



# 15-110 PRINCIPLES OF COMPUTING – S19

## LECTURE 7:

## LISTS AND TUPLES DATA STRUCTURES 2

TEACHER:

GIANNI A. DI CARO

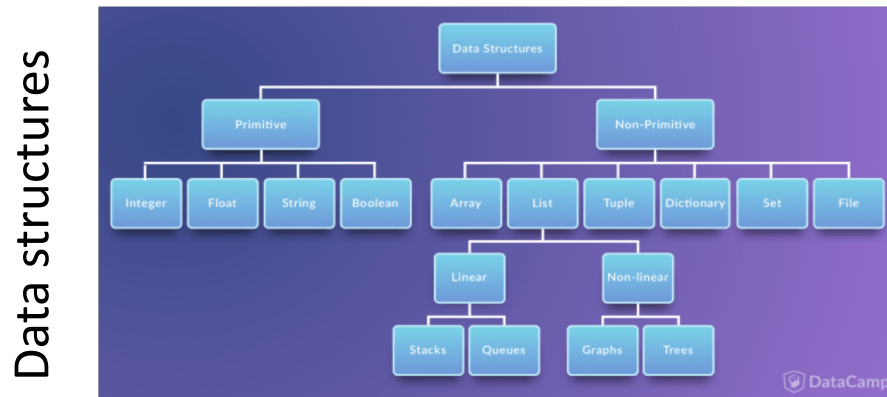
# Tuple and Lists: so far

- **Tuple:** (non-scalar type) ordered sequence of *objects* (any type, not need same type), immutable

```
person_info = ('Donald', 'Trump', 14, 'June', 1946, 'President')
```

- **List:** (non-scalar type) ordered sequence of *objects* (any type, not need same type), mutable

```
person_info = ['Donald', 'Trump', 14, 'June', 1946, 'President']
```



- **Access (read) operations**, at any position:

➤ get data by index `x[], x[:], x[::]` `x[][][]`

➤ find data by content

`a = x[1]`, `a = x[0:3]`, `a = x[1:6:2]`, `a = x[1][0][2]`

- **Update operations**, *at any position*, only for lists:

- **change** the value of an item
- **remove** one item
- **add** one or more new items anywhere

`x[], x[:], x[::]` `x[][][]`

`x[1] = 2`, `x[0:3] = [1,3,5]`, `x[1:6:2] = [2,3]`

`x[1][0][2]=3`

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

# Mutable vs. Immutable types: int

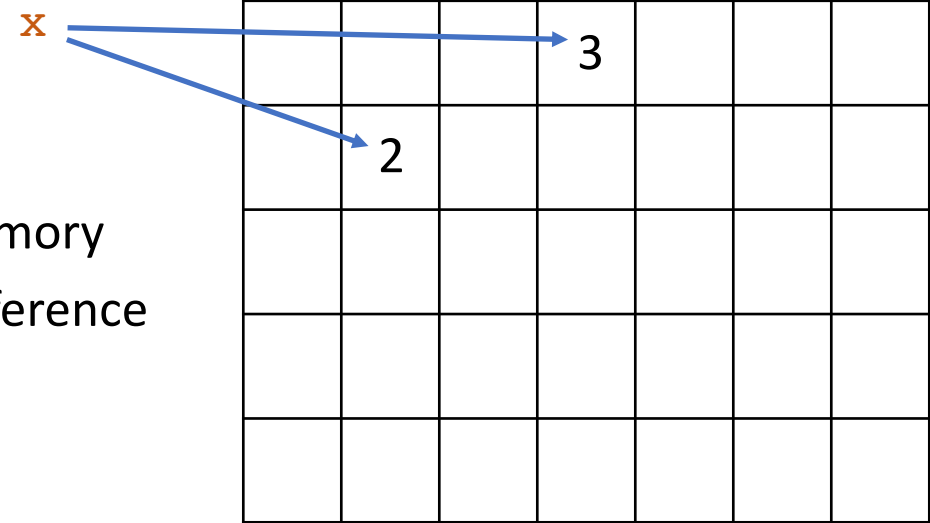
## 1. In the high-level program:

`x = 2` meaning that variable `x` has (`int`) value 2

### ■ Internal to python / computer:

variable `x` holds the **reference** (the address) to the memory location to where its value 2 is currently *stored* (the reference is associated to the identity `id()` of the variable)

```
print( id(x) )
```



## 2. Next instruction in the high-level program:

`x = 3` meaning that variable `x` has changed its (`int`) value to 3

### ■ Internal to python / computer:

variable `x` is of immutable type `int`, meaning that *its value in memory cannot be changed* → A new memory location is allocated to hold the integer literal 3.  
`x` now holds the reference to the new memory location (check `id(x)`!)

