Structures: Lists: Basics

CPSC 217: Introduction to Computer Science for Multidisciplinary Studies I
Fall 2020

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

Tuesday, September 8, 2020

UNIVERSITY OF CALGARY

What is a List?

- 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
    - Unique integer identifying its position in the list
  - A list is one type of data structure
    - A mechanism for organizing related data

Creating a List

- Format:

```
<list name> = [<value 1>, ..., <value n>]
```

- Examples:

```
names = []
```
→ defines an empty list

```
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 list memory is allocated for it

* Works on Lists?

Repetition Operator (*)

- Just like strings, you can use asterisk to repeat a list

```
>list = [0]*5
```


Produces a list of size 5 with all elements = 0

```
>newList = list*5
```


Produces a new list of size 25 with all elements = 0

Indices

Accessing Elements

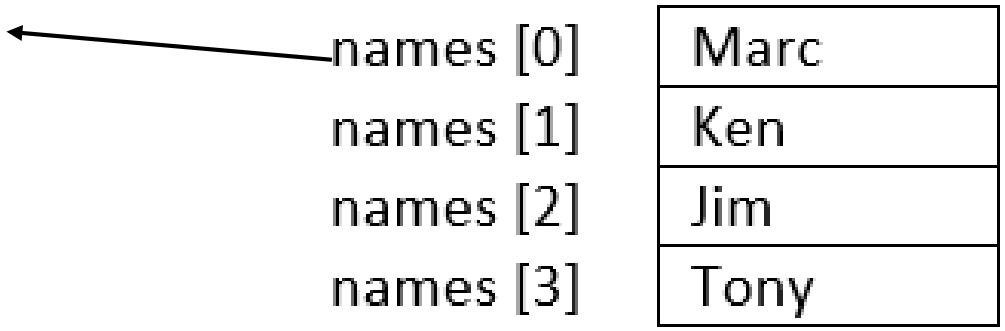
- Each list element has two unique indices, a positive one and a negative one:
  - Positive indices range from 0 to the length of the list minus one (*len(list)-1*)
  - Negative indices range from *-len(list) to –1*

0	1	2	3	4	5	6	7
A	B	C	D	E	F	G	H
-8	-7	-6	-5	-4	-3	-2	-1

Accessing Elements - Accessing a Single Element

- To access one element, use the name of the list, followed by the index of that element in square brackets
  - Use this one element just like any other variable

names[**index**] → returns the value stored at location **index**.



- names refers to the whole list
- len(names)→4
- names.index('Ken') → 1

Loop on List

Accessing Elements - Iterating Over List Items

- A for loop can be used iterates over the list values:

```
stuff = [1, "ICT", 3.14]
for item in stuff:
    print(item)
```

Accessing Elements - Iterating Over List Indices

- Sometimes we need a loop where the control variable varies over the indices rather than the values

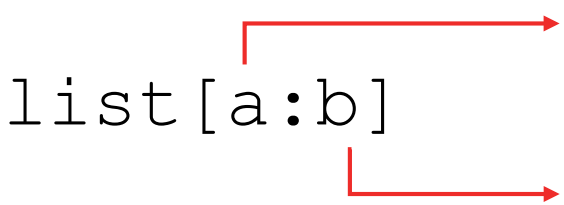
```
stuff = [1, "ICT", 3.14]
for i in range(0, len(stuff))
  print(stuff[i])
```

List length changes as elements are added/removed.
So, use *len()* function to determine the length of list.

Slicing

Slicing a List

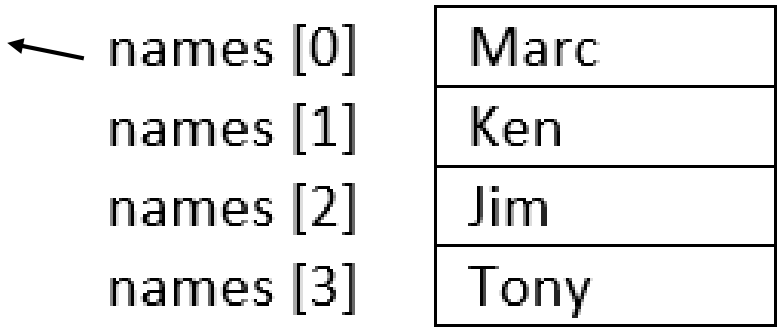
- You can produce copies and sub-lists of a list using the range of indices (:). The following produces a copy of *list* from *a* to *b-1:*



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[:] 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 be positive (increment) or negative (decrement).

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

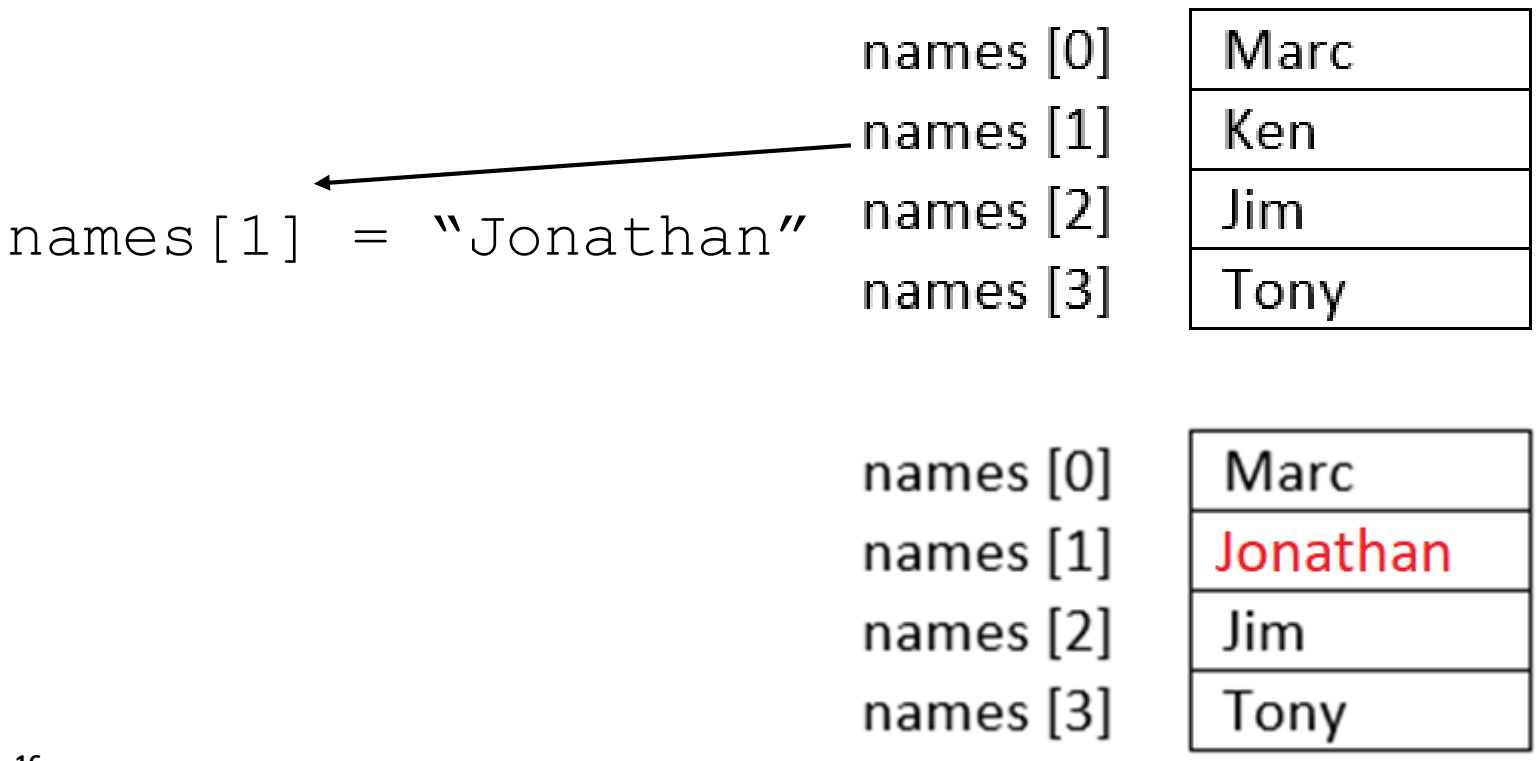
- `names[0:len(names):1]` returns a copy of list
- `names[::]` returns a copy of list
- `names[::-1]` returns a reversed list
- `names[-2::]` returns last two elements
- `names[::2]` returns a list with every other element in names is skipped

Modifying List

Modifying Elements

- Lists are mutable, so their elements can be changed as follows:

