Lists

CSE 1310 – Introduction to Computers and Programming
Vassilis Athitsos
University of Texas at Arlington

Motivating Exercise

- Let's write a program that:
  - Asks the user to enter three numbers.
  - Prints how many of those numbers are less than the last number entered.

- Example:
  - user enters: 10 35 15
  - program prints: 1 of those numbers are less than 15
  - explanation: 10 is less than the last number entered, which was 15.

Motivating Exercise

- Let's write a program that:
  - Asks the user to enter three numbers.
  - Prints how many of those numbers are less than the last number entered.

- Another example:
  - user enters: 100 35 10
  - program prints: 0 of those numbers are less than 10
  - explanation: none of the numbers entered is less than the last number entered (which is 10).

Motivating Exercise

- Let's write a program that:
  - Asks the user to enter three numbers.
  - Prints how many of those numbers are less than the last number entered.

- Example:
  - user enters: 10 35 105
  - program prints: 2 of those numbers are less than 105
  - explanation: 10 and 35 are less than the last number entered, which was 105.

Motivating Exercise

- Let's modify the previous program so that it:
  - Asks the user to enter four numbers.
  - Prints how many of those numbers are less than the last number entered.

- Example:
  - user enters: 10 5 20 15
  - program prints: 2 of those numbers are less than 15
  - explanation: 10 and 5 are less than the last number entered, which was 15.

Limits of This Approach

- Let's modify the previous program so that it:
  - Asks the user to enter 20 numbers.
  - Prints how many of those numbers are less than the last number entered.

- Or, how about we modify the previous program so that the user can enter as many numbers as they want (they can enter "q" when they are done).

Limits of This Approach

- Let's modify the previous program so that it:
  - Asks the user to enter 20 numbers.
  - Prints how many of those numbers are less than the last number entered.
  - Can be done, but is very tedious.

- Or, how about we modify the previous program so that the user can enter as many numbers as they want (they can enter "q" when they are done).
  - CANNOT BE DONE WITH WHAT WE KNOW

Another Program We Would Like to Write but Cannot

- Write a program that:
  - Asks the user to specify an integer N.
  - Asks the user to enter N names and phone numbers.
  - Then, whenever the user types a name, the computer outputs the corresponding phone number.
- Again, this cannot be done with what we know so far.

Containers

- A *container* is a data type that allows you to store not just one value, but a *set* of values.
- Container is a computer science term, not a Python term.
- Different programming languages have different (and usually multiple) names for containers.
  - A common name is *arrays* (Java, C++).

Containers and Lists in Python

- There are multiple types of containers in Python as well.
- The type we will cover at this point is called a **list**.
- Lists easily allow us to do the tasks we mentioned earlier.

A First Example

- Listing months and their lengths.
- Without containers:
  - 12 variables for month names.

```
month1_name = "January"
month2_name = "February"
month3_name = "March"
month4_name = "April"
month5_name = "May"
month6_name = "June"
…
```

A First Example

- Listing months and their lengths.
- Without containers:
  - 12 variables for month lengths.

```
month1_length = 31
month2_length = 28
month3_length = 31
month4_length = 30
month5_length = 31
month6_length = 30
…
```

A First Example

- Listing months and their lengths.
- Printing out this info requires explicitly mentioning each variable.

```
print month1_name, "has", month1_length, "days"
print month2_name, "has", month2_length, "days"
print month3_name, "has", month3_length, "days"
print month4_name, "has", month4_length, "days"
print month5_name, "has", month5_length, "days"
print month6_name, "has", month6_length, "days"
…
```

A First Example

- Listing months and their lengths.
- With containers (using lists):
  - One variable for month names.

```
month_names = ["January", "February", "March", "April",
  "May", "June", "July", "August", "September",
  "October", "November", "December"]
```

  - One variable for month lengths.

```
month_lengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
  30, 31]
```

A First Example

- Printing out months and lengths is easy:

```
month_names = ["January", "February", "March",
"April", "May", "June", "July", "August",
"September", "October", "November", "December"]

month_lengths = [31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31]

for i in range(0, 12):
    print(month_names[i], "has", month_lengths[i],
"days")
```

Why Is the List Solution Better?

Why Is the List Solution Better?

