Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 18
Lists
(Array Implementation)

FIT 1008
Introduction to Computer Science

Implementing a **List ADT** using arrays

- We will use **Python lists as our arrays**
- This means our implementation can only use the following operations:
  - **Create** an array/list: We will **assume** that the list created has a **fixed size**.
  - Access an item in position P
  - Obtain its length (number of elements already in the array).

Looking under the hood

- Many implementation of lists use arrays

- As we: arrays have **fixed size** (never changes)
  - Needs to be known when they are created
  - The size of an array is always known (kept with the array)
  - All elements are of the same type, or at least occupy same space

- But the number of elements in a list may change!

- So, lists implemented with arrays need two things:
  - The array itself already with a given **big size.** Some cells in the array will be empty (until it is full).
  - The number of elements currently in the list. That is, **how many array positions are used**.

Implementing your own List ADT

- How do we start? Easy:
  - Create a **new file** (called *my_list.py*)
  - Add any **operations/methods** users my need to use.

Implementing your own List ADT

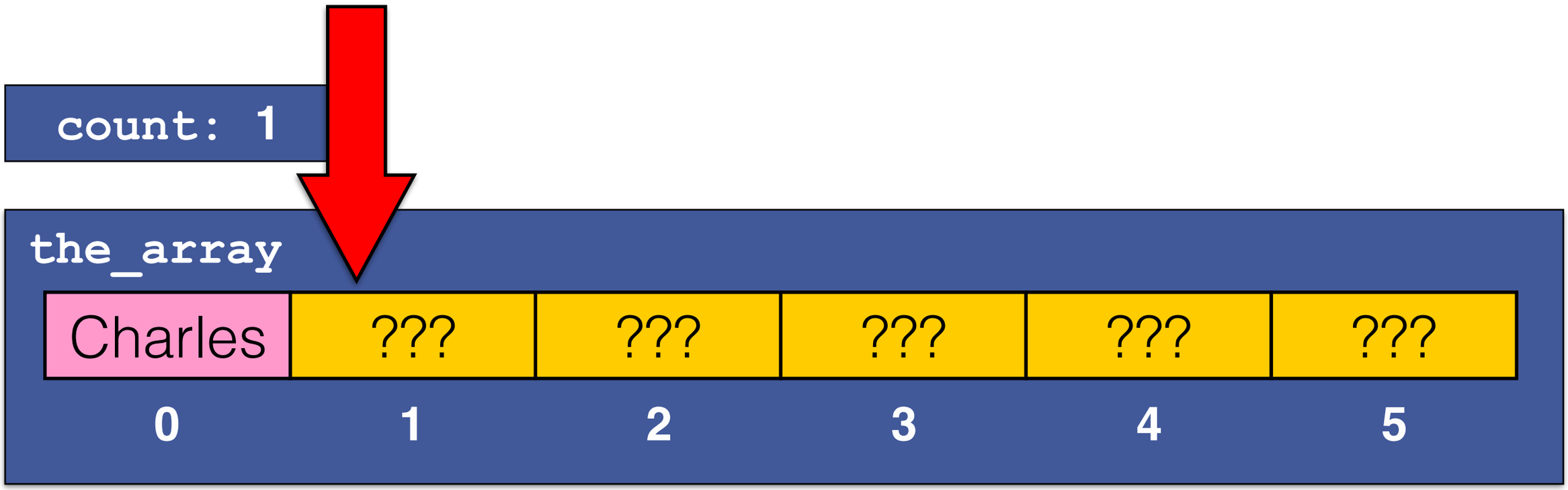
- How do we start? Easy:
  - Create a **new file** (called *my_list.py*)
  - Add any **operations/methods** users my need to use.

- What operations?
  - **Create a list**, **access** an element, compute the **length**
  - Determine whether **is empty**
  - Determine whether it **has a given item**
  - Find the **position of an item** (if in)
  - **Add/delete** an item
  - **Delete/insert** the item in position P

Visualising lists implemented with arrays

- Consider a list defined:
  - Over an array of size 6
  - Currently with one element (Charles)

- We will visualise it like this:

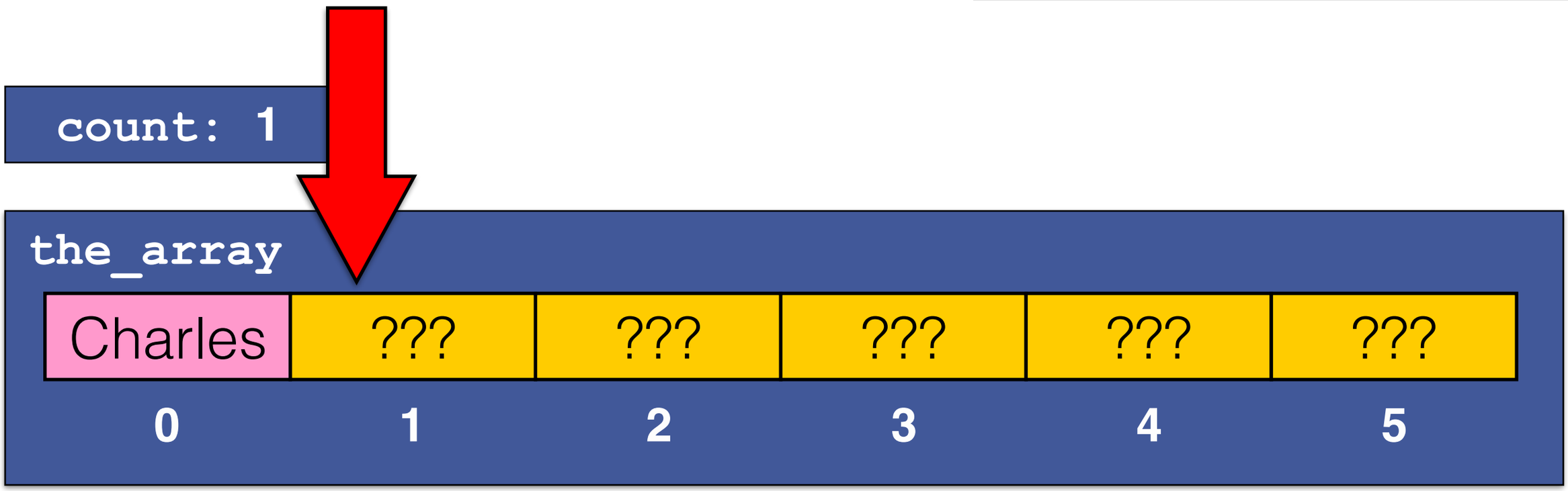


Visualising lists implemented with arrays

- Consider a list defined:
  - Over an array of size 6
  - Currently with one element (Charles)

- We will visualise it like this:

Visual Clarity:
Arrow, **Colour**, Counter (**`count`**) mean same thing…

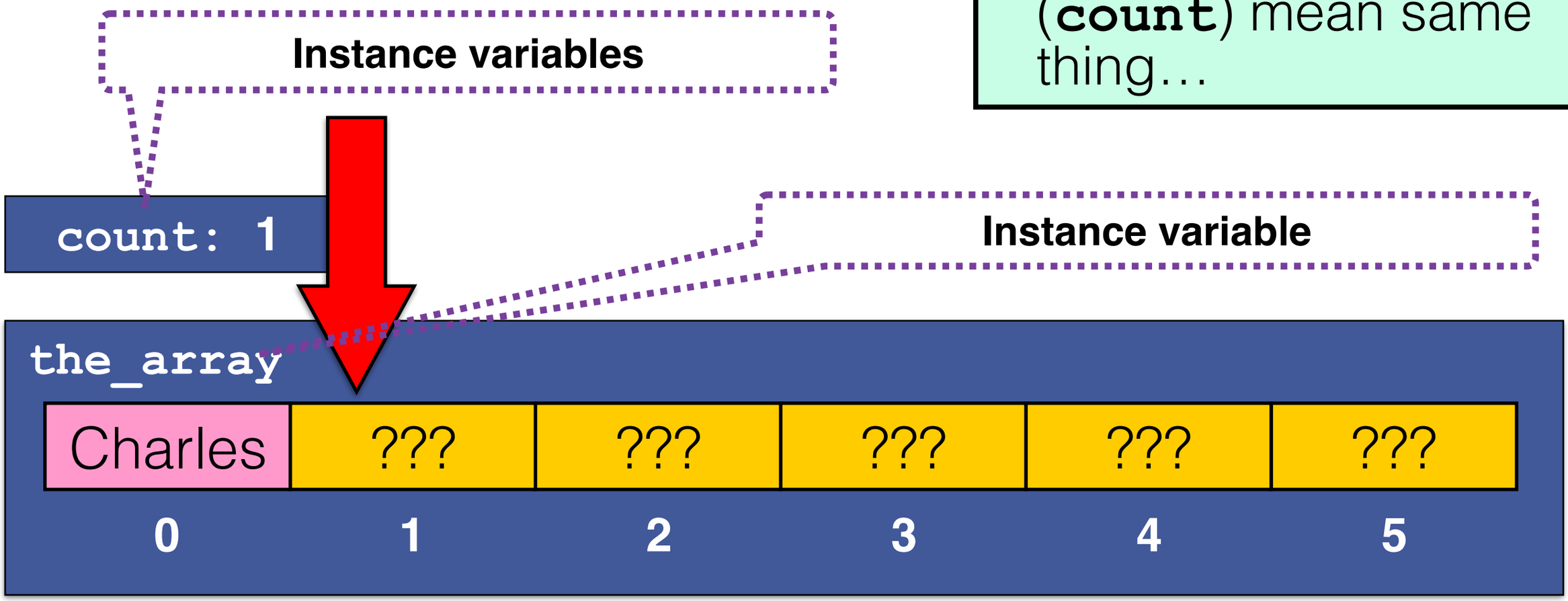


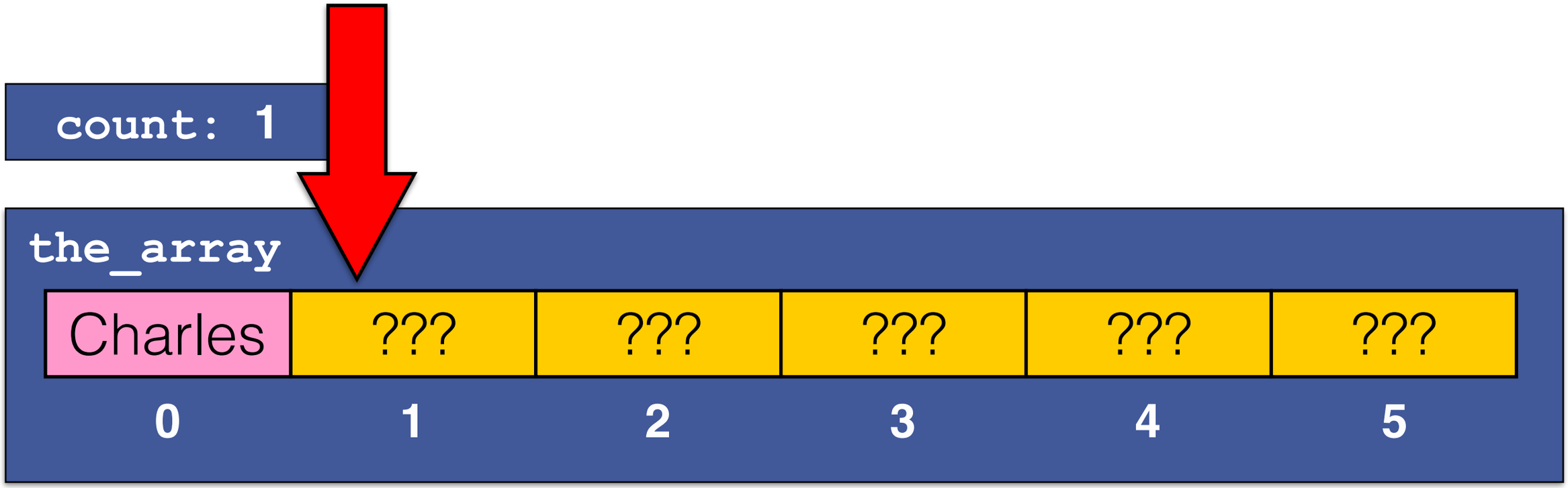
Visualising lists implemented with arrays

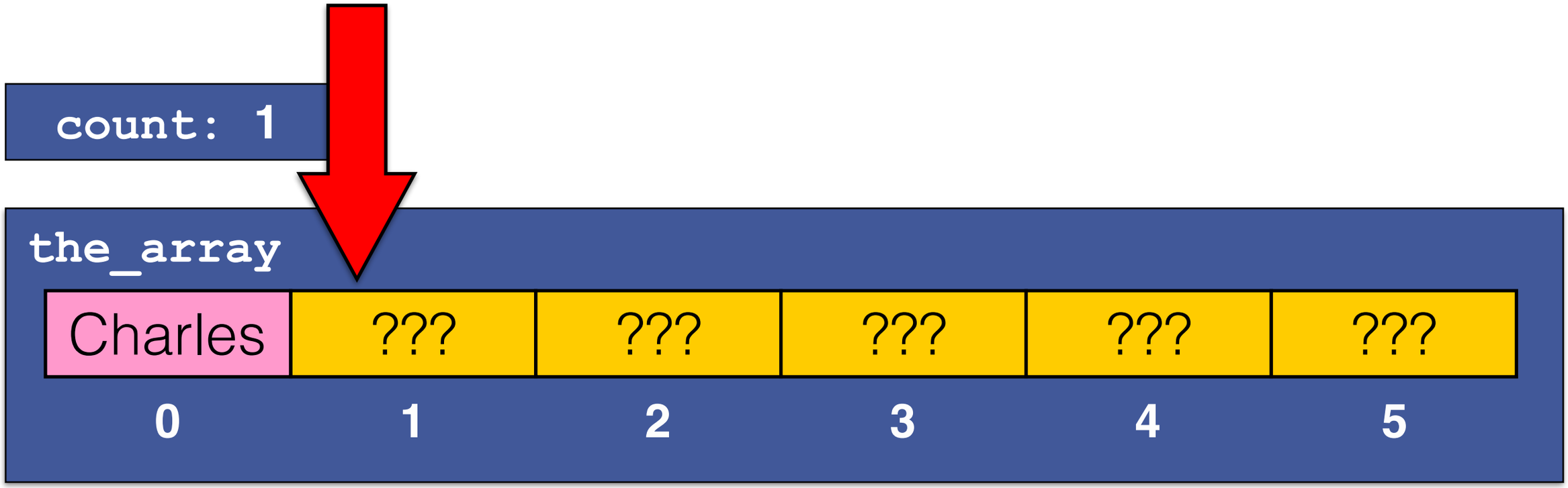
- Consider a list defined:
  - Over an array of size 6
  - Currently with one element (Charles)

