15-110 Principles of Computing – F21

Lecture 13:
Lists 3

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Lists Operations Summary

- Adding List Elements: +, *, L.append(item), L.insert(index, item), L.extend(seq)
- Removing List Elements: L.remove(item) , L.pop(index)

- Counting Element Occurrences in Lists/Tuples: L.count(item)
- Finding Element Position in Lists/Tuples: L.index(item)
- Comparing Lists/Tuples: **<, > , =, !=, <=, >=, ==**
- Finding Min/Max Elements in Lists/Tuples: min(L), max(L)
- Summing Elements of Lists/Tuples: sum(L)

- Reversing Lists: L.reverse(), L[::-1]
- Sorting Lists: sorted(L), L.sort()

+ and * operators

- The + operator **concatenates** two lists (and creates a NEW one)

```
primes = [2, 3, 5, 7, 11, 13]
primes2 = [17, 19, 23]
primes = primes + primes2
```

```
primes ?
→ [2, 3, 5, 7, 11, 13, 17, 19, 23]
```

- The * operator **replicates** a list *n* times

```
primes = [2, 3, 5, 7, 11, 13]
primes2 = 2 * primes
```

```
→ [2, 3, 5, 7, 11, 13, 2, 3, 5, 7, 11, 13]
```

- The / ** % // operators do not apply to lists

Practice

Implement the function `max_min(L)` that returns a tuple with the maximum and the minimum elements of a list L

```
def max_min(L):
    return max(L), min(L)
```

Practice

Given a list L and element e, write the function `count(L, e)` that counts the number of times `e` occurs in `L`.

- Use a list method

```
def count(L,e):
    return L.count(e)
```

- Use a for loop

```
def count_loop(L,e):
    cnt = 0
    for x in L:
        if x == e:
            cnt += 1
    return cnt
```

Practice

Implement the function `allTheSame(L)` that returns True if all elements of list L are the same, and False otherwise.

```
def allTheSame(L):
    s = L[0]
    for v in L[1:]:
        if v != s:
            return False
    return True
```

- Alternative ways?

```
def allTheSame(L):
    if min(L) == max(L):
        return True
    else:
        return False
```

```
def allTheSame(L):
    s = L[0]
    if L.count(s) == len(L):
        return True
    else:
        return False
```

```
def allTheSame(L):
    if sorted(L) == sorted(L, reverse=True):
        return True
    else:
        return False
```

Practice

Implement the function `sameLists(L1, L2)` that returns True if the two lists L1 and L2 have the same elements, and False otherwise.

```
def sameLists(L1, L2):
    if L1 == L2:
        return True
    else:
        return False
```

Practice

Implement the function `nonDecreasing(L)` that returns a list with the elements in L in non-decreasing order.

- Can change L

```
def nonDecreasing(L):
    L.sort()
    return L
```

- Cannot change L

```
def nonDecreasing_new(L):
    return sorted(L)
```

Practice

Implement the function `nonIncreasing (L)` that returns a list with the elements in L in non-increasing order.

- Can change L

```
def nonIncreasing(L):
    L.sort(reverse=True)
    return L
```

- Cannot change L

```
def nonIncreasing_new(L):
    return sorted(L, reverse=True)
```

Practice

Implement the function `times10(L)` that returns a list containing the same elements as `L`, but multiplied by 10.

For example, `times10([1,2,3])` should return `[12,20,30]`.

- The returned list must not change the original list

```
def times10(L):
    L10 = []
    for i in L:
        L10 += [i * 10]
    return L10
```

- The returned list can be the input list changed

```
def times10_overwrite(L):
    for i in range(len(L)):
        L[i] = L[i] * 10
    return L
```

Practice

Implement the function `listProdSum(L)` that returns a tuple with the sum of the integer and float elements of a list and their products. You MUST use the function `sum()`

```
def listProdSum(L):
    list_sum = []
    prod = 1
    for x in L:
        if type(x) == int or type(x) == float:
            prod = prod * x
            list_sum.append(x)
    return (sum(list_sum), prod)
```

Tuples vs. Lists

- Lists are **mutable** objects: can be changed!
- Tuples are **immutable** objects: cannot be changed!

```
L = [3, 5, 7, 11]
L[2] = -1
```

```
x = L[1:3]
x[1] = 0
```

```
T = (3, 5, 7, 11)
T[2] = -1
```

Error!

```
x = T[1:3]
```

Slicing Ok → `x` is a tuple!

```
x[1] = 0
```

Error!

