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

Lecture 22
Queues
(Array Implementation)

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING
This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act).The material in this communication may be subject to copyright under the Act.Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.
Do not remove this notice.

Container ADTs

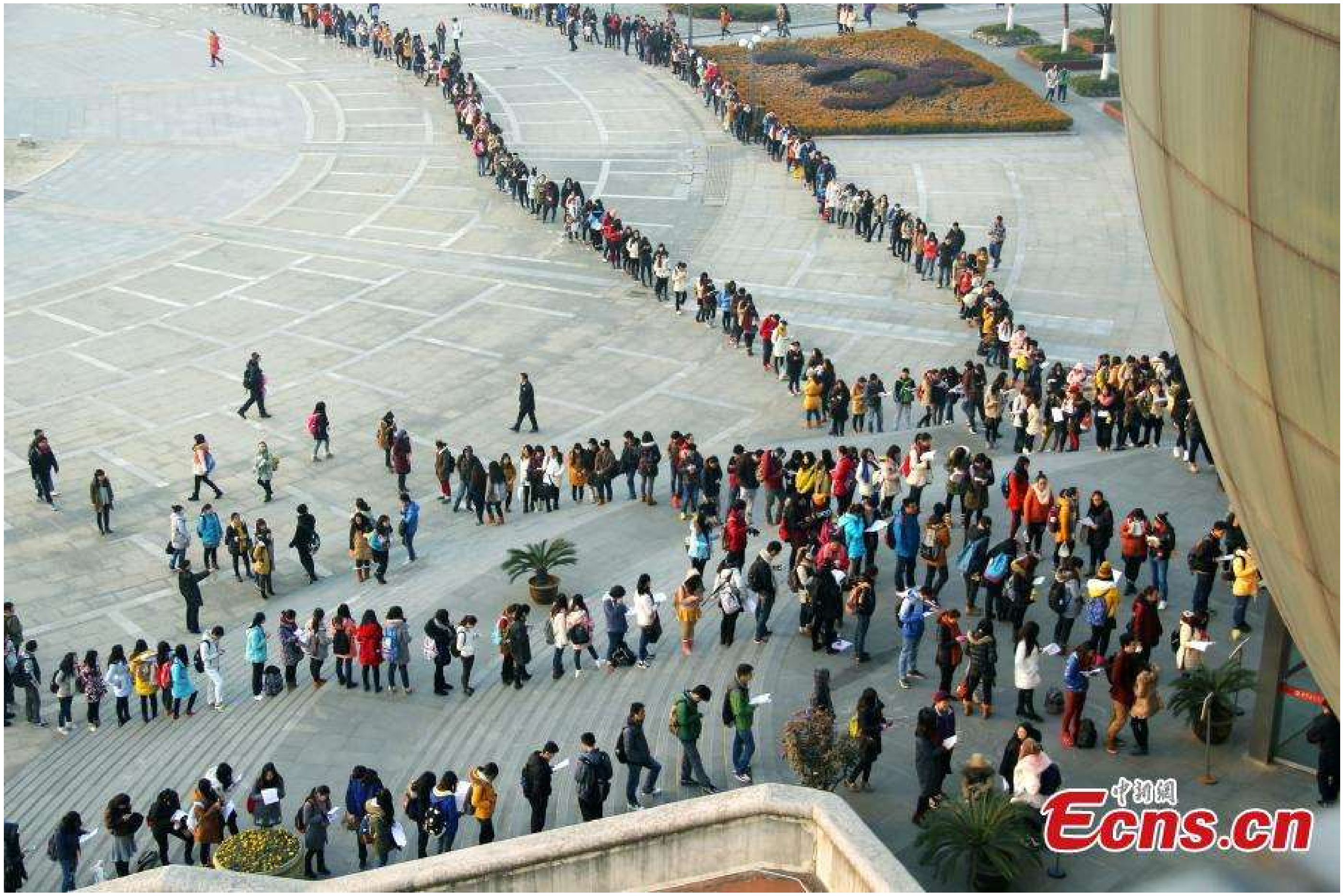
- **Stores** and removes items **independent of contents.**

- **Examples** include:
  - List ADT ☑
  - Stack ADT ☑
  - Queue ADT. ← **Today**

- Core **operations**:
  - ○ add item
  - ○ remove item

Queue

Like a list…
but…
the order in which items arrive is important



http://www.chinadaily.com.cn/china/2015-01/09/content_19282920.htm

FIFO

FIFO ≠ FIFA

- **FIFO** (First In First Out): The first element to arrive, is the first to be processed

- Data: The first element to be added, is the first to be deleted (or served)

- Access to any other element is unnecessary (and thus not allowed)

Queue Data Type

- Follows the **FIFO model**

- Its operations (interface) are:
  - **Create** the queue (Queue)
  - Add an item to the back (**append**)
  - Take an item off the front (**serve**)
  - Is the queue **empty**?
  - Is the queue **full**?
  - Empty the queue (**reset**)

Remember: you can only access the element at the front of the queue (first item inserted that is still in)

Possible implementation: linear queue

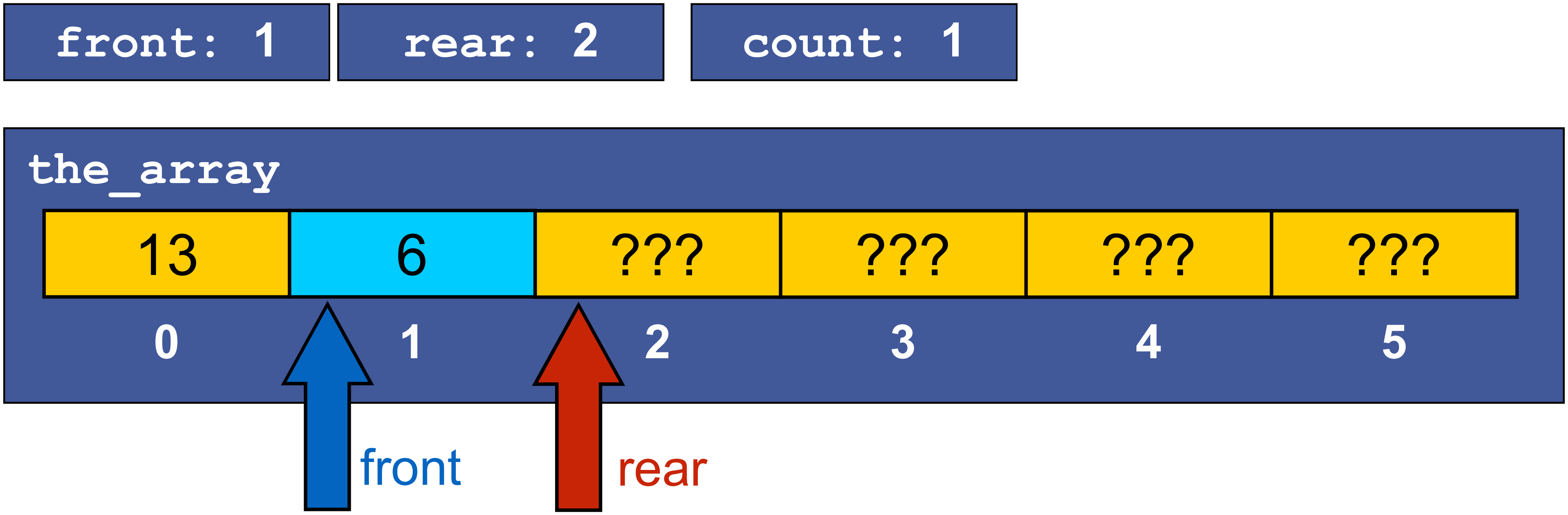
- We need to: **add items** at the **rear**. **take** items from the **front.**

  A single marker is not going to be enough.

- Lets try implementing queues using:
  - An **array** to store the items in the order they arrive.
  - An **integer** marking the front of the queue. Refers to the first element to be served.
  - An **integer** marking the rear of the queue. Refers to the first empty slot at the rear.
  - An integer **count** keeping track of the number of items.

