15-110 Principles of Computing – S19

Lecture 6:
Lists and Tuples Data Structures

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Tuple and Lists: Generalization and extension of strings

- **String:** (non-scalar type) ordered sequence of *characters,* immutable

H	e	l	l	o		J	o	e
0	1	2	3	4	5	6	7	8

H	e	l	l	o		J	o	e
-9	-8	-7	-6	-5	-4	-3	-2	-1

- **Indexing**: access (read) individual characters

```
s =“Hello Joe”
print(s[0], s[4], s[-2])
```

- **Slicing, Striding**: access subsequences of characters

```
s =“Hello Joe”
print(s[0:3], s[2:4], s[0::2], s[::-1])
```

- **Immutable:** cannot change its value(s)

```
s =“Hello Joe”
s[3] = ’L’
s[1:4] = ‘abc’
```

X Not allowed

```
s =“Hello Joe”
s = “New hello”
```

✓ *Generalization of strings*

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

Tuple type

- **Tuple:** (non-scalar type) ordered sequence of *objects* (any type, not need same type), immutable
- Purpose: ***group*** **items together in the same (immutable) object**
- **Syntax:** `(a,b,c, …, )`
  - Examples of tuple variables assigned with tuple literals

```
prime_numbers = (1, 3, 5, 7, 11)
irrational_numbers = (2.71828, 3.14159, 1.41421)
fruits = ('apple', 'pear', 'banana', 'orange')
colors = ('red', 'blue', green')
person_info = ('Donald', 'Trump', 14, 'June', 1946, 'President')
fruit_info = ('melon', 3.6, 'yellow', 12.5)
logical_values = (True, False, True, 255)
empty_sequence = ()
one_element_sequence = (5, )  → one_integer_var = (5)
```

Tuple type

- **Syntax:** `(a,b,c, …, )`
- … but also **any comma-separated sequence of objects**: `a, b, c, d`

```
prime_numbers = 1, 3, 5, 7, 11
irrational_numbers = 2.71828, 3.14159, 1.41421
fruits = 'apple', 'pear', 'banana', 'orange'
colors = 'red', 'blue', green'
person_info = 'Donald', 'Trump', 14, 'June', 1946, 'President'
fruit_info = 'melon', 3.6, 'yellow', 12.5
logical_values = True, False, True, 255
one_element_sequence = 5,
```

`a,b,c = 1,3,6.` → three variables of `int` type

`a = 1,3,6` → one variable of `tuple` type

List type

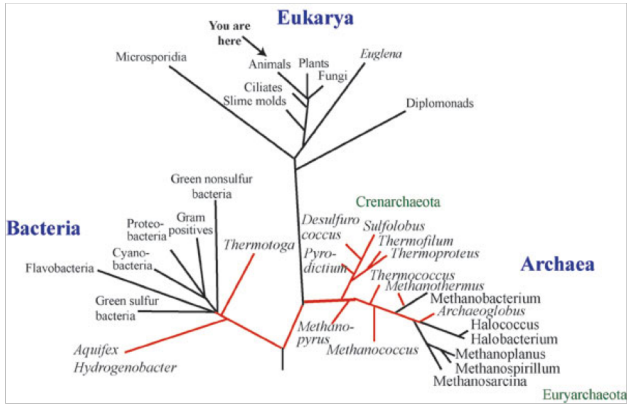
- **List:** (non-scalar type) ordered sequence of *objects* (any type, not need same type), mutable
- Purpose: ***group*** **items together in the same (mutable) object**
- **Syntax:** `[a,b,c, …, ]`

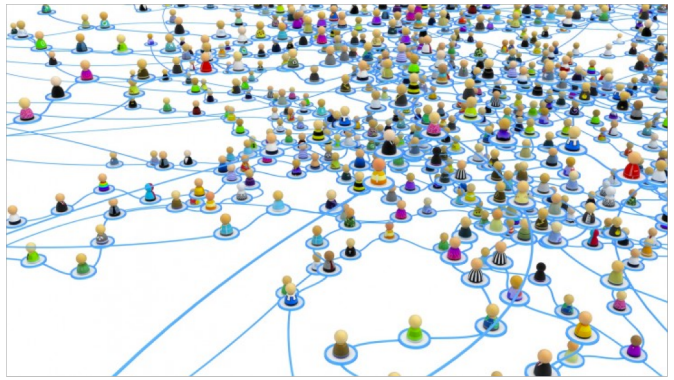
➢ Examples of list variables assigned with list literals

