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

Lecture 23
Linked Structures

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

Container ADTs

- **Stores** and removes items **independent of contents.**
- **Examples** include:
  - List ADT ☑
  - Stack ADT ☑
  - Queue ADT. ☑
- Core **operations**:
  - add item
  - remove item

http://s4.thisnext.com/media/largest_dimension/DFF83835.jpg

Implementation affects time complexity.

Unsorted List: Add

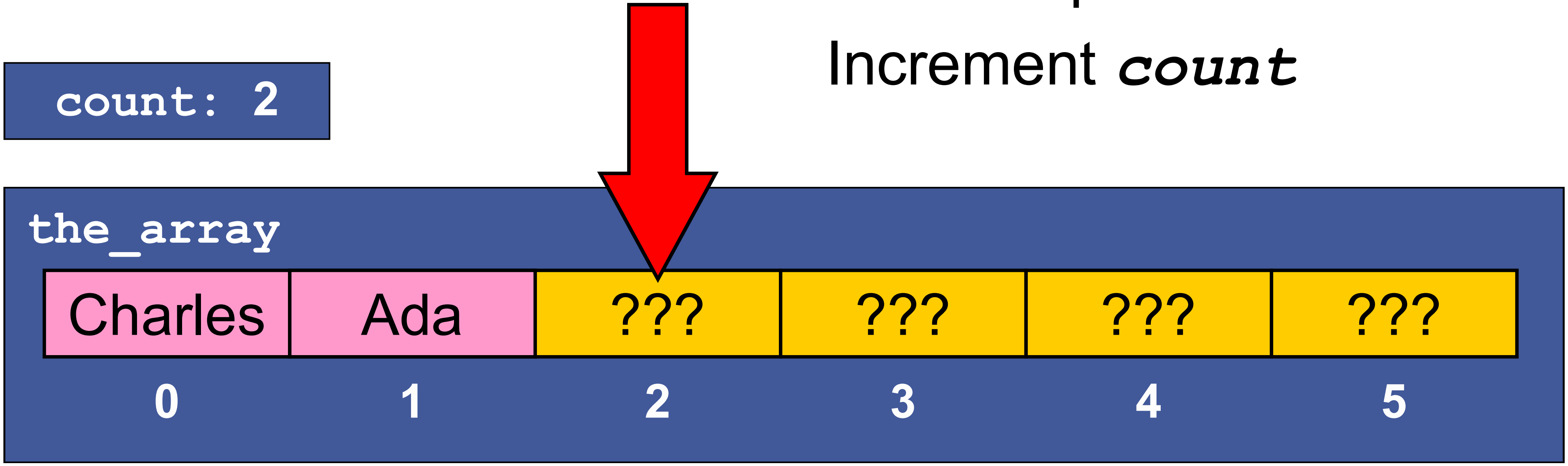
Adding an element

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

`Example: add “Ada”`

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

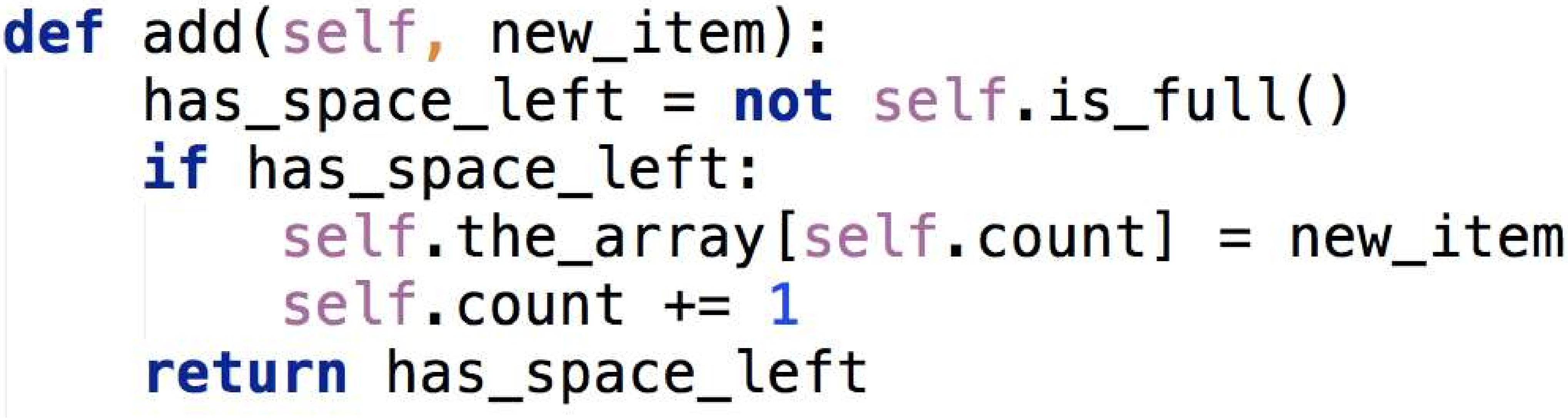
Increment ***`count`***



`count: 2`

`the_array`

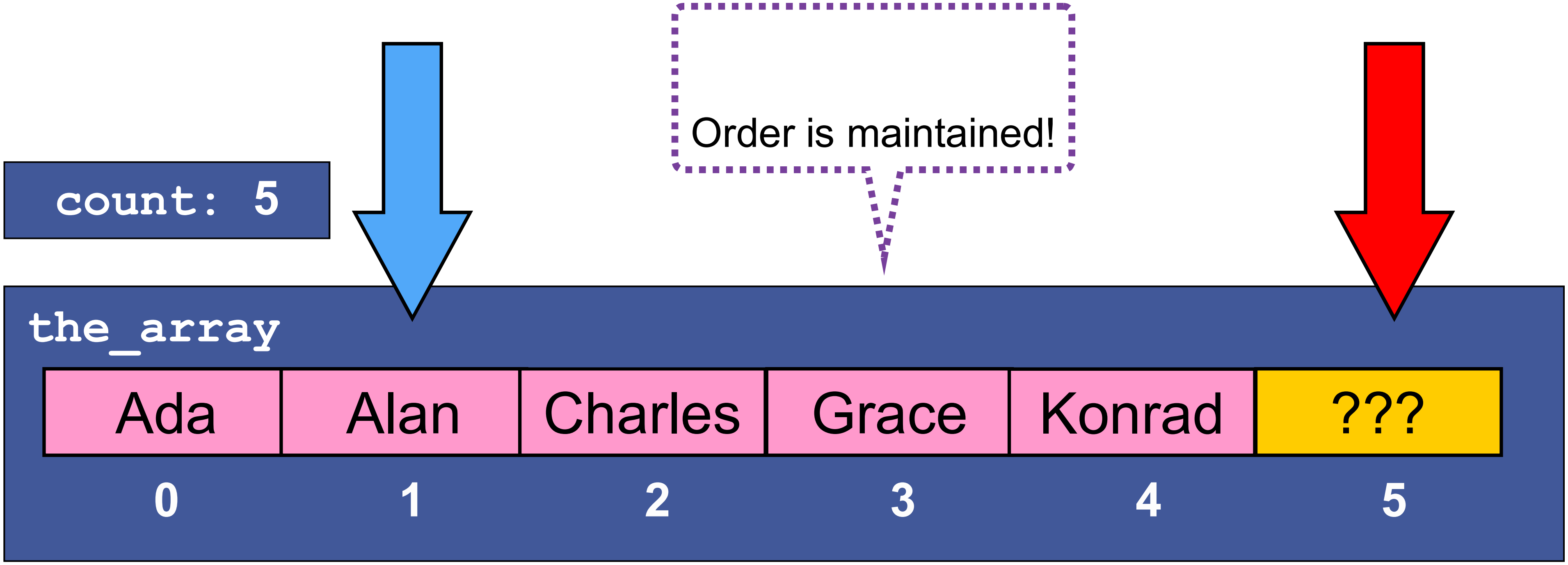
Charles	Ada	???	???	???	???
0	**1**	**2**	**3**	**4**	**5**



```python
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
```

Sorted List: Add

Example: add "Alan" to the sorted list.