- We will visualise it like this:

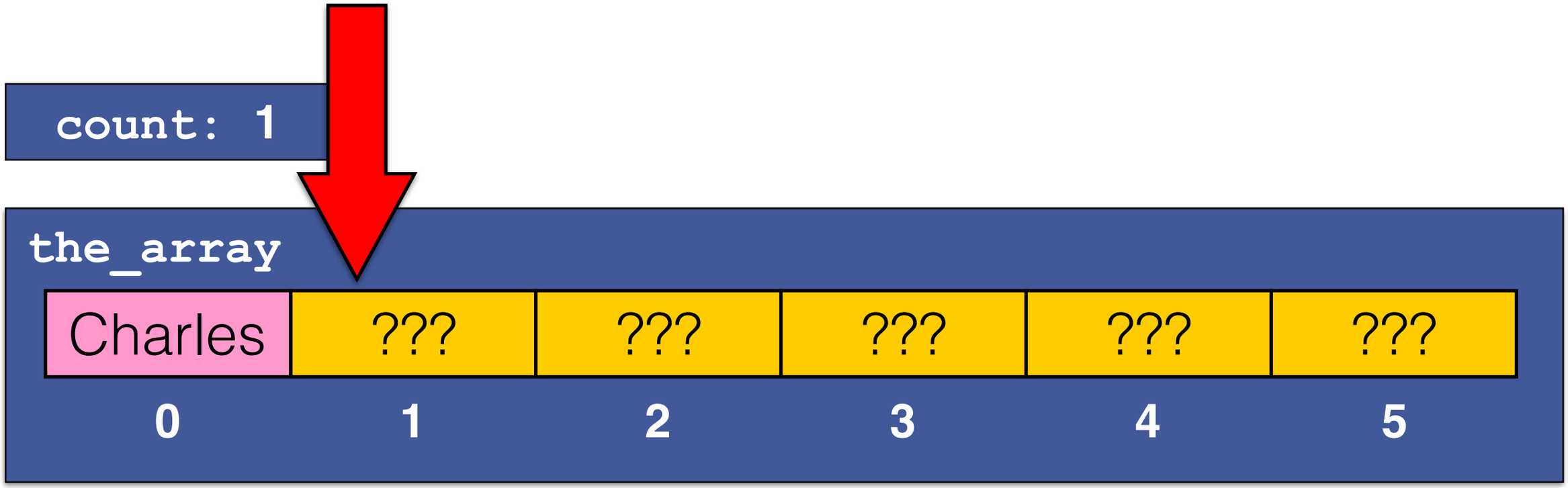
Visual Clarity:
Arrow, **Colour**, Counter (**`count`**) mean same thing…