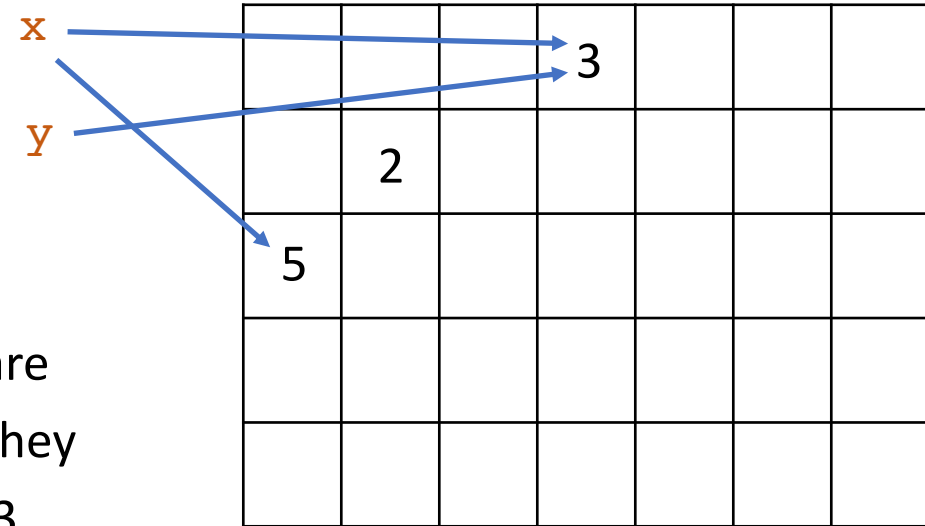
# Mutable vs. Immutable types: int

3. Next instruction in the high-level program:

`y = x` meaning that variable `y` gets the same value of `x`, and vice versa!

- Internal to python / computer

variable `y` gets the reference held by `x`, `y` and `x` are bound to the **same memory location** for their value, they are *temporary aliases* for referring to the same value 3



- `int` is an *immutable type*, such that any further change in the values of either `x` or `y` brakes their bound, creating a *new variable*

4. Next instruction in the high-level program:

`x = 5` meaning that variable `x` now gets a new value, which is allocated to a new, different memory location

- Internal to python / computer:

variable `x` gets the new reference associate to 5's memory location, `y` still references to value 3

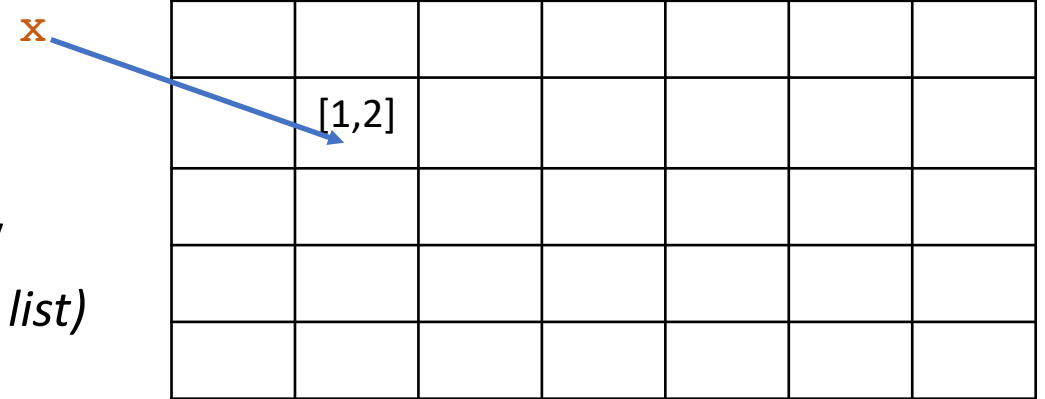
# Mutable vs. Immutable types: list

## 1. In the high-level program:

`x = [1, 2]` meaning that list `x` has value `[1, 2]`

### ■ Internal to python / computer:

variable `x` holds the **reference** (address) to the memory location to where the list values are stored (*head of the list*)

