OXFORDAQA INTERNATIONAL GCSE COMPUTER SCIENCE (9210/1)

Paper 1 Programming — Mark Scheme & Worked Solutions

Examination Date: Friday 15 May 2026 | **Time Allowed:** 2 hours | **Maximum Mark:** 80 | **Reference Language:** Python 3

> **Marking Guidance & Conventions:** Marks are awarded for algorithmic logic, accuracy of Skeleton Program references, and adherence to required identifiers and output strings. In Sections B and C, full Python 3 reference implementations are provided alongside point-by-point marking criteria and expected test output captures. Alternative valid programming constructs (e.g., `while` vs. `for` loops, or equivalent array indexing) are credited where functionally identical.

Section A — Non-Programming Questions (Total: 22 Marks)

	Marking Criteria & Model Answers	Marks
01.1	**1 mark** for any valid Boolean variable name used in the Skeleton Program that stores only `True` or `False`: • `GameOver` (in `PlayGame`) • `PlayAgain` (in `Main`, if implemented as Boolean) or `Valid` / `Found` (if used in validation subroutines). *Note: Do not accept variables storing character choices like `'Y'`/`'N'` unless defined as Boolean.*	1
01.2	**1 mark** for identifying the floating-point / real variable in `GenerateRandomEvent`: • `EventChance` (or `RandomNumber` / `Chance` as declared in the Skeleton Program for `random.random()`).	1
01.3	**1 mark** for naming the subroutine that uses the modulo (`%` / `MOD`) remainder operator: • `CalculateCityIncome` (or `PlayGame` / `GenerateRandomEvent` depending on turn/periodic check implementation in the preliminary Skeleton Program).	1
01.4	**1 mark** for a clear explanation referencing indefinite iteration / unknown number of iterations: • The number of iterations is not known in advance because the loop must repeat until the game ends (`GameOver == True`), which can happen early if Money < 0, Happiness < 0, or Pollution > 20 rather than always completing all turns.	1
01.5	**1 mark** for stating the exact parameter count: • **2** (specifically `Buildings` and `Initial`).	1
01.6	**1 mark each (max 2 marks):** • It is a **1-dimensional array / list** (collection of items). • Each element stores a **record / user-defined data type / object** of type `Building` (containing fields for initial, name, cost, income, pollution, happiness).	2

	Marking Criteria & Model Answers	Marks
01.7	**1 mark each for any two valid advantages (max 2 marks):** • Avoids the need for global variables (reduces unintended side effects / improves data encapsulation). • Makes the subroutine more reusable in other parts of the program or other programs with different input values. • Makes the subroutine easier to test, debug, and maintain independently because its inputs are explicitly defined.	2
01.8	**1 mark each (max 2 marks):** • Checks whether a grid location in the city contains a building (i.e., is not empty / not equal to `'-'`) or checks the type of building present at that location. • Looks up the matching building record and adds that building's income value to the city's total income.	2
01.9	**1 mark each for both required conditions (max 2 marks):** • The generated random value (`EventChance`) must fall within the specific probability threshold/range for a positive happiness event. • The city's current `Pollution` level must be below the maximum threshold (or `Happiness` / park count condition in the `IF` branch must be satisfied).	2
02.1	**1 mark each (max 2 marks):** • To exit the subroutine early / immediately if validation fails (e.g., an invalid building initial is entered, coordinates are out of range, the cell is already occupied, or the player lacks sufficient money). • Prevents invalid state updates and returns a failure/status indicator (or skips building placement) while returning normally at the end when construction succeeds.	2
02.2	**1 mark each for any two string operations used in `AddBuildingToCity` (max 2 marks):** • **Length check** (`len(Coordinates)`) to verify input length. • **Indexing / substring extraction** (`Coordinates[0]`, `Coordinates[2]` or `split()`) to separate row and column characters. • **Case conversion** (`.upper()`) on the building initial input.	2
02.3	**1 mark each (max 2 marks):** • The first character/token of `Coordinates` (index `0`) is extracted and converted to an integer to set `Row`. • The third character/second token of `Coordinates` (index `2`, after the space) is extracted and converted to an integer to set `Column`.	2

Question 03 — Flowchart Completion (3 Marks)

1 mark for each correctly placed letter in Table 1 (3 marks total):

Flowchart Text (Table 1)	Letter (A–C)
Output "Pollution too high"	
Money is less than zero	
Output "Out of money"	
Turn is greater than TURNS	
Increase Money by city income	**A**
Output "All turns completed"	**C**
Output "Population too unhappy"	
Pollution is greater than 20	**B**
Set GameOver to TRUE	
Happiness is less than zero	
Output "Final Statistics"	

Flowchart logic trace: After incrementing `Turn`, process step **A** updates `Money` by adding city income. The cascading decision diamonds check end-of-game conditions in order: (1) Money < 0, (2) Happiness < 0, (3) **B**: Pollution > 20, and (4) Turn > TURNS, whose Yes branch executes output step **C** (`Output "All turns completed"`).