```
names[index] = new_data
```



Adding Elements

- Lists are mutable, so we can add more elements to them.
- There are three ways to add elements to a list
  - `append(x)`: adds a single element to the end of the list

    ```
    names.append('Daniel')
    ```

  - `insert(i, x)`: inserts a single element into a list at index i, shifts elements up

    ```
    names.insert(3, 'Chris')
    ```

  - `extend(L)`: extends the list by appending the given second list to it

    ```
    names.extend(['Eric', 'Frank'])
    ```

Adding Elements

- Example:

```
names = []
name = input("Enter a name:")
names.append(name)
names_str = input("Enter names separated by comma:")
names.extend(names_str.strip().split(","))
print(names)
```

Printing List

Printing List

- There are many ways to print the content of a list.
- Two common ways are:
  - using *print()*

```
print('names = %s` , (names))
```

- Using a loop → allows us to print the list in a customized format:

```
for i in range(0, len(names), 1):
    print("names[%d] = %s" % (i, names[i]))
```

Copy List

Same List

- A list variable is a reference to the list.

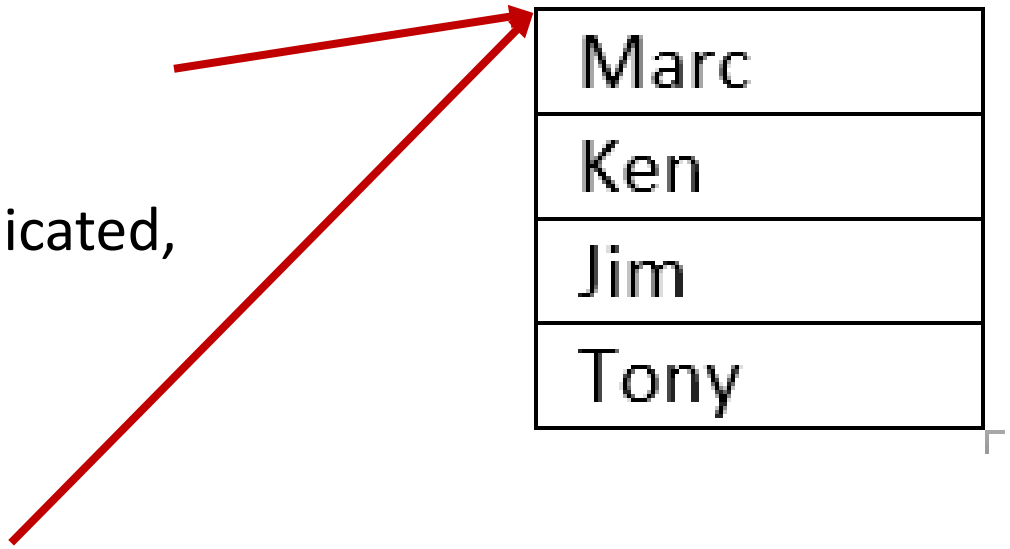
  names<address of the first byte of the list in memory>

- 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

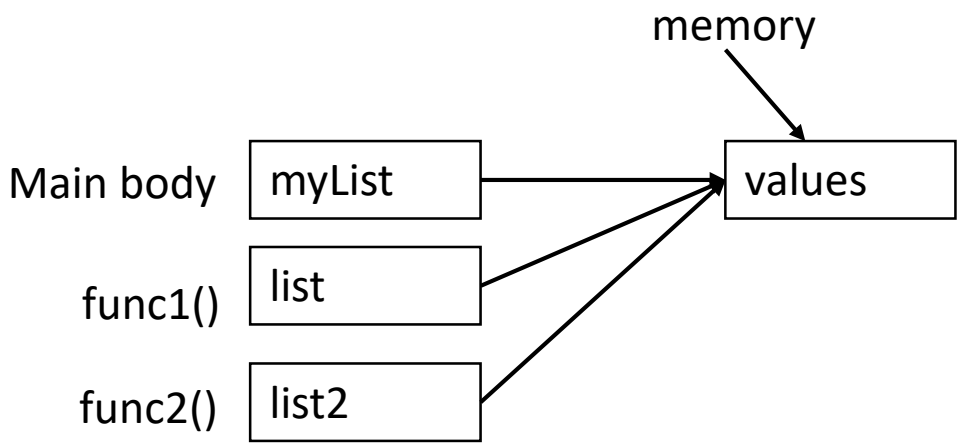
- When passing mutable types, such as lists, to functions, remember that any changes to the list, will be reflected in the original list in the caller's scope.

```
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])
```

Tuples?

Duplicate a List

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

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

Onward to …
more complicated lists.

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

UNIVERSITY OF CALGARY