

Structures: Lists: Complex

CPSC 217: Introduction to Computer Science for Multidisciplinary Studies I
Jul 2021 - CBE

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Wednesday, June 2, 2021

Copyright © 2021



**UNIVERSITY OF
CALGARY**

Slicing

Slicing a List

- You can produce copies and sub-lists of a list using the range of indices (:). The following produces a copy of *list* from *a* to *b-1*:

`list[a:b]`

a is the starting index of the slice. The default is 0.

b is the ending index of the slice. The default is `len(list)`.
b itself is excluded from the slice.

`names[start:end]` → to produce a sub-list

← names [0]	Marc
names [1]	Ken
names [2]	Jim
names [3]	Tony

- `names[:]` returns a copy of names
- `names[0:2]` returns the first two elements in names
- `names[-2:]` returns the last two elements in names

Slicing a List

- You can produce a sub-list of a list that consists of certain elements of a list using *step* in the range of indices

`list[a:b:step]`

a and *b* are defined in previous slide.

step is the amount by which *a* increments. The default is 1. *step* can be positive (increment) or negative (decrement).

`names[start:end:step]`
→ to produce a sub-list

<code>names [0]</code>	Marc
<code>names [1]</code>	Ken
<code>names [2]</code>	Jim
<code>names [3]</code>	Tony

- `names [0:len(names):1]` returns a copy of list
- `names [::]` returns a copy of list
- `names [::-1]` returns a reversed list
- `names [-2::]` returns last two elements
- `names [::2]` returns a list with every other element in names is skipped

Copy List

Same List

- A list variable is a reference to the list.
names<address of the first byte of the list in memory>

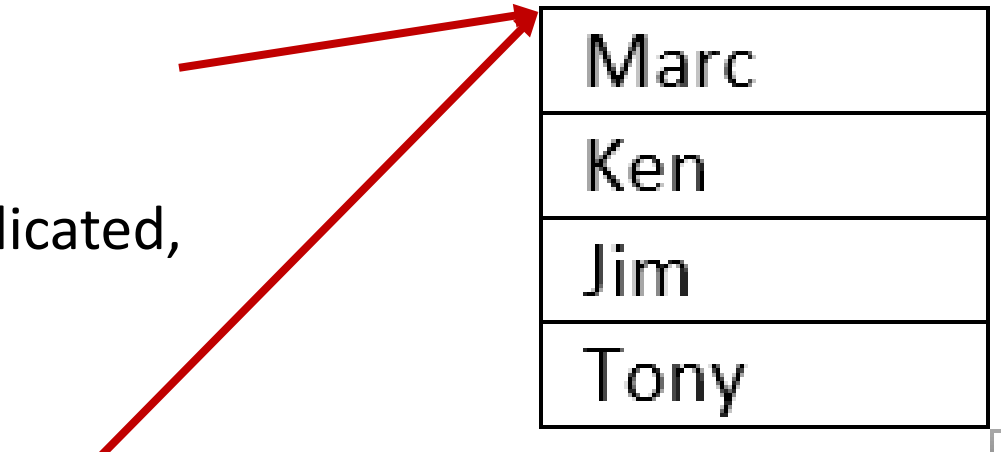
- When duplicating a list variable, the address is duplicated, not the actual list.

```
>new_names = names
```

If you change *names* you change *new_names*.
Also true the other way.

```
>new_names[0] = "Jonathan"
```

```
>print(names[0]) → 'Jonathan'
```



