

Programming

International Baccalaureate · IB Diploma · Computer Science · HL

Scope and prerequisites

This handout teaches Theme B2, Programming, for the course first assessed in 2027, using Python. Sections marked **AHL** are for Higher Level only.

Prerequisites: computational thinking from B1 (decomposition, abstraction, algorithms as flowcharts and pseudocode). You should be able to run a short Python program and read its output.

Programming fundamentals

■ Why does a working program fail on Monday?

A shop's checkout program has run for a month. On Monday a customer types **two** instead of 2 for the quantity and the program crashes. Nothing in the code changed: the input did. Good programs are written for every input they might meet, not only the ones the programmer tried.

■ Variables and data types

A **variable** is a named store for a value that can change while the program runs. A **constant** is a named value that should not change; in Python it is written in capitals by convention, such as `VAT_RATE = 0.2`.

Every value has a **data type**, which decides what it can store and what operations work on it.

Type	Python	Example	Use
integer	<code>int</code>	42	counts, indexes
real	<code>float</code>	3.75	measurements, money in a calculation
Boolean	<code>bool</code>	<code>True</code>	flags, conditions
string	<code>str</code>	"Ada"	text

- `int("42")` turns text into an integer and `str(42)` turns a number into text. This **casting** is needed because `input()` always returns a string.
- `7 / 2` is 3.5; `7 // 2` is 3 (**integer division**); `7 % 2` is 1 (the remainder, **modulo**).

■ Strings

A **string** is a sequence of characters, indexed from 0.

```
code = "ib-2027"
len(code)      # 7
code[0]         # "i"
code[3:7]       # "2027": from index 3 up to, not including, 7
code.upper()    # "IB-2027"
code.find("-")   # 2; -1 if not found
"2027" in code   # True
```

Python strings are **immutable** : `code.upper()` returns a new string and leaves `code` unchanged. Joining strings with `+` is **concatenation** .

■ Exceptions: handling bad input

An **exception** is an error raised while the program runs, such as `ValueError` for `int("two")` or `ZeroDivisionError` for `5 / 0`. Unhandled, it stops the program. A `try ... except` block catches it:

```
while True:
    text = input("Quantity: ")
    try:
        qty = int(text)
        break
    except ValueError:
        print("Enter a whole number.")
```

The loop repeats until the cast succeeds, so the program never crashes on `two`. A `finally` block, if present, runs whether or not an exception occurred: it is used to close files.

■ Debugging and trace tables

There are three kinds of error:

Kind	When it shows	Example
syntax error	before the program runs	a missing colon after <code>if x > 0</code>
runtime error	while it runs, as an exception	an index past the end of a list
logic error	never reported: the output is wrong	<code><</code> where <code><=</code> was meant

A **trace table** records the value of each variable after each step, by hand. It is the standard way to find a logic error and a common exam question.

```
total = 0
for k in range(1, 5):
    if k % 2 == 0:
        total = total + k
print(total)
```

k	k % 2 == 0	total
		0
1	False	0
2	True	2
3	False	2
4	True	6

Output: 6. Note that `range(1, 5)` stops **before** 5. Other debugging techniques are print statements at key points, breakpoints in a **debugger** that pause the program and show the variables, and stepping through one line at a time.

Data structures

■ Static and dynamic structures

A **static data structure** has a size fixed when it is created, like an array in Java or C. A **dynamic data structure** grows and shrinks as the program runs, like a Python list.

	Static	Dynamic
size	fixed	changes at run time
memory	reserved in advance, may be wasted	used as needed
access	direct by index, fast	may need extra work to grow
risk	overflow when full	running out of memory

■ One- and two-dimensional arrays

A one-dimensional **array** stores items of one type under one name, each reached by its **index**. In Python a list plays this role:

```
marks = [67, 82, 45, 90]
marks[2]          # 45
marks[2] = 50     # change one item
marks.append(71)  # add at the end: [67, 82, 50, 90, 71]
len(marks)        # 5
```

A **two-dimensional array** is a table: a list of rows, where each row is a list. The first index picks the row and the second the column.