count: 1
the_array
Charles
???
???
???
???
???
0
1
2
3
4
5



Invariant: count points to the first free position in the array

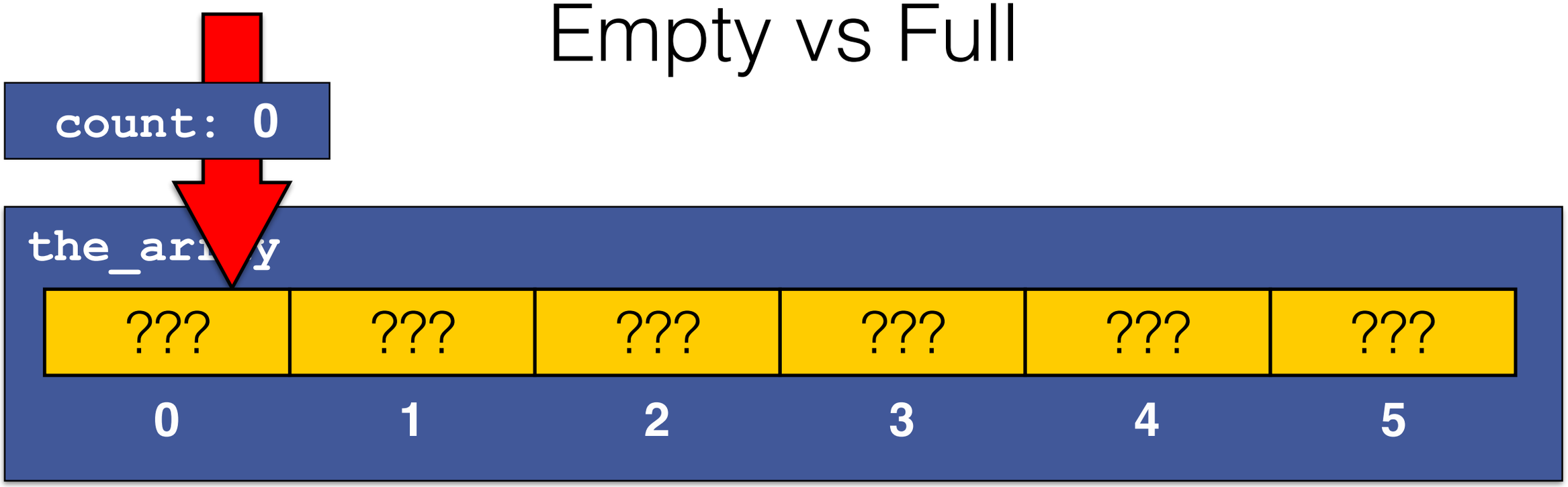


Invariant: count points to the first free position in the array

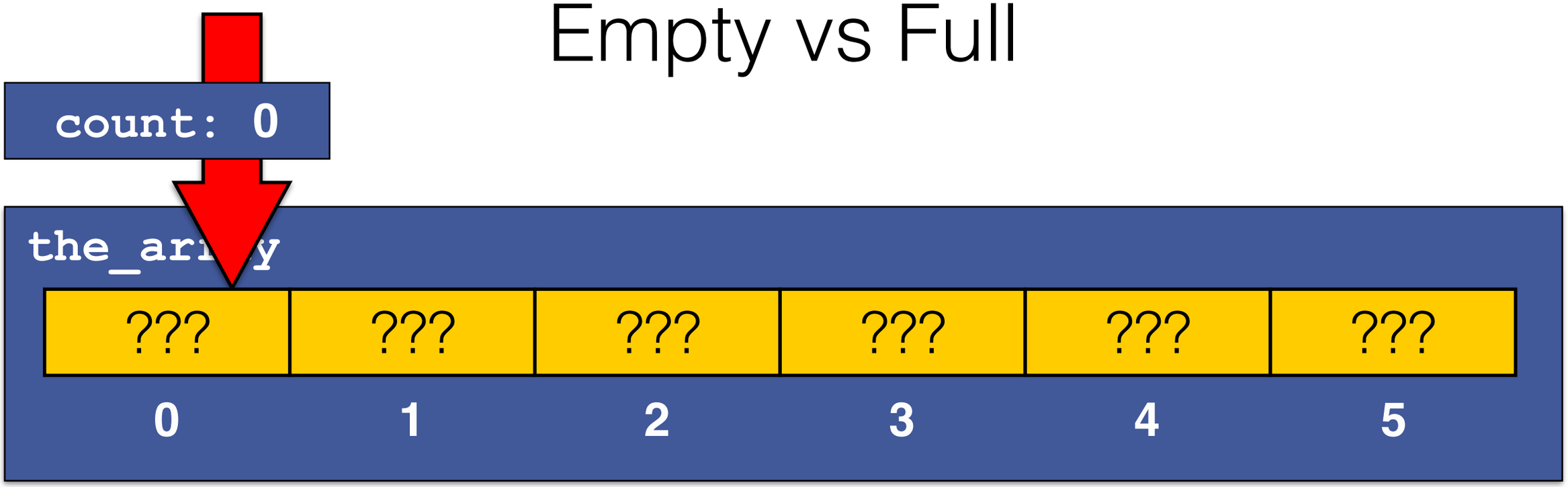
In other words: valid data appear in the 0..count-1 positions

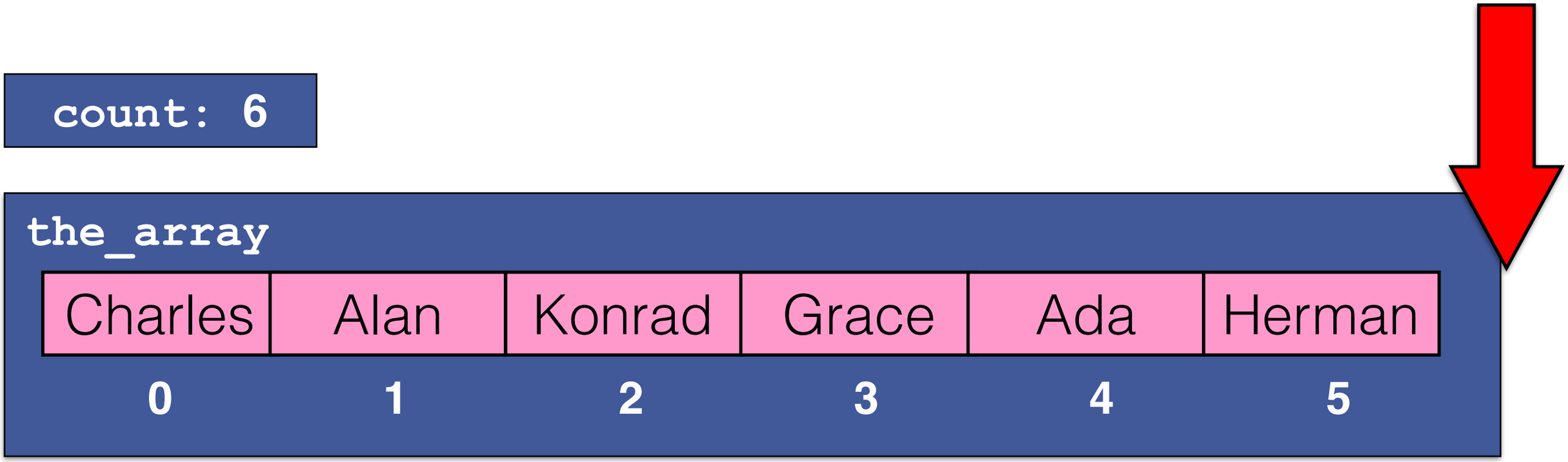
Empty vs Full

Empty vs Full



Empty vs Full





Creating a list

```
class List:
    def __init__(self, size):
        if size > 0:
            self.the_array = size*[None]
            self.count = 0
```

Creating a list

```
class List:
    def __init__(self, size):
        if size > 0:
            self.the_array = size*[None]
            self.count = 0
```

Instance variables

Creating a list

```
class List:
    def __init__(self, size):
        if size > 0:
            self.the_array = size*[None]
            self.count = 0
```

Instance variables

Built-in constant
Absence of a value

Simple methods

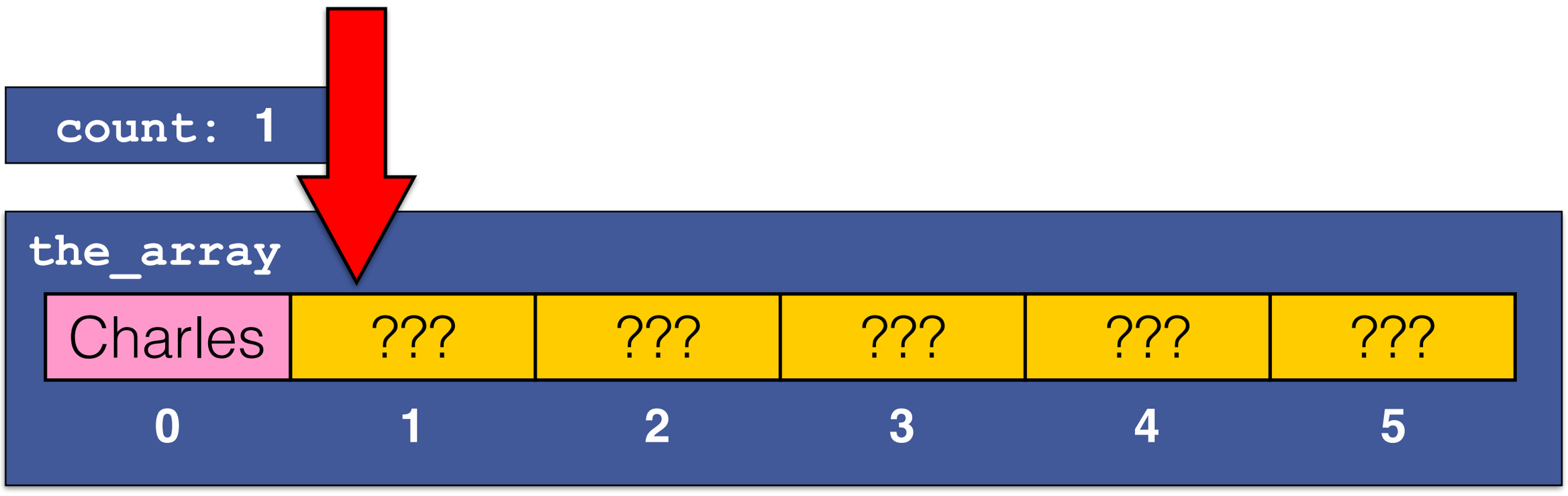
```
def length(self):
    return self.count

def is_empty(self):
    return self.count == 0

def is_full(self):
    return self.count >= len(self.the_array)
```