- **Invariant**: valid data appears in `front … rear-1` positions

- Create a new queue: no items
- Append item 13
- Append item 6
- Serve item 13

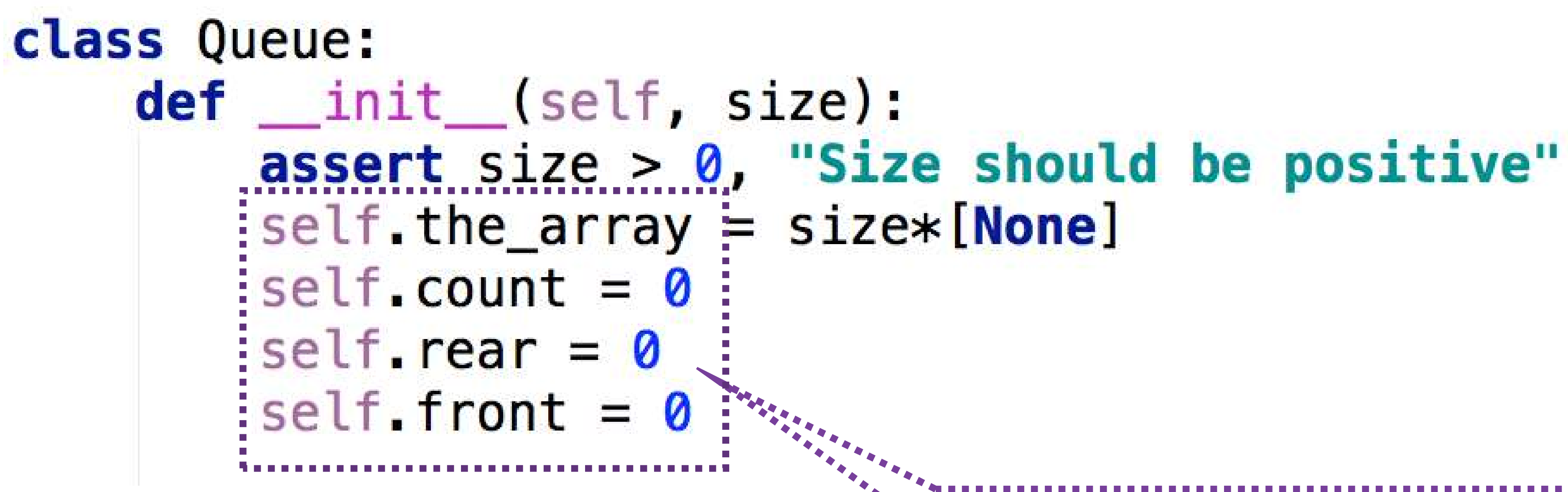


`front: 1` `rear: 2` `count: 1`

`the_array`

13	6	???	???	???	???
0	1	2	3	4	5

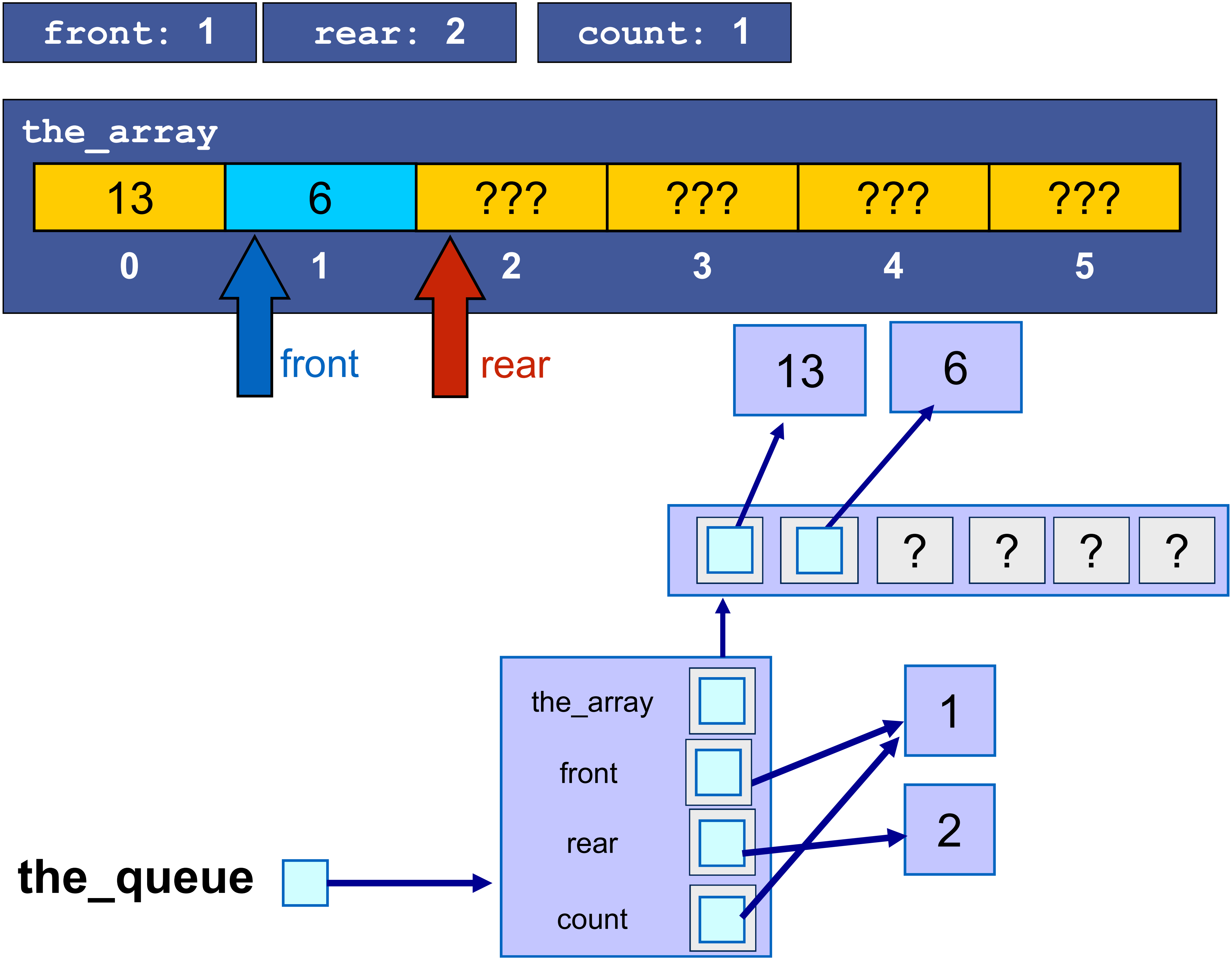
Creating a Linear Queue



```python
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0
```

Instance variables

Complexity is O(size)