Order is maintained!

count: 5

the_array

Ada	Alan	Charles	Grace	Konrad	???
0	1	2	3	4	5

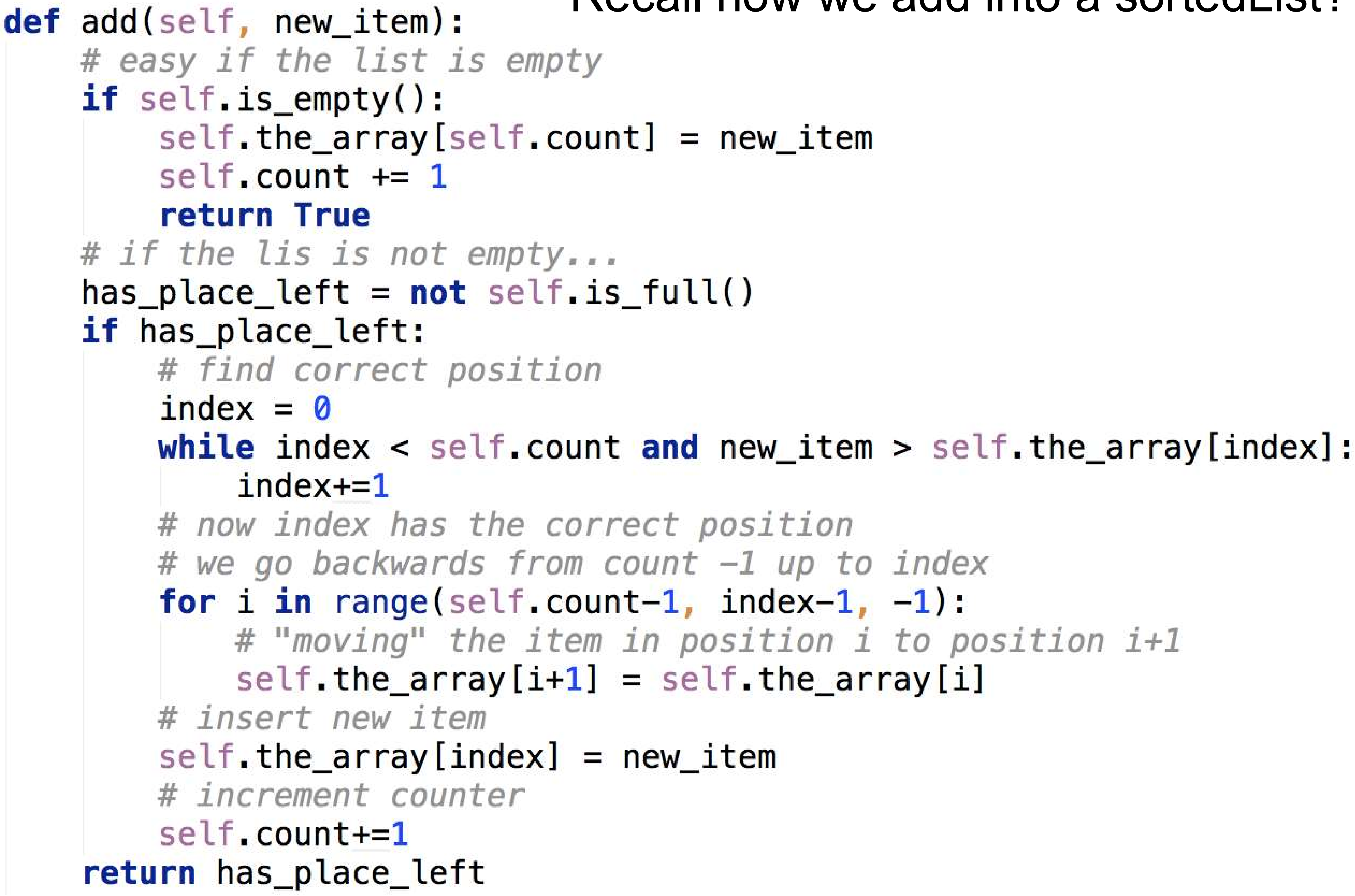
If there is space, find the correct position

Make room by moving all to the right.

Put item in position.

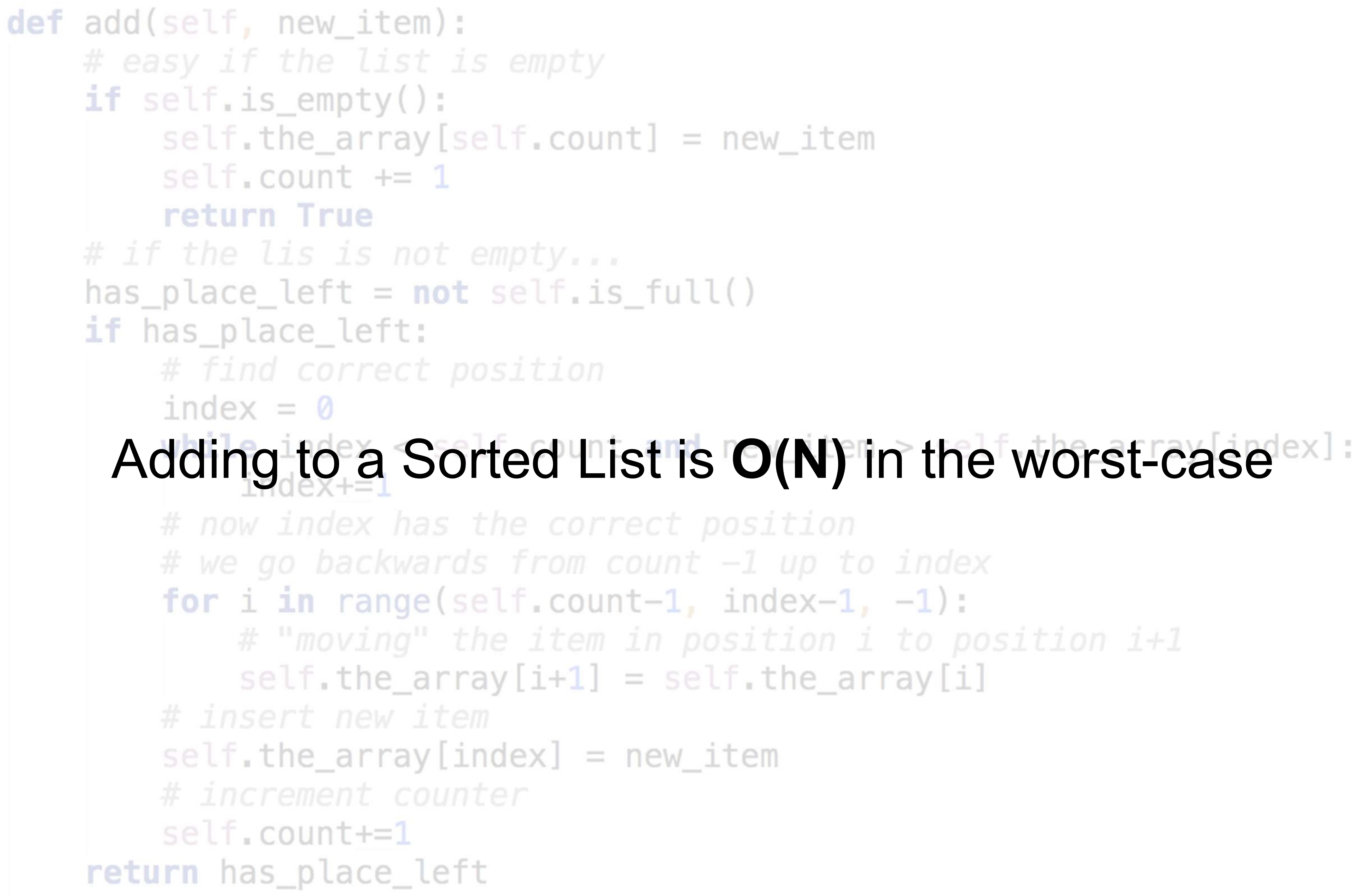
Update count then **return True**

Recall how we add into a sortedList?



```
def add(self, new_item):
    # easy if the list is empty
    if self.is_empty():
        self.the_array[self.count] = new_item
        self.count += 1
        return True
    # if the lis is not empty...
    has_place_left = not self.is_full()
    if has_place_left:
        # find correct position
        index = 0
        while index < self.count and new_item > self.the_array[index]:
            index+=1
        # now index has the correct position
        # we go backwards from count -1 up to index
        for i in range(self.count-1, index-1, -1):
            # "moving" the item in position i to position i+1
            self.the_array[i+1] = self.the_array[i]
        # insert new item
        self.the_array[index] = new_item
        # increment counter
        self.count+=1
    return has_place_left
```