Adding an element to a list

- **Input:**
  - List (in our case: array + count)
  - **Element to be added**
- **Output:**
  - List
  - Contains all original elements in the same order AND the input one (this is the *post-condition*)

Adding an element

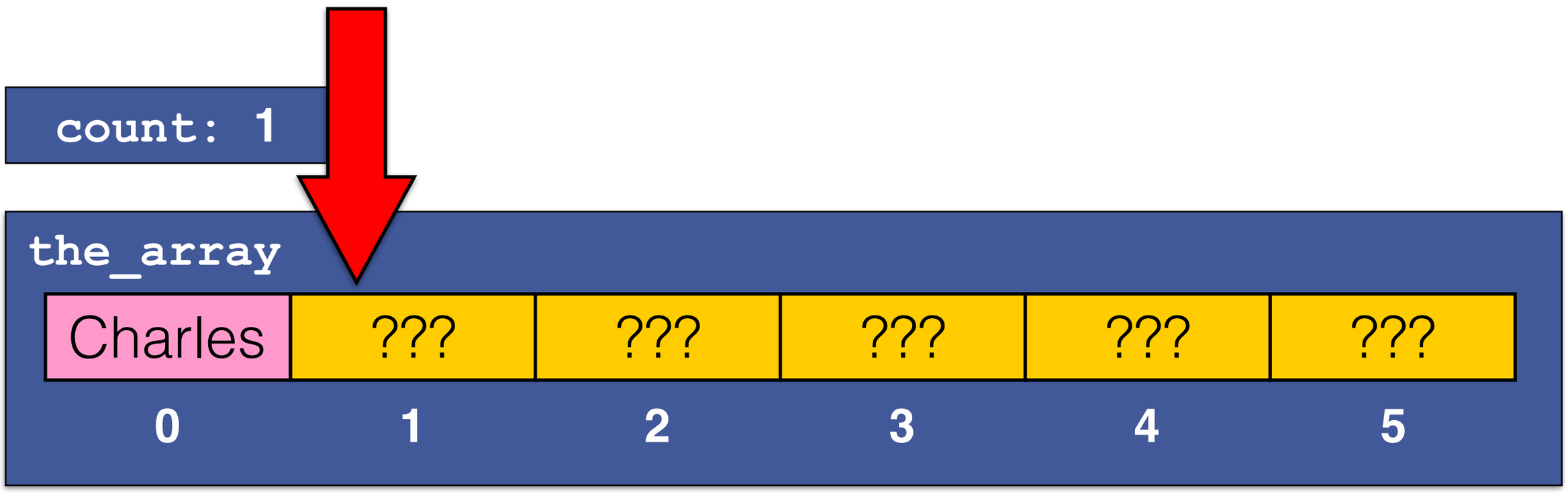
Example: add "Ada"



Adding an element

Recall: count indicates the first empty position (if any)

`Example: add "Ada"`

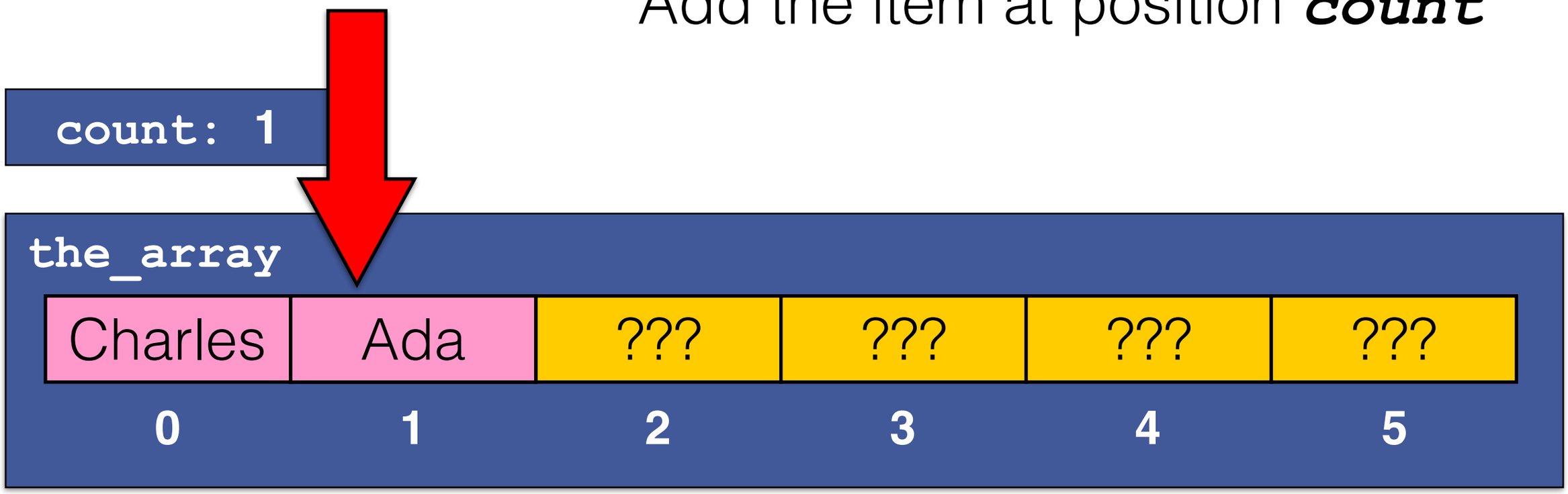


Adding an element

Recall: count indicates the first empty position (if any)

