
# 15-110 Principles of Computing – F19

## Lecture 12:
## Tuples, Lists 3

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

# So far about Python ...

- Basic elements of a program:
  - Literal objects
  - Variables objects
  - Function objects
  - Commands
  - Expressions
  - Operators

- Utility functions (built-in):
  - `print(arg1, arg2, …)`
  - `type(obj)`
  - `id(obj)`
  - `int(obj)`
  - `float(obj)`
  - `bool(obj)`
  - `str(obj)`
  - `input(msg)`
  - `len(non_scalar_obj)`

- Object properties
  - Literal vs. Variable
  - Type
  - Scalar vs. Non-scalar
  - Immutable vs. Mutable

- Conditional flow control

```
if cond_true:
    do_something
```

```
if cond_true:
    do_something
else:
    do_something_else
```

```
if cond1_true:
    do_something_1
elif cond2_true:
    do_something_2
else:
    do_something_else
```

- Data types:
  - `int`
  - `float`
  - `bool`
  - `str`
  - `None`
  - **`tuple`**
  - **`list`**

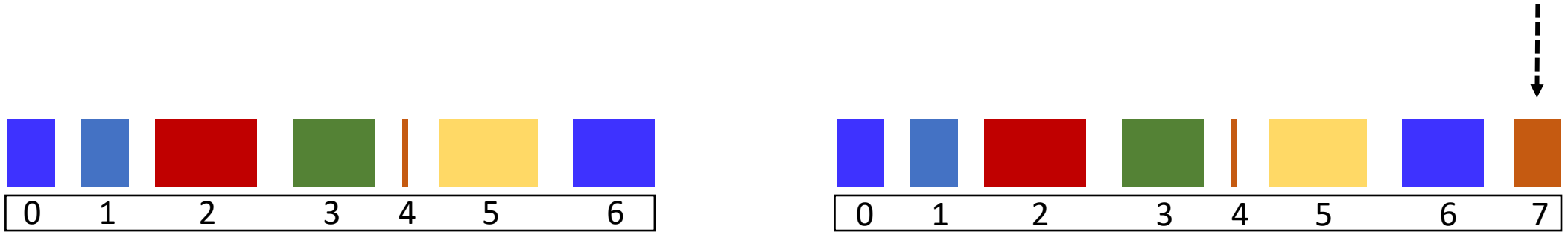
- Relational operators
  - `>`
  - `<`
  - `>=`
  - `<=`
  - `==`
  - `!=`
  - Logical operators
    - `and`
    - `or`
    - `not`

- Operators:
  - `=`
  - `+`
  - `+=`
  - `-`
  - `/`
  - `*`
  - `*=`
  - `//`
  - `%`
  - `**`
  - **`[]`**
  - **`[:]`**
  - **`[::]`**

- String methods

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

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

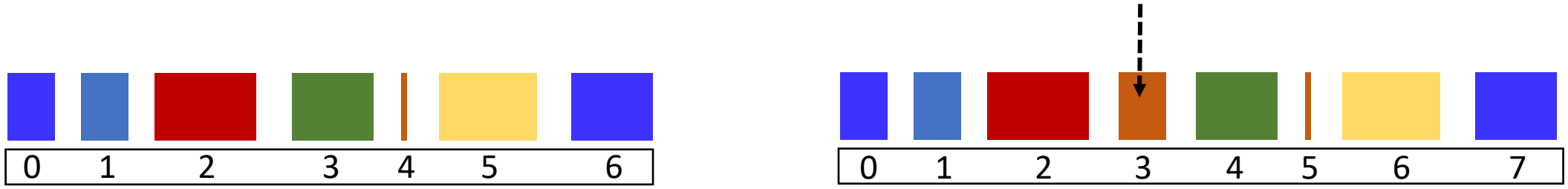



```
primes = [1, 3, 5, 7, 11, 13, 17]
```

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

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

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



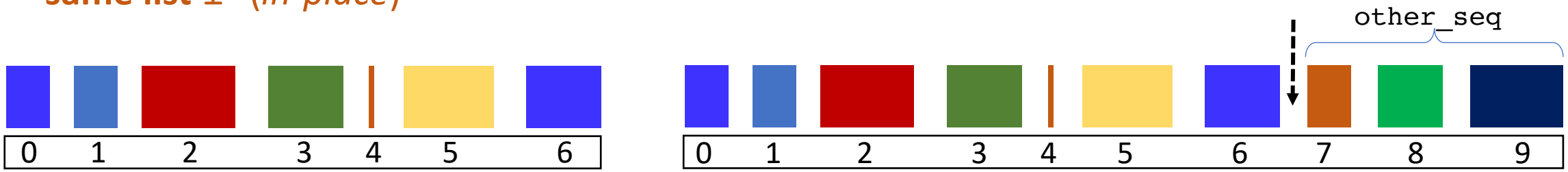

```
primes = [1, 3, 5, 7, 11, 13, 17]
```

