15-110 Principles of Computing – F21

Lecture 12:

Lists 2

Teacher:

Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر

Carnegie Mellon University Qatar

Tuples vs. Lists

- Lists: []
- Tuples: ()

```
L = [3, 5, 7, 11]

L = (3, 5, 7, 11)
```

Both are **sequences** of *anything* but ...

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

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]
x[1] = 0
```

Slicing Ok → `x` is a tuple!

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]
```

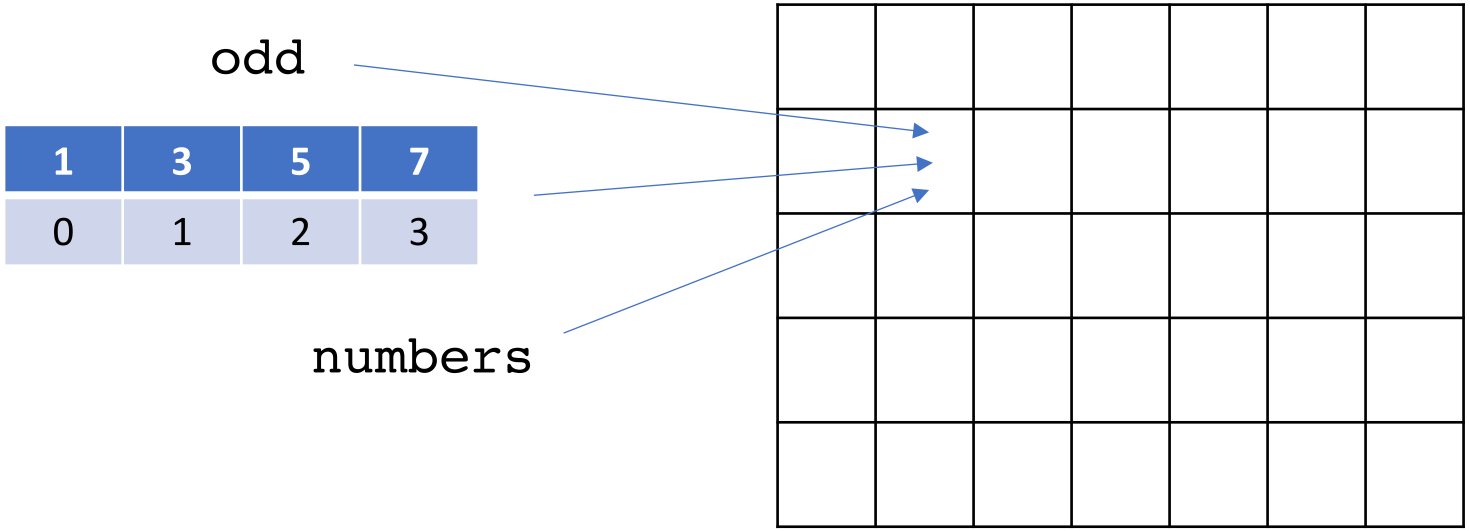
`L1 ??`   `L1 → [3, -1, 7, 11]`   The same as `L2!`

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
```



1	3	5	7
0	1	2	3

- `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)

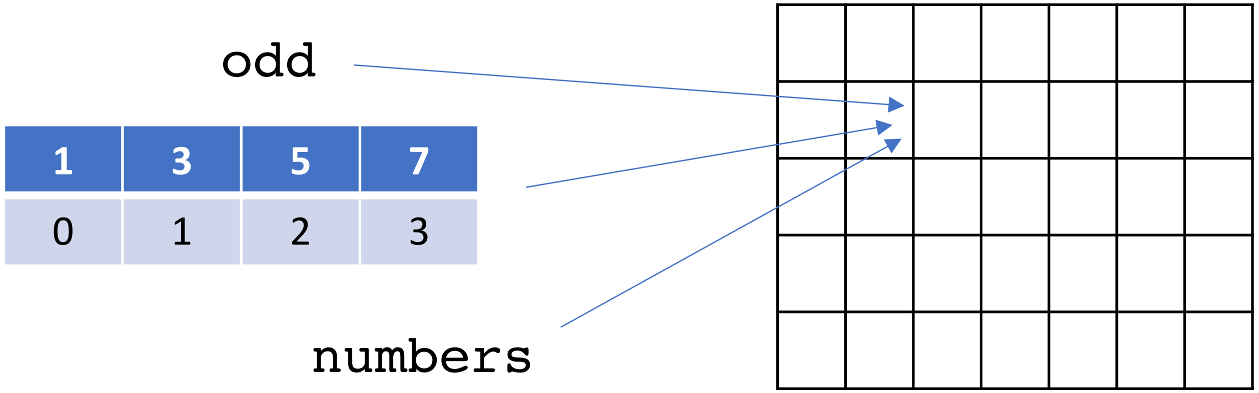
```
a = [2,4,1]
b = a.copy()