`Example: add "Ada"`

Add the item at position ***`count`***



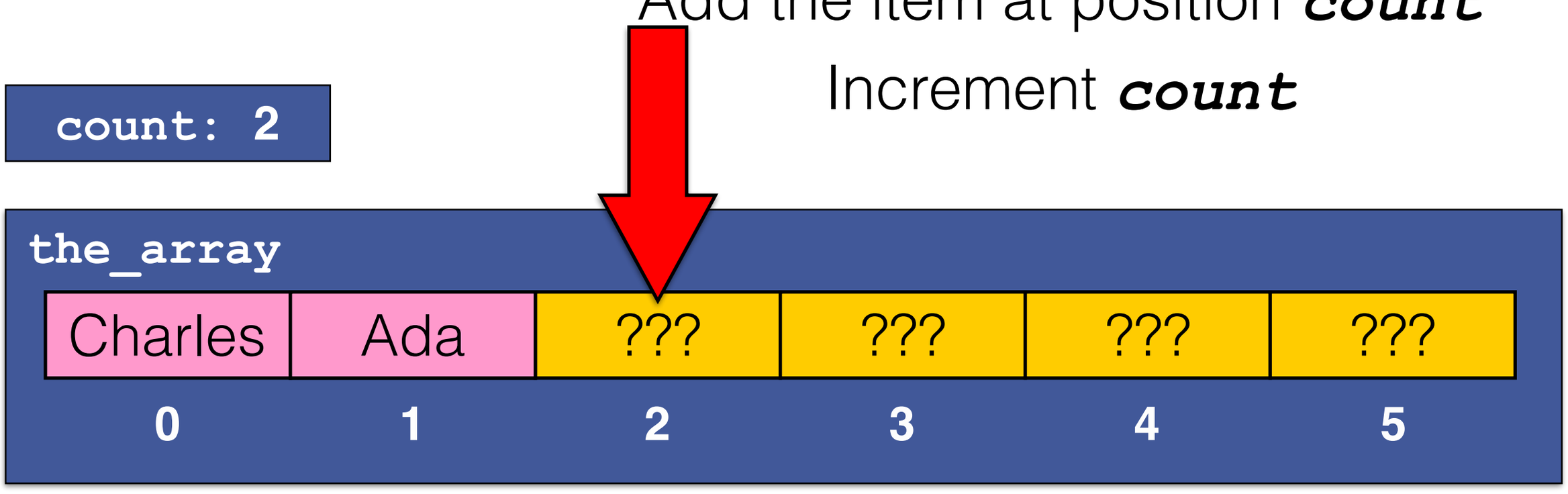
Adding an element

Recall: count indicates the first empty position (if any)

`Example: add "Ada"`

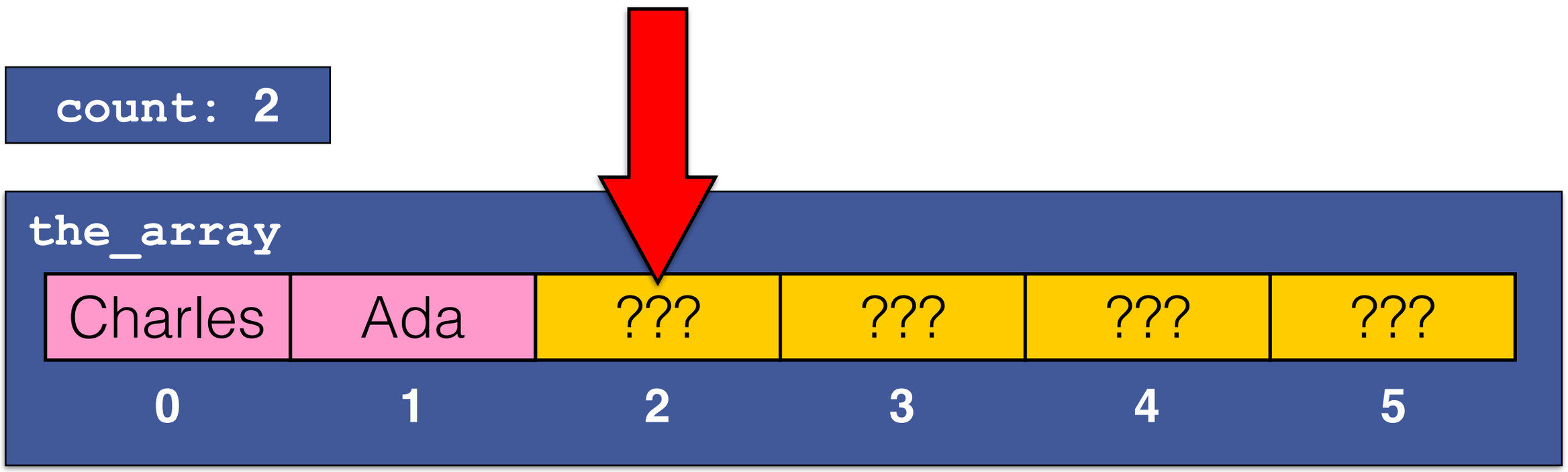
Add the item at position ***`count`***

Increment ***`count`***



Adding an element

- **Why did we add Ada at the end of the list?**
  - Because count gave us easy access to an empty spot
- **Why not at the beginning (position 0)?**
  - Because would have to move Charles somewhere



Adding an element

- Algorithm: add item to the_array, then increment count

Adding an element

- Algorithm: add item to the_array, then increment count
- Does it always work?

Adding an element

- Algorithm: add item to the_array, then increment count

- Does it always work?

- We are assuming we can always add…

Adding an element

- Algorithm: add item to the_array, then increment count

- Does it always work?

- We are assuming we can always add…

- What if it is full? What to do then?

Adding an element

- Algorithm: add item to the_array, then increment count

- Does it always work?

- We are assuming we can always add…

- What if it is full? What to do then?
  - One possibility: return **True** if we can, **False** otherwise
  - This changes the output AND the postcondition

Adding an element

- Algorithm: add item to the_array, then increment count

- Does it always work?

- We are assuming we can always add…

- What if it is full? What to do then?
  - One possibility: return **True** if we can, **False** otherwise
  - This changes the output AND the postcondition
  - Create a new larger array copy things over?

Adding an element

- Algorithm: add item to the_array, then increment count

- Does it always work?

- We are assuming we can always add…

- What if it is full? What to do then?
  - One possibility: return **True** if we can, **False** otherwise
  - This changes the output AND the postcondition
  - Create a new larger array copy things over?
  - What does Python do with its own lists? lists are never full…

Function add

```
def add(self, new_item):
```

Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
```

Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
```

Re-using existing method.

Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
```

Re-using existing method.

Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
    if has_space_left:
        self.the_array[self.count] = new_item
```

Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
    if has_space_left:
        self.the_array[self.count] = new_item
        self.count += 1
```

Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
    if has_space_left:
        self.the_array[self.count] = new_item
        self.count += 1
```