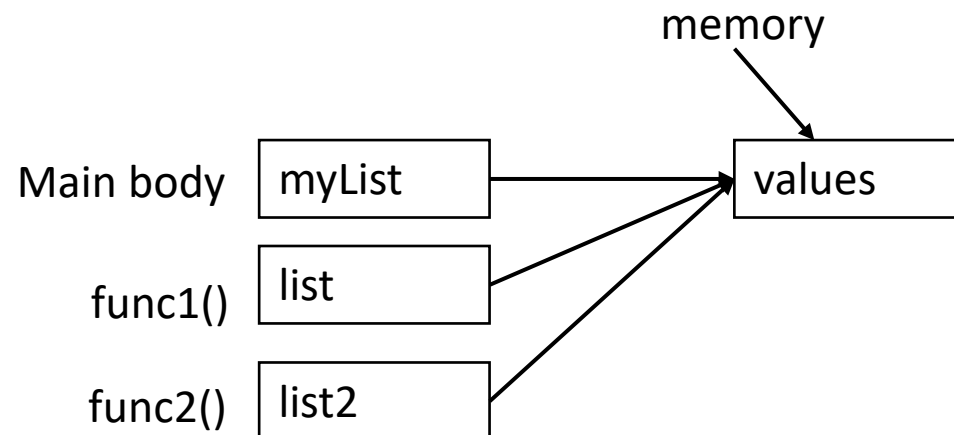
Passing List to Functions

- When passing mutable types, such as lists, to functions, remember that any changes to the list, will be reflected in the original list in the caller's scope.

```
def func2(list2):  
    ...
```

```
def func1(list):  
    list2 = list  
    func2(list2)
```

```
myList = [...]  
func1(myList) → Memory address is passed
```



Duplicate a List

- Many ways to create a copy of a list (also known as **shallow-copy**):

- Using **slice**:

```
new_names = names[:]
```

- Using the **repetition operator**:

```
new_names = names*1
```

- Using **extend()**:

```
new_names = []  
new_names.extend(names)
```

- Using a **loop** to duplicate the list element by element:

```
new_names = []  
for i in range (0, len(names), 1):  
    new_names.append(names[i])
```


Operations

List Operations and Methods

Operation	Example	Description
Indexing	<code>names[0]</code>	Access a list element
Membership	<code>if 'Alice' in names</code> <code>if 'Alice' not in names</code>	Query whether or not an item is in the list
Length	<code>len(names)</code>	Get the number of items in a list
Append	<code>names.append('Alice')</code>	Add an item to the end of the list
Insert	<code>names.insert(0, 'Alice')</code>	Insert an item at certain position
Sort	<code>names.sort()</code>	Sort the list
Reverse	<code>names.reverse()</code>	Reverse the items in the list
Count	<code>names.count('Alice')</code>	Count the number of occurrence of an item

Search/Remove List

Searching For Elements

- Use `in` to check if an item is present in a list

```
data = [1, 2, 3, 4, 5]
```

```
2 in data evaluates to True
```

```
8 in data evaluates to False
```

- Use `index` to determine where it is in the list

```
data = [11, 12, 13, 14]
```

```
data.index(12) evaluates to 1
```

```
data.index(8) results in a ValueError
```

Removing Elements

- How can we remove an item from a list?
 - Use the **remove** method
 - Removes the first occurrence of the item
 - Subsequent identical items remain in the list
 - Item must exist or a ValueError will occur

```
x = [1, 2, 1, 3, 4, 2, 1]
```

```
x.remove(1)
```

```
print(x)
```

Removing Elements

- What if we want to remove all occurrences of an item from a list?
- Use a while loop:

```
while x in myList:  
    myList.remove(x)
```

Removing Elements

- What if we know the index of the item we want to remove?
 - Use `pop(index)`
 - With no parameters: Removes last item
 - With one parameter: Removes item at the index specified
 - Returns the item that is removed

```
myList = [1, 2, 3, 4]
```

```
myList.pop()  
print(myList)
```

[1, 2, 3]

```
myList.pop(0)  
print(myList)
```

[2, 3]


```
myList.pop(myList.index(2))  
print(myList)
```

[3]

Sorting a List

Sorting

- Sorting is the process of ordering elements of a list in ascending or descending order.

[4, 2, 1, 3, 0]  **Unordered list**

[0, 1, 2, 3, 4]  **Ordered list in ascending order**

[4, 3, 2, 1, 0]  **Ordered list in descending order**

- How do we sort the list?