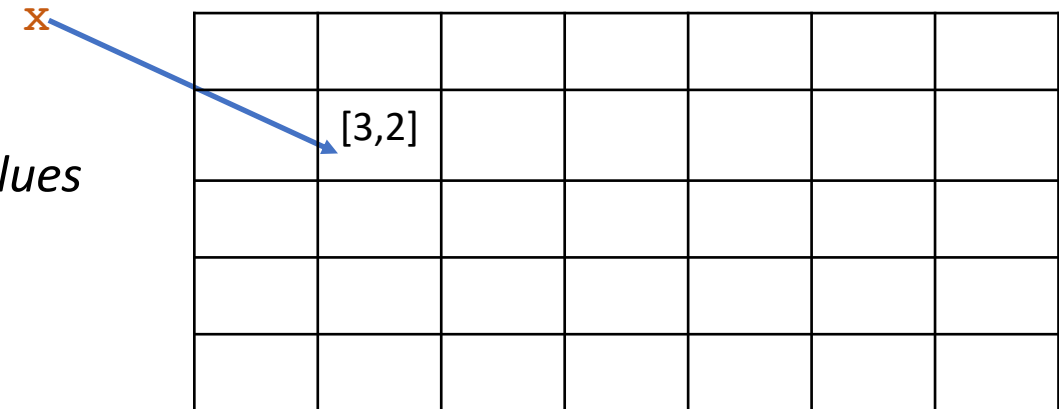


## 2. Next instruction in the high-level program:

`x[0] = 3` meaning that the value at index 0 has changed to 3

### ■ Internal to python / computer:

variable `x` is of mutable type `list`, meaning that *its values in memory can be changed* → The value at the memory location holding `x[0]` is updated with the value 3.  
`x` holds the same reference as prior this instruction



# Mutable vs. Immutable types

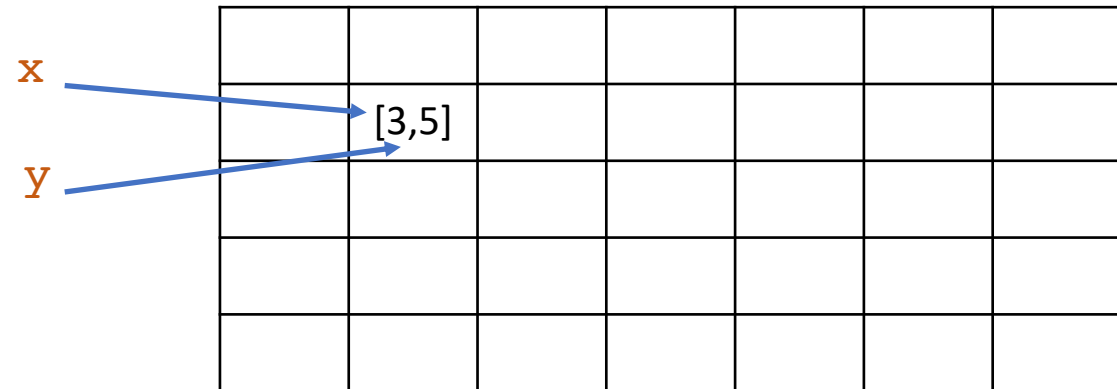
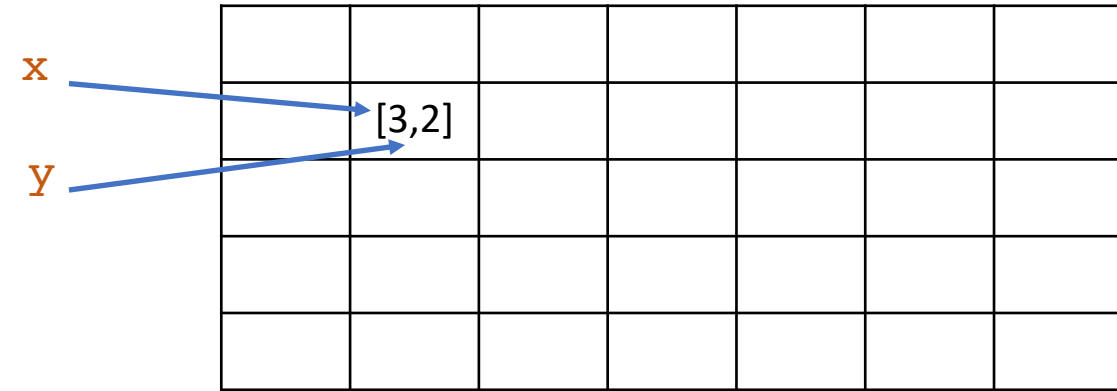
3. Next instruction in the high-level program:

`y = x` meaning that list variable `y` gets the same value of `x`, and vice versa!

