:

```
def average(numbers):  
    ...
```

AN EXAMPLE LIST FUNCTION

- ▶ The function below averages the values in a list:

```
def average(numbers):
    # numbers is assumed to be a list of floats
    if len(numbers) == 0:
        # We'll define the average value of [] as None
        return None

    # compute their sum
    sum = 0.0
    index = 0
    while index < len(numbers)
        sum = sum + numbers[index]
        index = index + 1

    # take their average
    return sum / len(numbers)
```

AN EXAMPLE LIST FUNCTION

- ▶ The function below averages the values in a list:

```
def average(numbers):  
    # numbers is assumed to be a list of floats  
    if len(numbers) == 0:  
        # We'll define the average value of [] as None  
        return None  
  
    # compute their sum  
    sum = 0.0  
    index = 0  
    while index < len(numbers)  
        sum = sum + numbers[index]  
        index = index + 1  
  
    # take their average  
    return sum / len(numbers)
```

This is a typical loop for scanning a list.

AN EXAMPLE LIST FUNCTION

- ▶ The function below averages the values in a list:

```
def average(numbers):  
    if len(numbers) == 0:  
        return None  
  
    sum = 0.0  
    index = 0  
    while index < len(numbers)  
        sum = sum + numbers[index]  
        index = index + 1  
    return sum / len(numbers)
```

- ▶ Here it is in use:

```
>>> xs = [2.0, 3.5, 7.5, 10.0]  
>>> average(xs)  
11.5  
>>>
```

AN EXAMPLE LIST FUNCTION

- ▶ The function below averages the values in a list:

```
def average(numbers):  
    if len(numbers) == 0:  
        return None  
  
    sum = 0.0  
    index = 0  
    while index < len(numbers)  
        sum = sum + numbers[index]  
        index = index + 1  
    return sum / len(numbers)
```

- ▶ Here it is in use:

```
>>> xs = [2.0, 3.5, 7.5, 10.0]  
>>> average(xs)  
5.75  
>>> average([])  
>>>
```

ANOTHER EXAMPLE LIST FUNCTION

- ▶ This function checks a list's contents:

```
def is_palindrome(xs):  
    ...
```

ANOTHER EXAMPLE LIST FUNCTION

- ▶ This function checks a list's contents:

```
def is_palindrome(xs):
    hi = len(xs) - 1
    lo = 0
    while hi > lo:
        if xs[lo] != xs[hi]:
            return False
        lo = lo + 1
        hi = hi - 1
    return True
```

ANOTHER EXAMPLE LIST FUNCTION

- ▶ This does the same using negative indexing

```
def is_palindrome(xs):  
    index = 0  
    middle = len(xs) // 2  
    while index < middle  
        if xs[index] != xs[-(index+1)]:  
            return False  
        index = index + 1  
    return True
```

YET ANOTHER EXAMPLE LIST FUNCTION

- ▶ A function that checks if two lists hold the same contents, in same order

```
def same_contents(xs, ys):
```

```
    ...
```

YET ANOTHER EXAMPLE LIST FUNCTION

- ▶ This checks to see if the contents of two lists are the same:

```
def same_contents(xs, ys):  
    if len(xs) != len(ys):  
        return False  
    i = 0  
    while i < len(xs):  
        if xs[i] != ys[i]:  
            return False  
        i = i + 1  
    return True
```

ONE MORE EXAMPLE LIST FUNCTION

- ▶ A function that checks if the value **y** is stored in list **xs**:

```
def contains(y, xs):  
    ...
```

ONE MORE EXAMPLE LIST FUNCTION

- ▶ This checks to see if the value **y** is stored in any of the slots of the list **xs**:

```
def contains(y, xs):  
    i = 0  
    while i < len(xs):  
        if xs[i] == y:  
            return True  
        i = i + 1  
    return False
```

SOME LIST CONTENT CHECKS

- ▶ Python has `contains` and `same_contents` built into its language:

```
>>> 4 in [1,2,4,8] # Does the list contain an element?
```

```
True
```

```
>>> 7 in [1,2,4,8]
```

```
False
```

```
>>> xs = [1,3,4]
```

```
>>> ys = [1,3,5]
```

```
>>> xs == ys # Are the lists' contents the same?
```

```
False
```

```
>>> xs != ys
```

```
True
```

```
>>> ys[2] = 4
```

```
>>> xs == ys
```

```
True
```

```
>>> xs != ys
```

```
False
```

```
>>> xs is ys # Are they the same list object?
```

```
False
```

- ▶ The operators `in` and `==` check list *contents*.