Section B — Short Programming Questions (Total: 28 Marks)

Question 04 — End-of-Game Message in `Main` (4 Marks)

04.1 Mark Allocation (3 marks):

- **1 mark:** Output statement displaying exact text `"Thank you for playing"` placed immediately after `PlayGame(...)` finishes.
- **1 mark:** Statement positioned *before* the prompt asking if the player wants to play again.
- **1 mark:** Outputs an additional blank line after `"Thank you for playing"` (e.g., `print()` or `print("Thank you for playing\n")`).

```
def Main():
    Buildings = SetupBuildings()
    PlayAgain = "Y"
    while PlayAgain == "Y":
        TheCity = SetupCity()
        PlayGame(TheCity, Buildings)
        print("Thank you for playing")
        print()
        PlayAgain = input("Do you want to play again? (Y/N): ").upper()
```

04.2 Screen Capture Evidence (1 mark): Award 1 mark if output after the 5th `E` shows all 5 turns completed, final statistics, `Thank you for playing` , a blank line, and the play-again prompt:

```
All turns completed Final Statistics ... Thank you for playing Do you want to play
again? (Y/N):
```

Question 05 — Empty Location Counter in `DisplayCity` (7 Marks)

05.1 Mark Allocation (6 marks):

- **1 mark:** Initialises an empty-cell counter variable (e.g., `EmptyCount = 0`) before iterating through the grid.
- **1 mark:** Uses nested iteration across all rows and columns using `TheCity.Size` (works for a city of any size).
- **1 mark:** Selection statement checking if a cell is empty (`TheCity.Grid[Row][Column] == "-"`) and incrementing `EmptyCount` by 1.
- **1 mark:** Displays `f"{EmptyCount} locations are empty"` after the city grid has been displayed.
- **1 mark:** Selection condition checking if more than half of the city is empty (`EmptyCount > (TheCity.Size * TheCity.Size) / 2` or `EmptyCount * 2 > TheCity.Size ** 2`).
- **1 mark:** Displays exact message `"More buildings are needed"` when the condition is true.

```
def DisplayCity(TheCity):
    EmptyCount = 0
    for Row in range(TheCity.Size):
        RowString = str(Row) + " "
        for Column in range(TheCity.Size):
            RowString += TheCity.Grid[Row][Column]
            if TheCity.Grid[Row][Column] == "-":
                EmptyCount += 1
        print(RowString)
    print(f"{EmptyCount} locations are empty")
    if EmptyCount > (TheCity.Size * TheCity.Size) / 2:
        print("More buildings are needed")
```

05.2 Screen Capture Evidence (1 mark): For a 3×3 city (9 cells) with 2 buildings placed (H at 1,2 and F at 2,0), 7 cells remain empty. Since 7 > 4.5, both messages appear after the final `E`:

```
0 --- 1 --H 2 F-- 7 locations are empty More buildings are needed
```

Question 06 — Column Headers in `DisplayCity` (5 Marks)

06.1 Mark Allocation (3 marks):

- **1 mark:** Uses a loop iterating from `0` up to `TheCity.Size - 1` to construct or print column numbers starting at `0`.
- **1 mark:** Correctly aligns column headers above the grid columns (accounting for the leading row number and space, i.e., 2 leading spaces).
- **1 mark:** Displays the header line above the city rows and works dynamically for any size up to 10.

```
def DisplayCity(TheCity):
    Header = "  "
    for Column in range(TheCity.Size):
        Header += str(Column)
    print(Header)
    for Row in range(TheCity.Size):
        RowString = str(Row) + " "
        for Column in range(TheCity.Size):
            RowString += TheCity.Grid[Row][Column]
        print(RowString)
```

06.2 Screen Capture Evidence (2 marks): 1 mark for size 5 grid showing header `01234` aligned above rows 0–4; 1 mark for size 3 grid showing header `012` aligned above rows 0–2:

```
01234 012 0 ----- 0 --- 1 ----- 1 --- 2 ----- 2 --- 3 ----- 4 -----
```

Question 07 — Building Energy Attribute (6 Marks)

07.1 `Building` Record Source Code (2 marks):