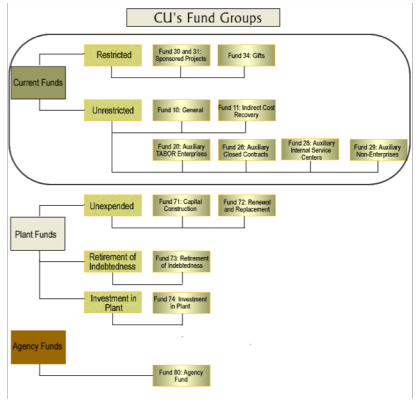
```
prime_numbers = [1, 3, 5, 7, 11]
irrational_numbers = [2.71828, 3.14159, 1.41421]
fruits = ['apple', 'pear', 'banana', 'orange']
colors = ['red', 'blue', green']
person_info = ['Donald', 'Trump', 14, 'June', 1946, 'President']
fruit_info = ['melon', 3.6, 'yellow', 12.5]
logical_values = [True, False, True, 255]
empty_list = []
one_element_list = [5]  or  one_element_list = [5,]
list_of_lists = [1, 'sedan', ['Toyota', 'Corolla', 1.8, 2012], 52000]
```

List and tuples: fundamental *data structures* in python

- **Data structure: Container used to to store data (information) in a specific organized form, a *layout***
- A given *layout* determines a data structure to be efficient in some operations and/or effective in the representation of parts of information, and maybe inefficient / ineffective in others
- E.g., the layout affects how we access *access / find / change / add / remove / sort/organize* data

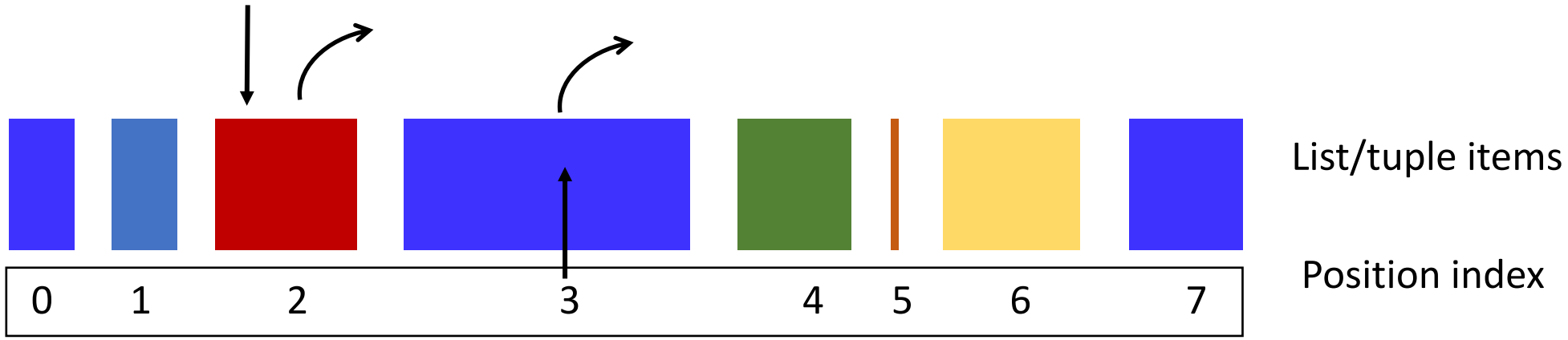






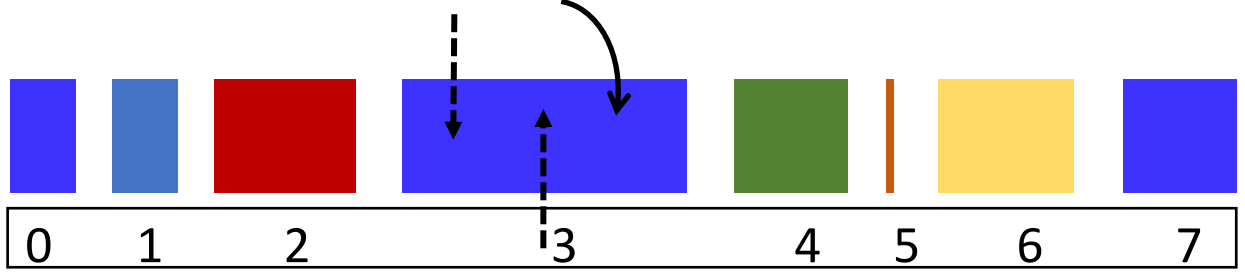
List and tuples data structures: basic operations

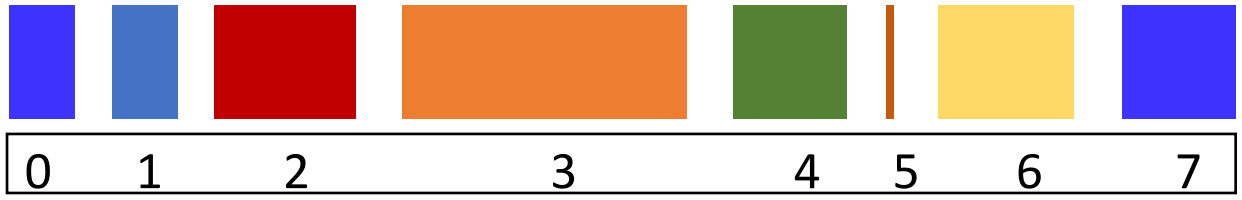
- **Data structure: Container used to to store data (information) in a specific organized form, a *layout***
- List/tuple layout is *linear*: based on a sequential arrangement of the data → position indexes
  - **Access (read) operations**, at any position, for both lists and tuples:
    - get data by index (in 2 steps time)
    - find data by content (linear number steps for scanning the list, depending on item's position)