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
```



odd

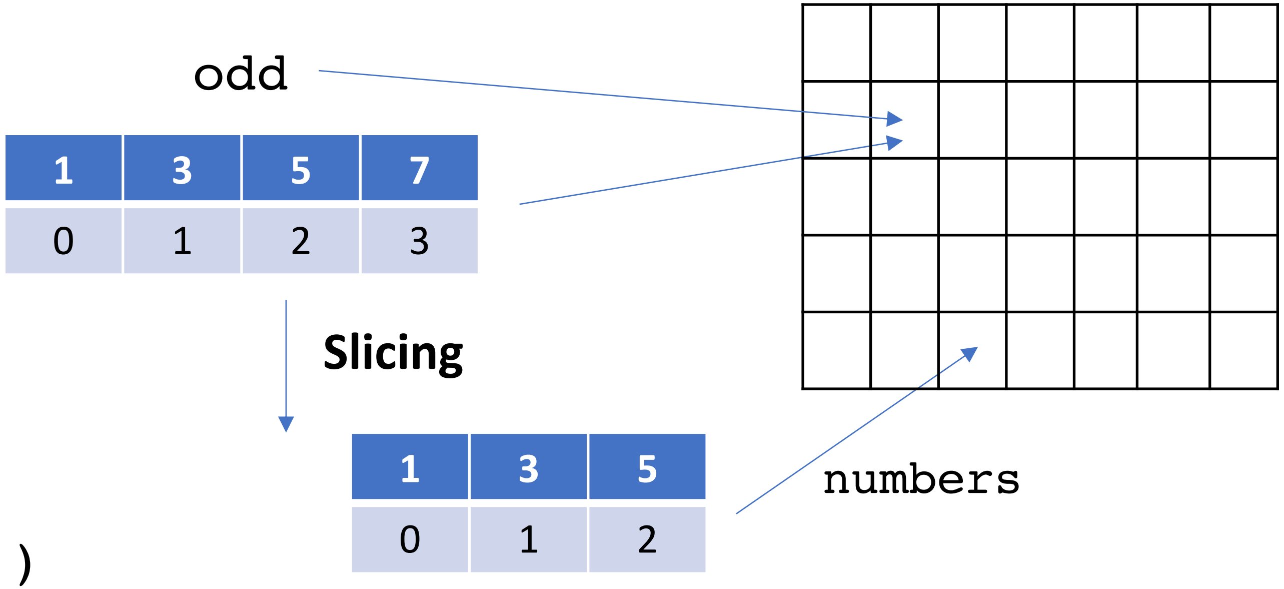
1	3	5	7
0	1	2	3

numbers

✓ 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))
```



odd

1	3	5	7
0	1	2	3

Slicing

1	3	5
0	1	2

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

Parallel assignments

```
T = (4,3,2,1)

a = T[0]
b = T[1]
c = T[2]
d = T[3]

print("a =", a)
print("b =", b)
print("c =", c)
print("d =", d)
```

A more compact way of making the same assignment:

```
T = (4,3,2,1)
a, b, c, d = T

print("a =", a)
print("b =", b)
print("c =", c)
print("d =", d)
```

- Number of values on the right must be the same as the number of variables on the left

```
L = [1,2,3]

a, b = L
```

```
ValueError: too many values to unpack (expected 2)
```

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 tuple `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)
```

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[2][0] = -1
a = [1, 2, [3,4], [5,6,7]]
                                 b → ?
b = copy.deepcopy(a)             [1, 2, [3,4], [5,6,7]]
```

Adding list elements: + operator

- 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]
```

➢ Is `primes` the *same* list as before? i.e., is primes at the same place in the memory?

No: a new list is created and stored in some (other) memory address → Expensive!

```
primes = [2, 3, 5, 7, 11, 13]
print('Original address of primes:', id(primes))
primes2 = [17, 19, 23]
primes = primes + primes2
print('New address of primes:', id(primes))
```

Adding single list elements: + operator

- We can use the + operator to add one single element to the list (need to use [])

```
primes = [2, 3, 5, 7, 11, 13]
primes = primes + [17]
                              primes ?
                              → [2, 3, 5, 7, 11, 13, 17]
```

➢ Remember: after this operation a new list is being created in memory

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
```