Sorting

- Sorting is an important task
 - Needed when working with large data sets
 - Frequently occurs as part of other algorithms
- Sorting has been studied extensively
 - Many algorithms, some of which are quite complex

Sorting - Bubble Sort

General idea (ascending order)

- go through list from beginning to end
 - compare adjacent elements
 - swap if previous element is larger than current element
- repeat until no swaps are performed

<https://www.youtube.com/watch?v=nmhjrl-aW5o>

- You can download a solution: *1_Bubble.py*

Sorting - Selection Sort

General idea (ascending order): The list is initially considered entirely unordered.

- Select the smallest element in the unordered portion of list
- Remove the element from unordered portion of the list and place it at the end of the ordered portion of the list.
- Repeat until no elements remain in the unordered portion of the list.

<https://www.youtube.com/watch?v=xWBP4IzkoyM>

Lets implement this!

You can download another solution: *2_Selection.py*

Sorting in Python

- Python makes sorting a list easy
 - Use the sorted function
 - Takes one parameter which is an unsorted list
 - Returns a new list sorted into increasing order
 - Use the *sort(order)* method
 - Order is a Boolean parameter. Default is True for ascending order. False sorts in descending order.
 - Invoked on a list using dot notation
 - Modifies the list

```
list=[0,1,6,2,10]
```

```
list.sort()
```

```
print(list)
```



```
[0, 1, 2, 6, 10]
```

List Example

Practice Example

- Compute the median of a list of values entered by the user
 - User will enter an unknown number of values
 - A blank line will be used to indicate that no additional values will be entered
 - If the list has an odd number of elements
 - Median is the middle value
 - If the list has an even number of elements
 - Median is average of the two middle values

Practice Example Design

- read values from user and store in a list (using append)
- sort list (put numbers in ascending order)
- if list length is odd, display middle value (`index = len(list) / 2`)
- if list length is even, display the average of two middle values (`index len(list)/2` and `len(list)/2 - 1`)

Lets code this!

Tracing

Trace The Code 1:

```
def f1(list1) :  
    list2 = list1  
    for index in range(len(list1)):  
        list2[index] = list1[index]+1  
    print(list1)  
    print(list2)
```



```
[2, 3, 4]  
[2, 3, 4]
```

```
f1([1, 2, 3])
```

Trace The Code 2:

```
def f2(list1) :  
    list2 = list1[:]  
    for index in range(len(list1)):  
        list2[index] = list1[index]+1  
    print(list1)  
    print(list2)
```



```
[1, 2, 3]  
[2, 3, 4]
```

```
f2([1, 2, 3])
```

Trace The Code 3:

```
def f3(list1) :  
    list2 = list1*2  
    for index in range(len(list2)) :  
        list2[index] += 1  
    print(list1)  
    print(list2)
```

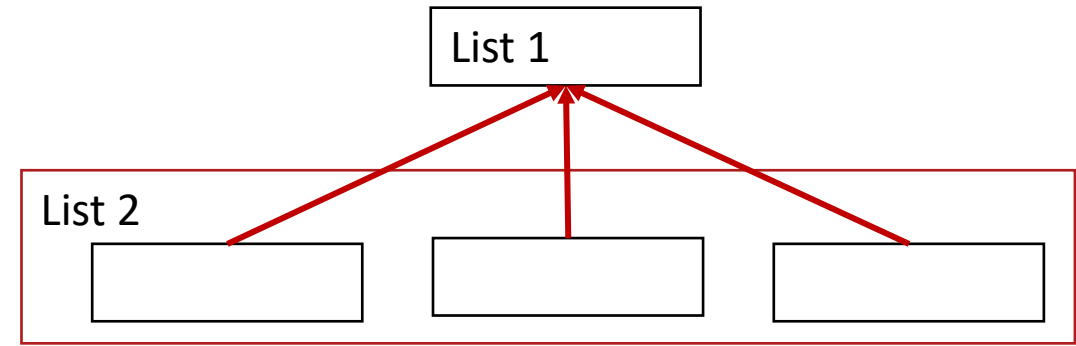


```
[1, 2, 3]  
[2, 3, 4, 2, 3, 4]
```

```
f3([1, 2, 3])
```