	column			
	0	1	2	3
row 0	4	7	1	9
row 1	2	8	5	3
row 2	6	0	7	4

`grid[1][2] == 5`
row first, then column

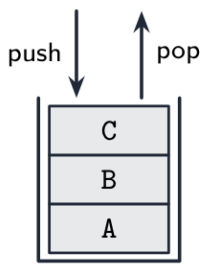
`grid[row][col]`: the row index comes first

```
grid = [[4, 7, 1, 9],  
        [2, 8, 5, 3],  
        [6, 0, 7, 4]]  
  
row_total = 0  
for col in range(len(grid[1])):  
    row_total = row_total + grid[1][col]  
print(row_total)    # 2 + 8 + 5 + 3 = 18
```

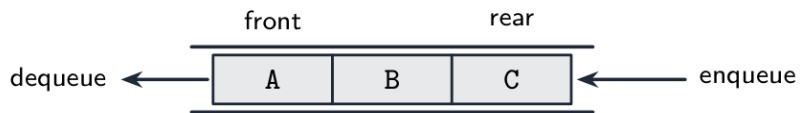
To visit every cell, nest two loops: the outer one over the rows, the inner one over the columns.

■ Stacks and queues

Both are **abstract data types** : defined by their operations, not by how they are stored.



stack: last in, first out



queue: first in, first out

Stack: one end does everything. Queue: add at the rear, remove from the front

	Stack	Queue
order	LIFO : last in, first out	FIFO : first in, first out
add	push onto the top	enqueue at the rear
remove	pop from the top	dequeue from the front
look	peek at the top	peek at the front
uses	undo, back button, function calls	print jobs, keyboard buffer, task scheduling

```

stack = []
stack.append("A"); stack.append("B"); stack.append("C")  # push
stack.pop()      # "C"

from collections import deque
queue = deque()
queue.append("A"); queue.append("B"); queue.append("C")  # enqueue
queue.popleft()   # "A"

```

Before a pop or dequeue, check isEmpty: removing from an empty structure is an **underflow**. Pushing onto a full fixed-size stack is an **overflow**.

Programming constructs

■ Sequence, selection and iteration

Every algorithm is built from three constructs: **sequence** (steps in order), **selection** (if, elif, else) and **iteration** (loops).

Loop	Python	Use when
count-controlled	<code>for i in range(n):</code>	the number of repeats is known
condition-controlled	<code>while condition:</code>	repeat until something happens
over a collection	<code>for item in items:</code>	visit each item once

A **while** loop whose condition never becomes false is an **infinite loop**. Nested selection or loops place one construct inside another; indentation shows which is inside which.

■ Functions, parameters and scope

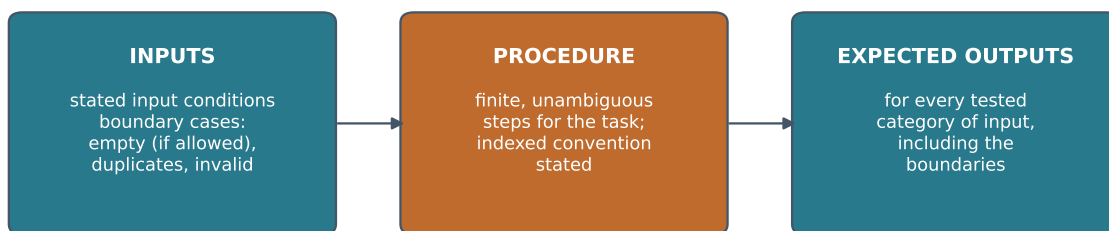
A **function** is a named block of code that can be called many times. It takes **parameters** and may **return** a value. Splitting a program into functions is **modularity**: each part is written, tested and reused on its own.

```
def mean(values):
    """Return the mean of a non-empty list of numbers."""
    total = 0
    for v in values:
        total = total + v
    return total / len(values)

print(mean([4, 8, 9]))    # 7.0
```

- A **local variable** such as `total` exists only inside the function while it runs.
- A **global variable** is defined at module level and can be read where Python scope and name-binding rules resolve to it; a local name can shadow it. Using many globals makes a program hard to test, because any function might change them.
- The docstring states the function's contract: what it needs (a non-empty list) and what it returns. `mean([])` breaks the contract and raises `ZeroDivisionError`.