TypeError: 'tuple' object does not support item assignment

Why to use tuples? → To ensure / represent that a list of values won't be changed!

A tuple is a *constant/fixed* list!

Lists and consequences of being mutable objects: aliases

- Lists are **mutable** objects: can be changed ... and *aliased,* or *cloned*

```
L1 = [3, 5, 7, 11]
L2 = L1
```

L2 → [3, 5, 7, 11]

```
L2[1] = -1
```

L2 → [3, -1, 7, 11]

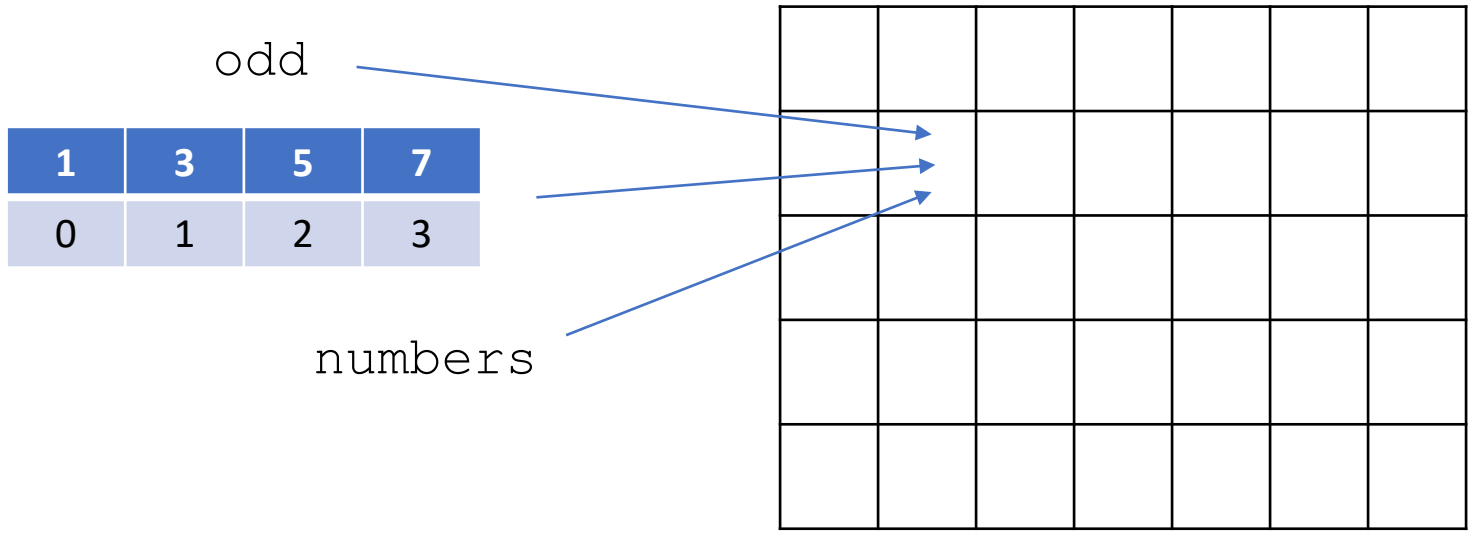
L1 ?? L1 → [3, -1, 7, 11] The same as L2 !

Writing L2 = L1 defines L2 as an **alias** of L1 and vice versa

- Changing L2 changes L1
- Changing L1 changes L2

Aliasing with mutable types

```
odd = [1, 3, 5, 7]
numbers = odd
```



- `numbers` and `odd` are *aliases* for the same mutable list in memory!
  - ✓ `numbers[1] = 29` has the same effects than `odd[1] = 29`

- The physical address / **identity** of a variable/literal: `print( id(odd), id(numbers))`

Be careful with aliasing!