Adding to a Sorted List is **O(N)** in the worst-case



```
def add(self, new_item):
    # easy if the list is empty
    if self.is_empty():
        self.the_array[self.count] = new_item
        self.count += 1
        return True
    # if the lis is not empty...
    has_place_left = not self.is_full()
    if has_place_left:
        # find correct position
        index = 0
        while index < self.count and new_item > self.the_array[index]:
            index+=1
        # now index has the correct position
        # we go backwards from count -1 up to index
        for i in range(self.count-1, index-1, -1):
            # "moving" the item in position i to position i+1
            self.the_array[i+1] = self.the_array[i]
        # insert new item
        self.the_array[index] = new_item
        # increment counter
        self.count+=1
    return has_place_left
```

Can we improve this by changing the representation?

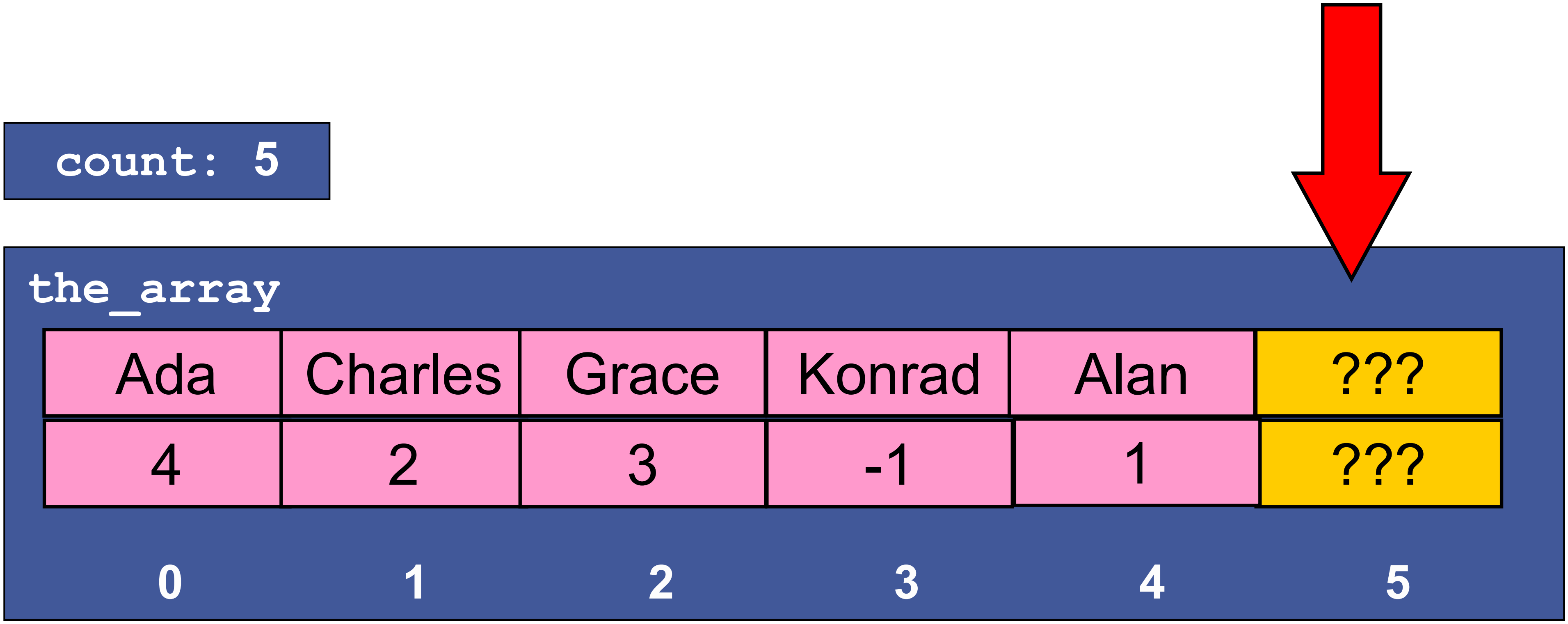
Welcome to the world of

ed structures

Legend of zelda
https://upload.wikimedia.org/wikipedia/en/3/39/Wakerlink.jpg

Array Linked List: Add

Example: add "Alan" to the sorted list.



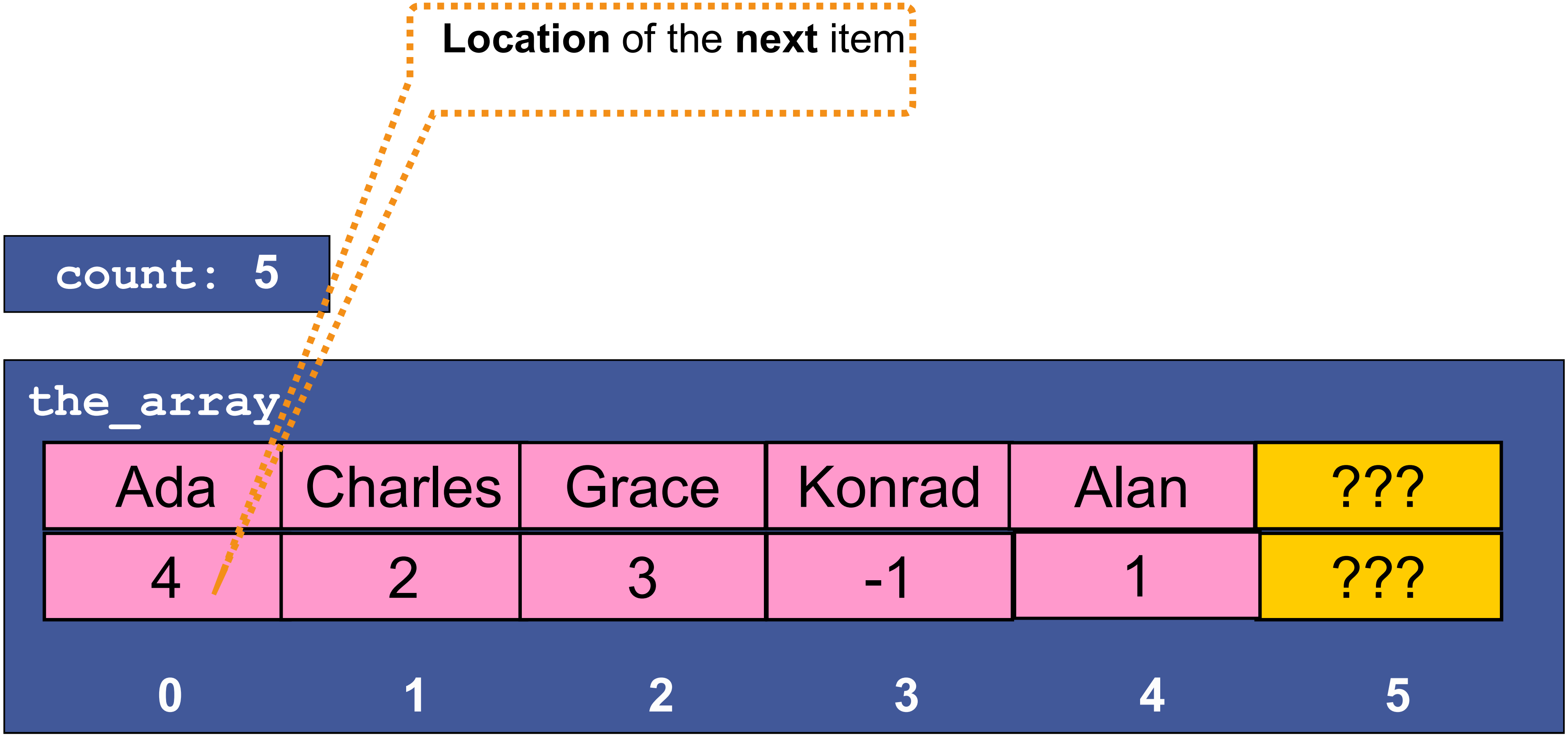
count: 5

the_array

0	1	2	3	4	5
Ada	Charles	Grace	Konrad	Alan	???
4	2	3	-1	1	???