- Internal to python / computer:

variable `y` gets the reference held by `x`, `y` and `x` are bound to the **same memory location** for their value, they are **permanent aliases**

- `list` is a *mutable type*, such that any further change in the values of either `x` or `y` will be reflected in the other list because of their bound



4. Next instruction in the high-level program:

`x[1] = 5` meaning that variable `x` updates to 5 its value at index 1

- Internal to python / computer:

the memory location referenced by both `x` and `y` gets updated in value → both `x` and `y` now have value `[3,5]`

# Cloning vs. Aliasing: Issues and opportunities of mutability

- Initializing a list with another list?

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

primes[1] = 29
```

what is the value of numbers?

```
print("Do the lists have the same contents?", numbers == primes)
print("Primes:", primes)
print("Numbers:", numbers)
print("Are the two lists the 'same' list?", id(primes) == id(numbers))
```

- Initializing a list with the contents of another list using a range?

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

primes[1] = 29
```

what is the value of numbers?

# Cloning vs. Aliasing: Issues and opportunities of mutability

From one variable to another: we can transfer data only or data & identity

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

```
numbers = primes
```

Variables are passed by identity,  
by *reference* address → **Aliasing**

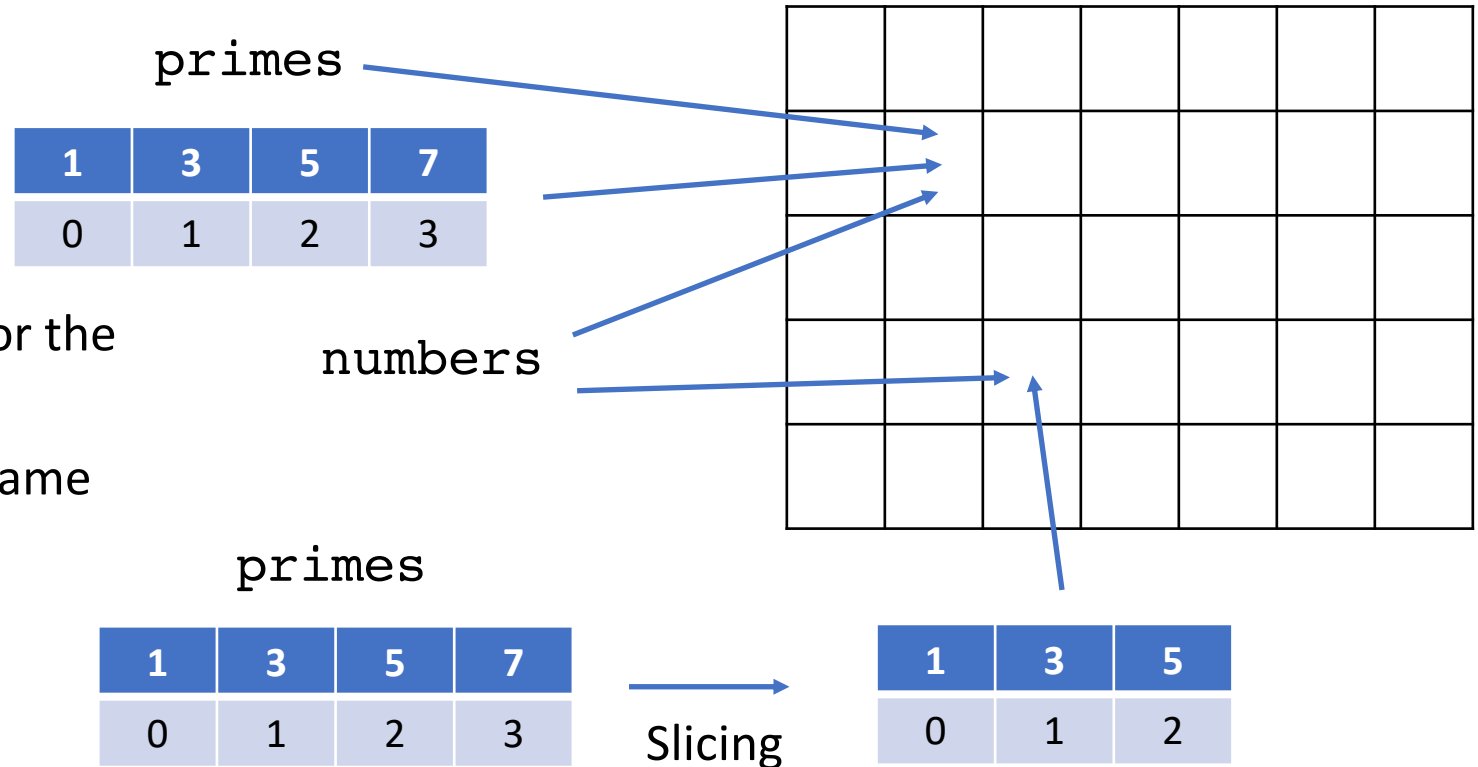
- numbers and primes are *aliases* for the same mutable list in memory!