- Going through all names and lengths requires many lines without containers.
  - Two lines with a list.
- Changing output from "xxx has yy days" to "there are yy days in xxx" requires 12 changes without containers.
  - One change using a list:
    - Replace

```
print month_names[i], "has", month_lengths[i], "days"
```

with

```
print "there are", month_lengths[i], "days in", month_names[i]
```

Lists Simplify Code

- Entering data remains painful.
  - Either way we must enter 12 names and 12 lengths.
  - We can live with this because:

Lists Simplify Code

- Entering data remains painful.
  - Either way we must enter 12 names and 12 lengths.
  - We can live with this because:
    - Data only needs to be entered once.
    - Often data is read from files (later we will learn how).

Lists Simplify Code

- Entering data remains painful.
  - Either way we must enter 12 names and 12 lengths.
  - We can live with this because:
    - Data only needs to be entered once.
    - Often data is read from files (later we will learn how).
- Manipulating data becomes much easier.
  - We can go through data using loops, as opposed to explicitly stating what to do with each value.
- How much easier does it get?
  - Savings proportional to number of values.

Lists Simplify Code

- How much easier does it get using lists?
  - Savings proportional to number of values.
- For 12 values, replacing 12 lines with 1.
- For 20,000 values, replacing 20,000 lines with 1.
- What type of real application would need 20,000 values?

Lists Simplify Code

- How much easier does it get using lists?
  - Savings proportional to number of values.
- For 12 values, replacing 12 lines with 1.
- For 20,000 values, replacing 20,000 lines with 1.
- What type of real application would need 20,000 values?
  - Saving and manipulating data on 20,000 people (students, citizens, customers)

Lists Simplify Code

- How much easier does it get using lists?
  – Savings proportional to number of values.
- For 12 values, replacing 12 lines with 1.
- For 20,000 values, replacing 20,000 lines with 1.

In practice:

YOU CANNOT CODE WITHOUT USING LOOPS AND CONTAINERS.

Accessing Single Elements

my_list = [10, 2, 5, 40, 30, 20, 100, 200]

- This is a list with 8 elements.
  - my_list[0] → 10, this is element 0 of the list.

    IMPORTANT: ELEMENT POSITIONS START WITH 0, NOT WITH 1.
  - my_list[5] → 20, this is element 5 of the list.
  - my_list[-1] → 200, this is the last element
  - my_list[-3] → 20, this is the third-from-last element .
  - my_list[8], my_list[-9] return errors.

Changing Single Elements

```
>>> my_list = [10, 2, 5, 40, 30, 20, 100, 200]
```

```
>>> my_list[0] = 15
```

- Sets value of element 0 to 15.

```
>>> my_list
[15, 2, 5, 40, 30, 20, 100, 200]
```

```
>>> my_list[3] = 23
```

- Sets value of element 0 to 15.

```
>>> my_list
[15, 2, 5, 23, 30, 20, 100, 200]
```

```
>>> my_list[-2] = 70
```

- Sets value of second-to-last element to 70.

```
>>> my_list
[15, 2, 5, 40, 30, 20, 70, 200]
```

Accessing Multiple Elements

```
my_list = [10, 2, 5, 40, 30, 20, 100, 200]
```

```
>>> my_list[2:5]
[5, 40, 30]
```

Returns list of elements from position 2 up to **and not including** position 5.

```
>>> my_list[3:]
[40, 30, 20, 100, 200]
```

Returns list of elements from position 3 until the end of the list.

```
>>> my_list[:4]
[10, 2, 5, 40]
```

Returns list of elements from start and up to **and not including** position 4.

`for` loops with lists

- A **`for`** loop, in the general form, is defined as follows:

```
for variable in set_of_values:
    line 1
    line 2
    …
    line n
```

- **`set_of_values`** can be a list.

Example 1: for loop with a list

```
my_list = [1, 'hello', 2, 'goodbye', [10, 20]]

for item in my_list:
    print(item)
```

Output:

```
1
hello
2
goodbye
[10, 20]
```

Example 2: for loop with a list

```
my_list = [1, 'hello', 2, 'goodbye', [10, 20]]

counter = 0
for item in my_list:
    print('item', counter, ':', item)
    counter = counter + 1
```

Output:

```
item 0 : 1
item 1 : hello
item 2 : 2
item 3 : goodbye
item 4 : [10, 20]
```

The len function

- len(my_list)
  - returns the length (number of elements) of the list.

```
>>> my_list = [10, 2, 5, 40, 30, 20, 100, 200]
>>> len(my_list)
8
```

Functions and Methods

- Functions and methods are almost identical concepts.
- Only difference: syntax of how we write a function call or a method call.
- Function call: len(my_list)
  - Form: function_name(argument1, argument2, …)
  - The expression starts with the function name.
- Method call: my_list.pop()
  - Form: object.method_name(argument1, argument2, …)
  - The expression starts with the object.