No shifting around!

If there is space, add it
then fix indexes…increment count



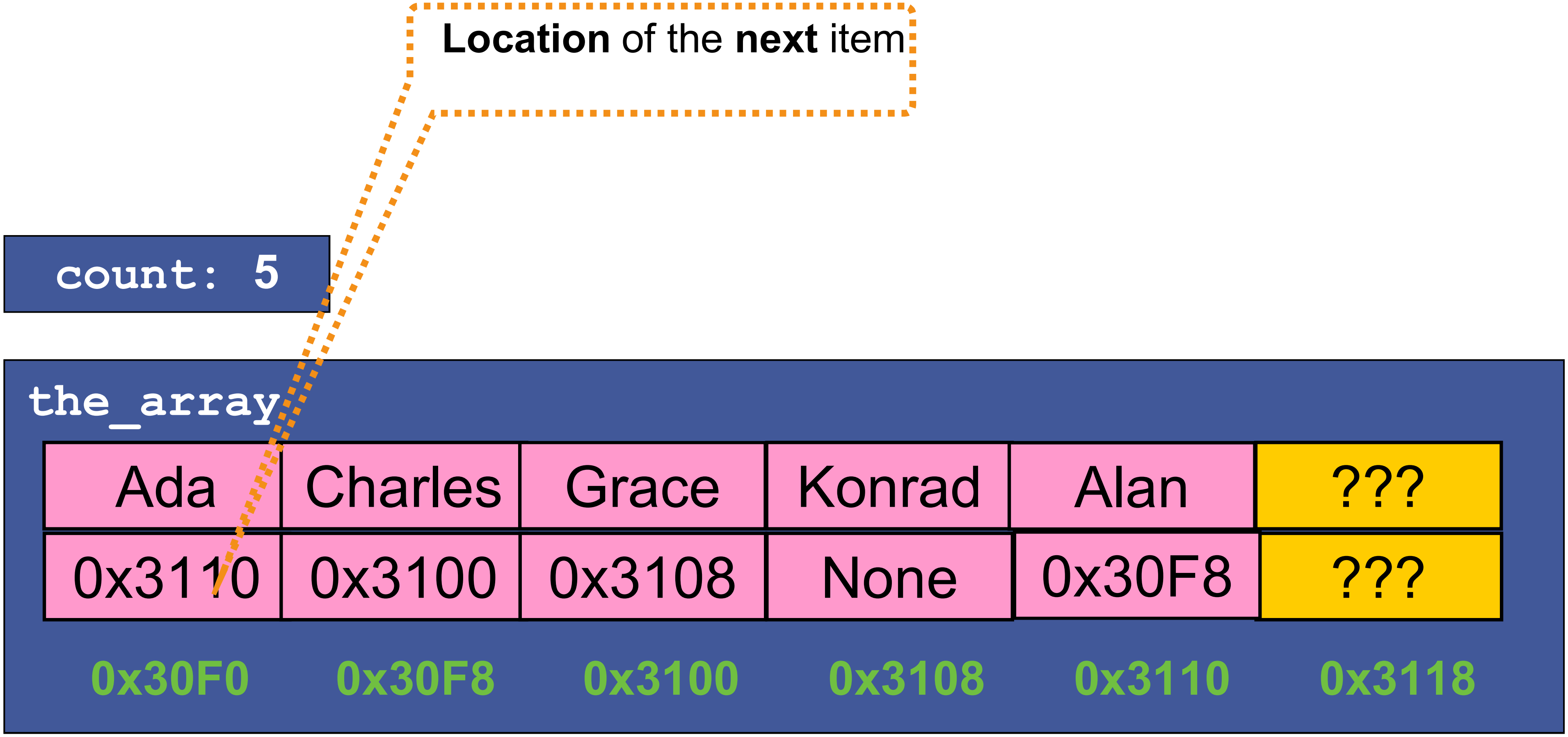
Location of the **next** item

`count: 5`

`the_array`

Ada	Charles	Grace	Konrad	Alan	???
4	2	3	-1	1	???
0	1	2	3	4	5

Why not use a **memory address** instead?

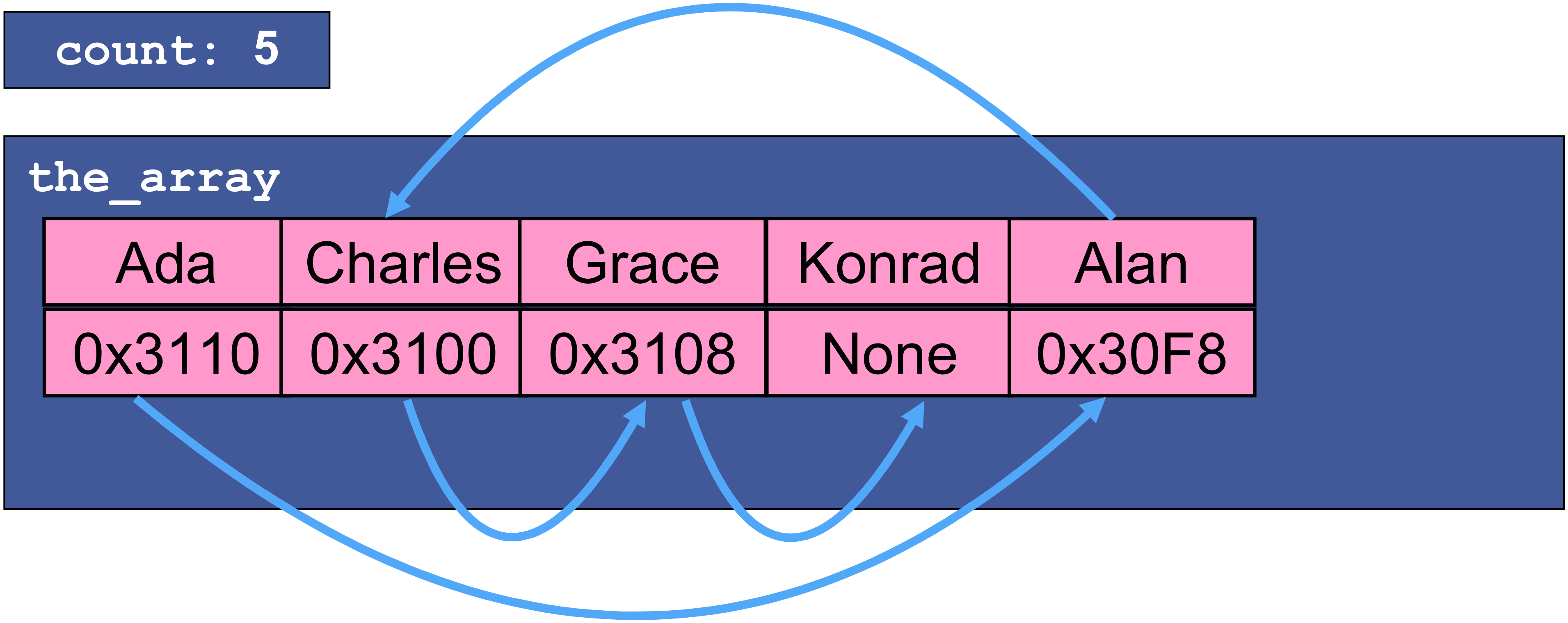


Location of the **next** item

count: 5

the_array

Ada	Charles	Grace	Konrad	Alan	???
0x3110	0x3100	0x3108	None	0x30F8	???
0x30F0	0x30F8	0x3100	0x3108	0x3110	0x3118



