Abstract Data Type, Stack, Queues

Abstract Data Type (ADT)

- Description of an organized way to store data in computer
- Tells what kind of things can be stored
- Tells what operations are provided to store, manipulate, and extract information within the ADT
- Does not specify: How the information is arranged in the memory.
    - Hence called the Abstract Data Type
    - A promise of what you can do with the data type—not how.
- ADT is not an implementation.
    - Only a description of the functionality we wish someone would implement for us
    - Details of how, figured out during implementation

Abstract Data Type (ADT)

- To translate ADT into an implemented data structure
  - Finalize how information is actually structured, stored
  - Implement the supporting functions

- An ADT can be implemented in many different ways
  - Each implementation has pros and cons
  - But all implementations must implement the ADT’s promised functionality

- Data structures and ADT implementation beyond the scope of 5001
  - We will focus on ADTs here

Some ADTs in Python (I)

- List—an indexable, ordered list ADT that supports enforcing strict order, numerical indexing, etc.

- Dictionary—a map ADT that supports key to associated data indexing

- Set—supports mathematical set operations, e.g., set union, set intersection, but does not support indexing and ordering

Some ADTs in Python (II)

- Note, List, Dictionary, Set are concrete implementations in Python

- But, when discussed in class, we did not visit how they are implemented
  - We focused on what each of those ADTs do
  - and how to use them

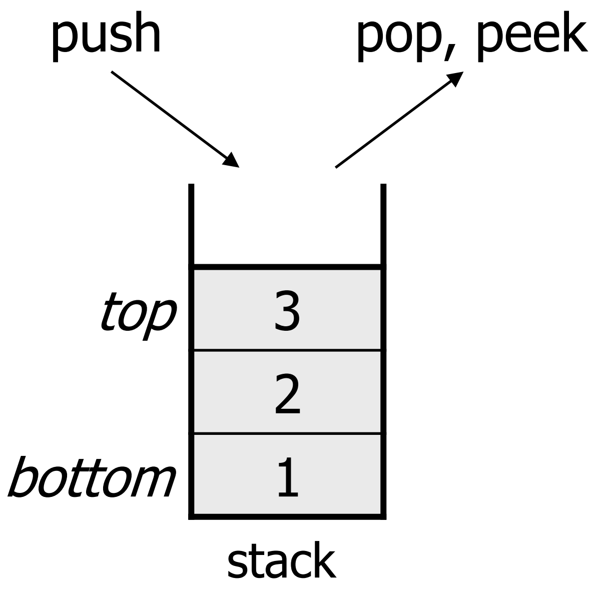
- Is an ADT’s implementation important? Absolutely!
  - Each implementation excels at some operations compared to others
    - “Excel” means the implementation is much faster in executing those operations
    - E.g., Python’s List implementation very fast in adding an item at the end but pretty slow inserting at the beginning of list

Intro to Stacks

- A collection based on the principle of adding elements and retrieving them in the opposite order.
    - Last-In, First-Out ("LIFO")
    - The elements are stored in order of insertion,
        - but we do not think of them as having indexes.
    - The client can only add/remove/examine
        - the last element added (the "top").

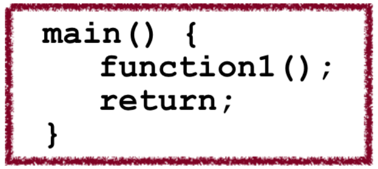
Basic stack operations:

- Necessary operations
  - push(value) - add an element onto the top of the stack
  - pop() - remove the element from the top of the stack and return it
    - Takes no arguments!
- Optional operations
  - peek() - look at the element at the top of the stack, but don’t remove it
  - isEmpty() – return True if the stack is empty; return False if it has at least one element.

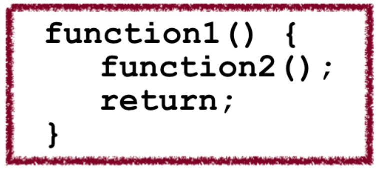


Stack applications (I)

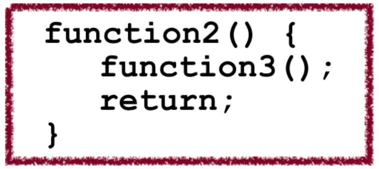
- Programming languages and compilers:
  - A stack is built in to every program running on your PC — the stack is a memory block that gets used to store the state of memory when a function is called, and to restore it when a function returns.
    - method calls are placed onto a stack *(call=push, return=pop)*
  - Why use stacks for function calls?
    - Assume the following program,



```
main() {
    function1();
    return;
}
```



```
function1() {
    function2();
    return;
}
```



```
function2() {
    function3();
    return;
}
```

    - main calls function1, which calls function2, which calls function3.
    - First, function3 returns, then function2 returns, then function1 returns, then main returns.
    - This is a LIFO pattern!

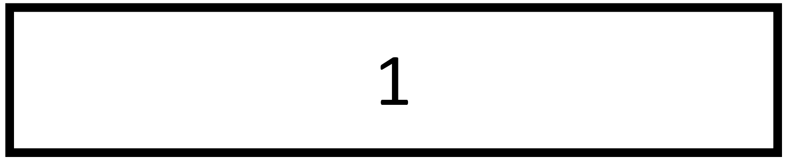
Stack applications (II)

- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } match
  - convert "infix" expressions to pre/postfix

- Sophisticated algorithms:
  - many programs use an "undo stack" of previous operations
    - We want to undo the most recent change done to the program
    - Undo things in the reverse order of how we did them

Example operations on stack

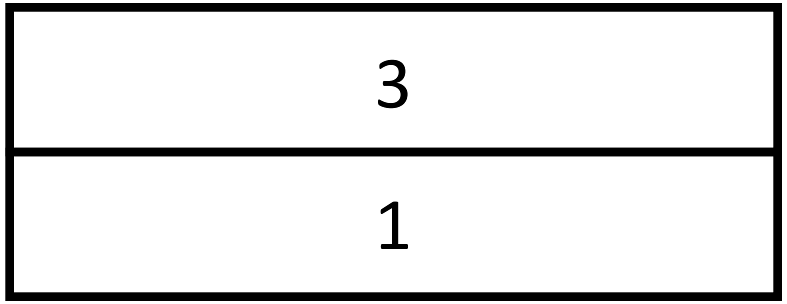
- Push(1)



1

Example operations on stack

- Push(3)

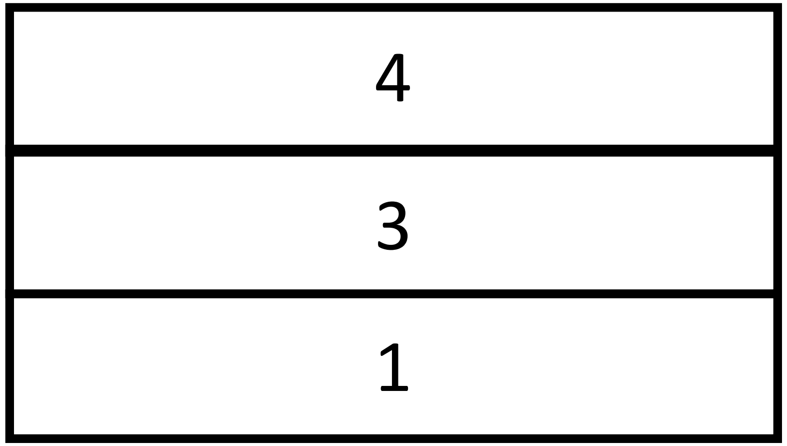


3

1

Example operations on stack

- Push(4)