A program contract: inputs, procedure, expected outputs



A failure is diagnosed as a wrong algorithm, a wrong implementation, or an incomplete requirement.

A function's contract: the inputs it accepts, including boundary cases, and the output it promises for each

Test a function against its contract: normal values, **boundary cases** at the edge of what it accepts (a one-item list), and invalid values it should reject (an empty list).

Algorithms

■ Measuring efficiency: Big O

Big O notation O gives an asymptotic upper bound for how a chosen cost grows with input size n . State whether the cost being analysed is worst-case, average-case or best-case. Constants and smaller terms are dropped.

Big O	Name	Example	Steps for $n = 1000$
$O(1)$	constant	reading <code>marks[2]</code>	1
$O(\log n)$	logarithmic	binary search	about 10
$O(n)$	linear	linear search	1000
$O(n^2)$	quadratic	bubble sort, selection sort	about 1 000 000

A single loop over the data is usually $O(n)$; a loop inside a loop over the same data is usually $O(n^2)$.

■ Linear and binary search

A **linear search** checks each item in turn. It works on any list, sorted or not, and is $O(n)$.

```
def linear_search(items, target):
    for i in range(len(items)):
        if items[i] == target:
            return i
    return -1
```

A **binary search** needs a **sorted** list. It checks the middle item and throws away the half that cannot hold the target, so each probe halves what is left: $O(\log n)$.

```
def binary_search(items, target):
    low, high = 0, len(items) - 1
    while low <= high:
        mid = (low + high) // 2
        if items[mid] == target:
            return mid
        elif items[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

	0	1	2	3	4	5	6	7	8
probe 1: $16 < 23$, go right	2	5	8	12	16	23	38	56	72
probe 2: $38 > 23$, go left	2	5	8	12	16	23	38	56	72
probe 3: found at index 5	2	5	8	12	16	23	38	56	72

Each probe discards half of what remains

Trace for 23: low 0, high 8, mid 4 ($16 < 23$) $\rightarrow$ low 5; mid $(5 + 8) // 2 = 6$ ($38 > 23$) $\rightarrow$ high 5; mid 5: found, return **5**. On an unsorted list binary search can miss a value that is there, so the contract must require sorted input.

■ Bubble sort and selection sort

A **bubble sort** compares each pair of neighbours and swaps them if they are in the wrong order. After each pass the largest remaining item has “bubbled” to the end.

```
def bubble_sort(a):
    n = len(a)
    for p in range(n - 1):
        swapped = False
        for i in range(n - 1 - p):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                swapped = True
        if not swapped:
            break
```

The **swapped** flag stops early when a pass makes no swaps: the list is already sorted.

A **selection sort** finds the smallest item in the unsorted part and swaps it into the next place.

```
def selection_sort(a):
    for i in range(len(a) - 1):
        smallest = i
        for j in range(i + 1, len(a)):
            if a[j] < a[smallest]:
                smallest = j
        a[i], a[smallest] = a[smallest], a[i]
```

Passes of each sort on [5, 1, 4, 2]:

After pass	Bubble sort	Selection sort
1	[1, 4, 2, 5]	[1, 5, 4, 2]
2	[1, 2, 4, 5]	[1, 2, 4, 5]
3	no swaps: stop	[1, 2, 4, 5]

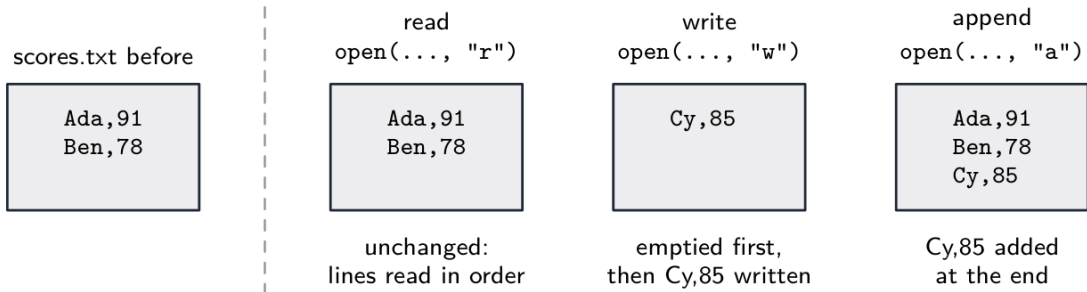
Both are $O(n^2)$. Selection sort makes at most $n - 1$ swaps; bubble sort can stop early on a nearly sorted list.

File processing

■ Reading and writing text files

A program’s variables are lost when it ends; a **file** keeps data between runs. Open the file with a **mode**, use it, then close it.

Mode	Meaning	If the file exists
"r"	read	read from the start
"w"	write	emptied first
"a"	append	add to the end



Mode “w” destroys what was there; mode “a” keeps it