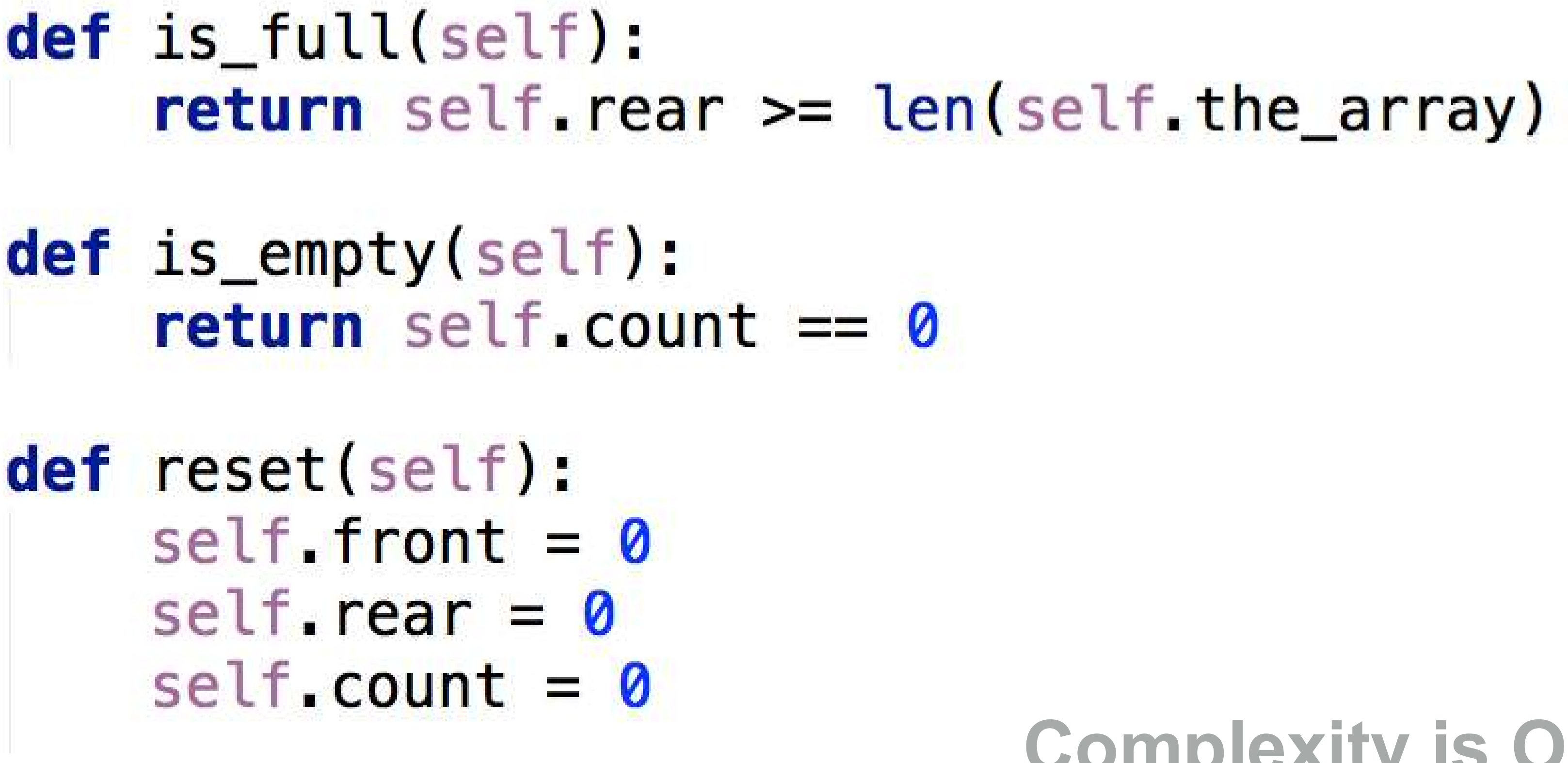


front: 1 rear: 2 count: 1

the_array

13	6	???	???	???	???
0	1	2	3	4	5

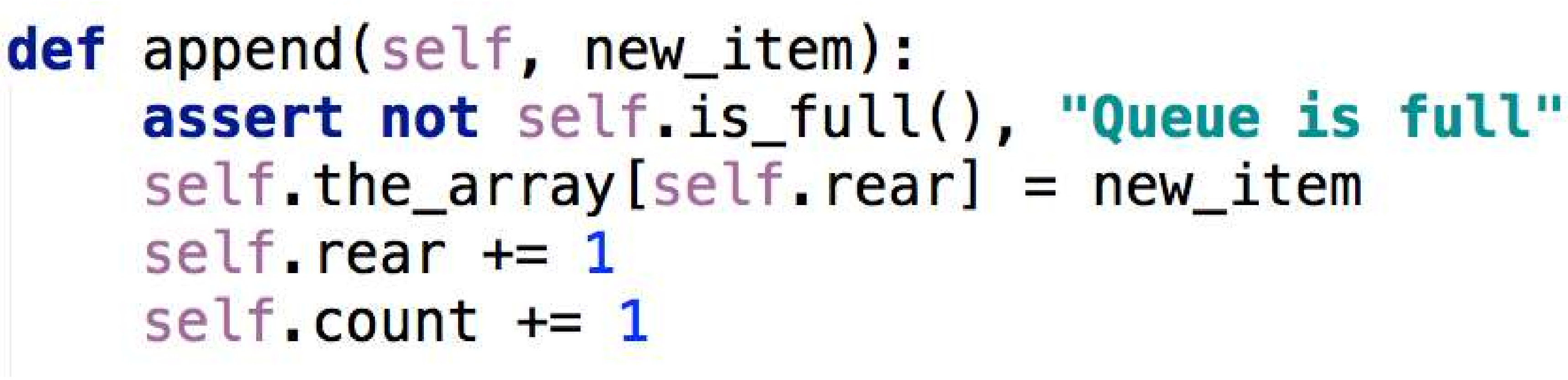
Simple methods



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

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Complexity is O(1)
for all of these methods.

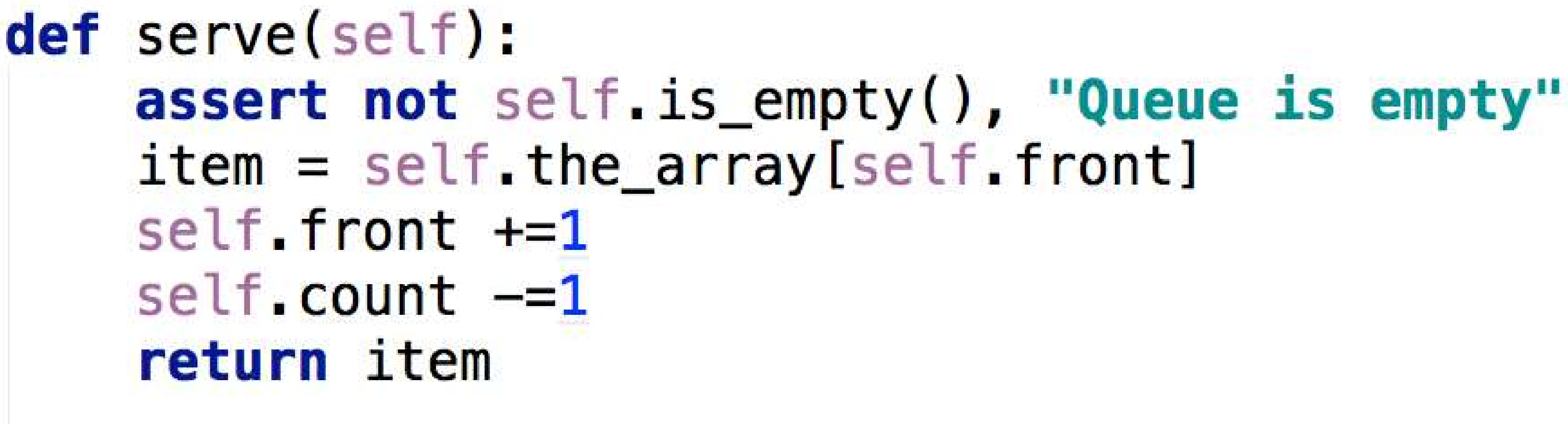
Implementing Append



```python
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    self.count += 1
```

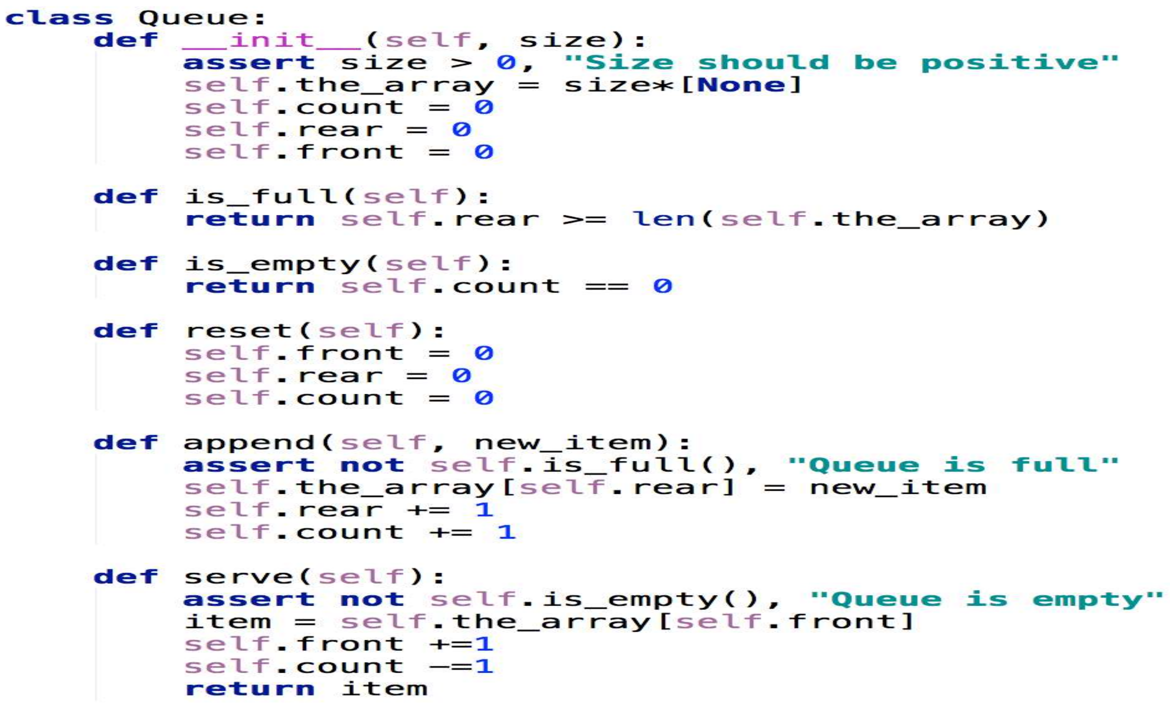
Complexity is O(1)

Implementing Serve

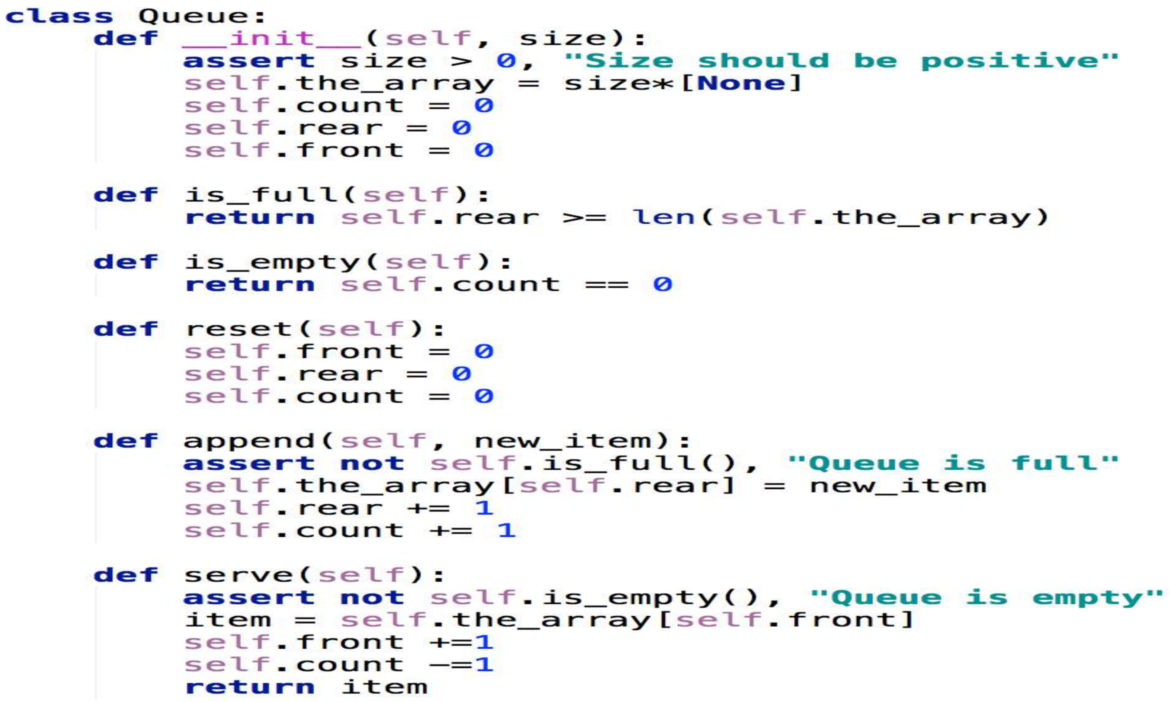


```python
def serve(self):
    assert not self.is_empty(), "Queue is empty"
    item = self.the_array[self.front]
    self.front +=1
    self.count -=1
    return item
```