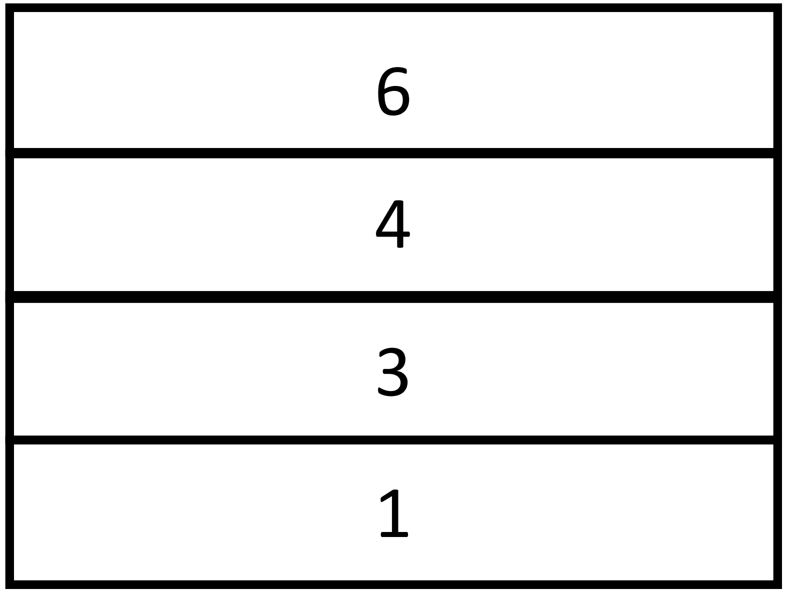
4

3

1

Example operations on stack

- Push(6)



6

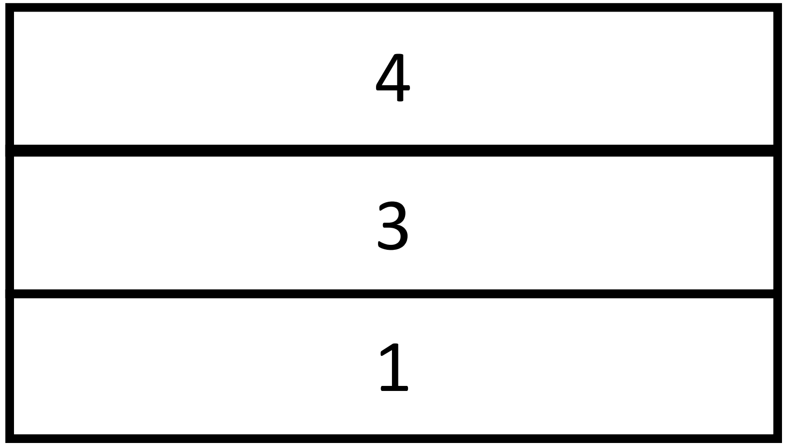
4

3

1

Example operations on stack

- Pop()
- Return value→6



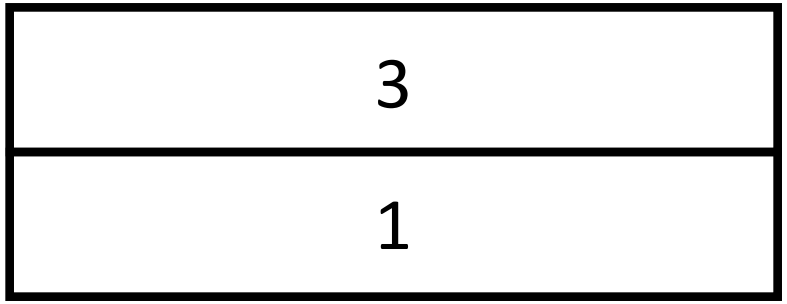
4

3

1

Example operations on stack

- Pop()
- Return value→4

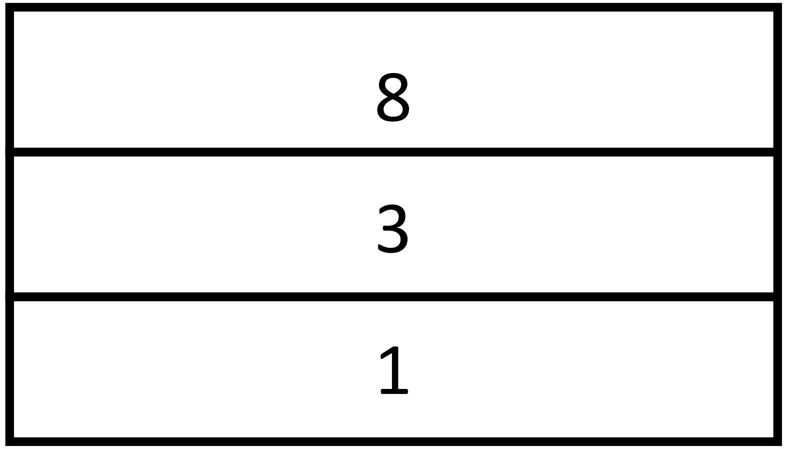


3

1

Example operations on stack

- Push(8)



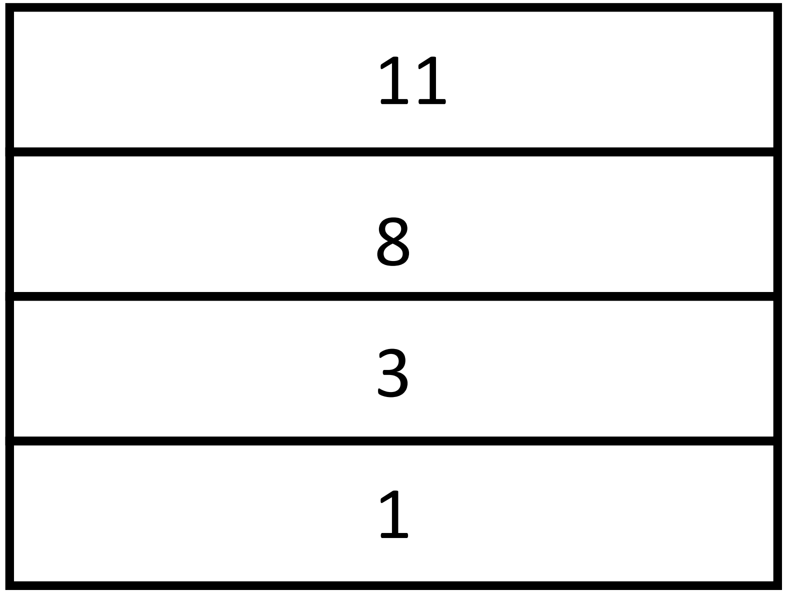
8

3

1

Example operations on stack

- Push(11)