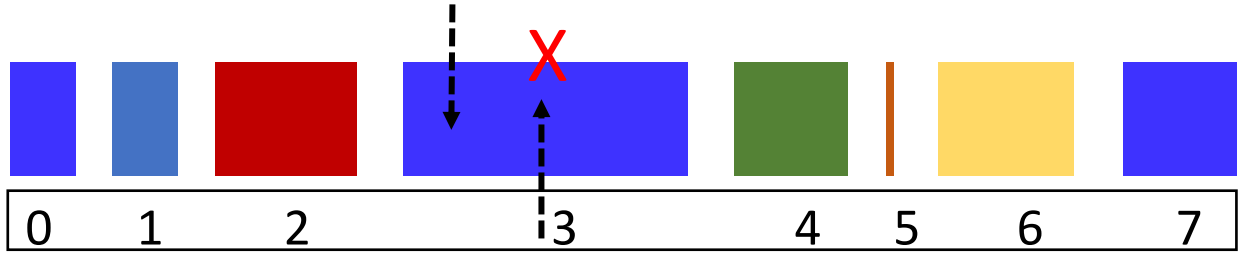


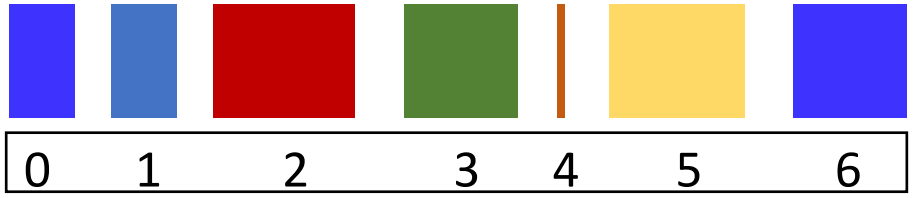
List data structures: basic operations

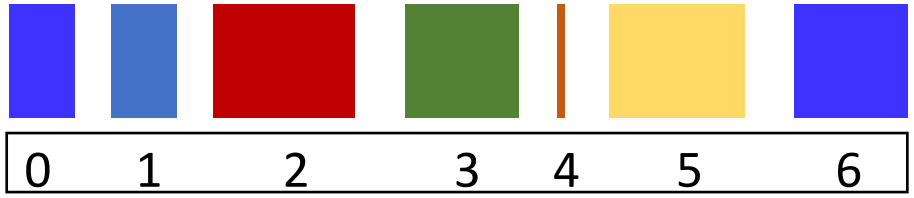
- **Update operations**, *at any position*, only for lists:
  - **change** the value of an item
  - **remove** one item
  - **add** a new item anywhere (at the end, at the beginning, in between)

