

Stacks and queues in Python

IB DP Computer Science HL • Friday 11 September 2026

Name _____ Date _____

Learn how stacks and queues control the order in which data is accessed. Build and test both structures, then review the Python skills needed for B2.4 Programming algorithms.

Reading: Baumgarten, Ganea and Turland, printed pages 352–357 in the supplied extract. These include B2.2.3 Stacks and B2.2.4 Queues.

How to use this handout

Lesson 1: work through pages 1–4. Lesson 2: complete the core exercises on pages 6–8. Page 5 is a complete textbook question checklist. Write programs in your editor and longer explanations in your notebook. Extra practice is labelled; it is not all due today.

Two rules for accessing data

An abstract data structure describes what operations are allowed. Its implementation describes how the data is stored. A Python list can implement either a stack or a queue, but your program must follow the chosen access rules.

Feature	Stack	Queue
Order	LIFO: last in, first out. Also called FILO.	FIFO: first in, first out.
Add an item	push at the top	enqueue at the rear
Remove an item	pop from the top	dequeue from the front
Inspect without removing	peek at the top	front at the front
Checks	isEmpty; isFull for bounded storage	isEmpty; isFull for bounded storage
Typical use	Undo the most recent action	Process print jobs in arrival order

A Predict before you run

Write the complete contents after each operation. In every row, show the stack from bottom to top and the queue from front to rear. Both start empty. Record the value removed or inspected.

Step	Stack operation and result	Queue operation and result
1	push("A") →	enqueue("A") →
2	push("B") →	enqueue("B") →
3	push("C") →	enqueue("C") →
4	pop() →	dequeue() →
5	peek() →	front() →

Explain why step 4 removes different values: _____

Build and test a stack

Reading the textbook diagrams

In the diagrams, `topIndex` identifies the occupied top position: `-1` means empty; `0` means one item; and `capacity - 1` means full. Push checks for space, increases `topIndex` and stores the new item. Pop checks for data and decreases `topIndex`. Data beyond the pointer is no longer part of the stack, even if an old value remains in the array.

The prose also mentions a pointer to the next free position. That is a different valid convention: empty is `0` and full is `capacity`. Do not mix the two conventions. Always state which one you use.

A static array reserves a fixed number of positions. A linked list can grow by linking new nodes and is accessed by following those links. Both can implement a stack: the storage differs, but the LIFO rule stays the same.

B A stack with capacity seven

Textbook programming exercise 1, page 357. Use a Python list for storage and enforce a maximum of seven items. The list can resize internally; the capacity rule makes this a bounded stack. Its top is the last list element, so no separate pointer is needed.

```
CAPACITY = 7
```

```
def is_empty(stack):
    return len(stack) == 0

def is_full(stack):
    return len(stack) == CAPACITY

def push(stack, value):
    if is_full(stack):
        raise OverflowError("Stack is full")
    stack.append(value)

def pop(stack):
    # TODO: reject an empty stack with IndexError.
    # TODO: remove AND return its last item.
    pass

def peek(stack):
    # TODO: reject an empty stack with IndexError.
    # TODO: return its last item without removing it.
    pass
```

1. Complete pop and peek. Start with an empty stack. Push these words in order: red, blue, green, gold, white. Display the top. Pop two words and display the top again. Predict both removed words and the top before and after the pops.

2. Check the boundaries in a separate test: fill a fresh stack with seven words, then try an eighth push. Also try pop and peek on an empty stack. Catch each expected exception with try/except and print a helpful message. Failed operations must leave the stack unchanged.

3. Explain the maximum size and what your program does when it is exceeded. This answers reflection activity 1 on page 357.

Underflow means attempting removal from an empty structure. Overflow means attempting insertion when bounded storage is full. An empty peek is also invalid, although it removes nothing.

Build and reverse a queue

Front and rear

The front identifies the next item to be removed; new items join at the rear. In the list implementation below, index 0 is the front and the last item is the rear. Dequeue returns the removed value so that another operation can use it.

```
def enqueue(queue, value):
    queue.append(value)

def dequeue(queue):
    if len(queue) == 0:
        raise IndexError("Queue is empty")
    return queue.pop(0)

def front(queue):
    if len(queue) == 0:
        raise IndexError("Queue is empty")
    return queue[0]
```

This queue has no chosen maximum capacity. A bounded version would check isFull before enqueue. Python list pop(0) shifts the remaining items; it is convenient for this exercise, but a circular buffer or deque avoids this shifting.

Why circular queues are useful

A fixed array can reuse positions freed by dequeues. After the last array position, a pointer wraps to position 0. For example, with capacity 4, advancing index 3 gives $(3 + 1) \% 4 = 0$. The physical array need not look like the logical front-to-rear sequence.

If front and rear can be equal in both the empty and full states, pointer equality alone is insufficient. Keep a count or another explicit state flag. With a count, empty means `count == 0` and full means `count == capacity`. You do not need to implement a circular queue today.