11

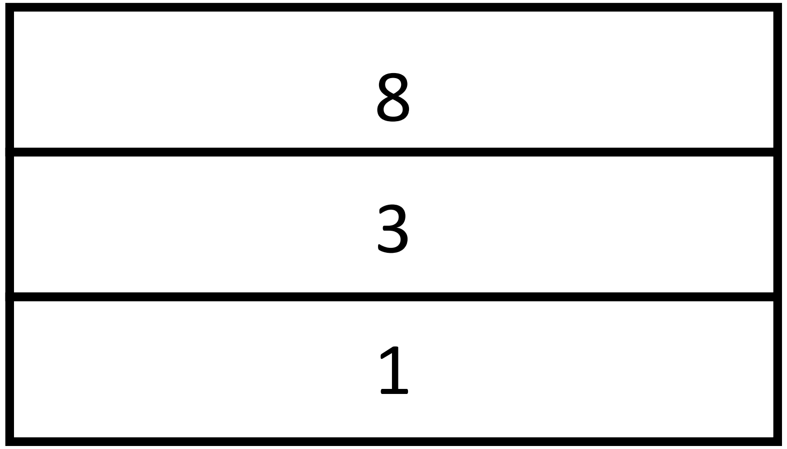
8

3

1

Example operations on stack

- Pop()
- Return value→11



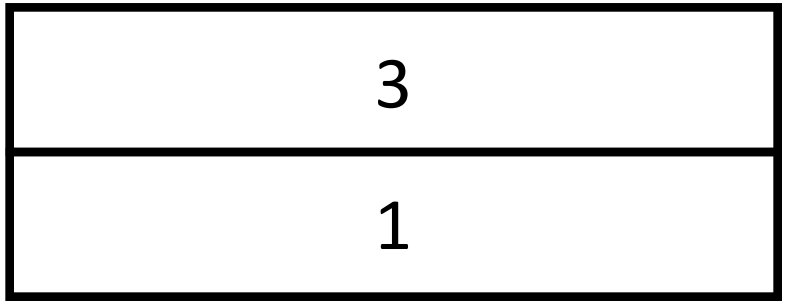
8

3

1

Example operations on stack

- Pop()
- Return value→8

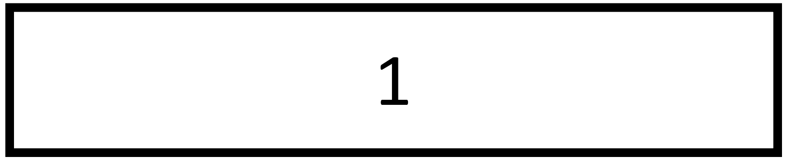


3

1

Example operations on stack

- Pop()
- Return value→3



1

Example operations on stack

- Pop()
- Return value→1
  - Stack empty!

Example operations on stack

- What if we do another Pop() on empty stack?
    - Would it return 0? -1? Error?
    - Not defined by the definition of ADT
    - Decided by the implementation!

Implementing Stack ADT!

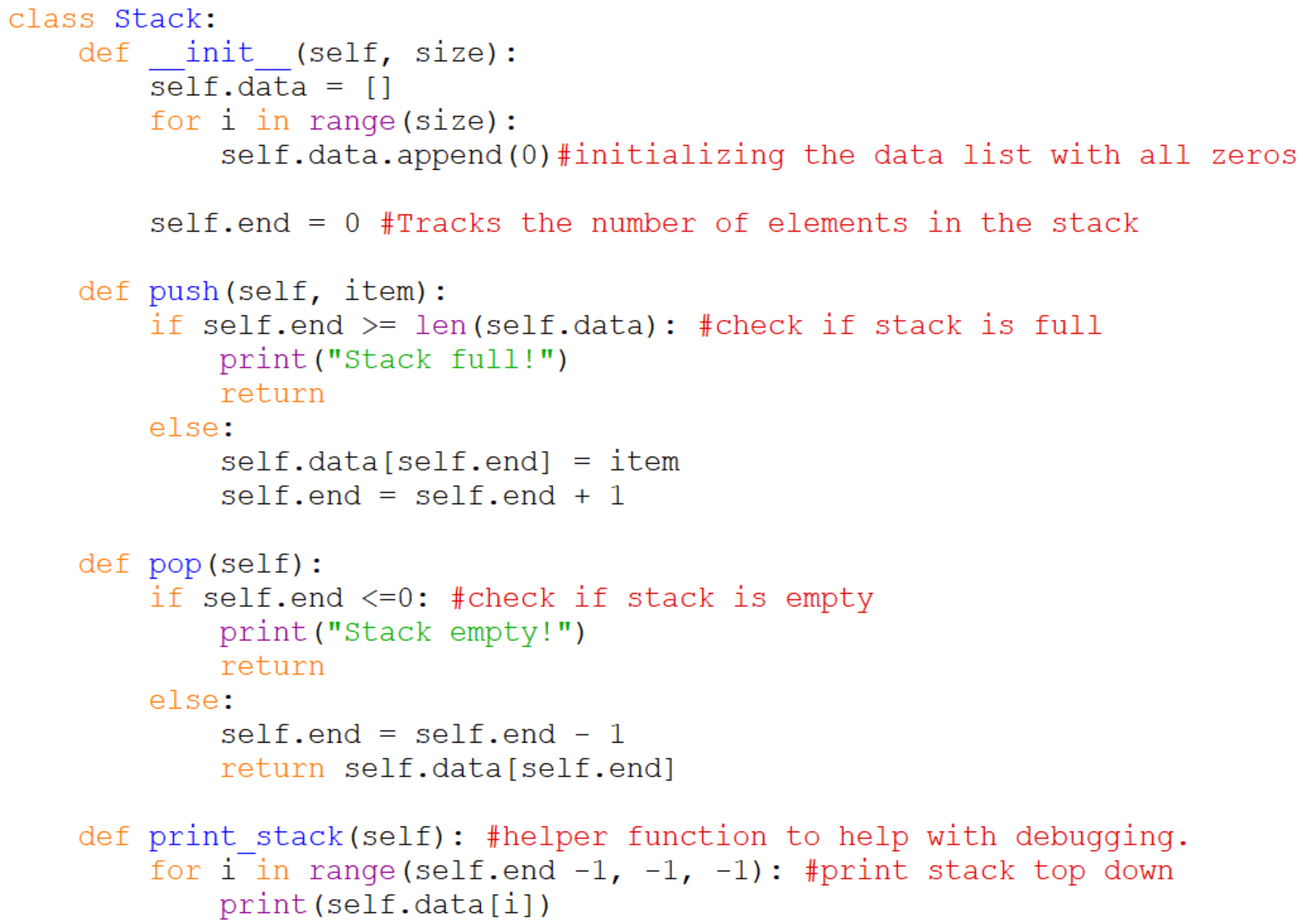
How to implement a stack?

- Design a way to store the information in computer
  - Need to store an ordered collection of items. How? Python List!
    - Assume, list index 0 stores bottom-most item and higher indices store upward items
  - We will design a stack with fixed max capacity
    - Start storing items from index 0
    - As more items pushed, increment the index for next item
      - Need to track current value of index in a data item