`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

- 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

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

- 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 (*in-place*) → 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()`

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

- 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 (*in-place*), the removed item is also returned by the function → 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]

`n = numbers.pop(0)` → `n` gets value 1

`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(), length of a tuple/list

- `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

**Remember**: `len(nonscalar_obj)` can be invoked on any non-scalar object!

# Test your knowledge

Write the function `methods(L1, L2, n)` that takes as input two lists `L1, L2,` and an integer, `n`.

- The function returns a tuple `T` with the following contents.
- `T` includes all the elements of `L2` and `L1`, concatenated (L2 first).
- Then, the element at position n must be removed and replaced by the list [1,2,3].
- The resulting tuple must be returned.
- The function also prints out the length of `T` and the number of times the number `n` appears in the returned list.

# Useful operations: getsizeof() function, `del` operator

- How many **bytes** are used <u>in memory</u> 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
```

# Shallow copy (cloning) of a list/tuple: copy() method

```
a = [2,4,1]
b = a.remove(4)
print(a,b)=  → [2, 1] None
```

```
a = [1,2]
b = a.extend([4,5])
print(a,b)          → [1, 2, 4, 5] None
```

**Most of the methods so far** operate *in-place*, they change the object but **do *not* return it**

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

```
a = [2,4,1]
b = a.copy()
print(a,b)
print(id(a), id(b))   → [2,4,1] [2,4,1]
                      →   4730312200   4695822984
```

a and b are now <u>different objects</u>

# Different ways of doing cloning / shallow copy

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

Slicing

```
a = b[:]

a = []
a = b[:]
```

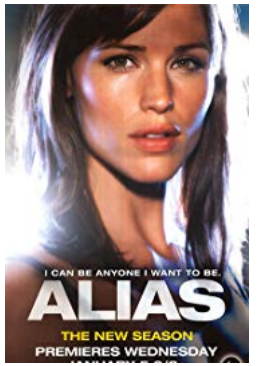
Concatenating

```
a = []
a = a + b

a = []
a += b
```

```
a = b
```

**Aliasing**

Watch out!

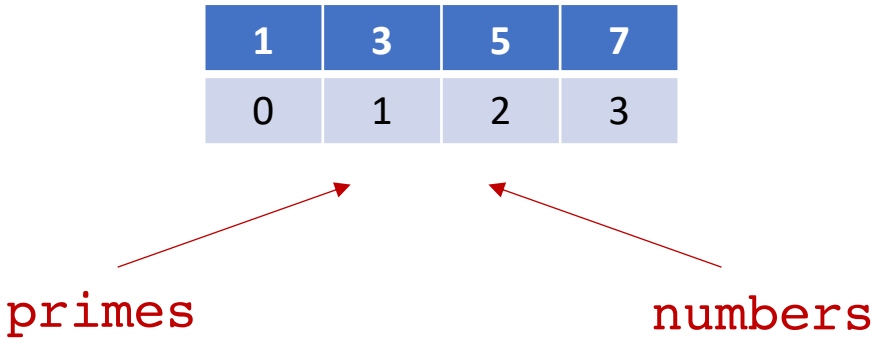



Aliasing ≠ Cloning

List methods

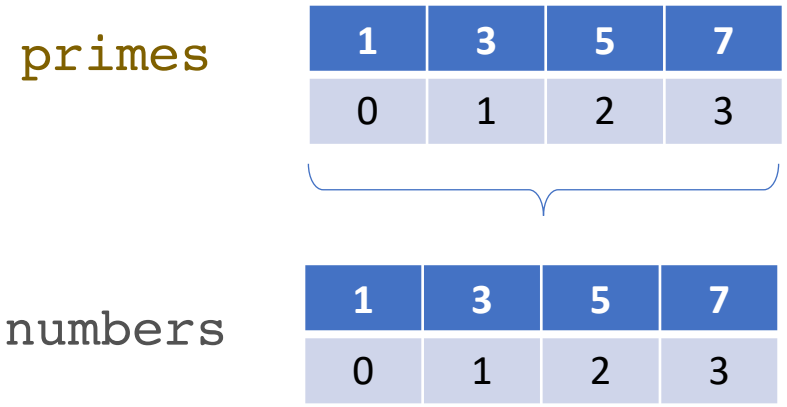
```
a = []
a.extend(b)

a = b.copy()
```

# Cloning vs. Aliasing




Aliasing,
same physical identity

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




Cloning, shallow copy,
different physical identity