List Methods - **append**

- **my_list.append(x):** adds x to the end of my_list.

```
>>> my_list = ["mon", "tue", "wed"]
>>> my_list.append("thu")
>>> my_list
['mon', 'tue', 'wed', 'thu']
```

List Methods - **pop**

- **my_list.pop():** removes **and returns** the last element of my_list.
  - This is an expression, not a statement.

```
>>> my_list = ["mon", "tue", "wed"]
>>> a = my_list.pop()
>>> my_list
['mon', 'tue']
>>> a
'wed'
```

List Functions - **del**

- **del(my_list[position]):** deletes the specified position, and moves forward by one position all elements coming after that position.

```
>>> my_list = [40, 10, 20, 80, 70]
>>> del(my_list[1])
>>> my_list
[40, 20, 80, 70]
>>> my_list[1]
20
```

List Functions - The **+** Operator

- **my_list1 = my_list2 + my_list3:** sets my_list1 to be the <u>**concatenation**</u> of my_list2 and my_list3.

```
>>> my_list2 = [10, 20, 30]
>>> my_list3 = [1, 2, 3]
>>> my_list1 = my_list2 + my_list3
>>> print(my_list1)
[10, 20, 30, 1, 2, 3]
```

List Methods - **insert**

- **my_list.insert(position, x):** inserts x **right before** the specified position.
  - After the insertion, my_list[position] is equal to x.

```
>>> my_list = [40, 10, 20, 80, 70]
>>> my_list.insert(3, 50)
>>> my_list
[40, 10, 20, 50, 80, 70]
>>> my_list[3]
50
```

List Methods - **sort**

- **my_list.sort():** sorts my_list in ascending order.

```
>>> my_list = [40, 10, 20, 80, 70]
>>> my_list.sort()
>>> my_list
[10, 20, 40, 70, 80]
```

List Methods - **sort**

- **my_list.sort():** sorts my_list in ascending order.
- NOTE: this also works with lists of strings.

```
>>> my_list = ["Sunday", "Monday", "Tuesday"]
>>> my_list.sort()
>>> my_list
['Monday', 'Sunday', 'Tuesday']
```

Not quote alphabetical order:

capital letters come before lower case letters.

List Methods - **sort**

- **my_list.sort():** sorts my_list in ascending order.
- NOTE: this also works with lists of strings.

```
>>> my_list = ['a', 'b', 'c', 'A', 'B', 'C', 'ant', 'bee', 'car']
>>> my_list.sort()
>>> my_list
['A', 'B', 'C', 'a', 'ant', 'b', 'bee', 'c', 'car']
```

Not quote alphabetical order:

capital letters come before lower case letters.

List Methods - **reverse**

- **my_list.reverse():** reverses the order of my_list.

```
>>> my_list = [40, 10, 20, 80, 70]
>>> my_list.reverse()
>>> my_list
[70, 80, 20, 10, 40]
```

The **in** operator

- We have used **in** before, to check if a letter appears in a string.

```
>>> 'a' in 'bravo'
True
```

- We can also use **in** to check if a value of any type is included in a list.

```
>>> 15 in [234, 15, 32]
True
>>> 'day' in ['have', 'a', 'good', 'day']
True
>>> 'day' in ['have', 'a', 'good', 'evening']
False
```

Example 1: Using the **in** operator

```
# find elements that are repeated twice in a list

my_list = [15, 10, 30, 20, 40, 30, 15, 40, 15]

for i in range(0, len(my_list)):
    item = my_list[i]
    if item in my_list[i+1:]:
        print(item, "appears more than once in the list.")
```

Example 1: Using the **in** operator

```
# find elements that are repeated twice in a list

my_list = [15, 10, 30, 20, 40, 30, 15, 40, 15]

for i in range(0, len(my_list)):
    item = my_list[i]
    if item in my_list[i+1:]:
        print(item, "appears more than once in the list.")
```

Issues:

- Repeats information about element 15.
- How can it also print that a value only appears once?

Example 2: Using the **in** operator

```
# Find elements that are repeated twice or more in a list.
# Also identify elements that are repeated only once in the list.

my_list = [15, 10, 30, 20, 40, 30, 15, 40, 15]
duplicates = []

for i in range(0, len(my_list)):
    item = my_list[i]
    if (item in duplicates):
        continue
    if item in my_list[i+1:]:
        duplicates.append(item)
        print(item, "appears more than once in the list.")
    else:
        print(item, "does not appear more than once in the list.")
```

Converting a String to a List

- To convert a string to a list, use the **list** function.