- Design implementations for push() and pop() operations

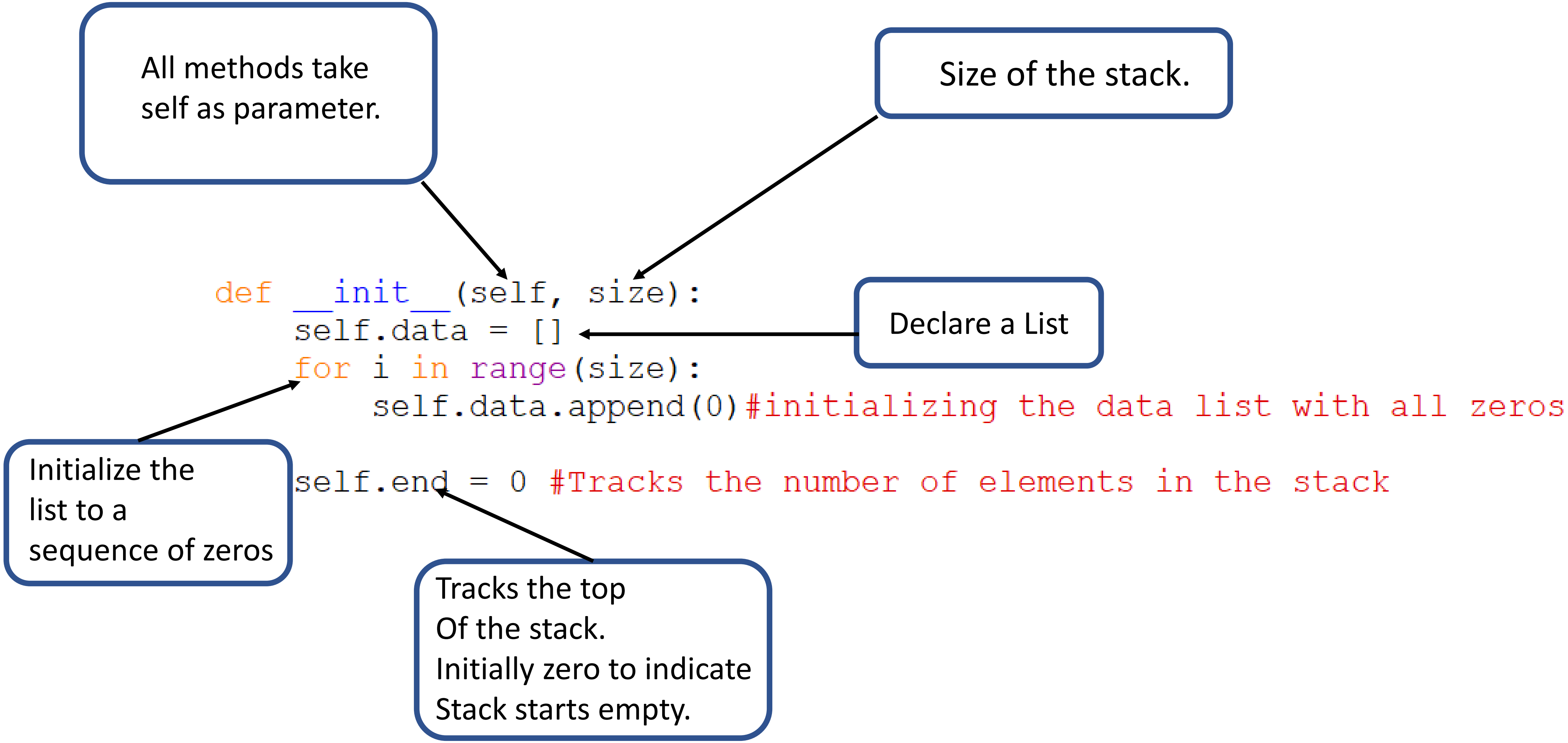
- Implement stack as a class of its own

Stack class



```
class Stack:
    def __init__(self, size):
        self.data = []
        for i in range(size):
            self.data.append(0)#initializing the data list with all zeros

        self.end = 0 #Tracks the number of elements in the stack

    def push(self, item):
        if self.end >= len(self.data): #check if stack is full
            print("Stack full!")
            return
        else:
            self.data[self.end] = item
            self.end = self.end + 1

    def pop(self):
        if self.end <=0: #check if stack is empty
            print("Stack empty!")
            return
        else:
            self.end = self.end - 1
            return self.data[self.end]

    def print_stack(self): #helper function to help with debugging.
        for i in range(self.end -1, -1, -1): #print stack top down
            print(self.data[i])
```

Constructor



All methods take self as parameter.

Size of the stack.

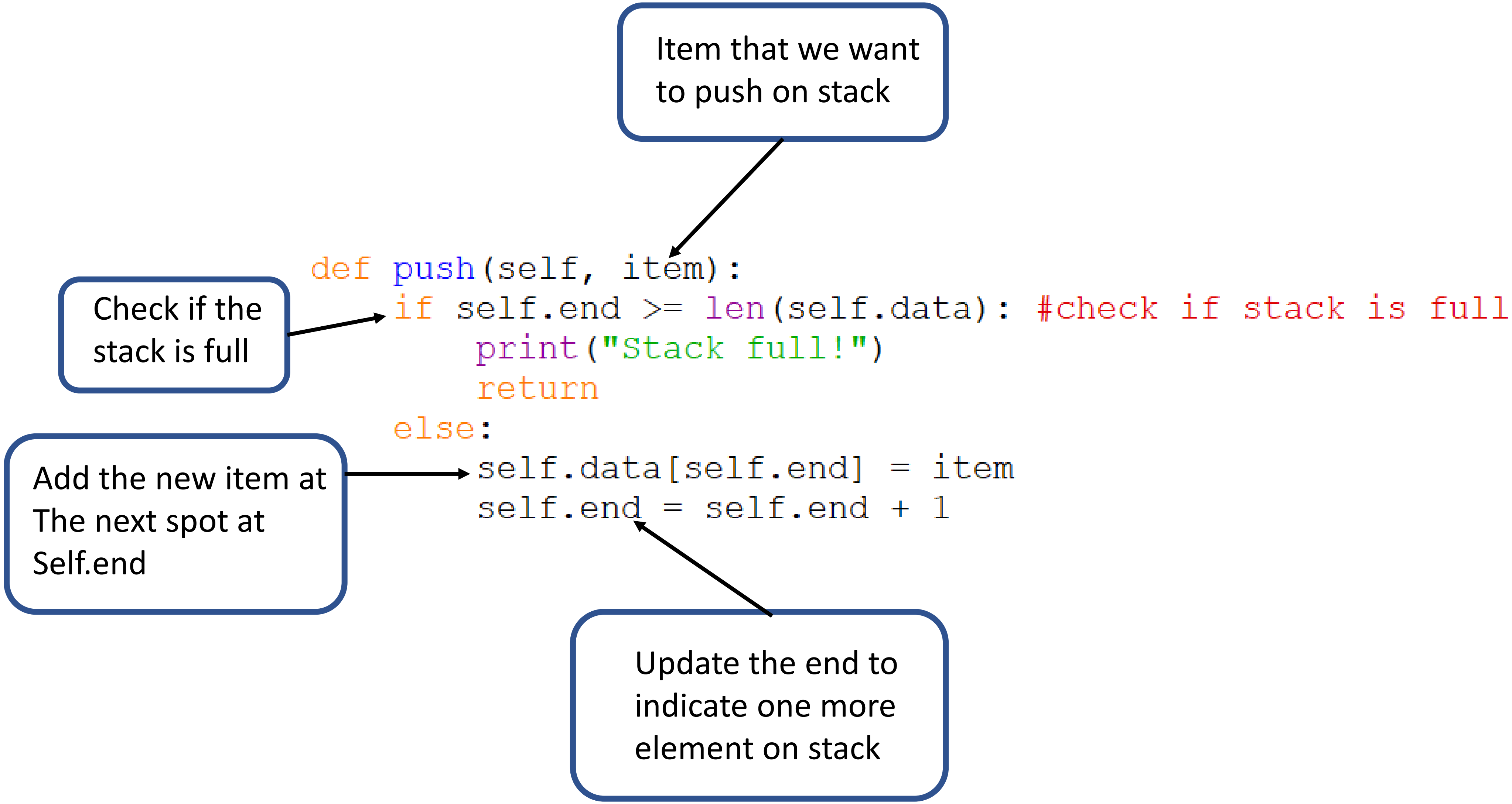
```
def __init__(self, size):
    self.data = []
    for i in range(size):
        self.data.append(0)#initializing the data list with all zeros

    self.end = 0 #Tracks the number of elements in the stack
```

Declare a List

Initialize the list to a sequence of zeros

Tracks the top
Of the stack.
Initially zero to indicate
Stack starts empty.