```
primes = [1, 3, 5, 7]
numbers = primes.copy()
```

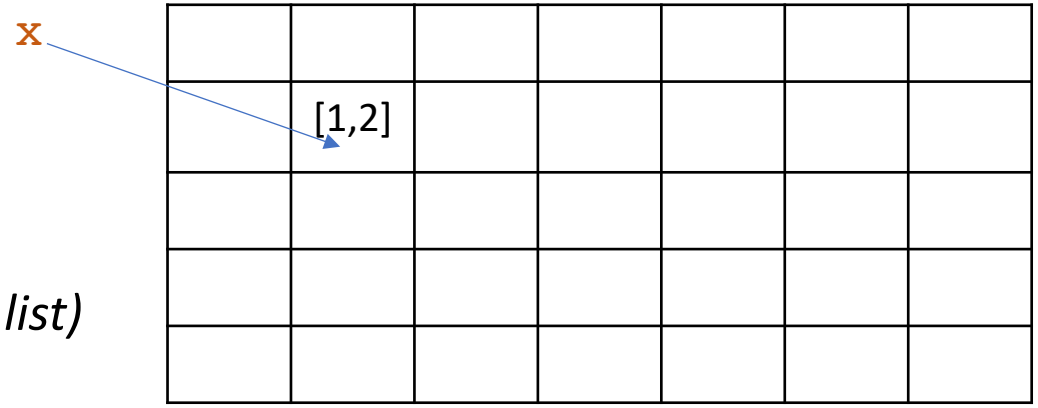
# (Optional) Mutable vs. Immutable types: lists

1. In the high-level program:

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

   - Internal to pyhton / computer:

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



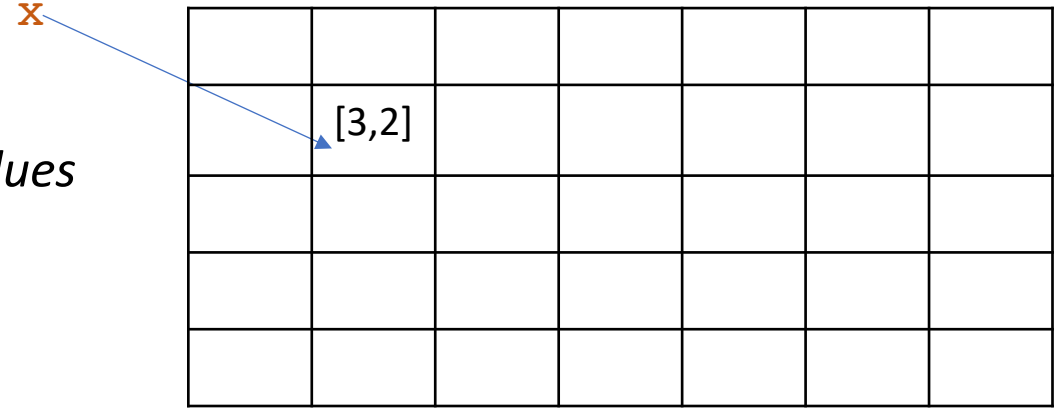

2. Next instruction in the high-level program:

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

   - Internal to pyhton / 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

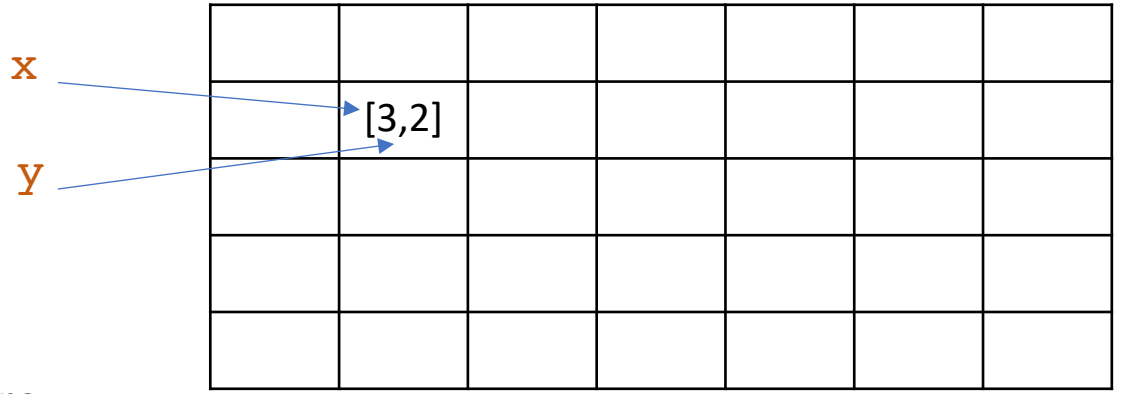


# (Optional) Mutable vs. Immutable types: list

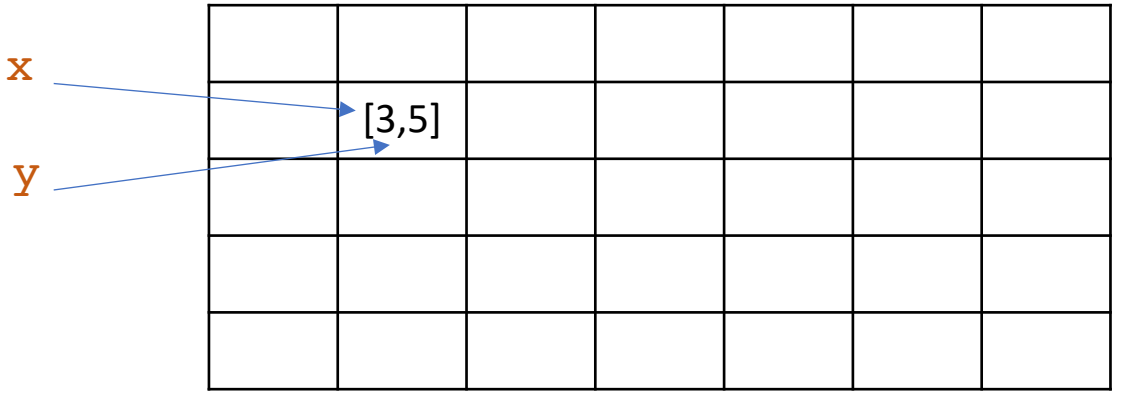
3. Next instruction in the high-level program:

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

   - Internal to pyhton / computer:
     variable `y` gets the reference held by `x, y` and `x` are bound to the **same memory location** for their value




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

# (Optional) Cloning vs. Aliasing: Issues, opportunities of mutability

- What about initializing a list with another list?

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

primes[1] = 29
```