C Reverse a queue using a stack

Textbook programming exercise 2, page 357. Start with `queue = ["A", "B", "C", "D"]`. Reuse your stack functions from page 2. Today the input queue must contain at most seven items so that they fit in that stack.

1. While the queue is not empty, dequeue an item and push it onto a temporary stack. Record the stack after this phase.
2. While the stack is not empty, pop an item and enqueue it. Record the queue after this phase. The same queue should now read D, C, B, A from front to rear.
3. Test an empty queue, a one-item queue and a queue containing duplicates. Explain why duplicates should not disappear. Use only stack and queue operations; do not use `reverse()`, slicing or sorting.

Test input	Predicted result	Actual result
<code>[]</code>		
<code>["A"]</code>		
<code>["A", "B", "A"]</code>		

Reflection activity 2: How could you reverse a stack using a queue instead? Describe the two transfer phases and identify which value becomes the new top. Code this as an extension.

Evaluate expressions and explain uses

D Reverse Polish notation

In reverse Polish notation (RPN), an operator follows its operands. Read from left to right. Push a number. For a binary operator, pop the right operand first, then the left operand. Calculate left operator right and push the result. A complete expression leaves exactly one value.

Worked example: 8 3 - gives stack [8], then [8, 3], then [5]. The calculation is $8 - 3$, not $3 - 8$. This order also matters for division.

Textbook review question 19 uses: 5 2 * 3 2 + - . Complete the trace. Write stack contents from bottom to top.

Token	Action or calculation	Stack after token
5	Push 5	[5]
2		
*		
3		
2		
+		
-		

Final value _____ Equivalent ordinary expression _____

Check: What would indicate an invalid RPN expression? Consider both too few operands during processing and too many values left at the end.

Explain how the structure is used

For an application question, name a concrete situation and explain the operations and order. "Stacks are used for undo" is incomplete. A fuller answer is: "A drawing program pushes each reversible action onto a stack. Undo pops the most recent action so it can be reversed first."

Other stack uses in the text include browser back navigation, saved processor state during nested interrupts, expression evaluation and recursive function calls. In recursion, a later call finishes before the earlier call that is waiting for it. You are explaining this use today, not writing a recursive program.

Queue uses include ordinary print jobs, a keyboard input buffer and a checkout simulation. For example, a print job is enqueued at the rear and the oldest waiting job is dequeued for printing. This model assumes jobs have equal priority.

Short discussion

Choose one stack use and one queue use. Explain why changing the removal order would cause a problem. Then describe a program that could use both structures, such as a task queue with a history stack for undoing completed actions.

Your examples _____

Check your understanding of the reading

These prompts cover every review question on printed pages 356–357. Use your own words. Answer the questions selected in class; use the remainder to check your understanding. Question 19 has a full trace on page 4 of this handout.

Definitions and operations

Q1–2. Define a stack and a queue. Include their access order, not just an everyday example.

Q3–4. List the main operations for each structure. Explain which operations change contents and which only inspect or check them.

Q5. Explain FIFO and LIFO and identify the structure that uses each rule.

Q14–15. Give two characteristics of a stack and two characteristics of a queue.

Notes _____

Choosing a structure

Q6–7. Give a real-life stack example and a real-life queue example. Explain where items enter and leave.

Q8–9. Give one situation better served by a stack and one better served by a queue. Explain how the required order is obtained directly. Neither structure is always faster; efficiency depends on the task and implementation.

Q10. Describe a program that combines a stack and a queue, giving the job of each.

Q11. You must retrieve numbers in the reverse of their insertion order. Choose a structure and justify it.

Q12–13. Explain three computing uses of stacks and three of queues. Connect each example to its operations, as modelled on page 4.

Notes _____

Storage and access

Q16. Compare a stack with an array or list. Include a similarity and a difference. A stack defines restricted access; an array or list can be used to implement it.

Q17. Compare static and dynamic structures in both storage and access. Discuss fixed capacity, growth and how an item is reached. Do not assume that every dynamic structure is a linked list: a Python list is a resizable array with indexed access.

Q18. Explain stack overflow and stack underflow, including the operation that causes each. Refer to your tests from page 2.

Notes _____

Use command terms carefully

Define gives a precise meaning. Identify names the required item. Outline gives a brief account. Explain connects what happens with why. Compare gives relevant similarities as well as differences.

Review dynamic lists through programming

These are the four programming exercises on printed page 352, expressed as Python tasks with specific test data. Complete E1 and E4 in class. E2 and E3 are short extra practice if time permits.

Reminder: indices start at 0. In “for item in values”, item receives a value, not an index. Use `range(len(values))` when you need positions for replacement. Run each exercise with a fresh list.

E1 Insert change and delete

```
colours = ["red", "blue", "green", "yellow", "purple"]
```

Insert orange immediately after the third colour. Change the second colour to black. Delete the last colour. Display the final list and its length. Predict the list before running your code.