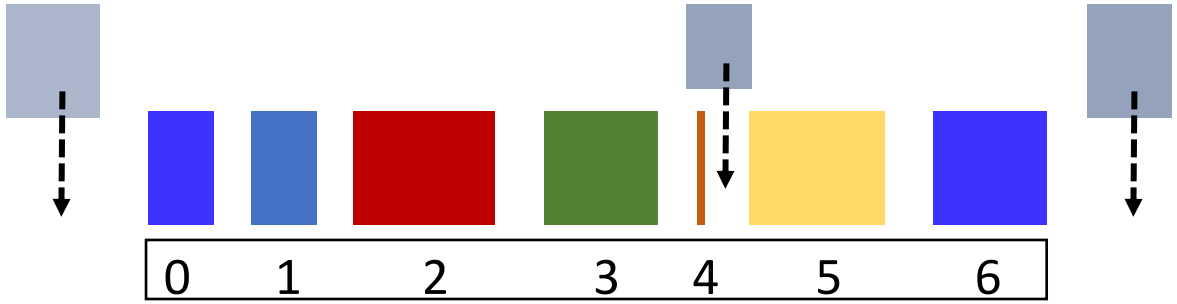




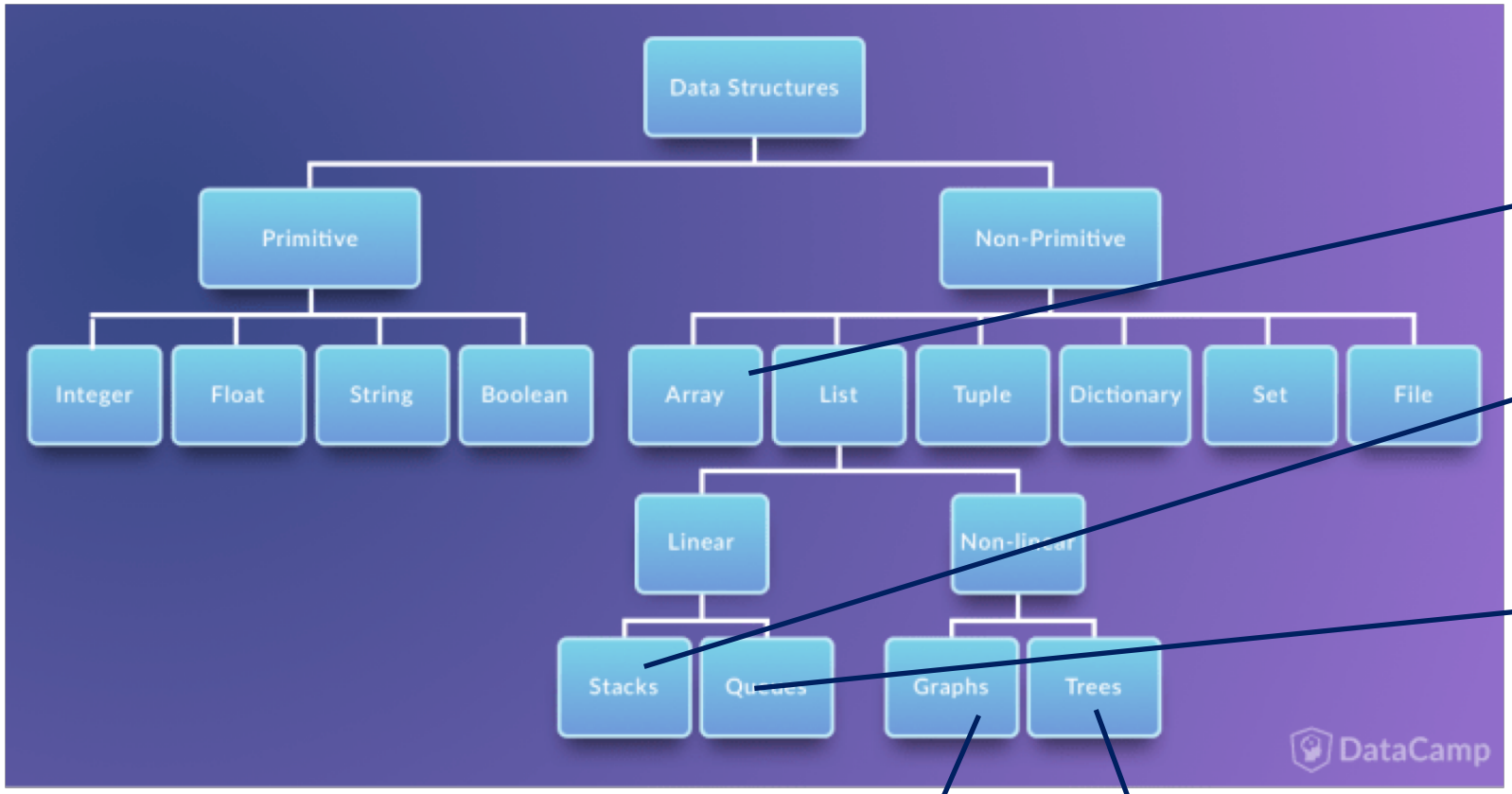


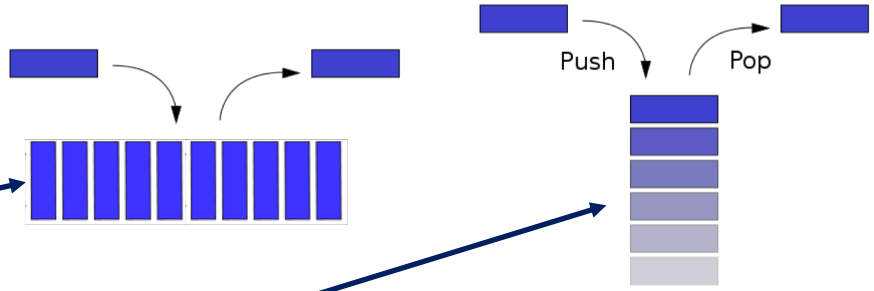


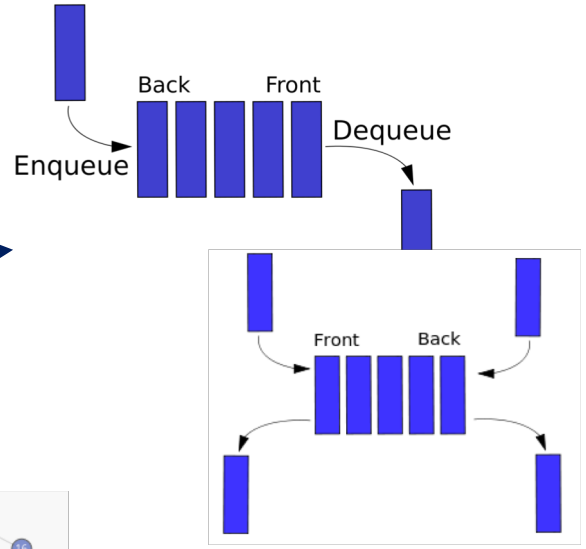


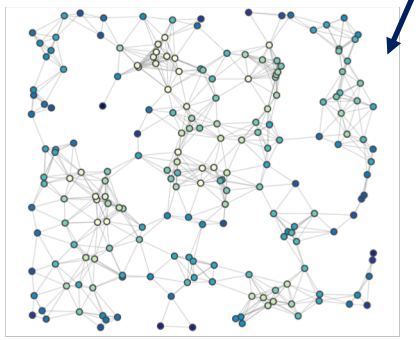


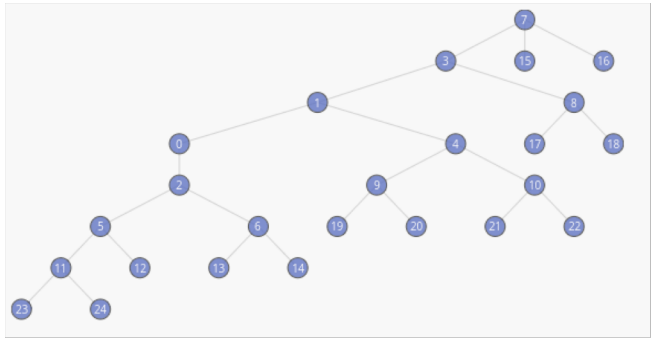
Other data structures in python (building on lists)











Accessing tuple / list values: [], [:], [::]

- **Syntax:** the same used for strings
  - *Single* tuple/list element: `[index]`
  - *Subsequence* of the tuple/list: `[start : end]`
  - *Subsequence with a stride*: `[start : end : step]`

tuple

```
prime_numbers = (1, 3, 5, 7, 11)
```

or

list

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

```
x = prime_numbers[0]   → x is of type int and has value 1
x = prime_numbers[2]   → x is of type int and has value 5
x = prime_numbers[-3]  → x is of type int and has value 5
x = prime_numbers[1:3] → x is of type tuple/list and has value (3,5)
                         corresponding to the tuple/list elements of index 1 and 2
x = prime_numbers[1:4:2] → x is of type tuple/list and has value (3,7)
                           corresponding to the tuple/list elements of index 1 and 3
```

Accessing tuple/list values that are inside tuples/lists

```
cars = [1, 'sedan', ['Toyota', 'Corolla', 1.8, 2012], 52000]
```

- How do we access the item `Corolla` in the list variable `cars`?

```
model = cars[2][1]
```

Equivalent to:

```
model = (cars[2])[1]
```

