ECOR 1041
Computation and Programming

Lists

Carleton University

References

- *Practical Programming*, 3rd ed.
  - Chapter 8, pp. 129-137:
    - Chapter introduction and the first four sections
      - Up to the end of “Operations on Lists”
  - Chapter 9, pp. 149-151, 152-154, 154-156
    - Chapter introduction and these sections:
      - Processing Items in a List
      - Looping Over a Range of Numbers
      - “Processing Lists Using Indices” up to and excluding *Processing Parallel Lists Using Indices*

Lecture Objectives

- Introduce Python's `list` type

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - List, index, type `list`
  - Concatenation, replication and deletion operators
  - Built-in `len`, `min` and `max` functions

Learning Outcomes

- Be able to evaluate expressions containing `list` objects, the `*`, `+` and `del` operators, indexing (*`list_name`*`[`*`index`*`]`) , and calls to the `len`, `min` and `max` functions.
- Be able to design and implement functions that process lists

Lists

- In computer science, a list is a finite sequence of ordered values; e.g., 7, 4, 9, 12, 7, 2 is a list of six integers
- The same value may occur more than once; e.g., two 7's
- The values are *ordered*: the 9 occurs after the 4 and before the 12
- Many programming languages (including Python) provide a built-in data type that represents lists

Python's `list` Type

- A list is created by an expression of the form `[`*`expression1`*`, `*`expression2`*`, ..., `*`expressionN`*`]`
- Example: the expression `[7, 4, 9, 12, 7, 2]` creates an object of type `list` that represents the integer sequence 7, 4, 9, 12, 7, 2

```
>>> type([7, 4, 9, 12, 7, 2])
<class 'list'>
```

- `list` is a built-in type, like `int`, `float`, and `str`

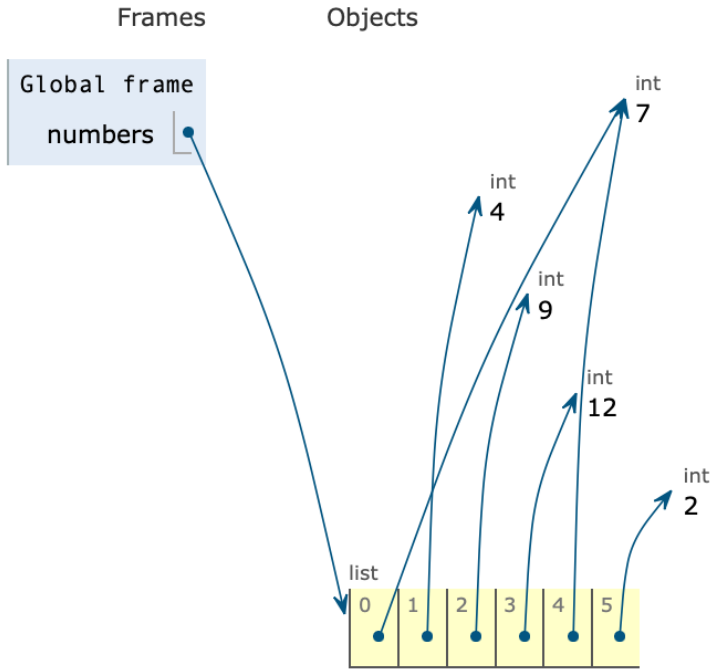
Python's `list` Type

- Lists can be assigned to variables:

```
>>> numbers = [7, 4, 9, 12, 7, 2]
>>> numbers
 [7, 4, 9, 12, 7, 2]
```

Python's `list` Type

- A list stores references to objects (memory addresses of objects)
  - `numbers` refers to a `list` object that stores six references to objects of type `int`



Constructing Lists

- Items in a list are often called *elements*
- When creating a new list, the elements are expressions
  - They do not have to be literal values

```
>>> x = 5
>>> y = 2
>>> a_list = [x, 2 * x, x * 2 + y * 2 + 1]
>>> a_list
[5, 10, 15]
```

Empty Lists

- An *empty list* is a list that has no elements
- We can create an empty list by using the expression `[]`

```
>>> empty = []
>>> empty
[]
```

Accessing List Elements

- Each element can be accessed by an integer *index* that indicates its position in the list
  - The first element is at index 0, the second element is at index 1, etc.
  - When a list has *n* elements, the index of the last element is *n*-1

Accessing List Elements

- Syntax: *list_name*[*index*]

```
>>> numbers = [7, 4, 9, 12, 7, 2]
>>> numbers[0]
7
>>> numbers[1]
4
>>> numbers[5]
2
>>> numbers[6]
IndexError: list index out of range
```