✓ `numbers[1] = 29` has the same effects than `primes[1]=29`

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

```
numbers = primes[0:3]
```

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



**Pay attention to statements resulting in cloning vs. aliasing!**



# Cloning vs. Aliasing: Issues and opportunities of mutability

---

- Previous reasoning apply also when we create list of lists:

```
p = [1, 3, 5, 7]
pp = [11, 13, 17]
n = [p, pp]
```

Any (*in-place*) change to either p or pp is reflected on n, and vice versa

# Cloning vs. Aliasing: Issues and opportunities of mutability

---

What about the following piece of code with `int` variables?

```
p = 1
n = p
print(n, p, id(n), id(p))
p = 29
print(p, n)
n = -1
print(p, n)
```

`int`, `float`, `bool`, `str` variables are immutable: we don't update the content at the same memory location, every time a change is made, a new memory variable (memory location) is potentially generated, that potentially has a new identity

# Cloning vs. Aliasing: Issues and opportunities of mutability

---

## When to use aliasing vs. cloning with lists?

- **Aliasing:** For instance, when we need to create aliases in the program such that it is convenient (or more clear) to refer to the same object using different names, maybe in different parts of the program

```
# sport, sedan and family lists have been defined already  
cars = [sport, sedan, family]
```

E.g., based on different input data (web requests, email data, files, keyboard inputs, ...), the program manipulates and updates data about sport, sedan, and family cars in separate parts / modules of the program, updating the specific list (`sport`, `sedan`, `family`) only

Using the alias, any updates to any sub-set of cars gets automatically reflected in updates in the general `cars` list, that can be accessed being always up-to-date with current data

# Cloning vs. Aliasing: Issues and opportunities of mutability

## When to use aliasing vs. cloning with lists?

- **Cloning:** If we are only interested in the *values* held by a certain list, such that we want to *use/transfer* its data without creating any binding between the variables

```
basic_primes = [1,3,5,7,11]
primes = basic_primes[:]
# later on additional primes can be added to primes, unaffected basic_primes
```

- Given an existing list *b*, a few equivalent alternatives (with the same macroscopic effects) are possible to create a new list *a* that inherits data from *b* but doesn't establish any binding aliases

Slicing

$$\left\{ \begin{array}{l} a = b[:] \\ a = [] \\ a = b[:] \end{array} \right.$$

Concatenating

$$\left\{ \begin{array}{l} a = [] \\ a = a + b \\ a = [] \\ a += b \end{array} \right.$$

List methods

$$\left\{ \begin{array}{l} a = [] \\ a.extend(b) \\ a = b.copy() \end{array} \right.$$

# Basic list/tuple operators: +, \*

## ■ Operator + : Concatenation of lists/ tuples

```
prime_numbers = [1, 3, 5, 7, 11]
```

```
other_primes = [13, 17, 19]
```

```
new_primes = prime_numbers + other_primes → new list/tuple with [1,3,5,7,11,13,17,19]
```

```
my_list = [1,2,3] + (1,3,5) → error! operator + needs same type operands
```

## ■ Operator \* : Duplication of lists/tuples

```
prime_numbers = [1, 3, 5]
```

```
repeat_primes = prime_numbers * 2 → new list/tuple with [1,3,5,1,3,5]
```

```
x = [1,1,1,1,1,1,1,1,1,1] → create a list/tuple with 10 elements all initialized to integer value 1
```

```
x = [1]*10 → create a list/tuple with 10 elements all initialized to the integer value 1
```

Cloning



# Basic list/tuple operators: in, not in

---

- Operator **in**: **Membership**, returns `True` if item belongs to the list/tuple, `False` otherwise

```
prime_numbers = [1, 3, 5, 7, 11]
```

```
is_prime = 5 in prime_numbers
```