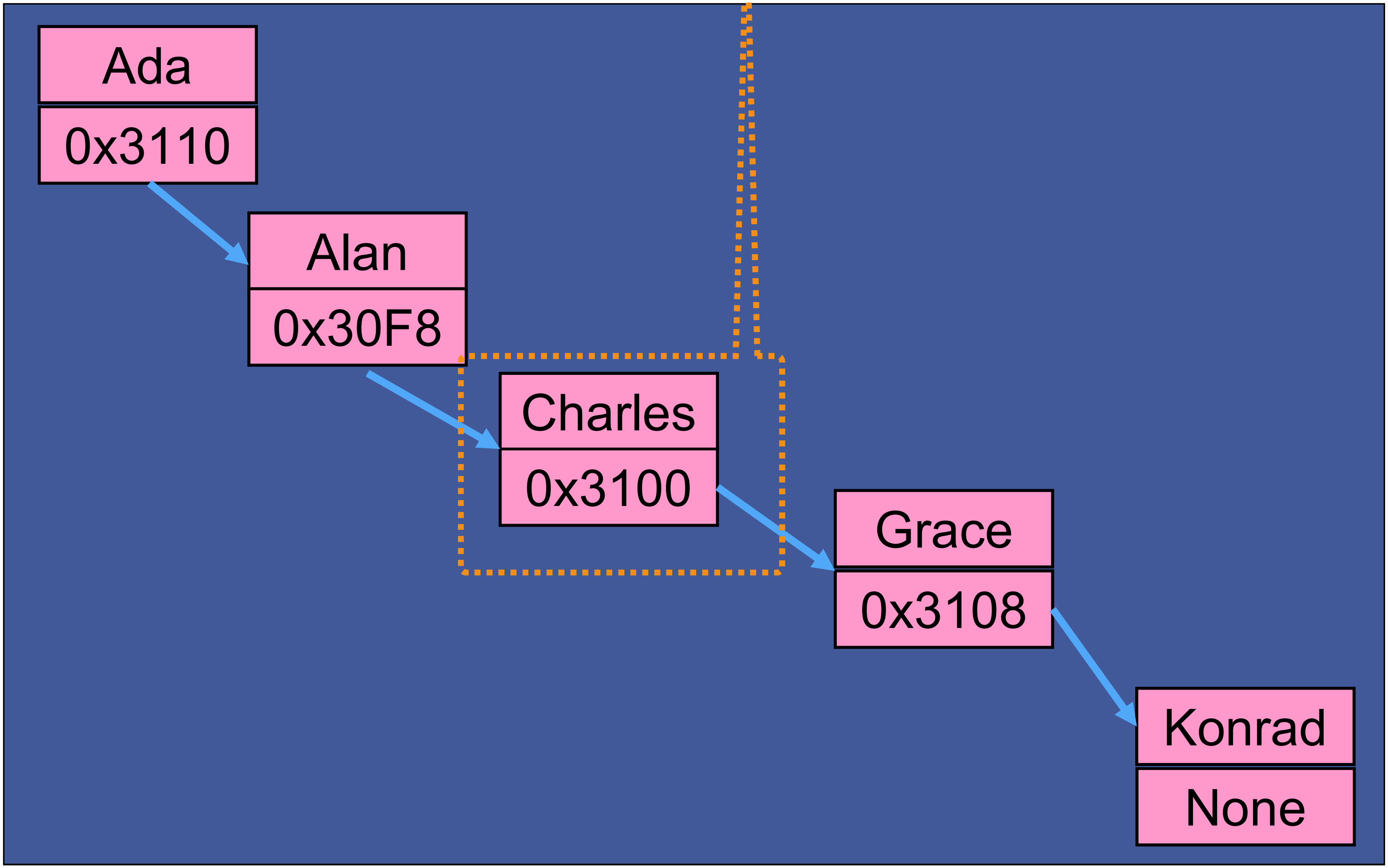
count: 5

the_array

Ada	Charles	Grace	Konrad	Alan
0x3110	0x3100	0x3108	None	0x30F8

`count: 5`

A **node**



Ada
0x3110

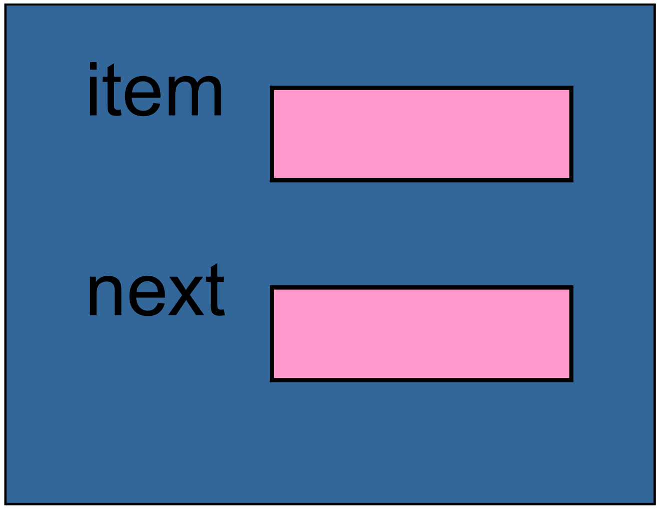
Alan
0x30F8

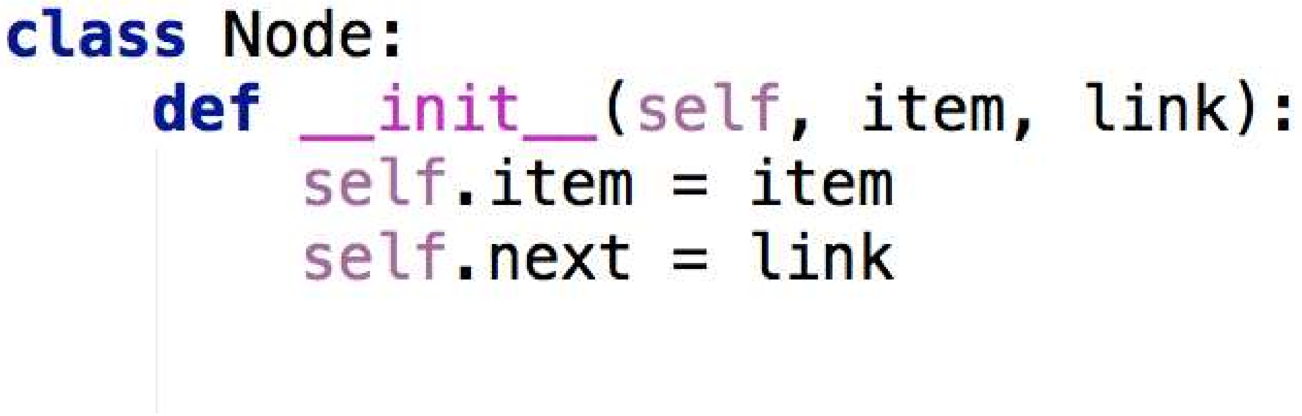
Charles
0x3100

Grace
0x3108

Konrad
None

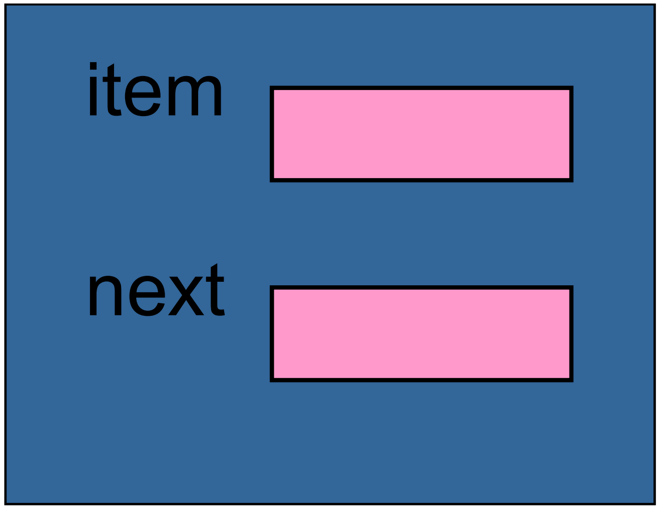
Node



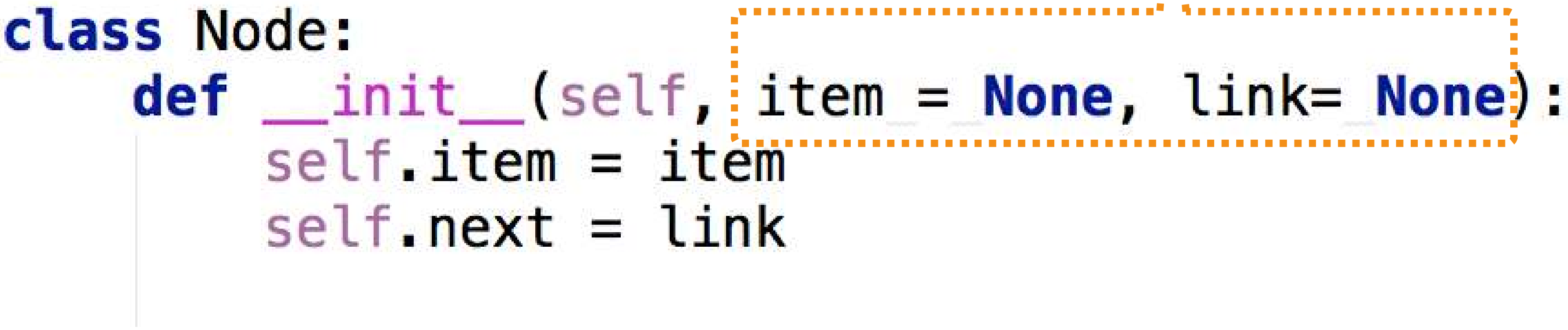