Complexity is O(1)



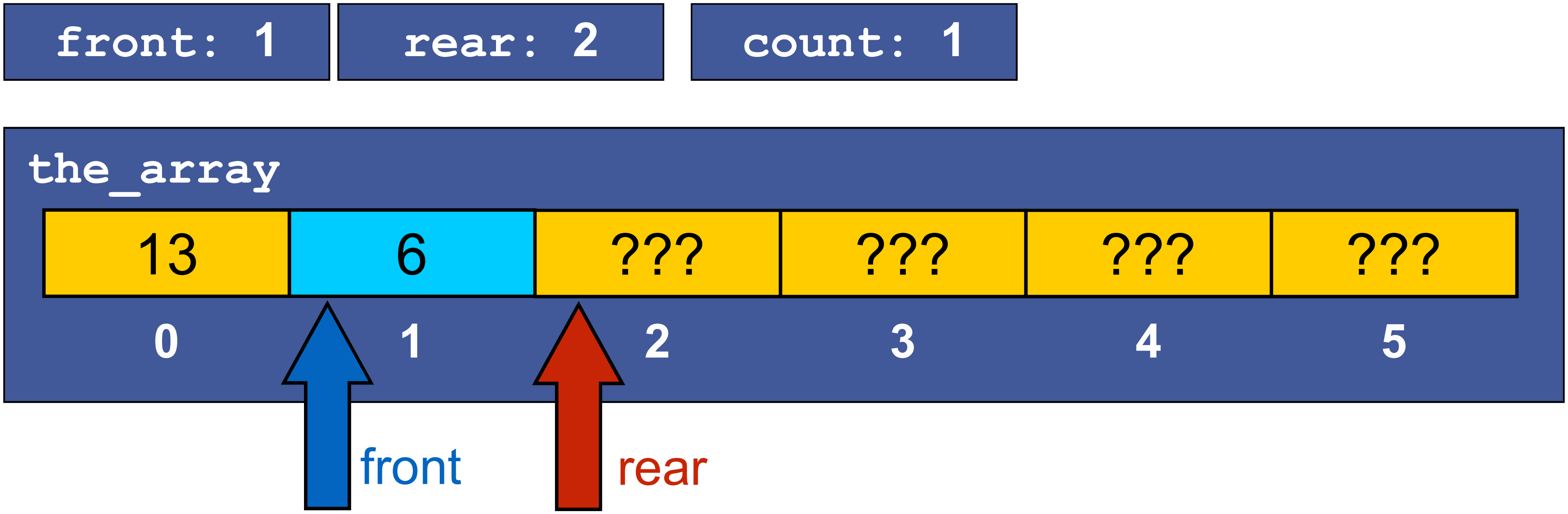
```python
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0

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

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

    def reset(self):
        self.front = 0
        self.rear = 0
        self.count = 0

    def append(self, new_item):
        assert not self.is_full(), "Queue is full"
        self.the_array[self.rear] = new_item
        self.rear += 1
        self.count += 1

    def serve(self):
        assert not self.is_empty(), "Queue is empty"
        item = self.the_array[self.front]
        self.front +=1
        self.count -=1
        return item
```



```python
class Queue:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
        self.rear = 0
        self.front = 0

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

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

    def reset(self):
        self.front = 0
        self.rear = 0
        self.count = 0

    def append(self, new_item):
        assert not self.is_full(), "Queue is full"
        self.the_array[self.rear] = new_item
        self.rear += 1
        self.count += 1

    def serve(self):
        assert not self.is_empty(), "Queue is empty"
        item = self.the_array[self.front]
        self.front +=1
        self.count -=1
        return item
```

ear Queue, shifting both pointers

… so much space… just wasted



`front: 1`

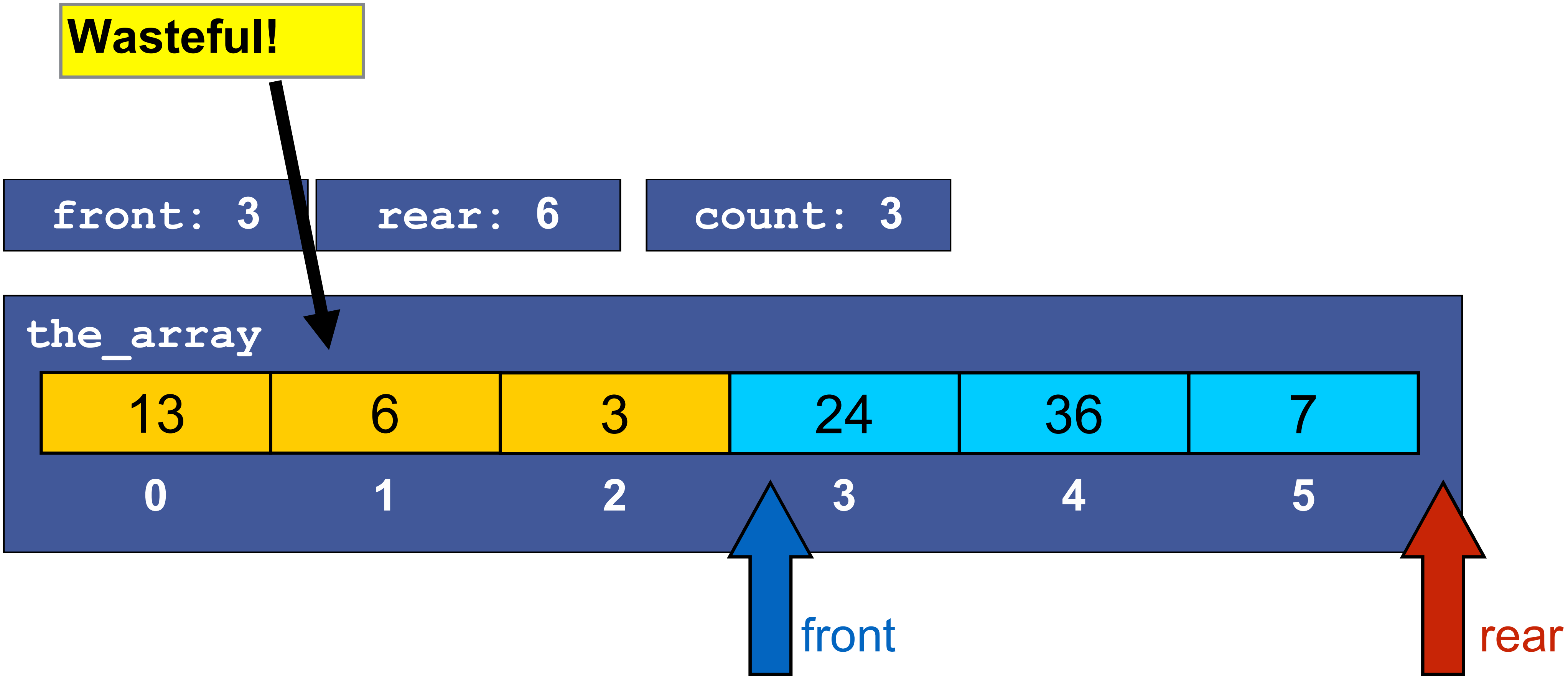
`rear: 2`

`count: 1`

`the_array`

13	6	???	???	???	???
0	1	2	3	4	5

Implementation problem



Wasteful!