```
with open("scores.txt", "a") as f:
    f.write("Ada,91\n")

best_name, best = "", -1
with open("scores.txt", "r") as f:
    for line in f:
        name, score = line.strip().split(",")
        if int(score) > best:
            best_name, best = name, int(score)
print(best_name, best)
```

- `with` closes the file automatically, even if an exception occurs.
- Each line read ends in `"\n"`; `strip()` removes it before `split(",")` breaks the line into fields.
- Opening a missing file for reading raises `FileNotFoundError`: catch it with `try ... except`.

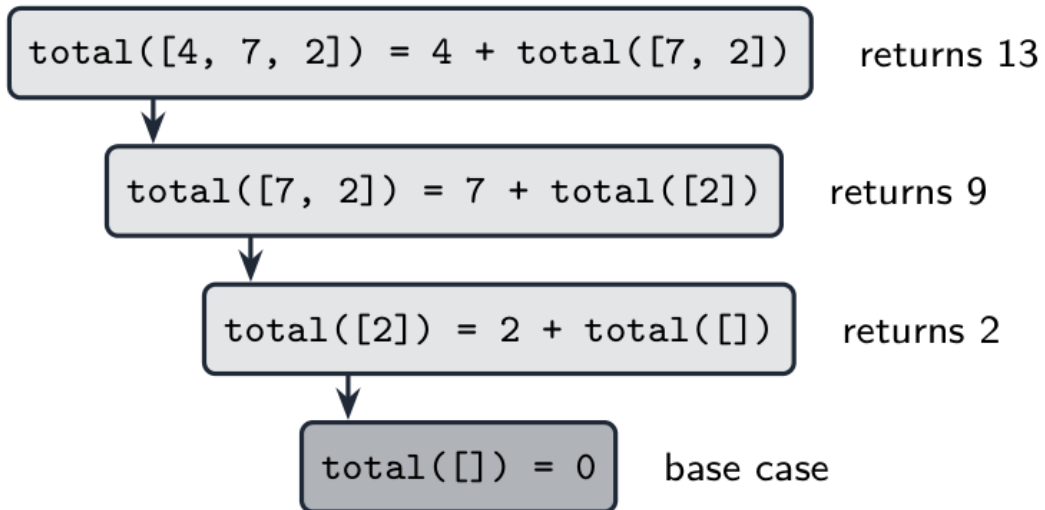
AHL: Recursion

■ A function that calls itself

A **recursive function** calls itself on a smaller version of the same problem.
It needs two parts:

1. A **base case** that is answered directly, with no further call.
2. A **recursive case** that calls the function on a smaller input, moving towards the base case.