Push()



Item that we want to push on stack

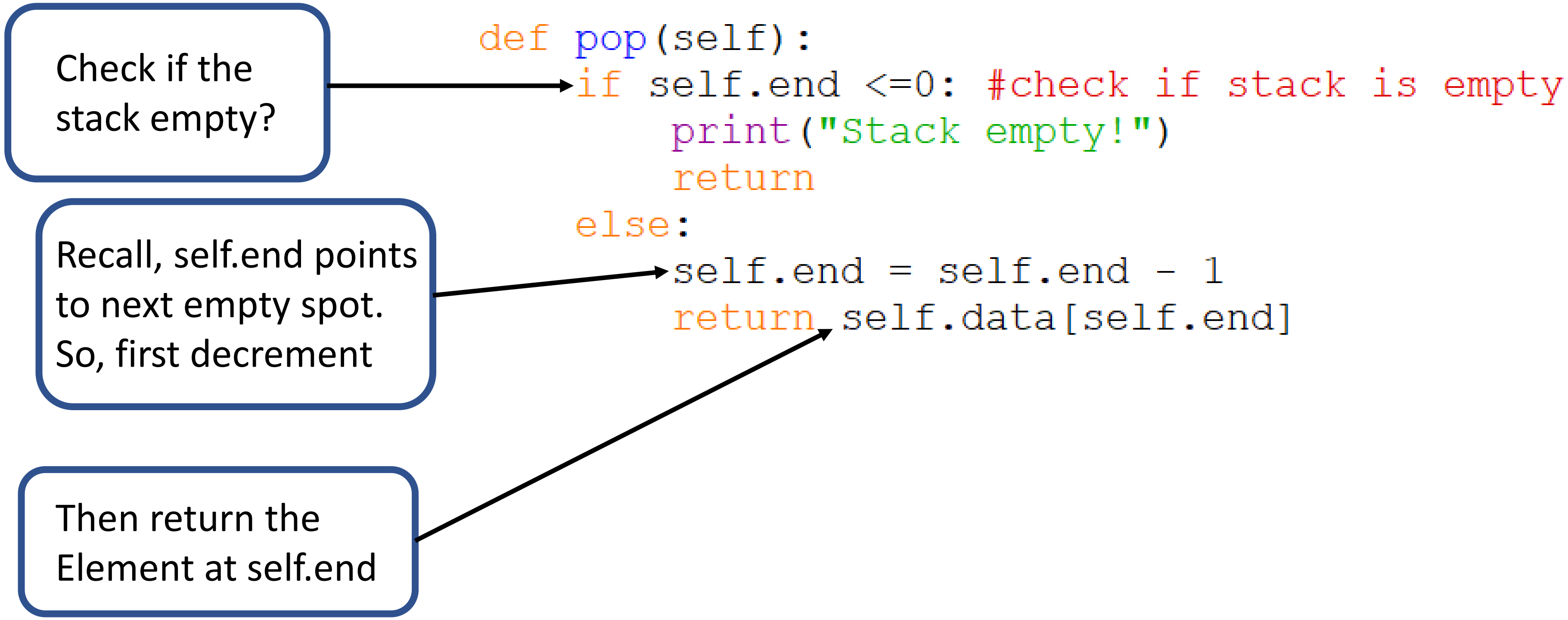
```
def push(self, item):
    if self.end >= len(self.data): #check if stack is full
        print("Stack full!")
        return
    else:
        self.data[self.end] = item
        self.end = self.end + 1
```

Check if the stack is full

Add the new item at The next spot at Self.end

Update the end to indicate one more element on stack

Pop()



Check if the stack empty?

Recall, self.end points to next empty spot. So, first decrement

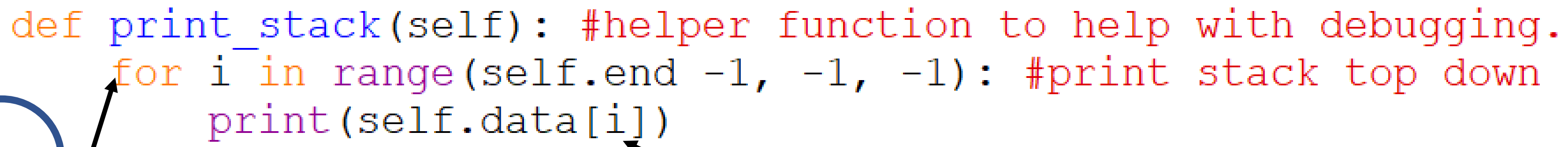
Then return the Element at self.end

```
def pop(self):
    if self.end <=0: #check if stack is empty
        print("Stack empty!")
        return
    else:
        self.end = self.end - 1
        return self.data[self.end]
```

Print_stack()

This is an optional function for help with debugging the implementation.

Print_stack() simply displays all the contents of the stack from top to bottom.



```
def print_stack(self): #helper function to help with debugging.
    for i in range(self.end -1, -1, -1): #print stack top down
        print(self.data[i])
```

Print top down
Hence range from
Self.end -1
To -1 (stop at 0).
Step size -1 to count
backwards

Print the item
At position i

Main script

Main script to test the stack.

```
my_stack = Stack(6)#create a new stack with size=6

#ask user what they want to do with the stack
while True:
    cmd = input("push, pos, print_stack, or exit?")
    if cmd == "push":
        value = input("Enter the item to push on stack: ")
        my_stack.push(value)
    elif cmd == "pop":
        value = my_stack.pop()
        print("pop() rturned ", value)

    elif cmd == "print_stack":
        my_stack.print_stack()
    elif cmd=="exit":
        break
    else:
        #Invalid option
        print("Please try again!")
```

Create a stack
Instance with
Size = 6

Test stack inside an
Infinite while loop

If “pop” entered,
pop a value
From stack

Ask user what to do?

If “push” entered, ask
User what value to push

Push the value on stack

If “print_stack” entered, call print_stack()

If “exit” entered, break out of the loop

If an invalid option entered, ask user to try again

Demonstration!

- Download and run the python file Stack_Implementation.py from Canvas

- Note that, we have only implemented push(), pop() methods in stack.
  - Implementation of the other optional methods is left as an exercise for you
  - Extend the implementation of the Stack class by including a peek() method

Python's Built-In Support for Stacks

- Python List supports the behaviors of Stack ADT
  - But doesn’t use the standard name push for adding items

- In Python List
  - Push: list.append(x)
    - Pushes an item(x) onto a list-based stack; returns no value
  - Pop: list.pop()
    - Removes and returns the last item (the one at the highest index) from the list
    - The list shrinks in length by one

- NOTE: list.append(x) and list.pop() automatically grow and shrink the list

Stack class using List

```
class Stack:
    def __init__(self, size):
        self.data = list() #an empty list to store stack data
        self.size = size


    def push(self, item):
        if len(self.data) >= self.size: #check if stack is full
            print("Stack full!")
            return
        else:
            self.data.append(item)

    def pop(self):
        if len(self.data) <=0: #check if stack is empty
            print("Stack empty!")
            return
        else:
            return self.data.pop()

    def print_stack(self): #helper function to help with debugging.
        for i in range(len(self.data) -1, -1, -1): #print stack top down
            print(self.data[i])
```

Key Changes:

1) Create an empty list. The size is optional. If no size used, Python will automatically grow and shrink the list.

2) No need to track the end of the stack. Python does that for us.