= self.count + 1

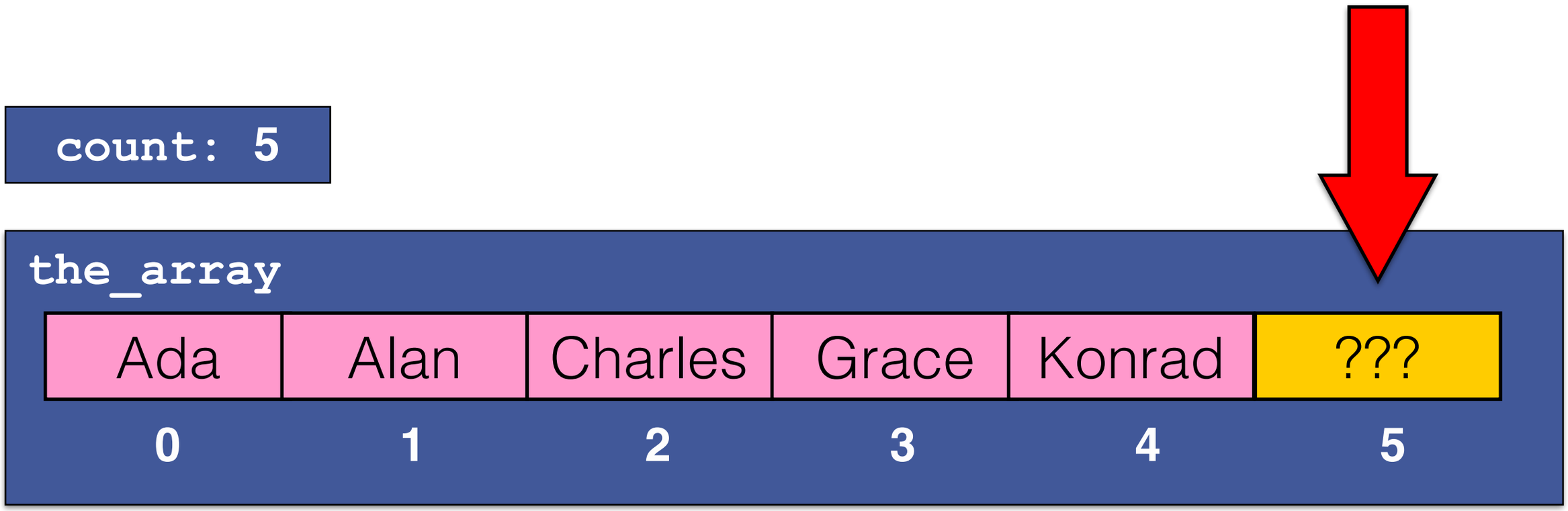
Function add

```
def add(self, new_item):
    has_space_left = not self.is_full()
    if has_space_left:
        self.the_array[self.count] = new_item
        self.count += 1
    return has_space_left
```

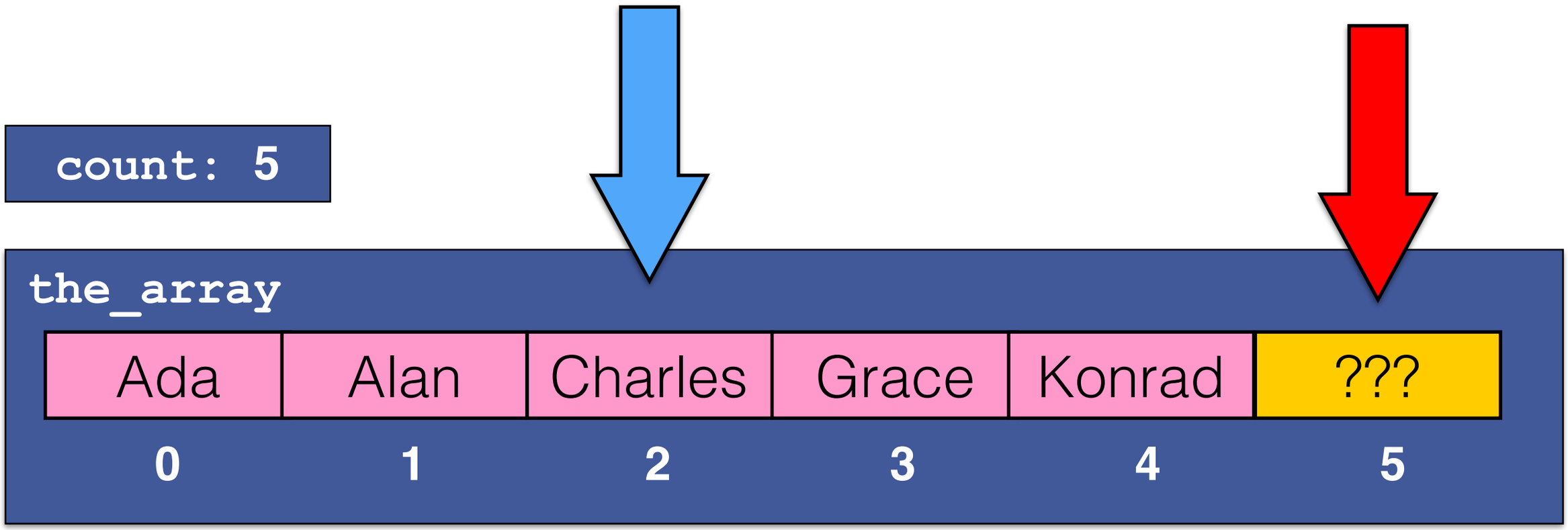
Deleting an element from a list

- **Input:**
  - List (in our case: array + count)
  - **Position of the element to be deleted**
- **Output:**
  - List
  - Contains all original elements EXCEPT the deleted element
  - **Assume:** Remaining elements retain initial ordering.