```
def total(items):
    if items == []:                # base case
        return 0
    return items[0] + total(items[1:]) # recursive case
```



Calls go down to the base case; results come back up

Each call is placed on the **call stack** with its own local variables. The calls wait until the base case returns, then finish in reverse order: last called, first finished, like a stack.

■ Tracing a recursive call

```
def power(b, e):
    if e == 0:
        return 1
    return b * power(b, e - 1)
```

$\text{power}(3, 3) \rightarrow 3 * \text{power}(3, 2) \rightarrow 3 * 3 * \text{power}(3, 1) \rightarrow 3 * 3 * 3 * \text{power}(3, 0) \rightarrow 3 * 3 * 3 * 1 = 27$: four calls in all.

■ Recursion or iteration?

	Recursion	Iteration
code	often shorter, matches the problem's definition	often longer
memory	one stack frame per call	one set of variables
risk	stack overflow if too deep or no base case	an infinite loop
natural for	trees, nested folders, divide and conquer such as quicksort	simple counting and running totals

Quicksort is recursive: pick a pivot, split the list into items less than and greater than the pivot, then quicksort each part. Its average case is $O(n \log n)$, much faster than the $O(n^2)$ sorts above for large n .

Practice

1. State the data type best suited to each: (a) the number of students in a class; (b) whether a door is locked; (c) a postcode such as "SW1A 1AA"; (d) a temperature of 21.5 °C.
2. `s = "Programming"`. Give (a) `len(s)`; (b) `s[3:7]`; (c) `s.find("g")`.
3. Classify each error as syntax, runtime or logic: (a) `print("Hi"`; (b) a loop that adds one item too few; (c) `int("abc")`.
4. Complete a trace table for `x = 1` then `while x < 20: x = x * 3`, and state the final value of `x`.
5. The items 10, 20 and 30 are pushed onto an empty stack, then one item is popped and 40 is pushed. State the item on top.
6. The same items are put through a queue: enqueue 10, 20, 30, dequeue once, enqueue 40. State the item at the front.
7. For the list `[3, 8, 12, 19, 25, 31, 40]`, list the values that a binary search for 31 compares with.
8. Show the list `[6, 3, 8, 2]` after each pass of a bubble sort.
9. Write a Python function `count_over(marks, limit)` that returns how many marks in the list are greater than `limit`.
10. Explain why opening `"log.txt"` with mode `"w"` each time a program starts loses data, and state the fix.
11. **AHL** For `power` above, state the number of calls made by `power(2, 5)` and the value returned.
12. **AHL** Explain what happens when a recursive function has no base case.

Answers.

1. (a) integer; (b) Boolean; (c) string, as it holds letters and spaces; (d) real.
2. (a) 11; (b) "gram"; (c) 3.
3. (a) syntax: a missing bracket; (b) logic; (c) runtime: a `ValueError`.
4. `x` takes the values 1, 3, 9, 27; the loop stops when `x` is 27. Final value **27**.
5. Push 10, 20, 30; pop removes 30; push 40. Top: **40**.
6. Enqueue 10, 20, 30; dequeue removes 10; enqueue 40. Front: **20**.
7. low 0, high 6: mid 3, value $19 < 31$, so low = 4; mid $(4 + 6) // 2 = 5$, value **31**: found. Compared with 19 and 31.
8. Pass 1: [3, 6, 2, 8]; pass 2: [3, 2, 6, 8]; pass 3: [2, 3, 6, 8].
9. One answer:

```
def count_over(marks, limit):
    count = 0
    for m in marks:
        if m > limit:
            count = count + 1
    return count
```

10. Mode "w" empties an existing file before writing, so the earlier log is lost. Open it with "a" to append.
11. `power(2, 5)` calls itself with `e = 4, 3, 2, 1, 0`: **6** calls, returning $2^5 = \mathbf{32}$.
12. It calls itself for ever; each call adds a frame to the call stack until memory runs out, a stack overflow (in Python a `RecursionError`).

Exam tips

- In a trace table, give a new row each time a variable changes, and show the output separately. Check where **range** stops.
- State the indexing convention (Python indexes from 0) before tracing an array or a search.
- For a binary search answer, say that the list **must be sorted** and give low, high and mid at every probe.
- Bubble sort and selection sort are both $O(n^2)$: compare them by swaps, early stopping and passes, not by Big O.
- "Explain" a stack or queue with its order (LIFO or FIFO), its two operations and a real use.
- When writing code, use clear names, close files with **with**, and handle bad input with **try ... except**.
- **AHL** A recursion answer names the base case, the recursive case, and how each call moves towards the base case.