Demonstration

- Download and run the python file Stack_built_in.py from Canvas

- Note that, we have only implemented push(), pop() methods in stack.
  - Implementation of the other optional methods is left as an exercise for you
  - Extend the implementation of the Stack class by including a peek() method

Demonstration (Solution)

```
 def peek(self):
        return self.data[len(self.data) -1]
```

In the main script

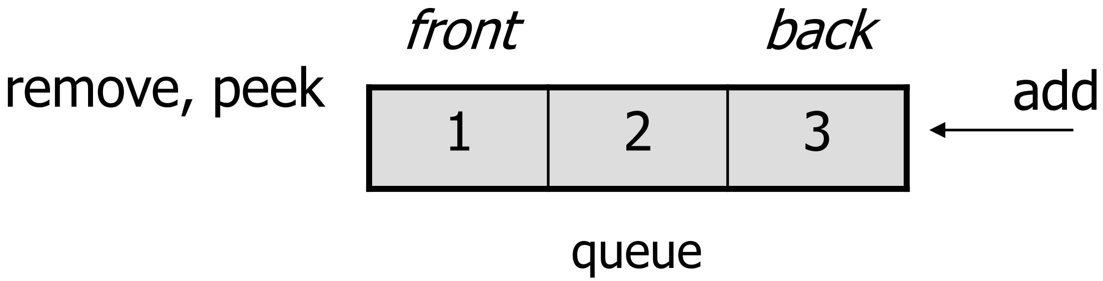
```
cmd = input("push, pop, print_stack, peek or exit?")
```

Add the following else-if statement

```
 elif cmd=="peek":
        value = my_stack.peek()
        print("peek() returned ", value)
```

Queue ADT

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.



front *back*

remove, peek

1 2 3

add

queue

Queue operations

- Basic queue operations:
  - **add** (enqueue): Add an element to the back/tail/end.
  - **remove** (dequeue): Remove and return the front/head element.

- Optional queue operations
  - **is_full()**: returns True if the queue is full, otherwise returns False
  - **is_empty()**: returns True if the queue is empty, otherwise returns False
  - **peek()**: returns the front element without removing from the queue.
  - **count()**: returns the number of elements present in the queue

Queues in Computer Science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send

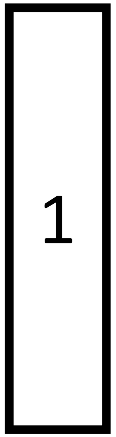
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Example operations on queue

- Enqueue(1)

- **HEAD** **TAIL**

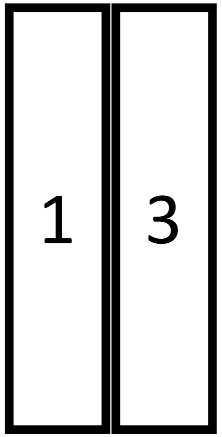


1

Example operations on queue

- Enqueue(3)

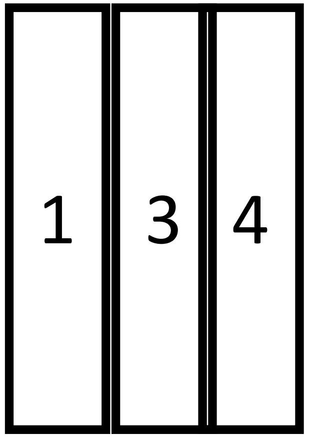
- **HEAD** **TAIL**



Example operations on queue

- Enqueue(4)

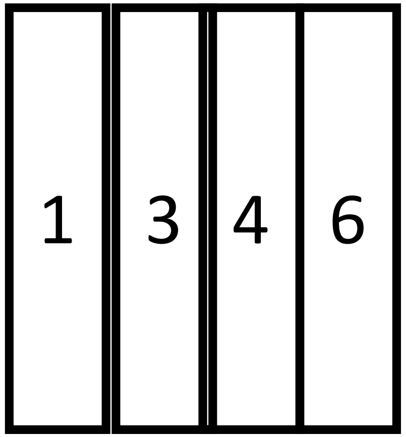
- **HEAD** **TAIL**



Example operations on queue

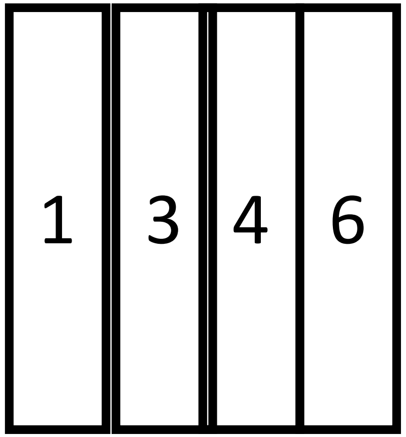
- Enqueue(6)

- **HEAD** **TAIL**



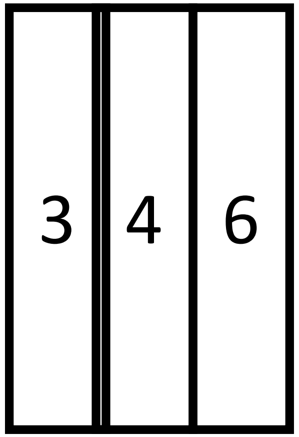
Example operations on queue

- Dequeue()
- Removes and returns → 1
- **HEAD** **TAIL**



Example operations on queue

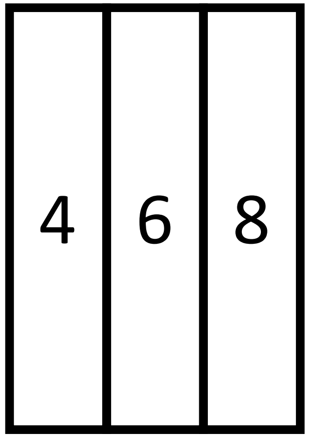
- Dequeue()
- Removes and returns → 3
- **HEAD** **TAIL**



Example operations on queue

- Enqueue(8)

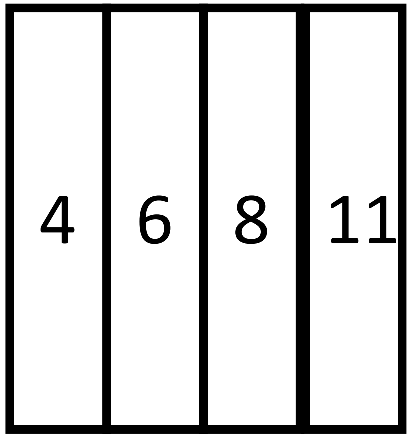
- **HEAD** **TAIL**



Example operations on queue

- Enqueue(11)

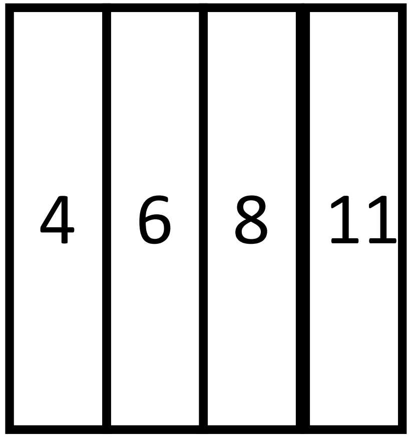
- **HEAD** **TAIL**



4 6 8 11

Example operations on queue

- Dequeue()
- Removes and returns → 4
- **HEAD** **TAIL**

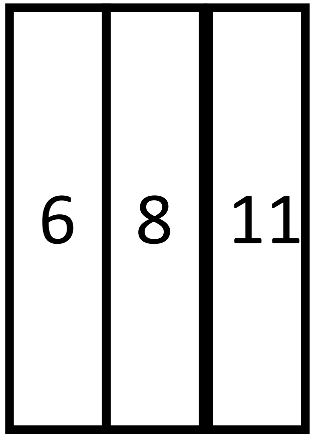


4	6	8	11

Example operations on queue

- Dequeue()
- Removes and returns → 6
- **HEAD**

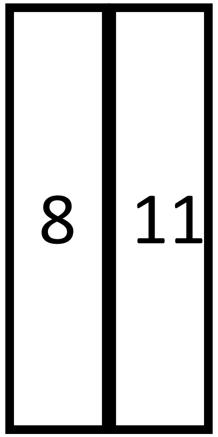
TAIL



6	8	11

Example operations on queue

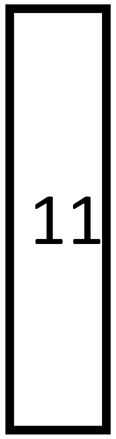
- Dequeue()
- Removes and returns → 8
- **HEAD** **TAIL**