what about `numbers`?

- What about initializing a list with the contents of another list using a range?

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

primes[1] = 29
```

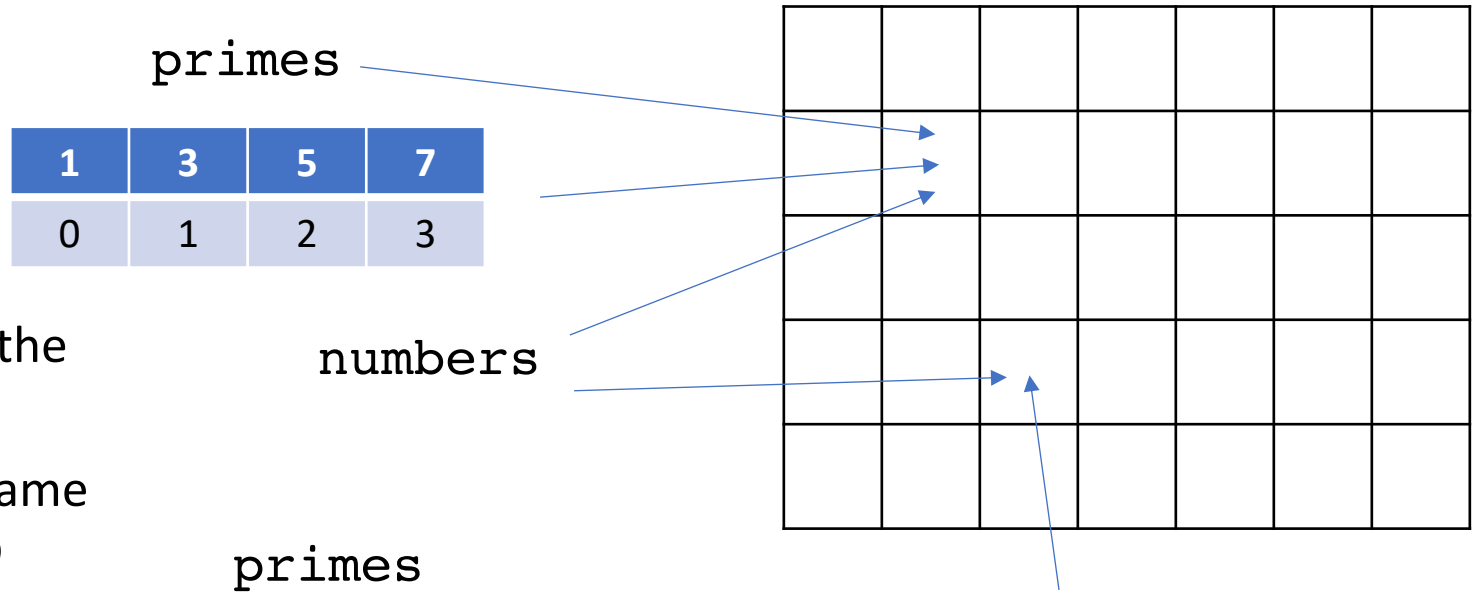
what about `numbers`?