Aliasing doesn't happen with immutable types!

Immutable types:

- `int`
- `float`
- `bool`
- `string`
- `tuple`

```
x = 29
y = x

y = 0
x ?    x → 29!
```

```
x = (27, 29, 30)
y = x

y = (28, 31)
x ?    x → (27, 29, 30)
```

```
hi = 'hello'
hi2 = hi

hi = 'how are you?'
hi2      hi2 → 'hello'
```

Shallow copy (*cloning*) of a list/tuple: `.copy()` method

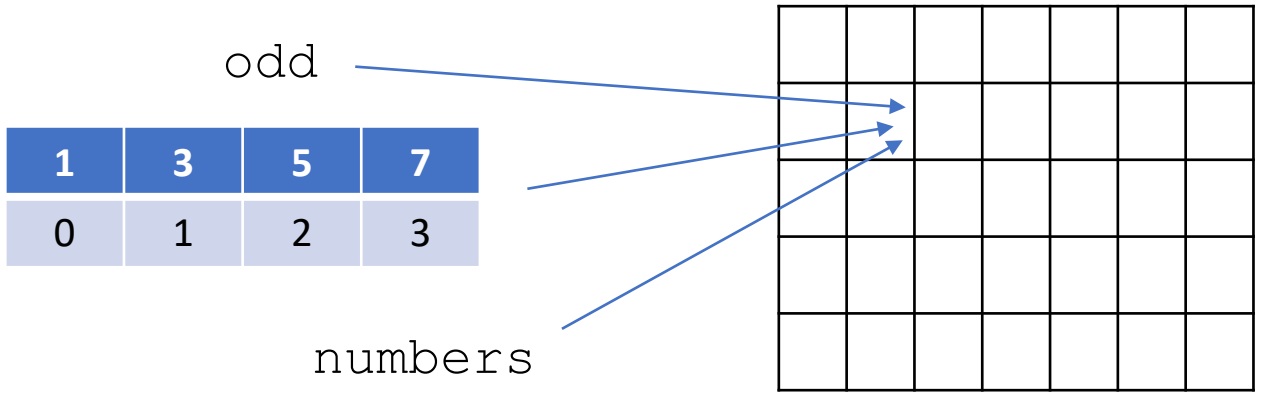
- Method `.copy()` returns a copy (**clone**) of the list/tuple (and does *not* affect the original)

```
a = [2,4,1]
b = a.copy()

print(a, b)                 → [2,4,1] [2,4,1]

print(id(a), id(b))  →   4730312200  4695822984    a and b are now different objects
```

Slicing makes a copy → Cloning!

Aliasing:

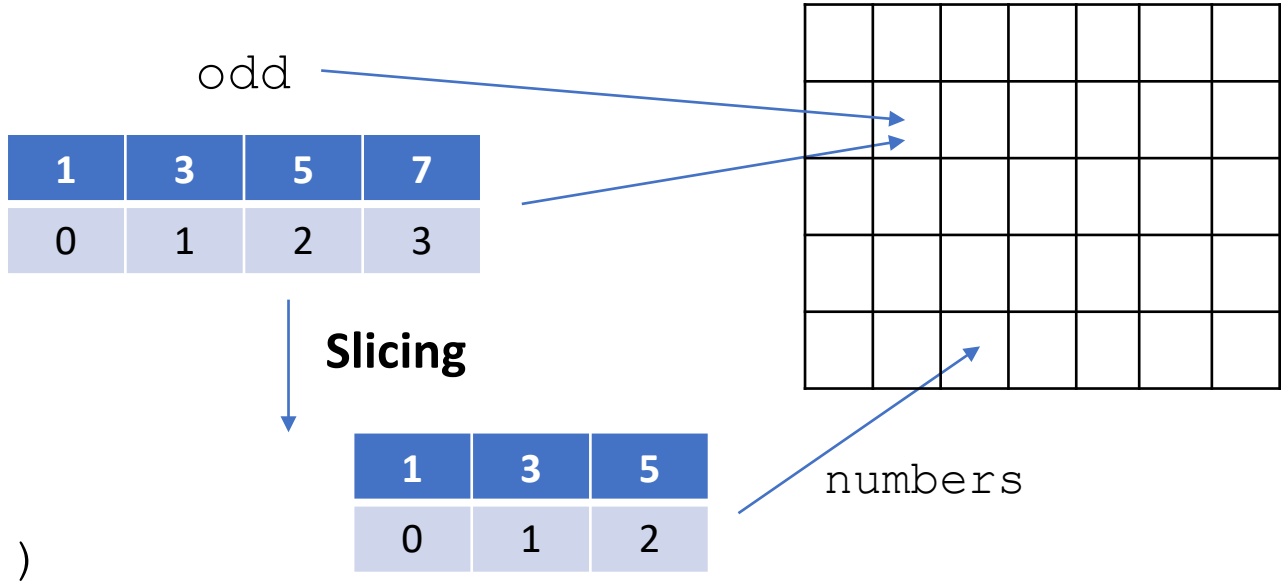
```
odd = [1, 3, 5, 7]
numbers = odd
```