8 11

Example operations on queue

- Dequeue()
- Removes and returns → 11
- **HEAD** **TAIL**



11

Example operations on queue

- What happens if we perform another Dequeue() on an empty queue?
  - Returns 0?
  - Returns -1?
  - Error message?

- Depends on the implementation.

- This behavior is not defined in the queue ADT

Implementing Queue ADT!

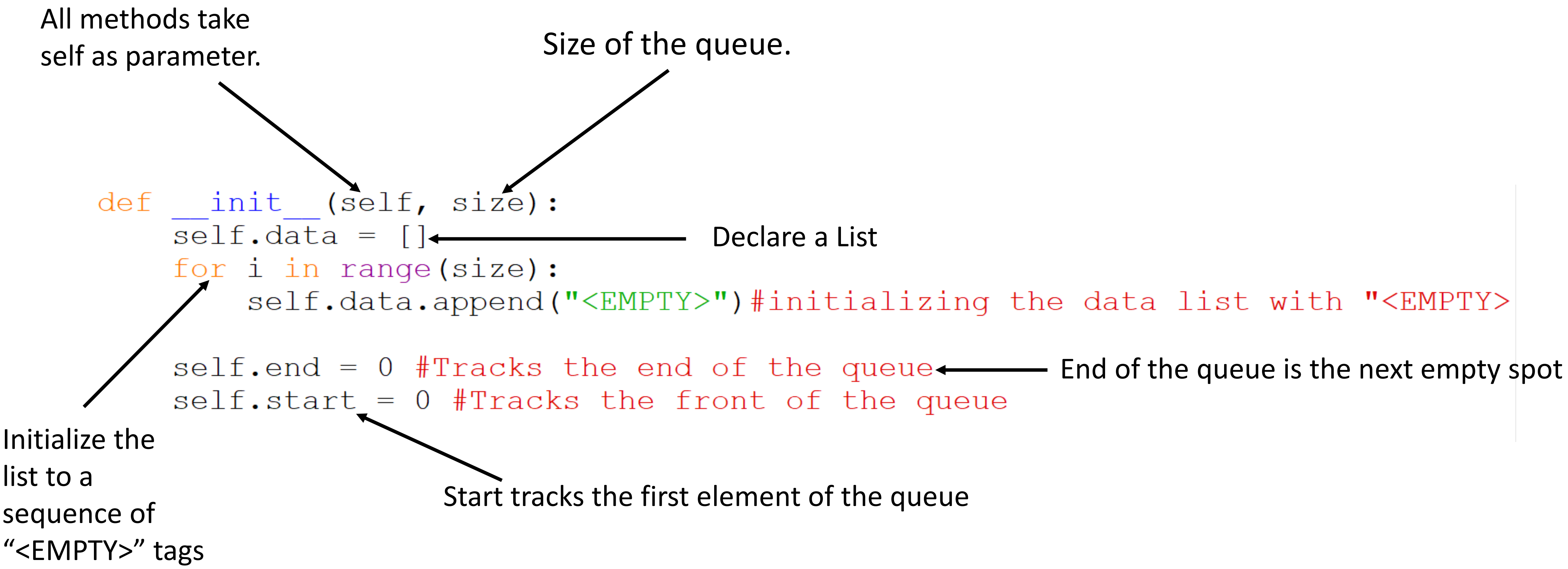
We will

- Use List to store data
- Pre size the queue to hold a max number of elements
- Start queuing at index 0
- Newly added items added to higher indexes
- Items dequeued from low index end
- Need to track both the front and the tail/back end of the queue

Queue class

```
class Queue:
    def __init__(self, size):
        self.data = []
        for i in range(size):
            self.data.append("<EMPTY>")#initializing the data list with "<EMPTY>

        self.end = 0 #Tracks the end of the queue
        self.start = 0 #Tracks the front of the queue

    def enqueue(self, item):
        if self.end >= len(self.data): #check if queue is full
            print("Queue full!")
            return
        else:
            self.data[self.end] = item
            self.end = self.end + 1

    def dequeue(self):
        if self.start == self.end: #check if queue is empty
            print("Queue empty!")
            return
        else:
            item = self.data[self.start]
            self.data[self.start] = "<EMPTY>"
            self.start = self.start + 1
            return item

    def print_q(self): #helper function to help with debugging.
        for i in range(self.start, self.end): #print queue start to end
            print(self.data[i])
```

Constructor



```
def __init__(self, size):
    self.data = []
    for i in range(size):
        self.data.append("<EMPTY>")#initializing the data list with "<EMPTY>

    self.end = 0 #Tracks the end of the queue
    self.start = 0 #Tracks the front of the queue
```

All methods take self as parameter.

Size of the queue.