→ new bool variable with value `True`

- Operator **not in**: **Membership**, returns `False` if item belongs to the list/tuple, `True` otherwise

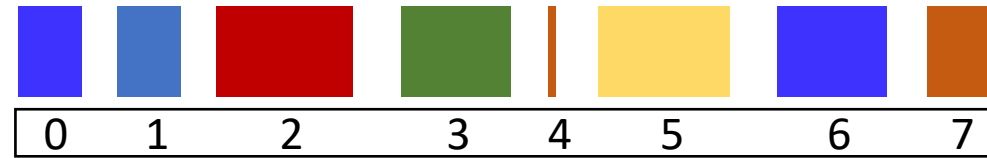
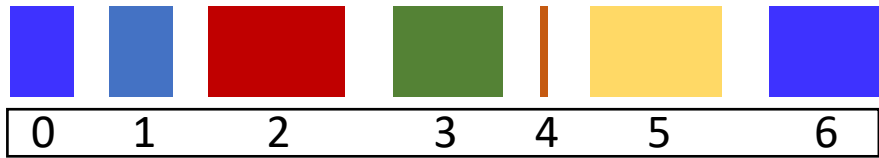
```
prime_numbers = [1, 3, 5, 7, 11]
```

```
is_prime = 5 not in prime_numbers
```

→ new bool variable with value `False`

# Adding single list elements: `append()`, `insert()`

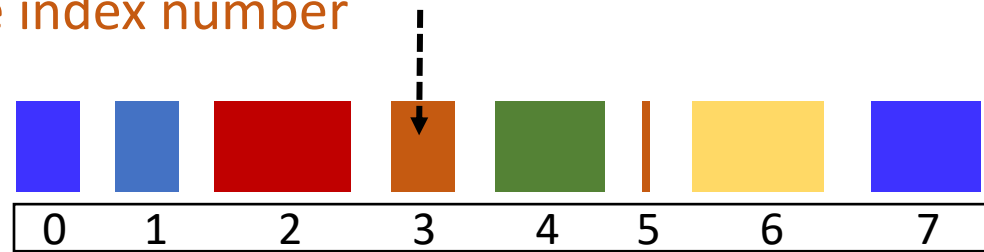
- Method `l.append(item)`: add an item at the end of the **same list** (*in-place*)



```
primes = [1, 3, 5, 7, 11, 13, 17]
```

```
primes.append(19) → same list, extended to the end by adding one int literal of value 19
```

- Method `l.insert(index, item)`: add an item at the index position of the **same list**, moving all the other items in the list up by one index number



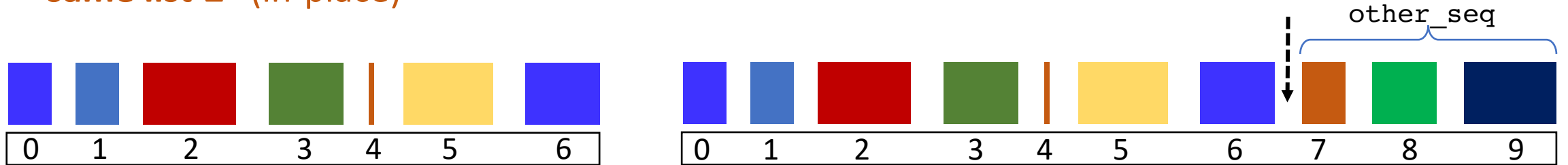
```
primes = [1, 3, 5, 7, 11, 13, 17]
```

```
primes.insert(3,19) → same list with new item: [1, 3, 5, 19, 7, 11, 13, 17]
```

```
primes.insert(0,23) → same list, with new item, all index shifted: [23, 1, 3, 5, 19, 7, 11, 13, 17]
```

# Adding multiple list elements: extend()

- Method `l.extend(other_seq)`: add items from another list/tuple onto the end of the same list l (in-place)



```
primes = [1, 3, 5, 7, 11, 13, 17]
```

```
other_primes = (19, 23, 29)
```

```
primes.extend(other_primes) → same list, extended to the end by adding the items of other_primes
```

```
primes.extend(other_primes[0:2]) → extended to the end by adding two items of other_primes
```

Cloning behavior!