Trace The Code 4:

```
def f4(list1) :  
    list2 = [list1]*3  
    for index in range(len(list2)) :  
        innerList = list2[index]  
        for innerIndex in range(len(innerList)) :  
            innerList[innerIndex] += 1  
  
    print(list1)  
    print(list2)
```



f4([1, 2, 3])

→ [4, 5, 6]
[[4, 5, 6], [4, 5, 6], [4, 5, 6]]

Trace The Code 5:

```
def f5(list1) :  
    list2 = [list1]*2  
    for index in range(len(list2)) :  
        innerList = list2[index]  
        innerList = innerList[:]  
        for innerIndex in range(len(innerList)) :  
            innerList[innerIndex] += 1  
        list2[index] = innerList  
    print(list1)  
    print(list2)
```

f5([1, 2, 3])



```
[1, 2, 3]  
[[2, 3, 4], [2, 3, 4]]
```

Tuples?

What is a Tuple?

- A collection of values
 - Values
 - May all have the same type, or
 - May have different types
 - Each item is referred to as an element
 - Each element has an index (ORDERED)
 - Unique integer identifying its position in the tuple
 - A tuple is one type of data structure
 - A mechanism for organizing related data

Main thing to remember!

- Similar to lists, but
 - length cannot be changed
 - **Items cannot be modified (immutable)**
 - () empty tuple, (3,) length one tuple

```
aTuple = (1, "ICT", 3.14)
```

Tuples

- Like a list, a tuple is a sequence type that its elements can be of any other type
- Support many of the same operations as lists
- Unlike lists, tuples are used to store data that should not be changed.
- Format
 - `<tuple name> = (<value 1>, <value 2>, ... , <value n>)`
- Example

```
student = ('Marc', 123456789, 9.5)
print (student)
print (student[1])
#student[2] = 10
```



```
('Marc', 123456789, 9.5)
123456789
```

TypeError: 'tuple' object does not support item assignment

Tuple

- Format:

<list name> = (<value 1>, <value 2>, ... , <value n>)

Examples:

nums = (10.0, 9.0, 8.5, 5.0, 7.5)

letters = ('a', 'b', 'c', 'd', 'e', 'f', 'g')

names = ('Marc', 'Jim', 'Ken')

mixed = (1.0, 1, "this", True)

By defining the tuple memory is allocated for it

names = (x,) → Singleton tuple of one time

Regular brackets () without comma are interpreted as empty tuple

Tuple operations

Operations	Example	Description
Indexing	<code>name[i]</code>	Access item by index
Slicing	<code>name[start:end:step]</code>	Get sub-tuple
Concatenation	<code>names1+names2</code>	Join two tuples into larger tuple
Update tuple	Immutable	Use slicing to get sub-tuple Use concatenation to get larger tuple
Length	<code>len(name)</code>	Get length of tuple
Repetition	<code>name*x</code>	Multiply to get tuple with int x copies of its contents in order
Membership	<code>n in name</code>	Boolean if item is in tuple at base level
Loop	<code>for x in name : print(x)</code>	Iterate through each item in tuple
Index	<code>name.index("Carl")</code>	Returns first index of item "Carl" in tuple name

Tuple

- In effect when we return multiple values from a function we are using tuples

- The same

```
def foo():
```

```
    return x,y
```

```
def foo():
```

```
    return (x,y)
```

- A number of common languages don't have tuples a structure like tuples, and are limited to returning a single pointer of data.

Packing/Unpacking

- You can define a tuple without brackets. Python will interpret variables/expressions separated by commas.

```
x = 1,2
```

```
print(x) -> (1,2)
```

```
print(x[0]) -> 1
```

```
print(x[1]) -> 2
```

```
a,b = x
```

```
print(a) -> 1
```

```
print(b) -> 2
```

The process seen here is generally called packing, and unpacking

Onward to ... dictionaries.

Jonathan Hudson
jwhudson@ucalgary.ca
<https://pages.cpsc.ucalgary.ca/~hudsonj/>



UNIVERSITY OF
CALGARY