```
['Toyota', 'Corolla', 1.8, 2012]
```

```
'Corolla'
```

- How do we access to the 4rd character in the `model` variable? (of type `str`)

Equivalent to:

```
model = ((cars[2])[1])[3]
```

```
model3 = cars[2][1][3]
```


```
['Toyota', 'Corolla', 1.8, 2012]
```

```
'Corolla'
```

```
'o'
```

Updating tuple values: No!

- **Updating tuple values: No! tuples are immutable types,** like strings
- *Invalid operations:*

```
prime_numbers = (1, 3, 5, 7, 11)
```

`prime_numbers[4] = 13` → an existing element cannot be modified

`prime_numbers[5] = 13` → it cannot *be extended* to an additional element

- "Update" the tuple by creating a new tuple from the existing one

```
prime_numbers = (1, 3, 5, 7, 11)
new_prime_numbers = (1, 3, 5, 7, 13)
new_prime_numbers = prime_numbers[0:4] + (13, )
new_prime_numbers = prime_numbers + (13, 17)
```

Updating list values: [], [:], [::]

- **Updating list values**: yes, they are mutable types, syntax is `L[index] = new_value`

`colors = ['red', 'green', 'blue', 'cyan']`

`colors[1] = 'yellow'` → same `colors` list, with modified value, `'yellow'`, for item in position 1

`colors[0] = None` → same `colors` list, with modified value and type, `None`, for item in position 0

`colors[4] = 'purple'` → error! the list doesn't include an item at position 4 and the list cannot be *extended* in this way

`numbers = []` → defines an empty list, `numbers[0]` does not exist (yet)!