# Adding multiple list elements: +, +=

- Concatenation operator `+` : add items from another list/tuple onto the end of the list

```
primes = [1, 3, 5, 7, 11, 13, 17]
```

```
primes = primes + [19, 23, 29] → list with the same resulting content as with extend()  
                                but different identity
```

**+ operator** : `primes` changes *identity*, *not in-place*

(check it with `print(id(primes))` before and after concatenation!)

- Compact notation for the above type of expressions using **operator addition**: `+=`

```
primes += [19, 23, 29] same high-level result as primes = primes + [19, 23, 29]
```

**+= operator**: `primes` doesn't change identity, *in-place*

(check it with `print(id(primes))` before and after `+=` )

- Also true for **operator multiplication**: `*=`

`a = a * b` is the same in value as `a *= b`

# Removing single list elements: remove(), pop()

---

- Method `l.remove(item)`: remove the (first) element with value `item` in the list, moving all the other items in the list up by one index number → Removal **by content**

```
numbers = [1, 3, 5, 4, 5, 5, 17]
```

```
numbers.remove(5) → same list, with the first element of value 5 being removed [1,3,4,5,5,17]
```

```
numbers.remove(15) → error! an item with value 5 is not found in the list: use in prior to remove()
```

- Method `l.pop(index)`: takes the argument `index` and removes the item present at that index, moving all the other items in the list up by one index number → Removal **by index**

```
numbers = [1, 3, 5, 4, 5, 5, 17]
```

```
numbers.pop(2) → same list, with the item at index 2, of value 5, being removed [1,3,4,5,5,17]
```

```
numbers.pop(8) → error! an index 8 is out of range for the list: use len() prior to pop()
```

# Getting information about elements: count(), index() methods

---

- `t.count(item)` : Returns the number of occurrences of `item` in the list/tuple `t`

```
scores = [1, 11, 5, 11, 4, 11, 7, 9, 0, 4]
```

```
n = scores.count(11) → n is an integer of value 3, the # of occurrences of 11 in scores
```

```
l = (True, False, True).count(True) → l is an integer of value 2 (two occurrences of True)
```

- `t.index(item)` : Returns the index of the first occurrence of `item` in the list/tuple `t`

```
scores = [1, 11, 5, 11, 4, 11, 7, 9, 0, 4]
```

```
n = scores.index(11) → n is an integer of value 1, the index of first occurrence of 11 in scores
```

```
n = scores.index(19) → generates an error since 19 is not in scores: to avoid the error use the  
operator in to check membership first
```

# Useful operations: len(), getsizeof(), del functions

- `len(l)` : Returns the **length** of a list/tuple `l` (the integer number of elements in the tuple/list)

```
prime_numbers = [1, 3, 5, 7, 11]
```

```
n = len(prime_numbers)    → n is an integer of value 5, the number of elements in the list/tuple
```

- How many **bytes** are used in memory for a list `l` (or any other object)? `sys.getsizeof(l)`

```
import sys
```

```
total_bytes_empty = sys.getsizeof([])
```

```
prime_numbers = [1, 3, 5, 7, 11]
```

```
total_bytes_five_int = sys.getsizeof(prime_numbers)
```

- Can we explicitly **delete an unused list** `l`, or, in general an **unused object**? (e.g., it occupies a lot of memory, and we can't / don't want to wait for the *garbage collector*): `del l`

```
my_unused_list = [1, 3, 5, 7, 11]
```

```
del my_unused_list
```

# Useful operations: sort (), reverse() methods

- `sort (key, reverse)` : Changes (in-place) the list `l` (not applicable to tuples!) with the elements sorted according to the (optional) criterion `key` for comparing the items ; the (optional) parameter `reverse`, if set to `True`, provides the result in *descending* order

```
numbers = [1, 4, 2, -7, 0, 6]
```

```
numbers.sort() → items are all integer, no criterion is required, numbers is sorted ascending:  
[-7, 0, 1, 2, 4, 6]
```

```
cars = ['toyota', 'BMW', 'nissan']
```

```
cars.sort(reverse=True) → items are str, no criterion is required, cars is sorted descending:  
['toyota', 'nissan', 'BMW']
```

```
mix = ['toyota', 'BMW', 'nissan', 3]
```

```
cars.sort() → error! items have mixed types, a comparison criterion is required
```

- `reverse ()` : Changes (in-place) the list `l` (not applicable to tuples!) putting the elements in the reverse order compared to the original list

```
numbers = [1, 4, 2, -7, 0, 6]
```

```
numbers.reverse() → numbers list is now: [6, 0, -7, 2, 4, 1]
```