```python
class Node:
    def __init__(self, item, link):
        self.item = item
        self.next = link
```

Node

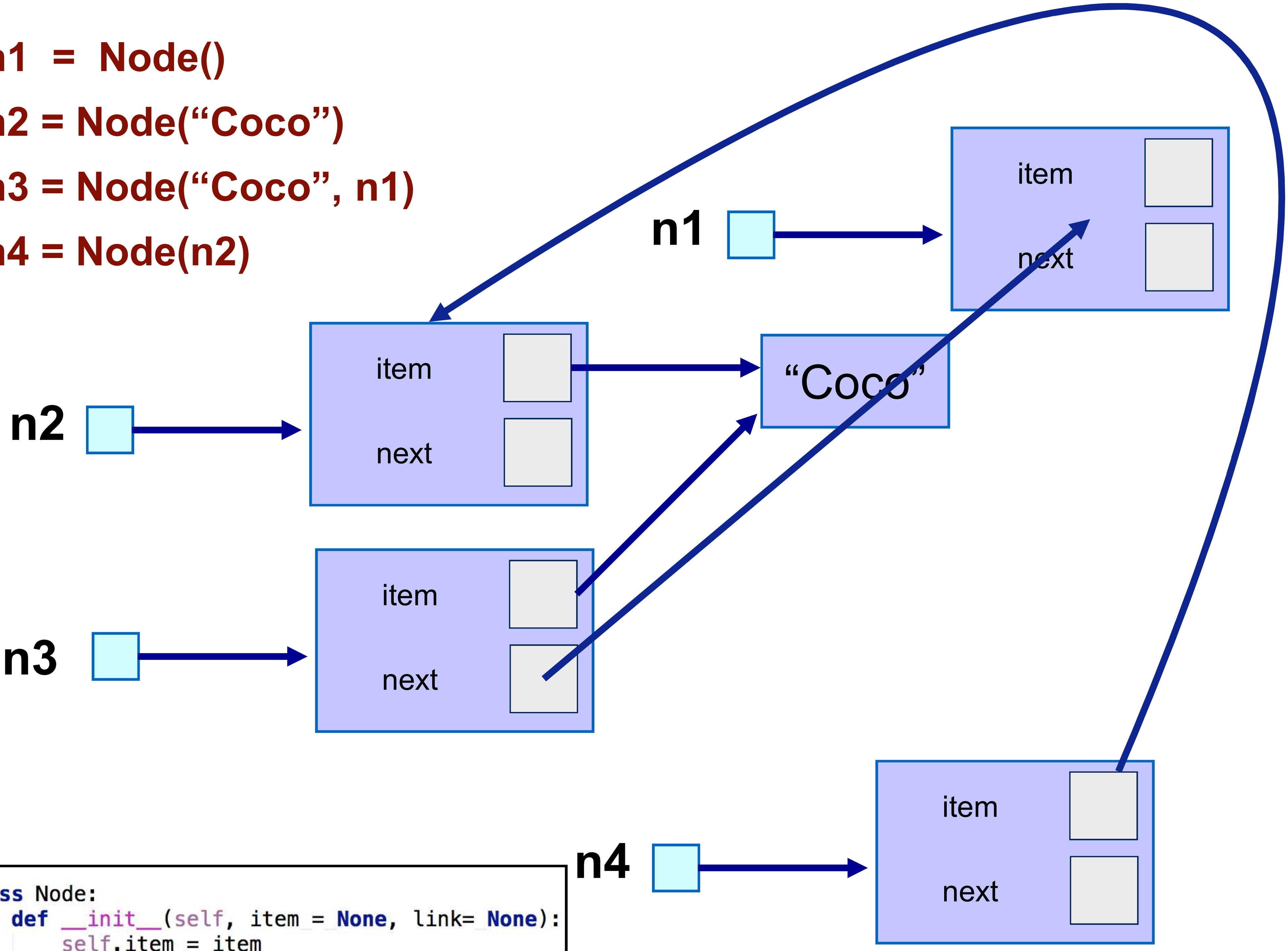


default values, if not supplied



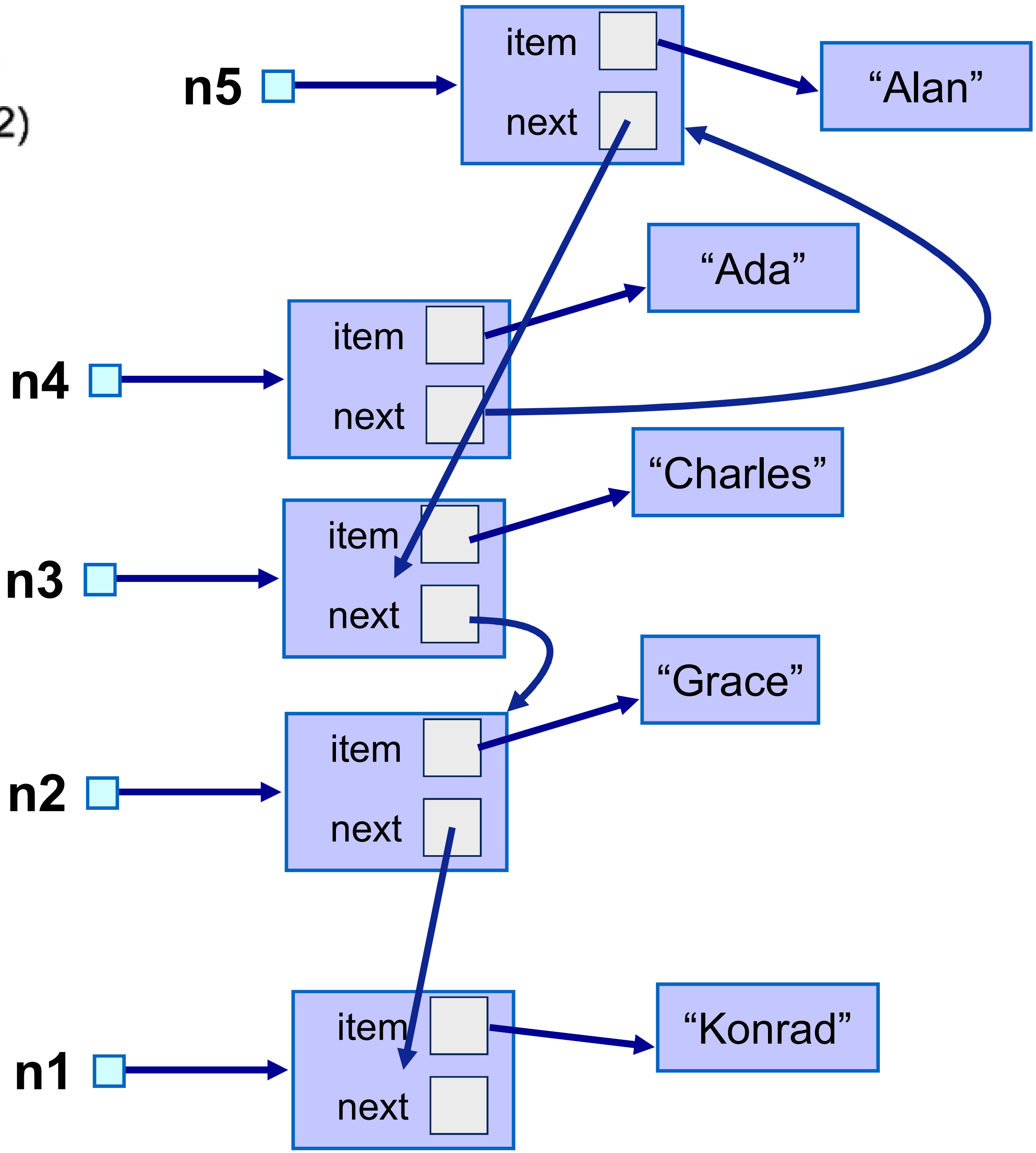
```
class Node:
    def __init__(self, item = None, link= None):
        self.item = item
        self.next = link
```