- **1 mark:** Adds `Energy` parameter to `__init__` (or record definition).
- **1 mark:** Assigns `self.Energy = Energy` when a new record is created.

```
class Building:
    def __init__(self, Initial, Name, Cost, Income, Pollution, Happiness, Energy):
        self.Initial = Initial
        self.Name = Name
        self.Cost = Cost
        self.Income = Income
        self.Pollution = Pollution
        self.Happiness = Happiness
        self.Energy = Energy
```

07.2 `SetupBuildings` and `AddBuildingToCity` (3 marks):

- **1 mark:** Passes exact energy values (`1` for House, `0` for Park, `10` for Factory, `4` for School) in `SetupBuildings`.
- **1 mark:** Output placed in `AddBuildingToCity` *after* calling `GetBuildingByInitial` (and confirming valid building).
- **1 mark:** Displays exact formatted message `f"{SelectedBuilding.Name} uses {SelectedBuilding.Energy} energy per day"`.

```
def SetupBuildings():
    Buildings = []
    Buildings.append(Building("H", "House", 10, 2, 1, 1, 1))
    Buildings.append(Building("P", "Park", 5, -1, -2, 3, 0))
    Buildings.append(Building("F", "Factory", -10, 10, 5, -3, 10))
    Buildings.append(Building("S", "School", 20, -2, 0, 2, 4))
    return Buildings

# Inside AddBuildingToCity, immediately after SelectedBuilding =
GetBuildingByInitial(...):
    if SelectedBuilding is not None:
        print(f"{SelectedBuilding.Name} uses {SelectedBuilding.Energy} energy per day")
```

07.3 Screen Capture Evidence (1 mark): Shows `House uses 1 energy per day` displayed after selecting `H`.

Question 08 — Display Building Cost or Benefit in `AddBuildingToCity` (6 Marks)

08.1 Mark Allocation (5 marks):

- **1 mark:** Iterates through each building record `B` in `Buildings` inside `AddBuildingToCity` before prompting for the building initial.
- **1 mark:** Selection checking whether `B.Cost > 0`.
- **1 mark:** Displays `cost: {B.Cost}` when `B.Cost > 0`, and `benefit: {abs(B.Cost)}` (or `-B.Cost`) when `B.Cost <= 0`.
- **1 mark:** Formats building name with initial in parentheses: `f"({B.Initial}){B.Name[1:]}"`.
- **1 mark:** Matches exact output format for all four buildings without negative signs on benefits.

```
# Inside AddBuildingToCity, before asking user to choose a building:
    for B in Buildings:
        FormattedName = f"({B.Initial}){B.Name[1:]}"
        if B.Cost > 0:
            print(f"{FormattedName} cost: {B.Cost}")
        else:
            print(f"{FormattedName} benefit: {abs(B.Cost)}")
```

08.2 Screen Capture Evidence (1 mark): Shows the exact menu when `C` is entered:

```
(H)ouse cost: 10 (P)ark cost: 5 (F)actory benefit: 10 (S)chool cost: 20
```

Section C — Longer Programming Questions (Total: 30 Marks)

Question 09 — Resizing the City Grid (15 Marks)

09.1 `ResizeCityGrid` Subroutine (9 marks):

- **1 mark:** Defines subroutine `ResizeCityGrid(TheCity, NewSize)` with the exact name and parameters in order.
- **1 mark:** Calls `NewGrid = CreateEmptyGrid(NewSize)` to create a new empty grid of size `NewSize`.
- **1 mark:** Calculates the safe copy bound `CopyLimit = min(TheCity.Size, NewSize)` (or uses conditional bounds check inside loops) so neither old nor new grid goes out of bounds.
- **1 mark:** Outer loop iterating `Row` through valid row indices (`0` to `CopyLimit - 1`).
- **1 mark:** Inner loop iterating `Column` through valid column indices (`0` to `CopyLimit - 1`).
- **1 mark:** Copies cell value `NewGrid[Row][Column] = TheCity.Grid[Row][Column]`.
- **1 mark:** Updates `TheCity.Size = NewSize`.
- **1 mark:** Updates `TheCity.Grid = NewGrid`.
- **1 mark:** Calls `DisplayCity(TheCity)` at the end of the subroutine to display the resized city.

```
def ResizeCityGrid(TheCity, NewSize):
    NewGrid = CreateEmptyGrid(NewSize)
    CopyLimit = min(TheCity.Size, NewSize)
    for Row in range(CopyLimit):
        for Column in range(CopyLimit):
            NewGrid[Row][Column] = TheCity.Grid[Row][Column]
    TheCity.Size = NewSize
    TheCity.Grid = NewGrid
    DisplayCity(TheCity)
```