```
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))
```

# (Optional) Cloning vs. Aliasing: Issues, opportunities of mutability

Two different ways of *copying* between data variables ( transfer data from one variable to another)

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



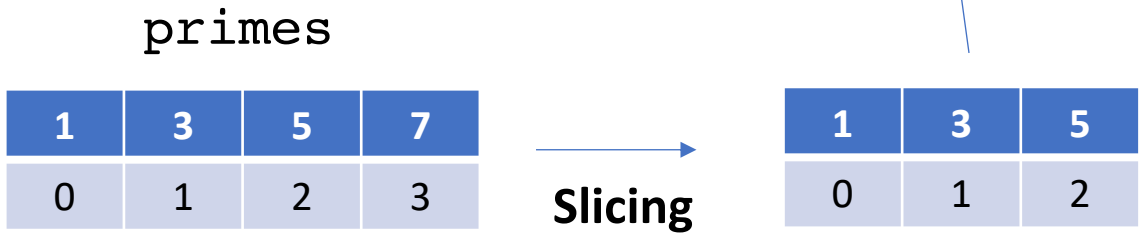

✓ Variables are passed by identity, by *reference* address → **Aliasing**

➢ `numbers` and `primes` are *alias* 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!**

# (Optional) Cloning vs. Aliasing: Issues, 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!

# (Optional) Cloning vs. Aliasing: Issues, 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 <u>immutable</u>: 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

# (Optional) Cloning vs. Aliasing: Issues, 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

# (Optional) Cloning vs. Aliasing: Issues, opportunities of mutability

## When to use cloning vs. aliasing 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 prime numbers can be added to `primes`, not unaffecting `basic_primes`

# Useful operations: reverse() method

- `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]

- Other way to obtain the same macroscopic result using `[ ]` operator:

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

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

- Watch out: a list with a *new* identity is being created (but the *macroscopic* effect is the same)

```
numbers = [1, 4, 2, -7, 0, 6]
print(id(numbers))          → 4729970376
numbers = numbers[::-1]
print(id(numbers))          → 4729921992
```

→ identity values (*memory addresses of list's content*) will be different on different executions / computers

# Useful operations: sort () method

- `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

# Useful operations: sort () method

- Note: A *list of lists/tuples of primitive types* is sorted according to the first element(s) of each list/tuple

```
my_tuples = [(1,2), (5,7,8), (-1,), (0,9,1,3)]
my_tuples.sort()
```

→ `[(-1,), (0, 9, 1, 3), (1, 2), (5, 7, 8)]`

```
my_tuples = [(-1,2), (5,7,8), (-1,3), (0,9,1,3)]
my_tuples.sort()
```

→ `[(-1,2), (-1,3), (0, 9, 1, 3), (5, 7, 8)]`

- *Ties* do not matter since the items become indistinguishable

```
my_tuples = [(-1,2), (5,7,8), (-1,2), (0,9,1,3)]
my_tuples.sort()
```

→ `[(-1,2), (-1,2), (0, 9, 1, 3), (5, 7, 8)]`