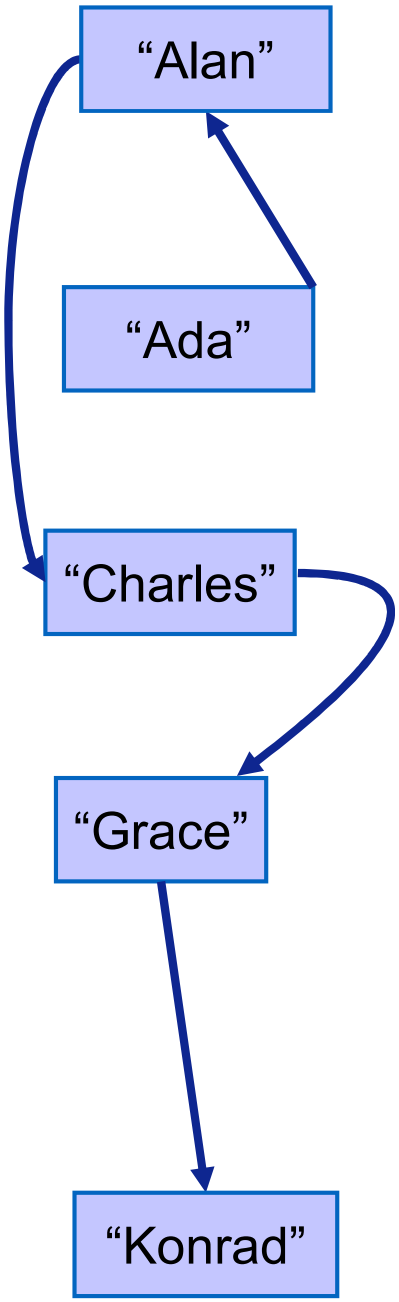
```
n1  =  Node()
n2 = Node(“Coco”)
n3 = Node(“Coco”, n1)
n4 = Node(n2)
```



```
class Node:
    def __init__(self, item = None, link= None):
        self.item = item
        self.next = link
```

```
>>> n1 = Node(“Konrad”)
>>> n2 = Node(“Grace”, n1)
>>> n3 = Node(“Charles”, n2)
>>> n4 = Node(“Ada”, n3)
>>> n5 = Node(“Alan”)
>>> n5.next = n3
>>> n4.next = n5
```





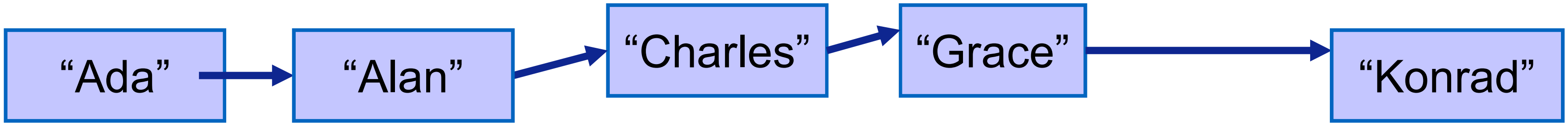
“Alan”

“Ada”

“Charles”

“Grace”

“Konrad”



“Ada”

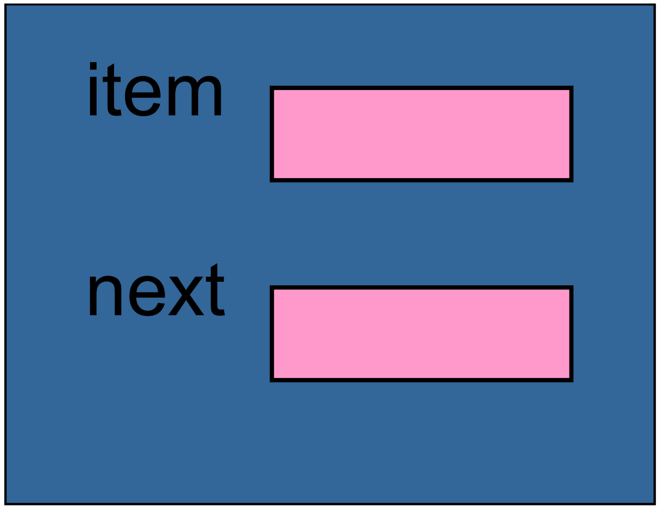
“Alan”

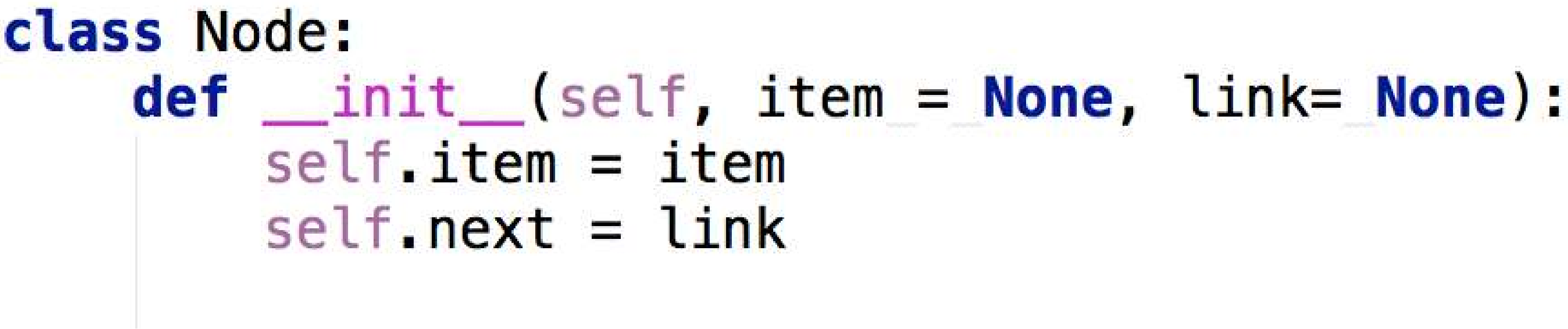
“Charles”

“Grace”

“Konrad”

Node



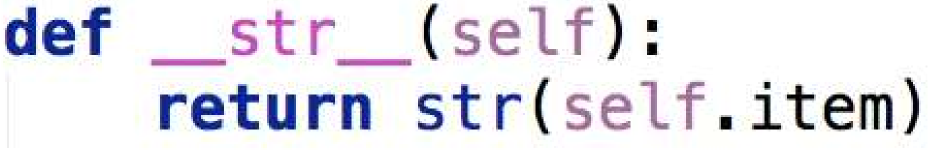


```python
class Node:
    def __init__(self, item = None, link= None):
        self.item = item
        self.next = link
```

