MODULE M PROGRAMMING · M28

Writing maintainable code: mark scheme

20 marks · OCR J277 spec 2.3.1 (maintainability: use of sub programs, naming conventions, indentation, commenting; why commenting is useful and applying it), 2.2.3 (the use of sub programs: functions)

How to use. 1 mark per bullet up to the Max shown. **/** separates alternative answers. Words in (brackets) are not needed for the mark. Code answers: mark the logic, not the language. Allow OCR Exam Reference Language or any high-level language, and allow any logically valid method. Do not penalise minor syntax slips (capitals, a missing colon, # instead of // for a comment). A recurring slip: penalise once, then follow through. Do not accept maths symbols such as ≥ or ≤ in code. Explain questions: a generic answer that does not refer to the program given scores max 1 where stated.

QUESTION	ANSWER	MARKS	GUIDANCE
1(a) J277 2.3.1	1 mark each: Adding comments that explain what each section of code does Indenting the statements inside each loop and `if` statement	**2**	More than two ticks: deduct 1 mark for each extra tick (minimum 0). The other three rows are input validation, authentication and testing, not maintainability.
1(b) J277 2.3.1	1 mark each to max 2: Explain what a section of code does / the purpose of the code Help other programmers (in a team) understand the code (quickly) Help the original programmer understand / remember the code when they return to it later Make errors easier to find and fix / make it easier to update or change the code Explain how a complex part / algorithm works / why it was written that way Can be used to temporarily stop a line of code running when testing (commenting out)	**2**	Two different reasons are needed: "easier to read" and "easier to understand" are the same point. Do not accept "makes the program run faster / better" or "makes the program more secure". Credit any other valid reason.
2(a) J277 2.3.1	1 mark each to max 2: Shows which statements are inside the `for` loop / are repeated 7 times (lines 04 to 08) Shows that line 07 only runs when the `if` condition is true / belongs to the `if` (selection) Makes the structure / where each block starts and ends easy to see, so another programmer can follow the code / find errors (such as a missing `endif` or `next`) more easily	**2**	"Easier to read" alone, with no reference to the loop, the `if` or blocks of code: max 1. Do not accept "the program will not run without it" (true of Python, but not the point; ERL does not need indentation).

QUESTION	ANSWER	MARKS	GUIDANCE

2(b)
J277 2.3.1
	1 mark each to max 5:
	Meaningful identifiers for `t` and `h`, e.g. `totalRain`, `heavyDays`
	Meaningful identifiers for `r` and `d`, e.g. `rainfall`, `day`
	New identifiers follow a naming convention (no spaces, one consistent style such as camelCase) and are used consistently everywhere (every use changed, same spelling)
	Lines 04 to 08 indented inside the `for` loop
	Line 07 indented further than the `if` (inside the `if`)
	At least one relevant comment that explains the purpose of the program or of a section (e.g. what counts as a heavy-rain day)
	`// Adds up a week of rainfall and counts heavy-rain days`
	`totalRain = 0`
	`heavyDays = 0`
	`for day = 1 to 7`
	`    rainfall = float(input("Enter rainfall in mm"))`
	`    totalRain = totalRain + rainfall`
	`    // more than 25 mm in a day is heavy rain`
	`    if rainfall > 25 then`
	`        heavyDays = heavyDays + 1`
	`    endif`
	`next day`
	`print("Total rainfall: " + str(totalRain))`
	`print("Heavy-rain days: " + str(heavyDays))`
		5	The program must still do the same job: if the rewrite changes the logic (e.g. `>= 25`, loop runs a different number of times), max 4. Allow `i` or `count` for the loop counter. Allow snake_case (`total_rain`) or any other consistent convention. Do not accept identifiers with spaces. A comment that only repeats the code (e.g. `// add 1 to h`) does not get the comment mark. Allow # comments. Allow inline comments at the end of a line.

3(a)
J277 2.2.3, 2.3.1
	1 mark each:
	Checks if `cost > 100`
	... and if so sets `cost` to 90% of its value / takes 10% off (`cost * 0.9` / `cost - cost * 0.1`)
	Returns the (new) cost in all cases
	`function applyDiscount(cost)`
	`    if cost > 100 then`
	`        cost = cost * 0.9`
	`    endif`
	`    return cost`
	`endfunction`
		3	Do not accept `>= 100`. Allow a new variable for the result (e.g. `newCost`) if it is set in both cases and returned. Allow `return cost * 0.9` inside the `if` with `return cost` after it. Printing the cost instead of returning it: do not award the third mark. Inputting the cost inside the function (overwriting the parameter): do not award the first mark.

3(b) J277 2.2.3	`adultCost = applyDiscount(adultCost)`	**1**	The returned value must be stored in `adultCost` (or in a new variable that is then used in line 11). Do not accept `applyDiscount(adultCost)` on its own. Do not accept `seniorCost` as the parameter.

QUESTION	ANSWER	MARKS	GUIDANCE
3(c) J277 2.3.1	1 mark each to max 3: The discount code is written once instead of twice / less repeated code / the program is shorter If the discount rule changes (e.g. a different percentage or limit), it only has to be changed in one place ... so there is less chance of changing one copy and missing the other / the two ticket types cannot end up with different rules by mistake The function can be tested on its own / errors in the discount only need fixing in one place The function name describes what the code does, so another programmer can understand line 03 / the program more easily The function can be reused / called again, e.g. if a new ticket type is added, without writing the discount code again	**3**	Answers must be about maintainability. Do not accept "the program runs faster" or "uses less memory". Generic points about sub programs with no link to this program (the discount, the two ticket types): max 2. Credit any other valid point.
4(a) J277 2.3.1	1 mark each to max 2: All the code looks the same / is consistent, so any programmer can read and understand code written by someone else Programmers can tell what kind of item an identifier is (e.g. a variable or a constant) from its style Fewer mistakes from names written in different styles (e.g. `ticketsSold`, `TicketsSold` and `tickets_sold` being treated as different identifiers) Saves time when code is changed / updated / debugged by a different member of the team	**2**	Do not accept "the program will run faster" or "makes the program more secure". "Easier to read" alone: 1 mark only. Credit any other valid point.

Total: 20 marks