# Useful operations: max(), min() functions

---

- `max(t, key)` : Returns the **item** of the list/tuple `t` with **maximum value**
  - Without a key (optional criterion for comparison), it can be applied only to homogeneous lists/tuples (all elements of the same type)
  - Return type depends on the type of the items

```
prime_numbers = [1, 3, 5, 7, 11]
```

```
n = max(prime_numbers) → n is an integer of value 11, the item of highest value
```

```
c = max('red', 'green', 'blue') → c is a string of value 'red', corresponding the item of  
highest code value (starting from 'r') in UTF-8
```

```
l = max('a', 'c', 'C') → l is a string of value 'c', that has highest code value in UTF-8
```

```
logical = max(True, False, True) → logical is a boolean of value True (1)
```

```
x = max(1, 3, True, 'red') → generates an error (how to compare different items?)
```

- `min(t, key)` : Returns the **item** of the list/tuple `t` with **minimum value**

```
l = min['a', 'c', 'C'] → l is a string of value 'C', that has lowest code value in UTF-8
```

```
logical = min(True, False, True) → logical is a boolean of value False (0)
```

# Use of a key for item comparison in sort(), max()/min(),...

---

- `sort(key, reverse)` : Changes (in-place) the list `l` (not applicable to tuples!) with the elements sorted according to the (optional) criterion `key` for comparing the items ; the (optional) parameter `reverse`, if set to `True`, provides the result in *descending* order
- **key** parameter:
  - specifies a **function** to be called **on each list element** prior to making comparisons
    - a function that takes a single input parameter, `F(x)`
  - the **return value** of the function is the key used for *comparison purposes*
  - return value must be a **primitive type**, such that python knows how to make comparisons among keys

```
my_tuples = [(1,2), (5,7,8), (1,), (2,2,0,0)]  
my_tuples.sort(key = len)
```

```
my_strings = ['hello', 'Good morning', 'I am', 'list example', 'zzTop']  
my_strings.sort(key = str.lower)
```

# Use of a key for item comparison in sort(), max()/min(),...

---

```
def cmp_on_second_element(item):  
    return item[1]
```

```
my_tuples = [(1,2), (5,7,8), (1,0), (2,2,0,0)]  
my_tuples.sort(key = cmp_on_second_element)
```

```
def cmp_on_second_element(item):  
    if len(item) >= 2  
        return item[1]  
    else  
        return -1
```

```
my_tuples = [(1,2), (5,7,8), (1,), (2,2,0,0)]  
my_tuples.sort(key = cmp_on_second_element)
```



# Iterating over (all) the elements of a list

---

➤ We might want to perform actions **on the entire list**, potentially on all items, or subsets of items

- **Initialize** a large list according to a given pattern that might depend on *index* values

- Set up a list of  $n$  (e.g., 1000) elements such that the element at position  $i$  has value  $i$   
`sequential_numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, ..., 999]`
- Set up a list of  $n$  (e.g., 1000) elements such that the element at position  $i$  has value  $\sum_{k=0}^i k$   
`incremental_sum = [0, 1, 3, 6, 10, 15, 21, ..., 4999500]`
- Set up a list of  $n$  (e.g., 256) elements such that each element is a unique string of 0 and 1  
`binary = ['00000000', '00000001', '00000010', ..., '11111111']`

# Iterating over (all) the elements of a list

---

- **Modify** or **use/extract** all values (or all values that satisfy a given condition) of a large list according to a given pattern that depends on *item* values
  - Scale all values by a factor 0.5 (e.g., price discount rate)  

```
articles = [['book', 15], ['toy', 25], ['cookies', 8], ...]  
articles ← [['book', 7.5], ['toy', 12.5], ['cookies', 4], ...]
```
  - Extract all items that are older than one week (e.g., food articles)  

```
articles = [['cheese', 10], ['milk', 2], ['butter', 8], ...]  
expiring ← [['cheese', 10], ['butter', 8], ...]
```
  - Find items satisfy a condition and perform an incremental operation (e.g., sum money invested in edge funds)  

```
investments = [['EF1', 100000], ['B1', 50000], ['EF4', 2000], ...]  
capital_in_EF ← 100000 + 2000 + ...
```

# Iterating over (all) the elements of a list

---

- How do we perform these list-level operations? → **Iterators**
- Constructs to repeat actions without explicitly enumerating all the elements to act upon

## Operators for iterations:

✓ `for i in sequence`

✓ `while condition_is_true`

Next time!