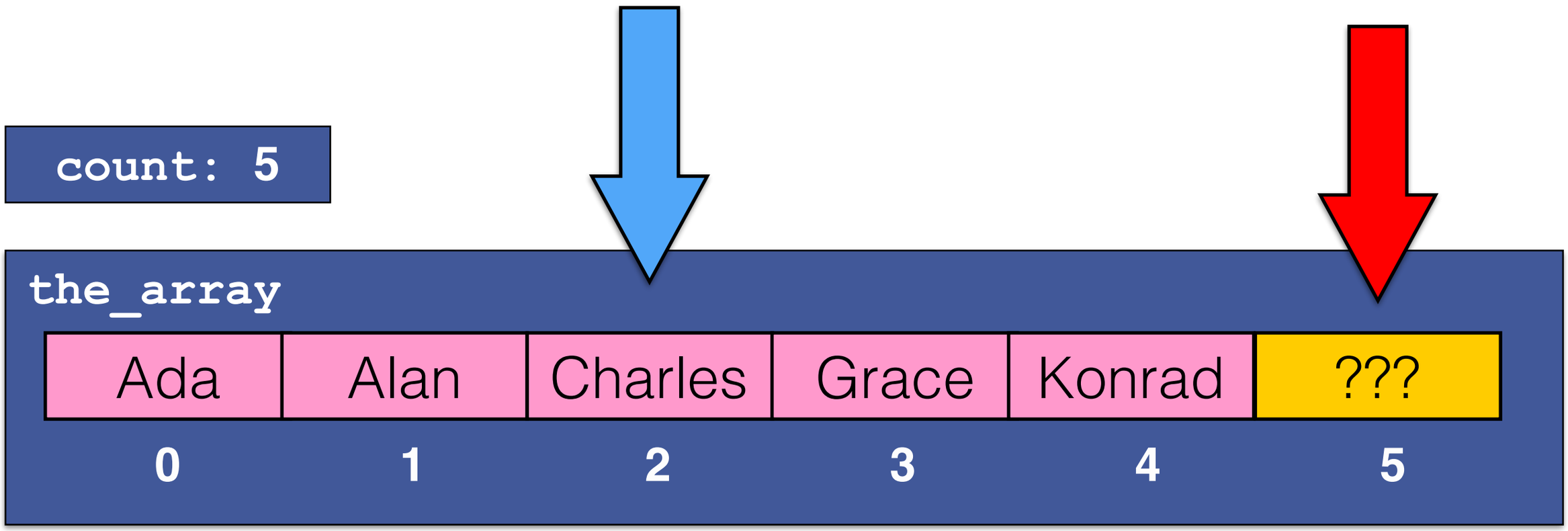
Example: delete item in position 2



Example: delete item in position 2

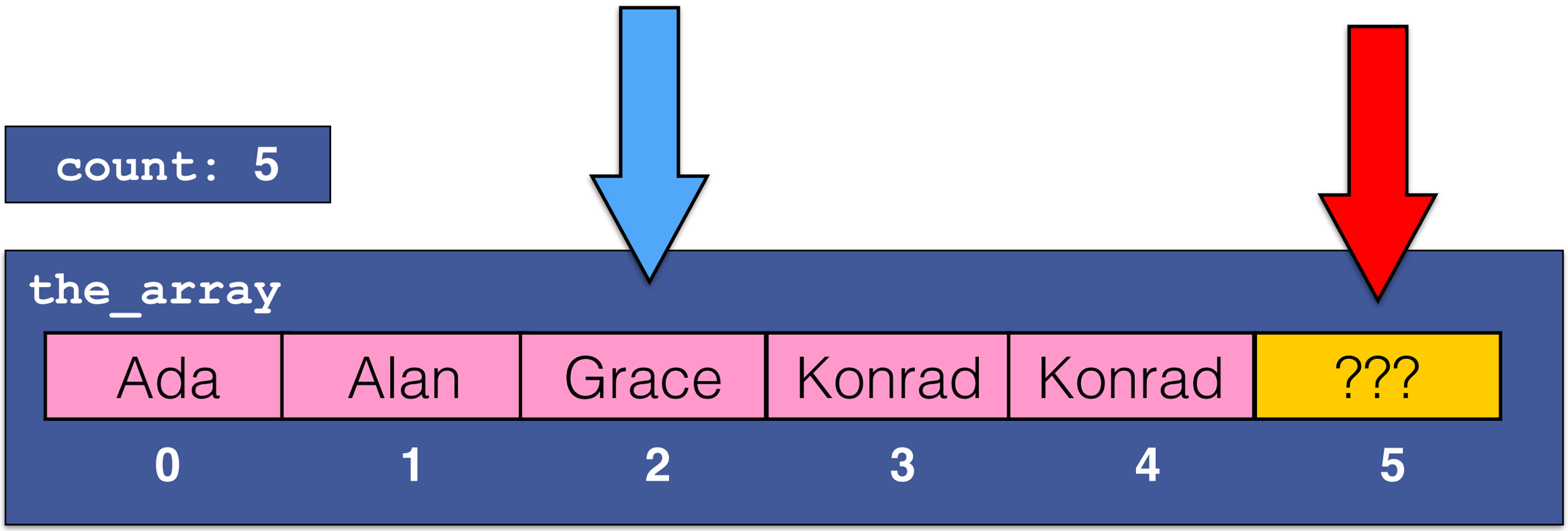


Example: delete item in position 2



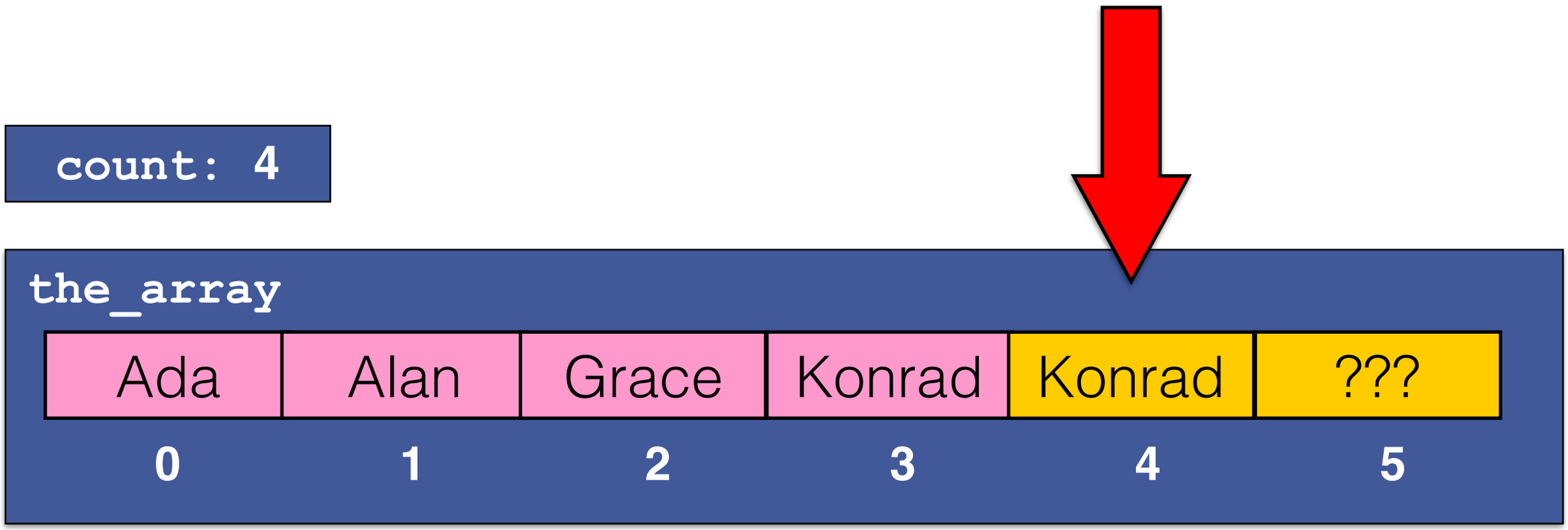
Move items appearing after the deleted item

Example: delete item in position 2



Move items appearing after the deleted item

Example: delete item in position 2



Move items appearing after the deleted item

Decrement count

```
def delete(self, index):
    valid_index = index >=0 and index < self.count
```

```
def delete(self, index):
    valid_index = index >=0 and index < self.count
    if (valid_index):
        for i in range(index, self.count-1):
            self.the_array[i] = self.the_array[i+1]
```

range limits
[ )

```
def delete(self, index):
    valid_index = index >=0 and index < self.count
    if (valid_index):
        for i in range(index, self.count-1):
            self.the_array[i] = self.the_array[i+1]
```

```
def delete(self, index):
    valid_index = index >=0 and index < self.count
    if (valid_index):
        for i in range(index, self.count-1):
            self.the_array[i] = self.the_array[i+1]
        self.count -=1
```

```
def delete(self, index):
    valid_index = index >=0 and index < self.count
    if (valid_index):
        for i in range(index, self.count-1):
            self.the_array[i] = self.the_array[i+1]
        self.count -=1
    return valid_index
```

```
def print(self):
    for i in range(self.count):
        print(self.the_array[i], end=" ")
```

Summary

- Implementing lists using arrays:
  - Class structure for a list
  - Add an element to an unsorted list
  - Delete an element