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a List to a String

- To convert a list to a string, **do not** use the **str** function.

```
>>> a = list('hello')
>>> a
['h', 'e', 'l', 'l', 'o']
>>> b = str(a)
>>> b
"['h', 'e', 'l', 'l', 'o']"
```

Converting a List to a String

- To convert a list to a string, use a **for** loop.

```
a = ['h', 'e', 'l', 'l', 'o']
b = ""

for letter in a:
    b = b+letter

>>> b
'hello'
```

Example: Strings to Lists and Back

```
# sort all the letters in a string.

text = input('Enter some text: ')
text_list = list(text)
text_list.sort()

new_text = ""
for letter in text_list:
    new_text = new_text + letter

print("The sorted text is:", new_text)
```

OUTPUT:

```
Enter some text: hello world
The sorted text is:  dehllloorw
```

Shallow Copies

```
>>> list1 = [40, 10, 20, 80, 70]
>>> list2 = list1
>>> list1 is list2
True
>>> list1[2] = 50
>>> list2
[40, 10, 50, 80, 70]
>>> list2.pop()
70
>>> list1
[40, 10, 50, 80]
```

This line makes list2 a **shallow copy** of list1. After this line, list2 and list1 refer to the same list in the computer's memory.

Thus, whenever that list changes, both list1 and list2 are affected.

list1 is list2 allows the programmer to check if two variables refer to the same actual list.

Breaking Links Caused by Shallow Copies

```
>>> list1 = [40, 10, 20, 80, 70]
>>> list2 = list1
>>> list1[2] = 50
>>> list2
[40, 10, 50, 80, 70]
>>> list1 = [1, 2, 3]
>>> list2
[40, 10, 50, 80, 70]
```

This line makes list2 a **shallow copy** of list1. After this line, list2 and list1 refer to the same list in the computer's memory.

Thus, whenever that list changes, both list1 and list2 are affected.

After this line, list1 and list2 refer to two different lists, and they are not connected anymore.

Level-1 Deep Copies

```
>>> list1 = [40, 10, 20, 80, 70]
>>> list2 = list1[:]
>>> list1 is list2
False
>>> list1.sort()
>>> list1
[10, 20, 40, 70, 80]
>>> list2
[40, 10, 20, 80, 70]
```

This line makes list2 a **level-1 deep copy** of list1. After this line, list2 and list1 refer to **different lists** in the computer's memory.

Thus, whenever we replace an element, insert an element, or delete an element from one list, that does **NOT** affect the other list.

Level-1 Deep Copies

```
>>> list1 = [[40, 10], 20, [80, 70]]
>>> list2 = list1[:]
>>> list2[2] = 1000
>>> list1
[[40, 10], 20, [80, 70]]
>>> list2
[[40, 10], 20, 1000]

>>> list2[0][1] = 77
>>> list1
[[40, 77], 20, [80, 70]]
```

list1 contains a list, an integer, and another list.

list2 is a level-1 deep copy of list1.

Changing list2[2] does not affect list1 (we are replacing an entire element).

However, changing list2[2][1] changes list1 as well (we are replacing **contents** of an element).

Deep Copies

```
>>> import copy
>>> list1 = [[40, 10], 20, [80, 70]]
>>> list2 = copy.deepcopy(list1)
>>> list2
[[40, 10], 20, 1000]

>>> list2[0][1] = 77
>>> list1
[[40, 10], 20, [80, 70]]
>>> list2
[[40, 77], 20, [80, 70]]
```

import copy must be executed before we call the copy.deepcopy method.

list1 contains a list, an integer, and another list.

list2 is a deep copy of list1. No change of list2 can possibly affect list1 anymore.

Note that list1 is now the same as before.

The **id** Keyword

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = a[:]
>>> id(a)
41460872L
>>> id(b)
41460872L
>>> id(c)
41417544L
>>> a is b
True
>>> a is c
False
```

- **`id(my_variable)`** returns a memory address, tells us where the computer stores information about **`my_variable`**.
- When variable2 is a **<u>shallow copy</u>** of variable1 (like **`a`** and **`b`** on the left), then both variables refer to the same underlying object, and their IDs are the same.
- When variable2 is **<u>not</u>** a shallow copy of variable1 (like **`a`** and **`c`** on the left), then the two variables refer to different underlying objects, with different IDs.

The **is** Keyword

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = a[:]
>>> id(a)
41460872L
>>> id(b)
41460872L
>>> id(c)
41417544L
>>> a is b
True
>>> a is c
False
```

- **`variable1 is variable2`** returns a boolean, that tells us whether both variables refer to the same underlying object. If this boolean is true, then it means that:
  - id(variable1) == id(variable2)
  - variable1 is a shallow copy of variable2 (and vice versa)
- Checking whether two variables are shallow copies of each other can be done using either **id** or **is**.
  - The **is** keyword is more readable.