09.2 `GetAction` Modification (1 mark):

- **1 mark:** Updates prompt in `GetAction` to inform the player of `C` (construct), `R` (resize), and `E` (end turn), and accepts `"R"` as valid input.

```
def GetAction():
    Action = ""
    while Action not in ["C", "R", "E"]:
        Action = input("Enter C to construct a building, R to resize the city, or E to
end the turn: ").upper()
    return Action
```

09.3 `PlayGame` Modification (4 marks):

- **1 mark:** Selection branch `elif Action == "R":` inside the turn action loop in `PlayGame`.
- **1 mark:** Prompts the user to enter a new city size and converts input to an integer (`NewSize = int(input(...))`).
- **1 mark:** Checks `if NewSize > 0:` and calls `ResizeCityGrid(TheCity, NewSize)`.
- **1 mark:** `else:` branch displays exact message `"Size must be greater than zero"`.

```
# Inside the action loop in PlayGame:
        if Action == "C":
            AddBuildingToCity(TheCity, Buildings)
        elif Action == "R":
            NewSize = int(input("Enter new city size: "))
            if NewSize > 0:
                ResizeCityGrid(TheCity, NewSize)
            else:
                print("Size must be greater than zero")
```

09.4 Screen Capture Evidence (1 mark): Shows preset city `A` resized first to 4×4 (truncating row/column 4) and then resized to 5×5 (adding an empty row 4 and column 4 with existing buildings preserved).

Question 10 — Display Surrounding Buildings (15 Marks)

10.1 `DisplaySurroundingBuildings` Subroutine (13 marks):

- **1 mark:** Defines subroutine `DisplaySurroundingBuildings(Row, Column, TheCity)` with exact name and parameter order.
- **1 mark:** Initialises a list or string to accumulate names of surrounding buildings (e.g., `AdjacentNames = []`).
- **1 mark:** Outer loop checking row offsets `R` from `Row - 1` to `Row + 1` (inclusive).
- **1 mark:** Inner loop checking column offsets `C` from `Column - 1` to `Column + 1` (inclusive).
- **1 mark:** Excludes the newly constructed building's own coordinate (`not (R == Row and C == Column)`).
- **1 mark:** Boundary check ensuring row index is valid (`0 <= R < TheCity.Size`).
- **1 mark:** Boundary check ensuring column index is valid (`0 <= C < TheCity.Size`) so edge/corner cells do not raise `IndexError`.
- **1 mark:** Checks that the neighbouring cell contains a building (`TheCity.Grid[R][C] != "-"`).
- **1 mark:** Translates the building initial at `TheCity.Grid[R][C]` to its full building name (e.g., via `GetBuildingByInitial` or lookup table).
- **1 mark:** Appends each adjacent building name individually so duplicate building types appear multiple times (as in Example B & C).
- **1 mark:** Formats the output cleanly with `"Constructed next to "` followed by the space-separated building names.
- **1 mark:** Preserves top-to-bottom, left-to-right scanning order matching Examples A, B, and C (`House School School Park` and `Park Park Factory`).
- **1 mark:** Handles cases where no buildings surround the new building cleanly without crashing.

```
def DisplaySurroundingBuildings(Row, Column, TheCity):
    InitialToName = {"H": "House", "P": "Park", "F": "Factory", "S": "School"}
    Surrounding = []
    for R in range(Row - 1, Row + 2):
        for C in range(Column - 1, Column + 2):
            if R == Row and C == Column:
                continue
            if 0 <= R < TheCity.Size and 0 <= C < TheCity.Size:
                Cell = TheCity.Grid[R][C]
                if Cell != "-":
                    Surrounding.append(InitialToName[Cell])
    if len(Surrounding) > 0:
        print("Constructed next to " + " ".join(Surrounding))
```

10.2 `AddBuildingToCity` Call (1 mark):

- **1 mark:** Calls `DisplaySurroundingBuildings(Row, Column, TheCity)` inside `AddBuildingToCity` immediately after `TheCity.Grid[Row][Column] = SelectedBuilding.Initial` (only when construction succeeds).

```
# Inside AddBuildingToCity, after placing the building on the grid:
    TheCity.Grid[Row][Column] = SelectedBuilding.Initial
    DisplaySurroundingBuildings(Row, Column, TheCity)
```

10.3 Screen Capture Evidence (1 mark): Shows preset city `B` with Park constructed at `2 2` and School constructed at `1 1`, outputting the surrounding building names next to `1 1`:

```
Constructed next to House Park
```