ANSWERS

A2.5 Procedural, functional and data abstraction

Recursion and computational thinking · A level · OCR H446 2.1.1, AQA 7517 4.4.1.4 · about 20 min

What this lesson is about

Information hiding, hiding the values, the method and the representation, and swapping a data type's insides.

- Information hiding
- Procedural abstraction
- Functional abstraction
- Data abstraction

Questions

6 marks in all

1. What is information hiding? [1 mark]

- ☐ **A** Hiding all details of an object that do not contribute to its essential characteristics
- ☐ **B** Encrypting data so it cannot be read
- ☐ **C** Removing comments from code
- ☐ **D** Storing data in a hidden file

> **Answer:** A. Users of a component see its interface, not how it works inside.

2. print(30 * 20) is replaced by def area(width, height): return width * height. Which kind of abstraction is this step? [1 mark]

- ☐ **A** Procedural abstraction: the actual values are abstracted away, leaving a computational method
- ☐ **B** Functional abstraction: the method is hidden
- ☐ **C** Data abstraction: the representation is hidden
- ☐ **D** Abstraction by generalisation

> **Answer:** A. The particular values have gone, but the method (multiply) is still visible.

3. Two functions both return the smallest reading: one scans the list, one sorts it. A caller uses either without knowing which. What does this show? [1 mark]

- ☐ **A** Functional abstraction: the particular computation method is disregarded
- ☐ **B** Procedural abstraction
- ☐ **C** Decomposition
- ☐ **D** Composition

> **Answer:** A. Only the relationship between input and output matters to the caller.

4. A stack is used only through push, pop and is_empty. Inside, it could be an array with a top pointer or a linked list. What is this? [1 mark]

- ☐ **A** Data abstraction
- ☐ **B** Procedural decomposition
- ☐ **C** Problem reduction
- ☐ **D** Representational abstraction

> **Answer:** A. How the data is represented is hidden behind the operations, so the representation can change without affecting the code that uses it.

5. What does this program print? [1 mark]

```
def new_time(minutes, seconds):
    return minutes * 60 + seconds

def add_seconds(t, s):
    return t + s

def show(t):
    return f"{t // 60}:{t % 60:02d}"

lap = add_seconds(new_time(2, 45), 30)
print(show(lap))
```

> **Answer:**
>
> ```
> 3:15
> ```
>
> The time is stored as 165 seconds; adding 30 gives 195, which show turns back into minutes and seconds.

6. What are benefits of hiding a component's implementation behind its interface? [1 mark]

Tick every answer that is true.

- ☐ **A** The inside can be changed without breaking code that uses it
- ☐ **B** Users have less to understand
- ☐ **C** Nobody can depend on an internal detail by accident
- ☐ **D** The component always runs faster

> **Answer:** A, B, C. Information hiding is about managing change and complexity, not speed.

The task: swap the insides

The robot stands in the middle of the mat, turns through eight 45 degree steps and logs a distance at each. The starter's log is a list, and the program at the bottom uses it only through five operations: `new_log()`, `add_reading(log, cm)`, `count(log)`, `mean(log)` and `nearest(log)`. Change the **representation** without changing the program at the bottom: - a log must now be a dictionary holding only three figures: how many readings there have been, their running total, and the smallest reading so far; - `new_log()` returns an empty log; `add_reading(log, cm)` updates the figures for a reading `cm` (a number of cm) and returns nothing; `count(log)` returns the number of readings; `mean(log)` returns the mean reading; `nearest(log)` returns the smallest reading; - do not keep the readings, and do not use `append`, `len` or `sum` anywhere. The program at the bottom must still print `count 8`, then `mean <m>` and `nearest <n>` with exactly the values the list version prints.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def new_log():
    return []

def add_reading(log, cm):
    log.append(cm)

def count(log):
    return len(log)

def mean(log):
    return sum(log) / len(log)

def nearest(log):
    return min(log)

# the program that uses the log: do not change anything below this line
log = new_log()
for i in range(8):
    add_reading(log, distance())
    turn_right(40, angle=45)
print("count", count(log))
print("mean", round(mean(log), 1))
print("nearest", nearest(log))
```

The hint students can ask for: Decide what the three questions need, rather than what the readings were: how many there have been, what they add up to, and the smallest so far. A new log starts those figures off; each new reading updates them. The program at the bottom must not notice the change.

A solution

```
from bugbot import *
connect()

def new_log():
    return {"count": 0, "total": 0.0, "nearest": None}

def add_reading(log, cm):
    log["count"] = log["count"] + 1
    log["total"] = log["total"] + cm
```

```
    if log["nearest"] is None or cm < log["nearest"]:
        log["nearest"] = cm

def count(log):
    return log["count"]

def mean(log):
    return log["total"] / log["count"]

def nearest(log):
    return log["nearest"]

# the program that uses the log: do not change anything below this line
log = new_log()
for i in range(8):
    add_reading(log, distance())
    turn_right(40, angle=45)
print("count", count(log))
print("mean", round(mean(log), 1))
print("nearest", nearest(log))
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.