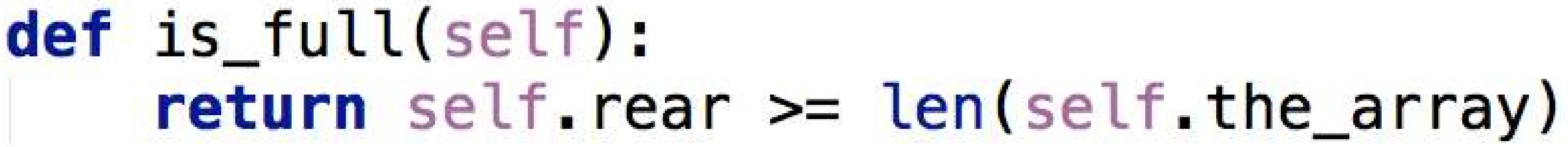
front: 3

rear: 6

count: 3

the_array

13	6	3	24	36	7
0	1	2	3	4	5



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

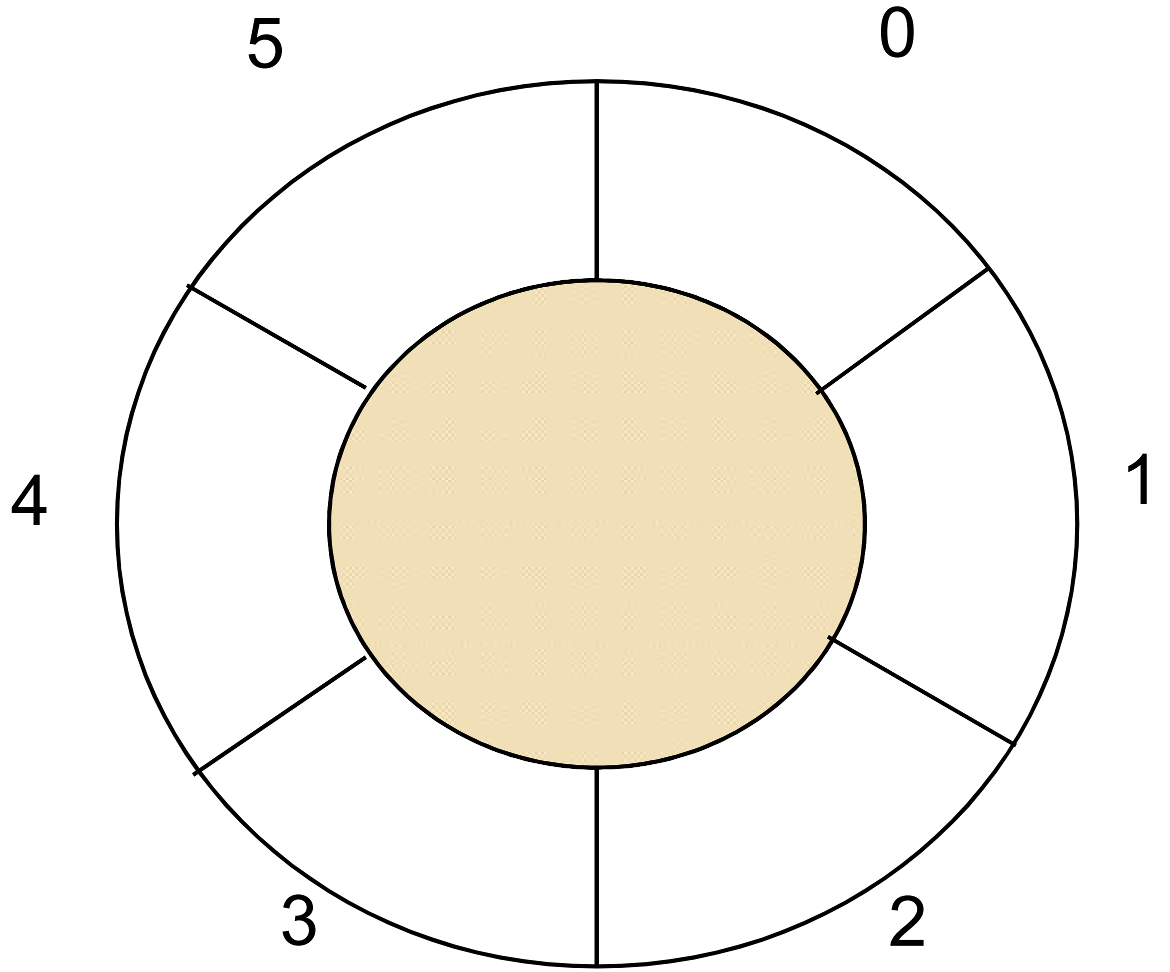
But my constant time!

Can we have an Array Queue without wasting space and STILL have constant time push and pop?