Shallow vs. Deep Copies

- Shallow copy: list1 = list2
- Level-1 deep copy: list1 = list2[:]
  - In a level-1 deep copy, list2[i] is a shallow copy of list1[i].
- Deep copy: list2 = copy.deepcopy(list1)
  - Line "import copy" must be executed beforehand.
  - With copy.deepcopy, list2 and list1 do not share any memory, changing one of them does not affect the other one.
- **BE AWARE OF THESE ISSUES, THEY MAY CAUSE HARD-TO-FIND BUGS**

Lists vs. Tuples

```
>>> a = [1, 2, 3]
>>> a[0] = 100
>>> a
[100, 2, 3]
>>> b = (1, 2, 3)
>>> b[0] = 100

error message…
```

- Tuples are basically lists, …
- with one important difference: you cannot change the contents of a tuple.

Lists vs. Tuples

```
>>> a = [1, 2, 3]
>>> a[0] = 100
>>> a
[100, 2, 3]
>>> b = (1, 2, 3)
>>> b[0]
1
>>> b[1:3]
(2, 3)
>>> len(b)
3
>>> print b
(1, 2, 3)
```

- Any operation that you can do on lists, and that does **NOT** change contents, you can do on tuples.
- Examples:
  – Indexing, e.g., b[0]
  – Slicing, e.g., b[1:3]
  – Taking the length, e.g., as len(b)
  – Printing, e.g., print(b)
- Operations that change contents of lists, produce errors on tuples.
- Examples: list methods pop, insert, reverse, sort, append

Lists vs. Tuples

```
>>> a = [1, 2, 3]
>>> a[0] = 100
>>> a
[100, 2, 3]
>>> b = (1, 2, 3)
>>> b[0]
1
>>> b[1:3]
(2, 3)
>>> len(b)
3
>>> print b
(1, 2, 3)
```

- Creating a tuple can be done easily, just use parentheses around the elements, instead of brackets.
  - See red lines on the left.
- When you print a tuple, you also see parentheses instead of brackets.

Lists vs. Tuples

```
>>> a = [1, 2, 3]
>>> b = tuple(a)
>>> b
(1, 2, 3)
```

```
>>> b = (1, 2, 3)
>>> a = list(b)
>>> a
[1, 2, 3]
```

- You can easily copy lists into tuples, and tuples into lists, as shown on the left.

Lists vs. Tuples

```
>>> a = [1, 2, 3]
>>> b = tuple(a)
>>> b
(1, 2, 3)
```

```
>>> type(a)
<class 'list'>
>>> type(b)
<class 'tuple'>
```

- Lists are of type 'list', and tuples are of type 'tuple'

Protection Against Shallow Copies

```
>>> tuple1 = ([40, 10], 20, [80,
    70])
>>> tuple2 = tuple1
>>> tuple1[2] = 1000

<error message>

>>> tuple2[0][1] = 77
>>> tuple1
([40, 77], 20, [80, 70])
```

- tuple1 contains a list, an integer, and a second list.
- tuple2 is a shallow copy of tuple1.
- Trying to replace the value at position 2 does not work (tuples cannot be modified).
- However, modifying tuple2[0][1] **also modifies tuple1**.
- This problem is caused by having a list as element of the tuple.

Protection Against Shallow Copies

```
>>> tuple1 = ((40, 10), 20, (80,
    70))
>>> tuple2 = tuple1
>>> tuple1[2] = 1000

<error message>

>>> tuple2[0][1] = 77

<error message>
```

- tuple1 contains a tuple, an integer, and a second tuple.
- tuple2 is a shallow copy of tuple1.
- Trying to replace the value at position 2 does not work (tuples cannot be modified).
- Modifying tuple2[0][1] **also doesn't work.**
- Thus, we avoid the problem of inadvertently modifying multiple variables.

Do We Care About Tuples?

- Be aware that they exist.
  - Know what they are if you see them in other people's code.
  - Use them if you find it beneficial.
  - Use them if you want to ensure that modifying one variable will NOT affect any other variable.
- We will see more uses of tuples later.
- Now you know these types:
  - int, float, str, bool, list, tuple.