Nodes

- Basic <u>building blocks</u> for simple **linked structures**.
- Allow easy addition of items
- Allow easy deletion of items
- Allow dynamic structures

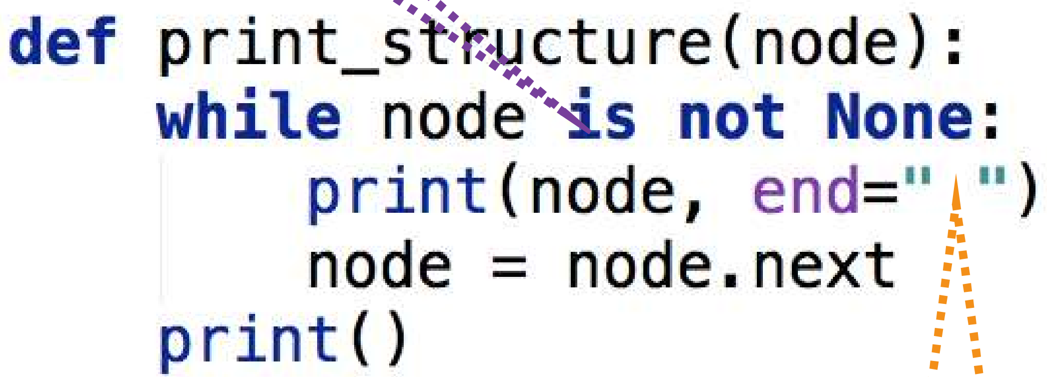
Print Node



```
def __str__(self):
    return str(self.item)
```

Print Linked Structure

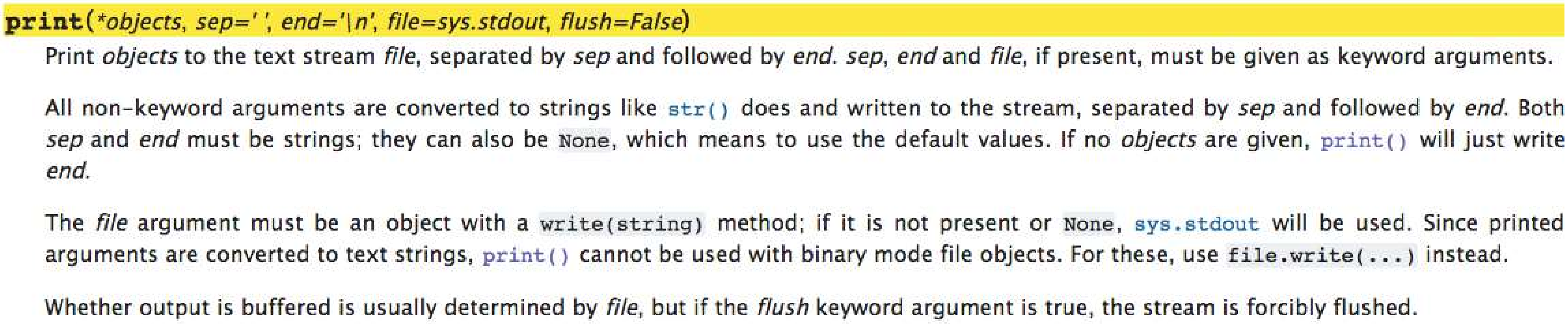
a is b, True if variables a and be point to the same object



```python
def print_structure(node):
    while node is not None:
        print(node, end=" ")
        node = node.next
    print()
```

by default end of **print** is a new line, we just want a space here

a == b, true if the objects referred to by variables a and b are equal.



`print`(**objects, sep=' ', end='\n', file=sys.stdout, flush=False*)

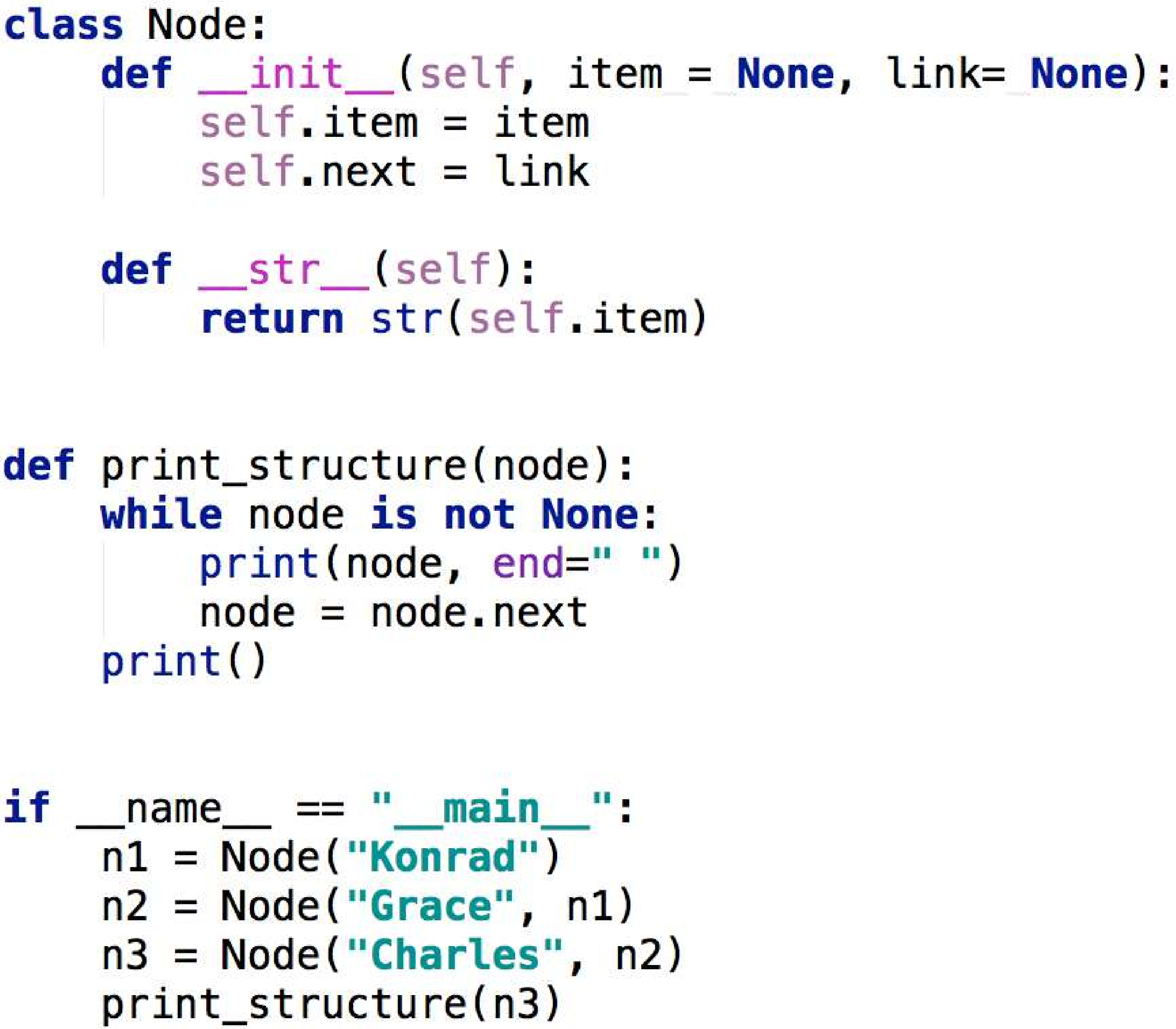
Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end* and *file*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

https://docs.python.org/3/library/functions.html#print



```python
class Node:
    def __init__(self, item = None, link= None):
        self.item = item
        self.next = link

    def __str__(self):
        return str(self.item)


def print_structure(node):
    while node is not None:
        print(node, end=" ")
        node = node.next
    print()


if __name__ == "__main__":
    n1 = Node("Konrad")
    n2 = Node("Grace", n1)
    n3 = Node("Charles", n2)
    print_structure(n3)
```

Summary

- Linked Structures
- Nodes