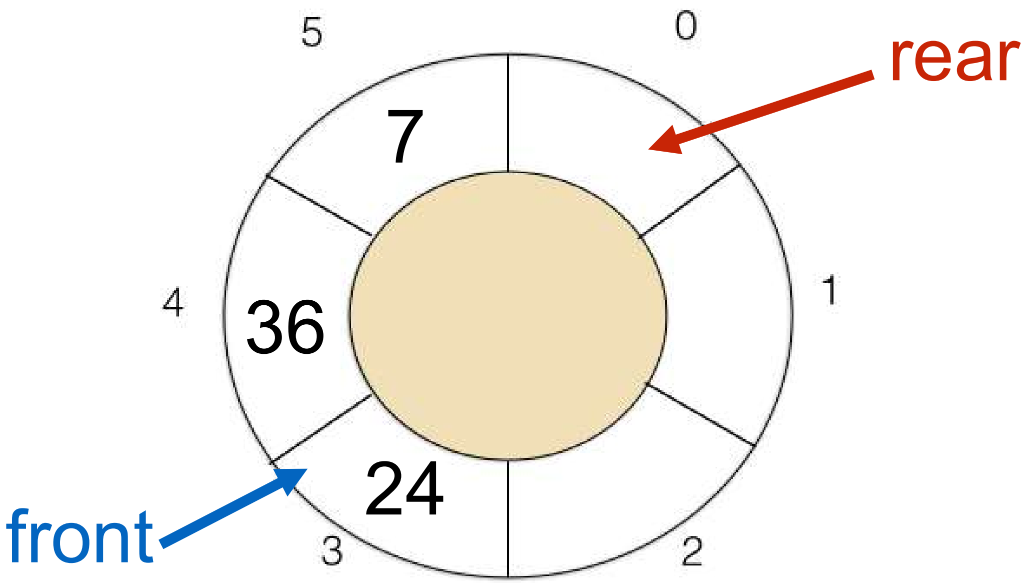
A) Yes
B) No

http://www.sunhills.org/blog/why-you-cant-have-your-cake-and-eat-it-too/

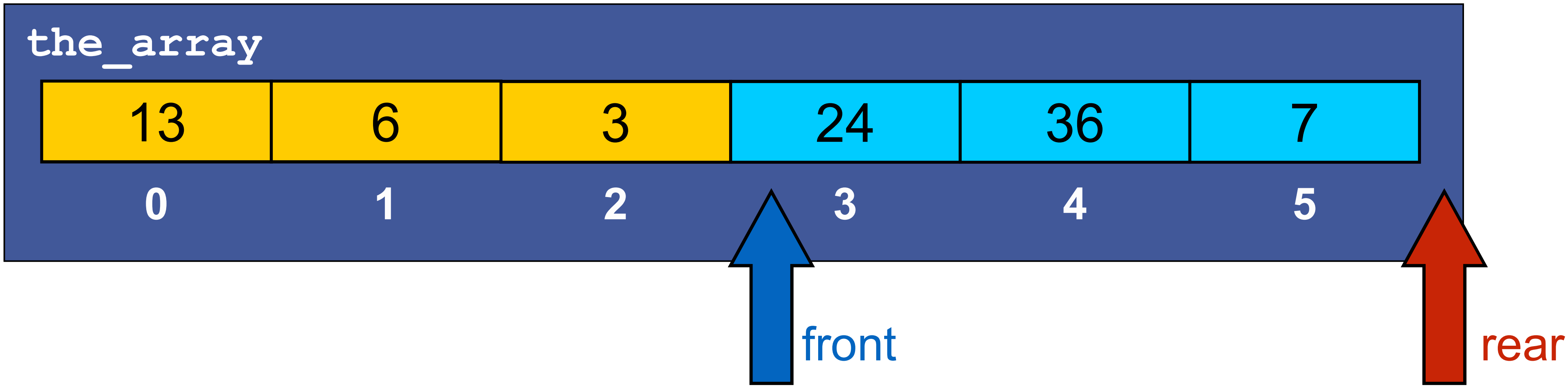
Solution: Circular Queues



Simulated by allowing **rear** and **front** to wrap around each other



`front: 3` `rear: 6` `count: 3`

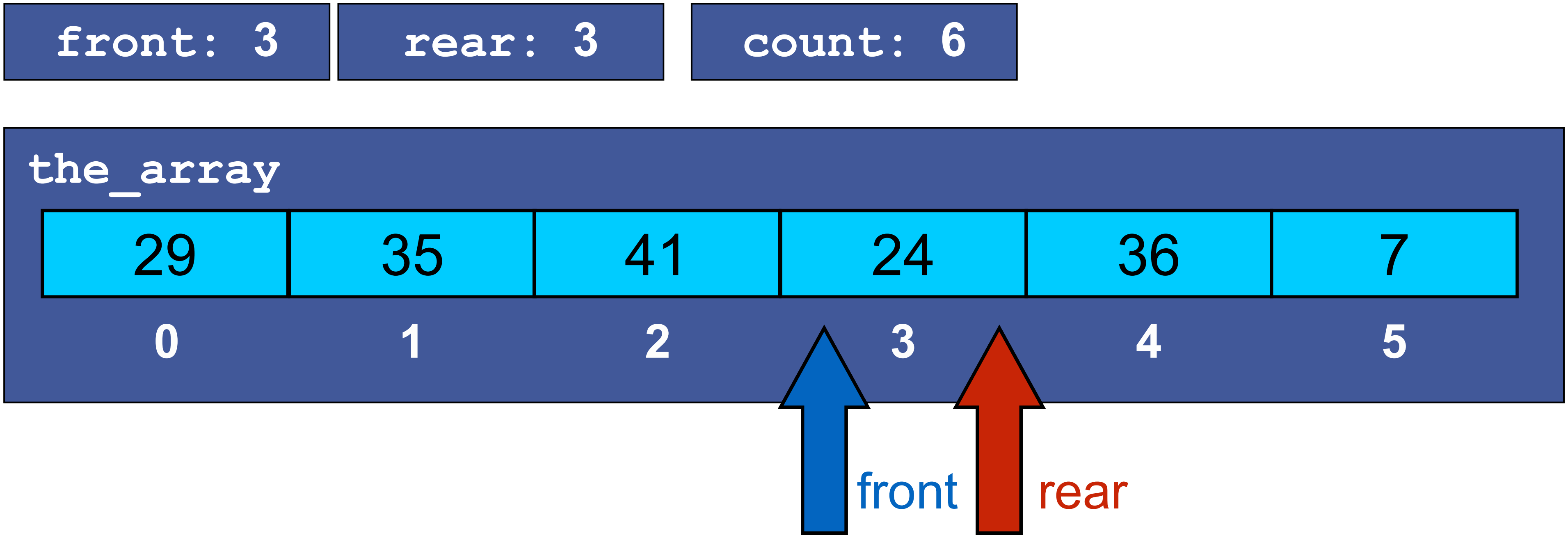


`the_array`

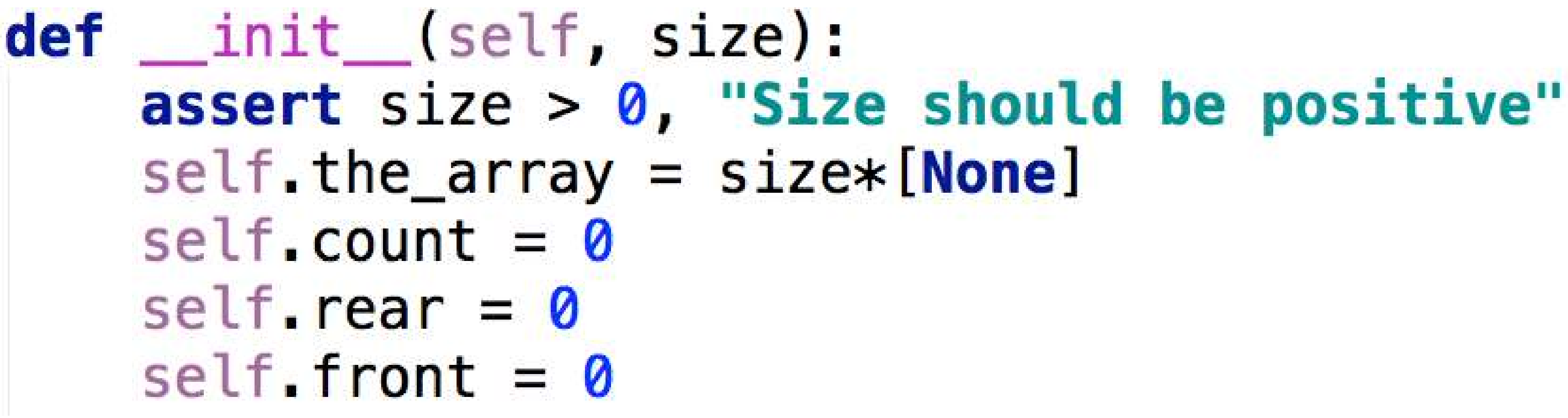
13	6	3	24	36	7
0	1	2	3	4	5

- After appending 7
- Append 29
- Append 35
- Append 41

Important: Instead of using rear, to determine if the queue is full, we now need to use count.



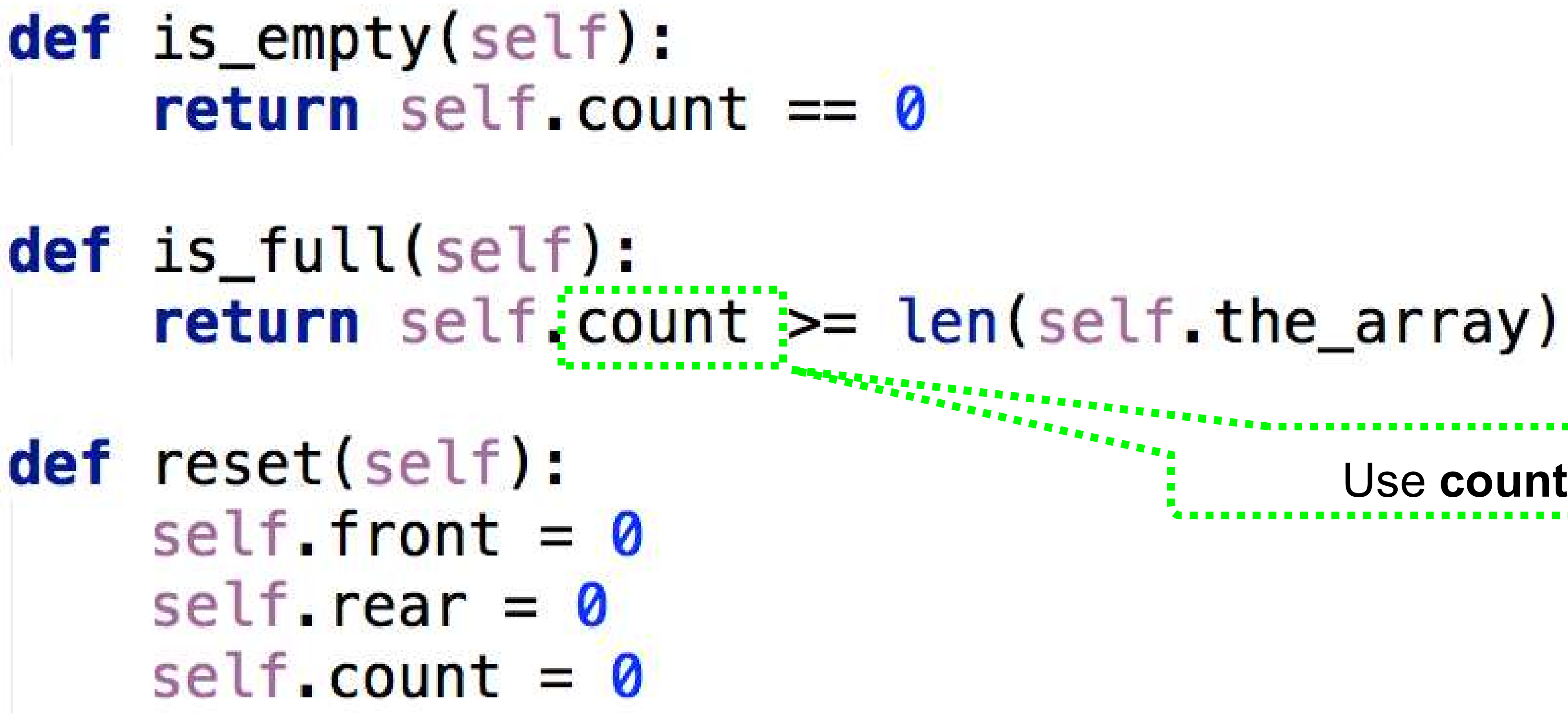
Creating a circular Queue



```
def __init__(self, size):
    assert size > 0, "Size should be positive"
    self.the_array = size*[None]
    self.count = 0
    self.rear = 0
    self.front = 0
```

Complexity is O(size)

Methods for Circular Queue

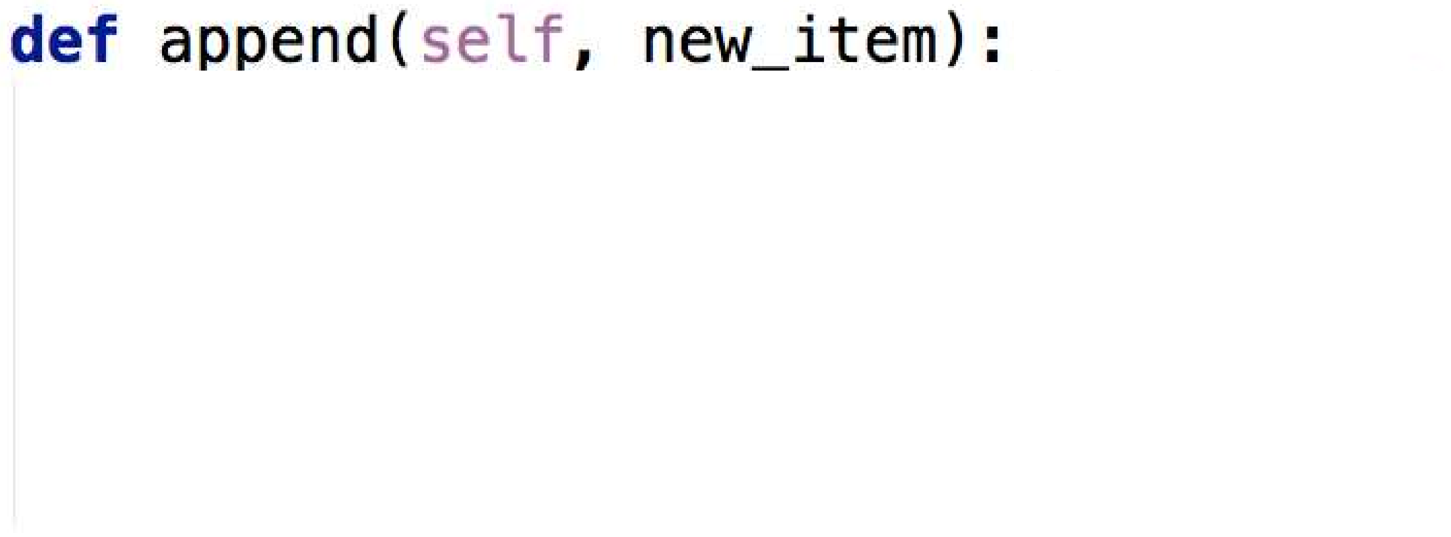


```python
def is_empty(self):
    return self.count == 0

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

def reset(self):
    self.front = 0
    self.rear = 0
    self.count = 0
```