`numbers[1]` → error! the list doesn't include an item at position 1 and the list cannot be *extended* in this way

`colors[0:3] = ('brown', 'magenta', 'pink')` → updating a subsequence of adjacent items

`colors[0:3:2] = ('brown', 'magenta')` → updating a subsequence of non-adjacent items

Mutable vs. Immutable types: int

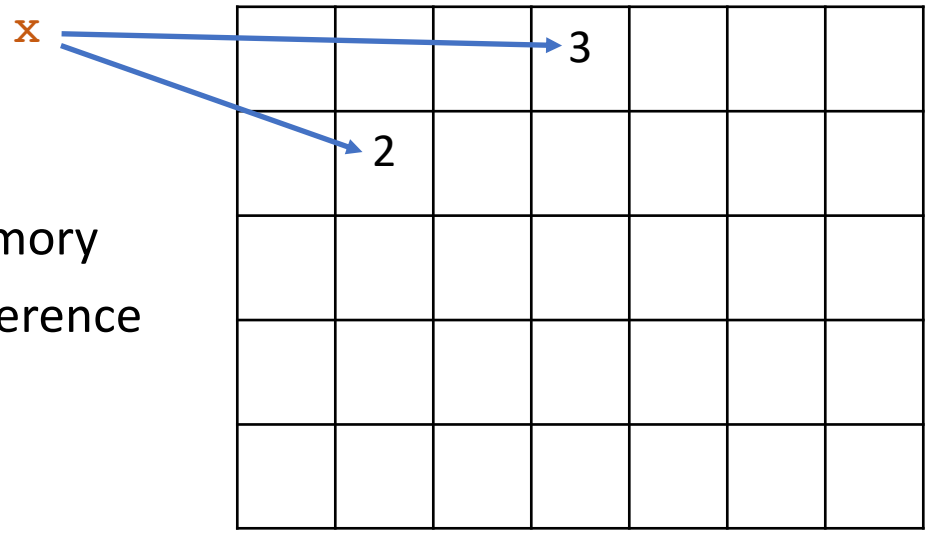
1. In the high-level program:

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

   - Internal to pyhton / 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 pyhton / 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)`!)

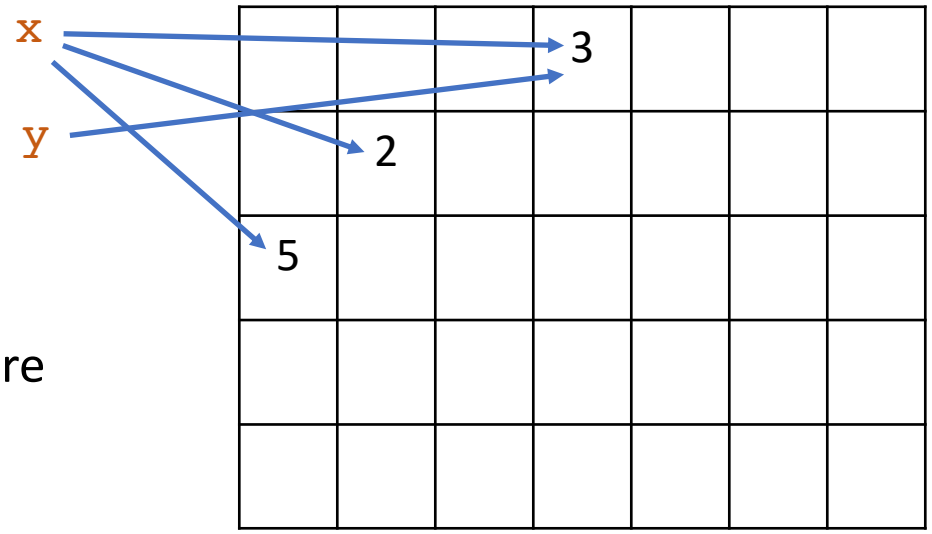
Mutable vs. Immutable types: int

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 pyhton / computer:

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



- `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 pyhton / computer:

     variable `x` gets the new reference associate to 5's memory location

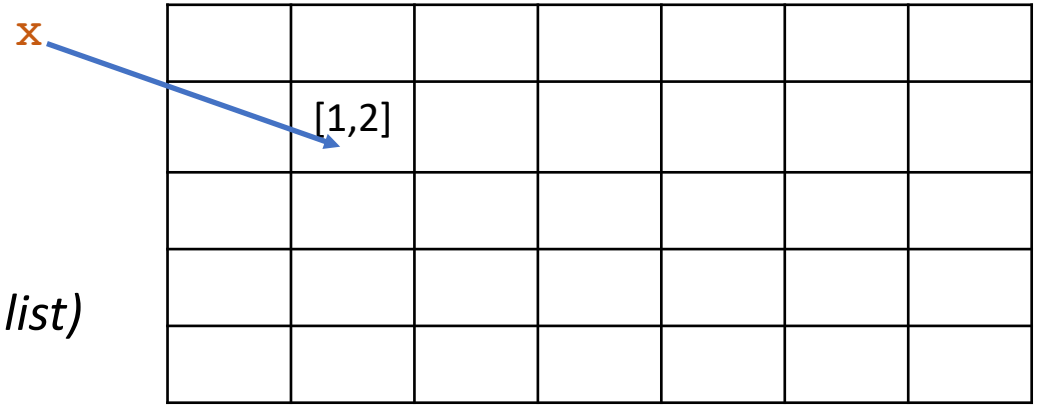
Mutable vs. Immutable types: list

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)*



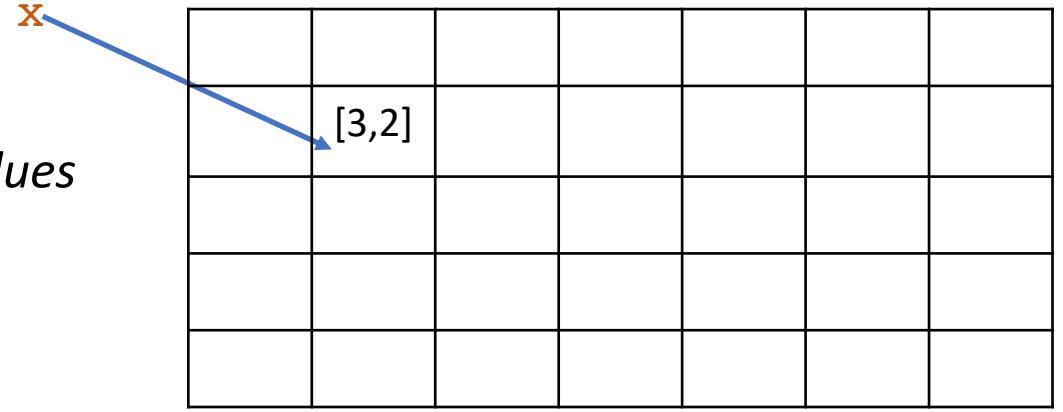
2. Next instruction in the high-level program:

   `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

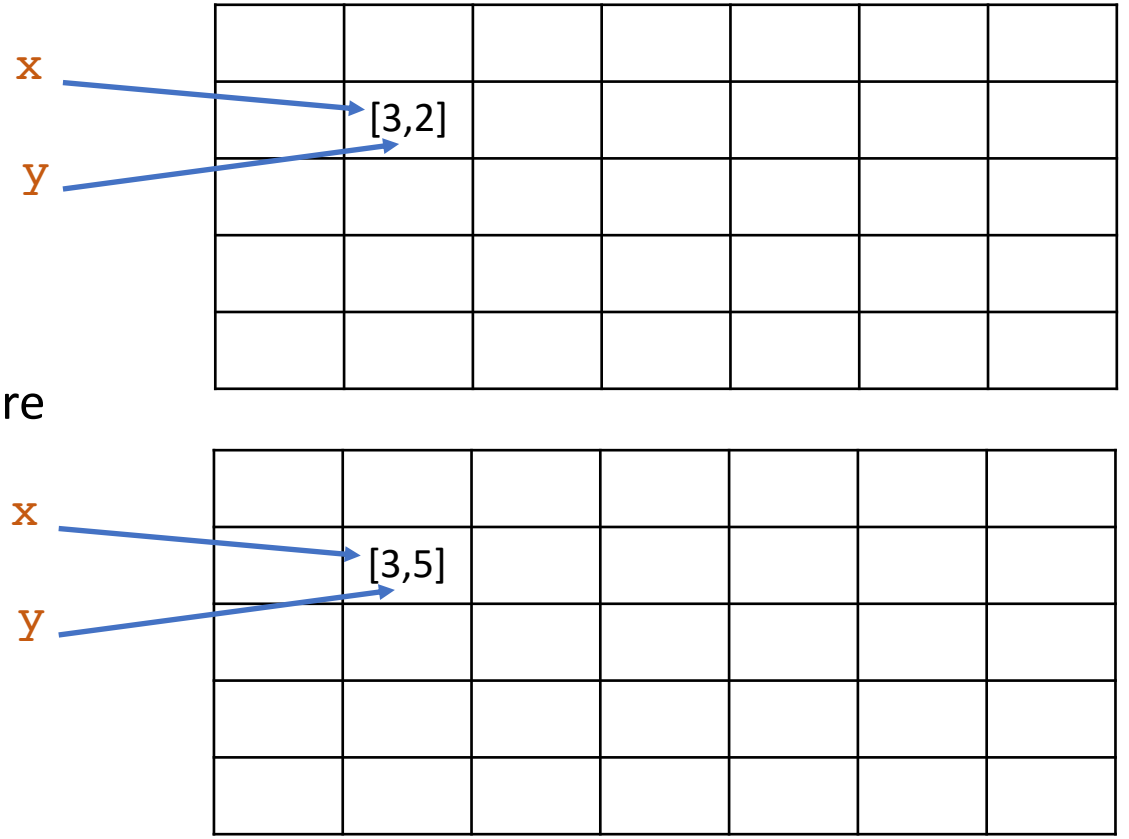


Mutable vs. Immutable types: list

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]

Cloning vs. Aliasing: Issues and 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`?