Your prediction _____

Explain why “after the third” uses insertion index 3: _____

E2 Traverse and replace

```
numbers = [4, 7, 9]
```

Append 2, 6 and 8. Display the size and the first element. Store the first element in a variable, then use a loop to replace every other element with that saved value. Display the new list. The first element should remain unchanged.

Your prediction _____

Why is assigning to the loop variable in “for item in numbers” insufficient?

E3 Duplicate a value and find its first occurrence

```
numbers = [12, 24, 36]
```

Store the second value in a variable and append it to the list. Display the index of its first occurrence. Explain why this is different from the index of the item you have just appended.

Final list _____ First index _____

E4 Swap two positions

```
values = ["north", "east", "south", "west"]
```

Display the list. Swap its second and third elements using a temporary variable, then display the result. Afterwards, try Python's simultaneous assignment as a second solution.

Your prediction _____

A useful debugging habit

For each exercise, keep the original input, predicted result and actual result. If they differ, inspect the list after each change. Check positions first: the second item is at index 1, and the third is at index 2.

Review functions and input validation

F Process workshop bookings

Core review. Practise decomposition, strings, conversion, selection, iteration, lists, functions and exception handling. A booking contains a name and an integer number of seats from 1 to 5. The fields are separated by one comma. Remove surrounding spaces; reject an empty name, an invalid number or the wrong number of fields.

```
records = [" Ada,3 ", "Ben,0", "Cy,two", "Dee,5", "Eli,2"]
```

```
def parse_booking(text):
    parts = text.split(",")
    if len(parts) != 2:
        raise ValueError("Use name,seats")
    name = parts[0].strip()
    seats = int(parts[1].strip())
    # TODO: reject an empty name or seats outside 1 to 5.
    return name, seats
```

```
names = []
counts = []
for text in records:
    try:
        name, seats = parse_booking(text)
        # TODO: append to names and counts.
    except ValueError as error:
        print("Rejected:", text, error)
```

1. Before coding, write a three-step plan for reading, validating and summarising the records. Identify what the function returns and why invalid records must not be added.
2. Complete the program. Then use a loop to find the total seats and calculate the mean seats per accepted booking. Handle the case where no records are accepted. Do not use `sum()` for this review.
3. Display the names whose accepted bookings have at least three seats. Use an index to connect each name to its count. State the types of text, seats, names and the calculated mean.

Test	Expected behaviour
The five supplied records	Accept Ada, Dee and Eli; counts [3, 5, 2]; total 10; mean about 3.33; large bookings Ada and Dee.
"Fox,1" and "Gia,5"	Accept both boundary values.
"Hal,6", ",2", "Ivo", "Jen,2,3"	Reject each with a useful message.
An empty input list	Report that there are no accepted bookings; do not divide by zero.

Extension: replace the fixed input with a while loop that accepts entries until the user types quit. Invalid input should produce a message and allow another attempt. Explain why finally is not needed just to catch an invalid value.

Debug test and prepare for algorithms

G Trace a faulty function

Core review. This function is intended to calculate the mean. Trace it before running it with [2, 4, 6]. There are two logic errors; an empty list also causes a runtime error.

```
def mean(values):
    total = 0
    for i in range(1, len(values)):
        total = values[i]
    return total / len(values)
```

Point	i	total	Returned value
Start	—	0	
First iteration			
Second iteration			
Return	—		

1. Predict the returned value and the correct value. Identify both logic errors and repair them. Decide that an empty list will raise ValueError with a clear message.
2. Set a breakpoint on the assignment inside the loop. Watch i, values[i] and total as you step. If no debugger is available, print these values inside the loop. Explain what each observation reveals.
3. Test [2, 4, 6], [5], [0, 0] and []. Explain why try/except alone cannot fix an incorrect calculation. What does a finally block do after a try statement?

H Review two dimensional data

Extra practice. Write a function row_totals(grid) that uses nested loops to return one total for each row. Do not use sum(). Test the following data, then test [] and [[], [4]].

```
scores = [[3, 5, 2], [4, 4, 1], [5, 2, 5]]
# Expected totals: [10, 9, 12]
```

Explain the difference between the grid, one row and one individual score. Then write a function count_at_least(grid, threshold) to count all scores at or above a supplied threshold. For threshold 4, the answer is 5.

Exit check before Monday

Without running code: push 4, push 9, pop, push 2. Give the final stack and its top. Repeat with enqueue and dequeue and give the final queue and its front.

Explain one test that distinguishes a correct program from a program that only works for typical input. Name one Python skill you can use independently and one you need to revisit before B2.4.

Submit your core programs B, C, E1, E4, F and G with a few test results. Homework, if assigned: spend no more than 25 minutes finishing a core task or completing E2 and E3 plus selected questions on page 5. H is an extension.