Use **count,** not rear

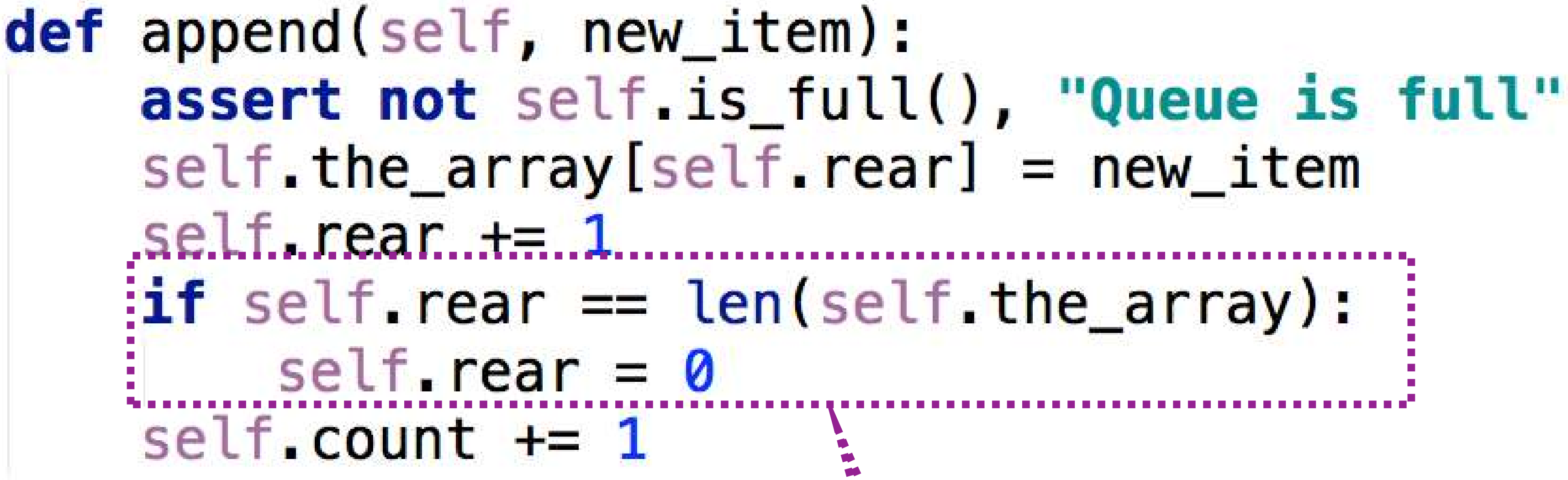
Complexity is O(1)

Implementation of Append for a Circular Queue



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

Implementation of Append for a Circular Queue



```python
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
    self.count += 1
```

If rear points outside of the_array
but I know **the queue is not full**

You know that *len(self.the_array) = 6* and *self.rear = 5*

(self.rear + 1) % len(self.the_array) is equal to…

(self.rear + 1) % len(self.the_array)

(5 + 1) % 6

6 % 6 = 0

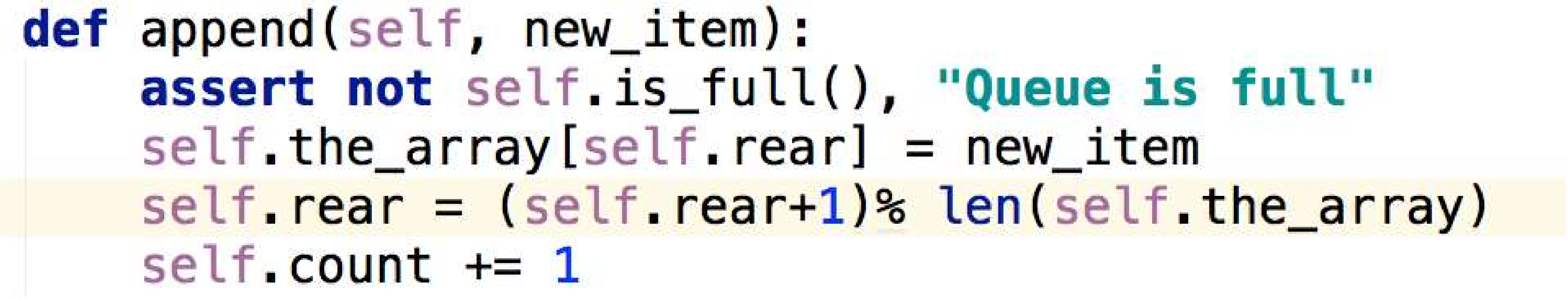
You know that *len(self.the_array) = 6* and *self.rear = 4*

(self.rear + 1) % len(self.the_array) is equal to…

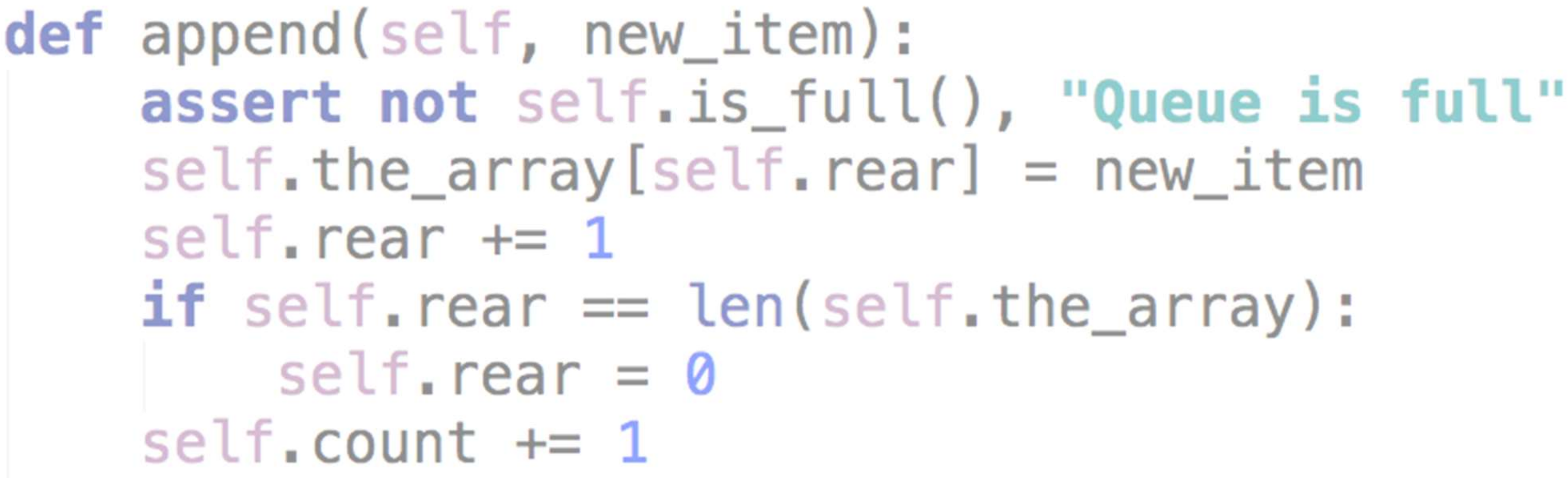
(self.rear + 1) % len(self.the_array)

(4 + 1) % 6

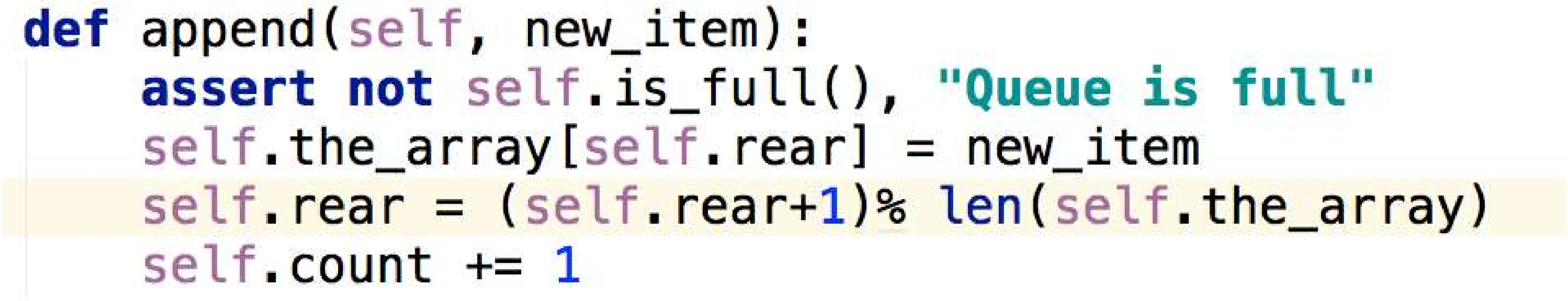
5 % 6 = 5



```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear = (self.rear+1)% len(self.the_array)
    self.count += 1
```