✓ Slicing **extracts** content from one list, makes a *copy* of it, and pass it to the receiving list → **Cloning**

Cloning:

```
odd = [1, 3, 5, 7]
numbers = odd[0:3]

print(odd, numbers)

print(id(odd), id(numbers))
```



Slicing & .copy()

- To make a copy / clone of a list/tuple:

```
odd = [1, 3, 5, 7]

numbers_slice = odd[:]

numbers_copy = odd.copy()
```

Equivalent in terms of effects

Lists of lists

- A list can include elements that are lists (or tuples) → List of lists/tuples

```
L = [ [11,12,13], [21,22,23], [31,32,33], 99, (1,2,3) ]
```

What is the **length** of the list `L`? → `len(L)` → 5

`L[1]` ? → `[21, 22, 23]`

How do we access the third element of of the list `L[1]`?

Using the indexing operator, `[]` ! → `L[1][2]`

How do we access the second element of of the list `L[2]`? → `L[2][1]` → 32

Lists of lists

- Write function `printNestedLists(L)` that takes as input a list L that can contain list or tuple elements (i.e., nested lists), and prints out, one by one, all the individual elements

```
def printNestedLists(L):
    for v in L:
        if (type(v) == tuple) or (type(v) == list):
            for i in v:
                print(i)
        else:
            print(v)
```

```
L = [1, [2, 3, 4], 5, 6, [7,8], 9]
```

Output:

```
1
2
3
4
5
6
7
8
9
```

Using `range()` and double indexing

```
def printNestedLists_Range(L):
    for i in range( len(L) ):
        if (type(L[i]) == tuple) or (type(L[i]) == list):
            for j in range( len(L[i]) ):
                print( L[i][j] )
        else:
            print( L[i] )
```

List of lists and `copy.deepcopy()`

```
a = [1, 2, [3,4], [5,6,7]]

b = a.copy()        b → [1, 2, [3,4], [5,6,7]]

a[2][0] = -1        b → ?
                    [1, 2, [-1,4], [5,6,7]]
```

`.copy()` doesn't perform a nested copy: if there are list elements in the list, these are aliased ☹

✓ `copy.deepcopy()` solve the problem, making a deep, nested copy of all complex data structures!

```
import copy

a = [1, 2, [3,4], [5,6,7]]

b = copy.deepcopy(a)
```

```
a[2][0] = -1

b → ?
[1, 2, [3,4], [5,6,7]]
```

Test your knowledge

Write the function `operations(L, n)` that takes as input a list `L` and an integer, `n`. The function returns a copy of the list `L` and a list `LL` with the following contents. `LL` includes first all the elements of `L` at the odd positions, and then all the elements of `L` at even positions. If the length of `L` is less than `n`, the function prints out `"Short list!"`

For instance, `operations([9, 6, 4, 2, 1, 6, 7], 10)` returns the list `[6,2,6,9,4,1,7]` and will make the print.

```
def operations(L, n):
    LL = L[1::2]
    LL = LL + L[0::2]
    if len(LL) < n:
        print("Short list!")
    return L.copy(), LL
```