Accessing List Elements

- Python supports negative indices
- Index -1 accesses the last element, index -2 accesses the second-last element, etc.

```
>>> numbers = [7, 4, 9, 12, 7, 2]
>>> numbers[-1]
2
>>> numbers[-2]
7
>>> numbers[-6]
7
```

Accessing List Elements: Continued

- Python supports negative indices
- Index -1 accesses the last element, index -2 accesses the second-last element, etc.

```
>>> numbers = [7, 4, 9, 12, 7, 2]
>>> numbers[-7]
IndexError: list index out of range
```

Accessing List Elements

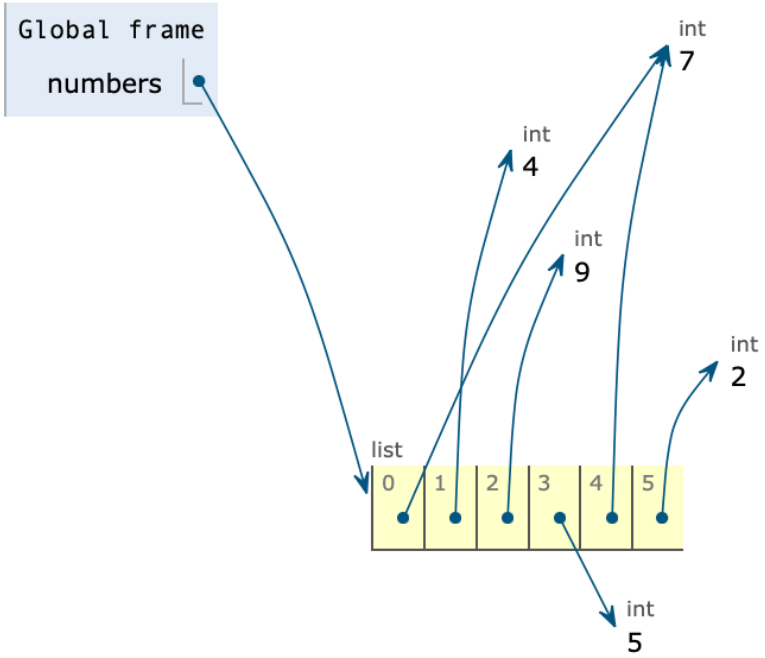
- Indices are expressions that evaluate to integers, so they are not limited to `int` literals

```
>>> i = 2
>>> numbers[i]
9
>>> numbers[i - 1]
4
>>> numbers[i + 3]
2
```

Lists are Mutable

- An assignment operation can be used to replace the list element at a specified location
- Example: replace the 4th element with 5

```
>>> numbers[3] = 5
>>> numbers
[7, 4, 9, 5, 7, 2]
```



List Operations - Length

- The built-in `len` function returns the number of elements in a list

```
>>> len(numbers)
6
>>> len(empty)
0
```

List Operations - Maximum & Minimum

- The built-in `max` and `min` functions returns the maximum and minimum values in a list

```
>>> max(numbers)
9
>>> min(numbers)
2
```

List Operations - Concatenation

- The + operator is the *concatenation* operator when both operands are lists
- + creates a new list that is the concatenation of the left-hand and right-hand lists

```
>>> a_list = [7, 10, 9]
>>> a_list + [4, 9, 3]
[7, 10, 9, 4, 9, 3]
```

List Operations - Replication

- The `*` operator is the list-replication operator when one operand is an integer *n* and the other operand is a list
- `*` creates a new list containing *n* copies of the list operand

```
>>> a_list
[7, 9, 2]
>>> a_list * 2
[7, 9, 2, 7, 9, 2]
>>> 2 * a_list
[7, 9, 2, 7, 9, 2]
```

List Operations - Element Deletion

- The `del` (deletion) operator removes the element at a specified index and shifts all subsequent elements to the left

```
>>> numbers = [7, 4, 9, 12, 7, 2]
>>> del numbers[2] # Removes 9 at index 2
>>> numbers
[7, 4, 12, 7, 2]
```

Iterating Over Lists

- A list is a sequence of elements, so we can use a `for` loop to iterate over every element in a list, starting with the first one:

```
for variable in list:
    block
```

- At the beginning of each iteration, *`variable`* is assigned the next element in the list

Iteration Example: Count Occurrences

- Count the number of elements in a list that are equal to a specified value

```
count = 0
for elem in a_list:
    if elem == value:
        count += 1
```