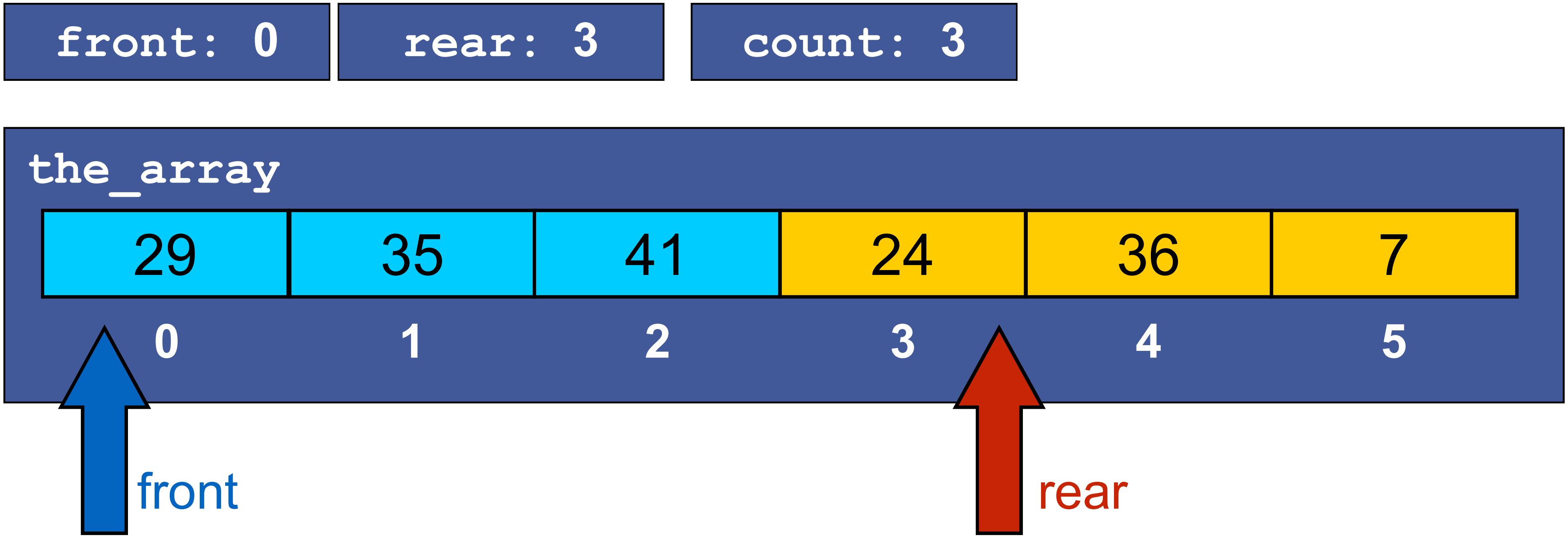
```
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear += 1
    if self.rear == len(self.the_array):
        self.rear = 0
    self.count += 1
```



```python
def append(self, new_item):
    assert not self.is_full(), "Queue is full"
    self.the_array[self.rear] = new_item
    self.rear = (self.rear+1)% len(self.the_array)
    self.count += 1
```

Circular queue: both front and rear wrap

- Serve item (returns 24)
- Serve item (returns 36)
- Serve item (returns 7)

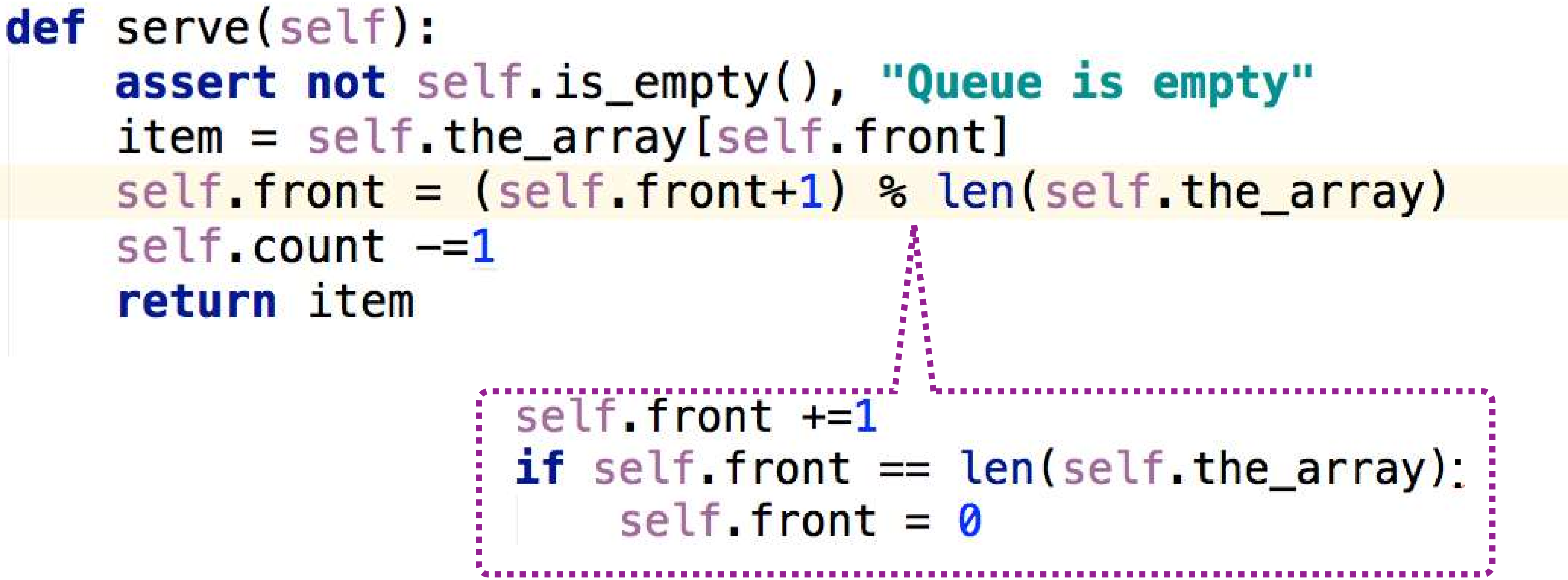


`front: 0` `rear: 3` `count: 3`

`the_array`

29	35	41	24	36	7
0	1	2	3	4	5

Implementation of Serve for a Circular Queue



```python
def serve(self):
    assert not self.is_empty(), "Queue is empty"
    item = self.the_array[self.front]
    self.front = (self.front+1) % len(self.the_array)
    self.count -=1
    return item
```

```python
self.front +=1
if self.front == len(self.the_array):
    self.front = 0
```

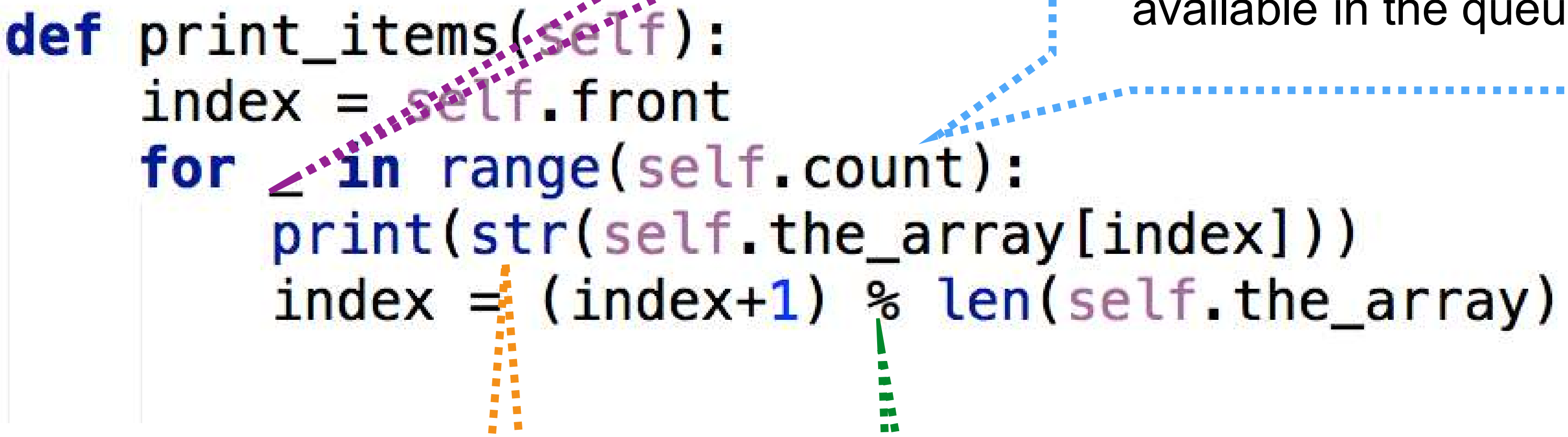
Print Queue

- Lets implement it as a function within the Queue ADT. So, it has access to the implementation.
- Do not modify the queue, just **print** its elements

Print Queue

Anonymous variable

print as many items as available in the queue



```
def print_items(self):
    index = self.front
    for _ in range(self.count):
        print(str(self.the_array[index]))
        index = (index+1) % len(self.the_array)
```

Convert to string whatever is stored

Increase index or make it zero if it points to outside of the_array

Some Queue Applications

- Scheduling and buffering
  - Printers
  - Keyboards
  - Executing asynchronous procedure calls

Summary

- Queues
  - Array implementation
    - Linear
    - Circular
  - Basic operations
  - Their complexity