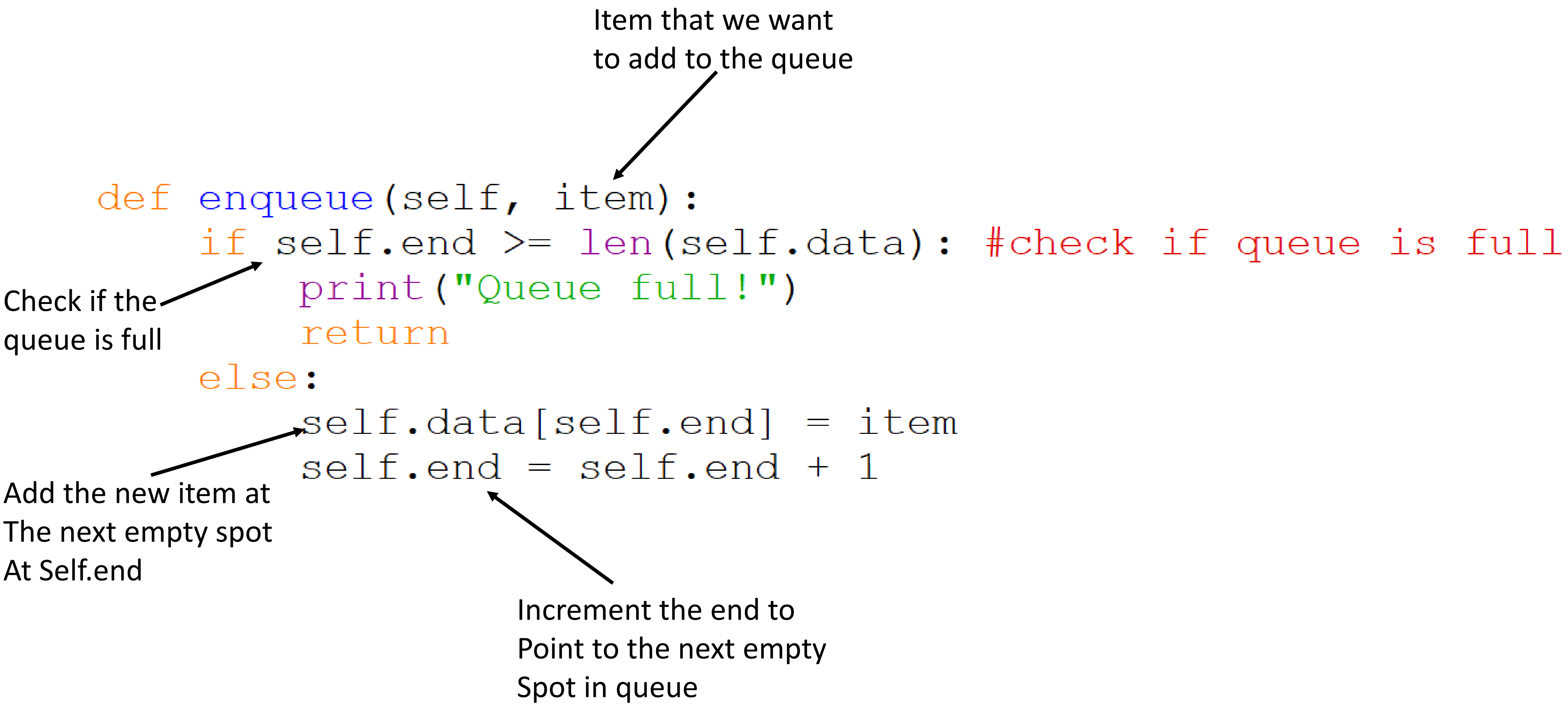
Declare a List

Initialize the list to a sequence of “<EMPTY>” tags

End of the queue is the next empty spot

Start tracks the first element of the queue

Enqueue()



Item that we want
to add to the queue

```
def enqueue(self, item):
    if self.end >= len(self.data): #check if queue is full
        print("Queue full!")
        return
    else:
        self.data[self.end] = item
        self.end = self.end + 1
```

Check if the
queue is full

Add the new item at
The next empty spot
At Self.end

Increment the end to
Point to the next empty
Spot in queue

Dequeue()

```
def dequeue(self):
    if self.start == self.end: #check if queue is empty
        print("Queue empty!")
        return
    else:
        item = self.data[self.start]
        self.data[self.start] = "<EMPTY>"
        self.start = self.start + 1
        return item
```

Check if the
queue empty?

Recall, items
Removed from
the front of
queue

Returned item is
Deleted from queue

Increment start
To point to the
New front element

Return the item

Print_q()

Print all items from
Start to end

Print the item
At position i

```
def print_q(self): #helper function to help with debugging.
    for i in range(self.start, self.end): #print queue start to end
        print(self.data[i])
    print(self.data, "Start: ", self.start, ", End", self.end)
```

Print all items in
Self.data list along
With queue’s start
and end indices

Demonstration!

- Download and run the python file Queue_buggy.py from Canvas

- TODO: Test the queue implementation thoroughly.
  - There is a logical bug in the Queue’s current implementation
  - What is the bug?

Demonstration (key)!

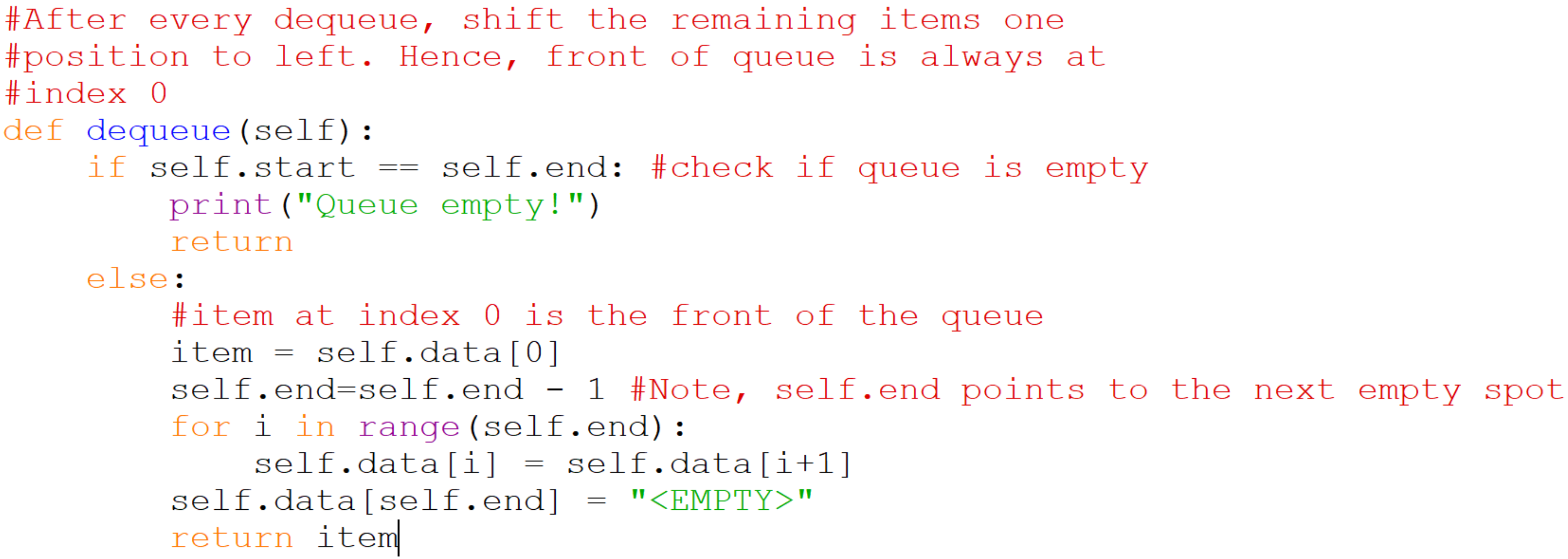
- The bug will show up if you follow the following steps
    - Add a few elements to the queue
    - Dequeue some of the elements
    - Now add more elements such that the queue gets full
    - Print the queue contents
    - How many elements does the queue have?
    - What is the queue’s capacity?

- You will see that the queue still has room but the program complains that it is full!

How to fix this bug?

- Need to shift the queue elements one position to left after every dequeue
  - All the empty spots will be at the end
  - Item at index = 0 is always the front element of the queue
  - self.start attribute is obsolete (not needed) now

Updated dequeue()



```
#After every dequeue, shift the remaining items one
#position to left. Hence, front of queue is always at
#index 0
def dequeue(self):
    if self.start == self.end: #check if queue is empty
        print("Queue empty!")
        return
    else:
        #item at index 0 is the front of the queue
        item = self.data[0]
        self.end=self.end - 1 #Note, self.end points to the next empty spot
        for i in range(self.end):
            self.data[i] = self.data[i+1]
        self.data[self.end] = "<EMPTY>"
        return item
```

Issues with updated dequeue()

- Every dequeue() copies and shift the remaining items to left

- The copy and shift may take long as the queue grows

Demonstration!

- Download and run the python file Queue_fixed.py from Canvas

- TODO: Implement a count() method in the Queue class that returns the total number of items in the queue.

Demonstration (solution)!

```
def count(self):
        return self.end-self.start
```

In the main script:

```
cmd = input("enq, deq, print_q, count, or exit?")
```

Add the following else-if statement

```
elif cmd == "count":
        value = my_queue.count()
```