Lists as Function Arguments

- A function that takes a `list` and returns `True` if a target value is in the list

```
def contains(a_list: list, target) -> bool:
```

  - No type annotation on `target` means it can be any type

```
>>> contains([1, 5, 7, 3, 9, 2, 9, 9], 9)
True
>>> contains([1, 5, 7, 3, 9, 2, 9, 9], 6)
False
```

Lists as Function Arguments

```
def contains(a_list: list, target) -> bool:
    for elem in a_list:
        if elem == target:
            return True
    return False
```

The `in` Operator

- Searching a list for a specific object is such a common operation that Python provides the `in` operator, which does the same thing as function `contains`

```
>>> x = 9
>>> x in [1, 5, 7, 3, 9, 2, 9, 9]
True

>>> x = 6
>>> x in [1, 5, 7, 3, 9, 2, 9, 9]
False
```

Type Annotations for Lists

- Suppose we want to specify that all elements of a list parameter have a specific type
- For the type annotation, write `list[`*`element-type`*`]`
- Example: a function that returns the average of a list of floats

```
def average(numbers: list[float]) -> float:
    """Return the average of the values in numbers."""
```

Type Annotations for Lists

- Type annotations have changed since *Practical Programming*, 3rd ed. was published
- It is no longer necessary to import type `List` from the `typing` module, as described in Chapter 8, section *Type Annotations for Lists*
- That format has been *deprecated* (will be phased out and eventually no longer supported by Python)

Processing Lists Using Indices

- We have seen that `range(`*n*`)` returns an object that generates the sequence of integers 0, 1, 2, ..., *n* - 1
- We can use `range` to produce the sequence of indices for a list's elements

Processing Lists Using Indices: Example 1

```
>>> a_list = [1, 3, 7, 2]
>>> len(a_list)
4
>>> for i in range(len(a_list)):
...     print(i)
...
0
1
2
3
```

Processing Lists Using Indices: Example 2

```
>>> a_list = [1, 3, 7, 2]
>>> len(a_list)
4
>>> for i in range(len(a_list)):
...     print(a_list[i])
...
1
3
7
2
```

Example: Find Smallest Value

- A function that returns the location (index) of the smallest value in a list

```
def find_smallest(a_list: list) -> int:
```

- The function loops over the sequence of element indices
- Local variable `smallest` stores the index of the smallest value found so far as the loop "visits" each element

Example: Find Smallest Value

```
def find_smallest(a_list: list) -> int:
    # Assume that the first element in the list is
    # the smallest one.
    smallest = 0

    # Check rest of the list for smaller elements.
    for i in range(1, len(a_list)):
        if a_list[i] < a_list[smallest]:
            # Save the index of the smallest element
            smallest = i
    return smallest
```

Lists as Function Return Values

- Write a function that takes a list of numbers and returns a a new list in which every element is twice the value of the corresponding element in the original list
- Approach:
  - Create a new list the same length as the original list
  - Loop over the indices of all elements in the original list
  - The element at index *i* in the new list is assigned 2 * the element at index *i* in the original list

Lists as Function Return Values

```
def double(numbers: list[float]) -> list[float]:
    doubled = [0] * len(numbers)
    for i in range(len(numbers)):
        doubled[i] = 2 * numbers[i]
    return doubled

>>> nums = [1, 2, 3]
>>> double(nums)
[2, 4, 6]
```

List Topics Not Covered in ECOR 1041

- Chapter 8
  - Slices (*Slicing Lists*, pp. 137-139)
  - Aliasing (*Aliasing: What's in a Name*, pp. 139-140)
  - Functions that modify their list arguments (*Mutable Parameters*, pp. 140-141)
  - List methods (*List Methods*, pp. 141-142)
  - Nested lists (*Working with Lists of Lists*, pp. 142-144)

List Topics Not Covered in ECOR 1041

- Chapter 9
  - Parallel lists (*Processing Parallel Lists Using Indices,* page 156)
  - Processing lists using nested loops (*Nesting Loops in Loops*, *Looping Over Nested Lists*, *Looping Over Ragged Lists*, pp. 156-160)

Recap of Learning Outcomes

Carleton University

Learning Outcomes (Vocabulary)

- Know the meaning of these words and phrases
  - List, index, type `list`
  - Concatenation, replication and deletion operators
  - Built-in `len`, `min` and `max` functions

Learning Outcomes

- Be able to evaluate expressions containing `list` objects, the `*`, `+` and `del` operators, indexing (*`list_name`*`[`*`index`*`]`) , and calls to the `len`, `min` and `max` functions.
- Be able to design and implement functions that process lists