LIST IDENTITY CHECKS

- ▶ There is also sometimes a need to check whether you are working with the same list object.

```
>>> list1 = [1, 3, 4, 8]
>>> list2 = [7, 3, 6]
>>> xs = list1
>>> xs is list1      # Are they the same list object?
True
>>> xs is list2
False
>>> xs[2] = xs[2] + 100
>>> list1
[1, 3, 104, 8]
>>>
```

- ▶ The operator `is` checks list *identity*.

MODIFYING LISTS: ADDING AND INSERTING

- ▶ We can add more slots to a list object:

```
>>> xs = [13,5,71]
>>> xs
[13, 5, 71]
>>> xs.append(-57)           # Adds a new slot to the end.
>>> xs
[13, 5, 71, -57]
>>> xs.extend([7,8,9])      # Adds several slots to the end.
>>> xs
[13, 5, 71, -57, 7, 8, 9]
>>> xs.insert(2,100)        # Adds a slot in the middle.
>>> xs
[13, 5, 100, 71, -57, 7, 8, 9]
```

MODIFYING LISTS: REMOVING

- ▶ We can remove slots from a list object:

```
>>> xs
[13, 5, 100, 71, -57, 7, 8, 9]
>>> xs.pop()      # Remove the last slot; return its value.
9
>>> xs
[13, 5, 100, 71, -57, 7, 8]
>>> xs[2]
100
>>> del xs[2]     # Remove a slot at a certain index.
>>> xs
[13, 5, 71, -57, 7, 8]
>>> xs[2]        # The other items shift left.
71
```

RETURNING LISTS

- ▶ This function builds a list of integers:

```
def count_up(n):  
    i = 1  
    counts = []  
    while i <= n:  
        counts.append(i)  
        i = i + 1  
    return counts
```

```
>>> count_up(7)  
[1, 2, 3, 4, 5, 6, 7]
```

RETURNING LISTS

- ▶ This function builds a number's divisor sequence:

```
def divisor_list(number):  
    sequence = [1]  
    divisor = 2  
    while divisor < number:  
        if number % divisor == 0:  
            sequence.append(divisor)  
    sequence.append(number)  
    return sequence
```

```
>>> divisor_list(35)  
[1, 5, 7, 35]  
>>> divisor_list(1)  
[1]  
>>> divisor_list(7)  
[1, 7]  
>>> divisor_list(36)  
[1, 2, 3, 4, 6, 9, 12, 18, 26]
```

PROCEDURES THAT MODIFY A LIST

- ▶ This function modifies a list.

```
def rotate_right(xs):  
    if len(xs) > 1:  
        last = xs.pop()  
        xs.insert(0, last)
```

- ▶ Calling `rotate_right` has the *side effect* of changing the list you give it:

```
>>> dsForSixteen = divisor_list(16)  
>>> dsForSixteen  
[1, 2, 4, 8, 16]  
>>> rotate_right(dsForSixteen)  
>>> dsForSixteen  
[16, 1, 2, 4, 8]  
>>> rotate_right(dsForSixteen)  
>>> dsForSixteen  
[8, 16, 1, 2, 4]
```

READING FOR PYTHON LISTS

▶ **Reading:**

- PP Ch 2.2
- TP Ch 9
- CP Ch 2.3

PYTHON LIST SUMMARY

- ▶ List description via enumeration/listing:

```
[3,1,7]  []
```

- ▶ Accessing contents by index; list length:

```
xs[3]  xs[-1]  len(xs)
```

- ▶ Updating contents by indexed assignment:

```
xs[3] = 5
```

- ▶ Modifying/mutating a list object:

```
xs.append(5)    xs.extend([8,9,10])    xs.insert(2,357)  
xs.pop()        del xs[6]
```

- ▶ Checking membership, content equality, and identity:

```
3 in xs        xs == [1,2,3]        xs is ys
```

- ▶ Scan using an integer index using a **while** loop:

```
i = 0  
while i < len(xs):  
    print(xs[i])  
    i = i + 1
```