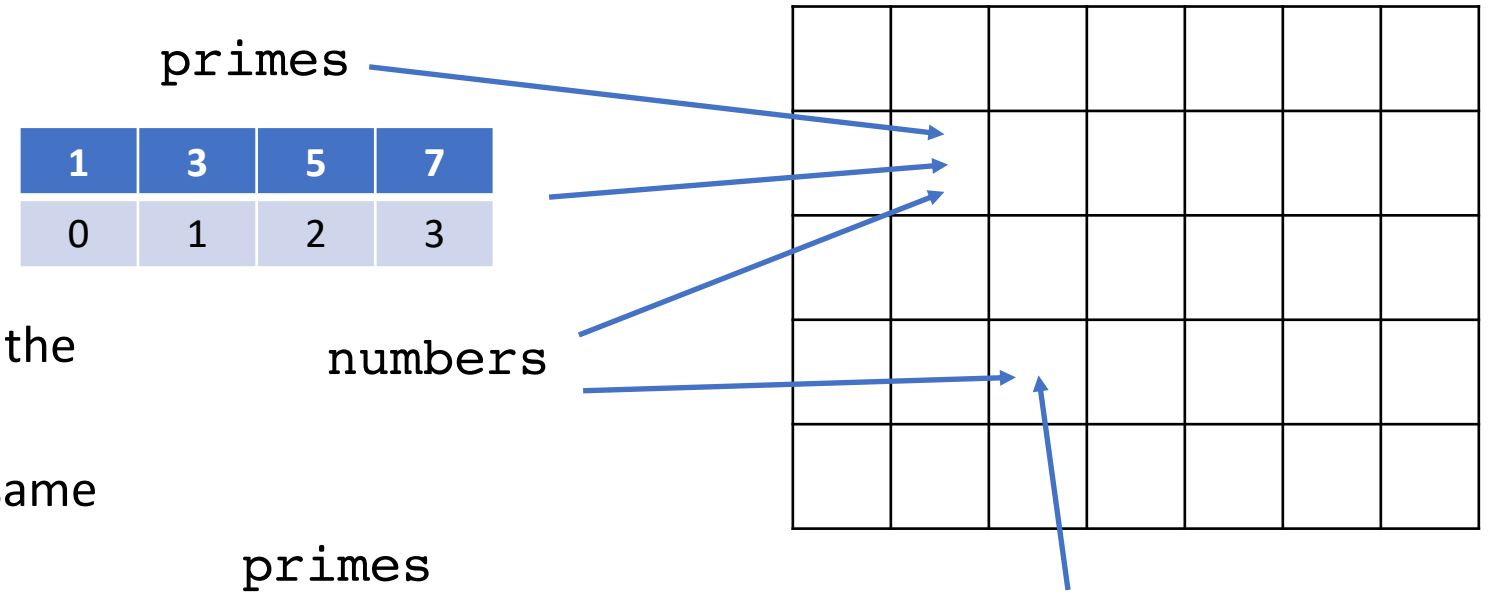
```
print("Do the lists have the same contents?", numbers == primes)
print("Primes:", primes)
print("Numbers:", numbers)
print("Are the two lists the 'same' list?", id(primes) == id(numbers))
```

Cloning vs. Aliasing: Issues and opportunities of mutability

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

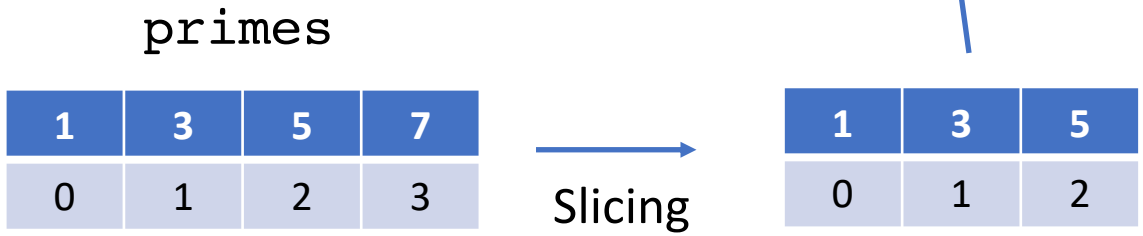
```
primes = [1, 3, 5, 7]
numbers = primes
```

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!

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