ANSWERS

A2.6 Decomposition, composition and automation

Recursion and computational thinking · A level · OCR H446 2.1.3, AQA 7517 4.4.1.9 · about 20 min

What this lesson is about

Thinking procedurally, compound procedures and compound data, and putting a model into action.

- Decomposition
- Composition
- Automation

Questions

5 marks in all

1. What is procedural decomposition? [1 mark]

- ☐ **A** Breaking a problem into sub-problems that each accomplish an identifiable task, which may be further subdivided
- ☐ **B** Combining procedures into a compound procedure
- ☐ **C** Removing unnecessary detail from a problem
- ☐ **D** Running a model to solve a problem

> **Answer:** A. Decomposition takes a problem apart; composition builds a solution up.

2. Which are examples of composition? [1 mark]

Tick every answer that is true.

- ☐ **A** A fetch procedure built from drive_to and grip
- ☐ **B** A tree made of nodes, each holding a list of smaller trees
- ☐ **C** A route stored as a list of shapes, each a list of legs
- ☐ **D** Removing the colours from a map

> **Answer:** A, B, C. Composition combines procedures into compound procedures, or data objects into compound data. Removing detail is abstraction.

3. Put the steps of automation in order. [1 mark]

Number the lines 1 to 4 to put them in the right order.

___ `Execute the code`
___ `Create an algorithm`
___ `Implement the model in data structures`
___ `Implement the algorithm in program code`

Answer:

```
Create an algorithm
Implement the algorithm in program code
Implement the model in data structures
Execute the code
```

Automation puts a model into action: algorithm, code, the model as data, then running it.

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

```
def double(x):
    return x * 2

def add_three(x):
    return x + 3

def compose(f, g):
    def both(x):
        return f(g(x))
    return both

h = compose(double, add_three)
print(h(4))
```

Answer:

```
14
```

compose(double, add_three) does add_three first, then double: (4 + 3) * 2 = 14.

5. When thinking procedurally about a problem, what should you identify? [1 mark]

Tick every answer that is true.

- ☐ **A** The components of the problem
- ☐ **B** The components of the solution
- ☐ **C** The order of the steps needed
- ☐ **D** The sub-procedures necessary
- ☐ **E** The programming language's keywords

Answer: A, B, C, D. These are the four parts of OCR's thinking procedurally. The language is a detail for later.

The task: a route built from pieces

A **leg** is a tuple `(direction, cm)`, where `direction` is one of `"forward"`, `"backward"`, `"left"` or `"right"` and `cm` is a distance. A **shape** is a list of legs. A **route** is a list of shapes. The starter builds this route from two shapes: `step = [("forward", 20), ("right", 20)]`, `square = [("forward", 20), ("right", 20), ("backward", 20), ("left", 20)]`, `route = [step, square, step]` Write three procedures, each built from the one before: - `drive_leg(leg)` prints `leg <direction> <cm>` and drives that leg at speed 60, sliding sideways for `left` and `right` (the robot never turns); - `drive_shape(shape)` drives every leg of a shape, using `drive_leg`; - `drive_route(route)` drives every shape of a route, using `drive_shape`. Drive the route, then print `route done: <shapes> shapes, <legs> legs`, which for this route is `route done: 3 shapes, 8 legs`. The robot should finish 40 cm right of and 40 cm up from where it started.

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

step = [("forward", 20), ("right", 20)]
square = [("forward", 20), ("right", 20), ("backward", 20), ("left", 20)]
route = [step, square, step]

def drive_leg(leg):
    direction, cm = leg

def drive_shape(shape):
    pass

def drive_route(route):
    pass
```

The hint students can ask for: Write the smallest piece first and test it on one leg. A shape is only a list of legs, so driving it is the smaller piece used once per item; a route is a list of shapes, so the same idea again one level up. Count the legs as they are driven, or work them out from the data.

A solution

```
from bugbot import *
connect()

step = [("forward", 20), ("right", 20)]
square = [("forward", 20), ("right", 20), ("backward", 20), ("left", 20)]
route = [step, square, step]

def drive_leg(leg):
    direction, cm = leg
    print("leg", direction, cm)
    if direction == "forward":
        forward(60, distance=cm)
    elif direction == "backward":
        backward(60, distance=cm)
    elif direction == "left":
        left(60, distance=cm)
    else:
        right(60, distance=cm)
    return 1
```

```
def drive_shape(shape):
    legs = 0
    for leg in shape:
        legs = legs + drive_leg(leg)
    return legs

def drive_route(route):
    legs = 0
    for shape in route:
        legs = legs + drive_shape(shape)
    return legs

legs = drive_route(route)
print(f"route done: {len(route)} shapes, {legs} legs")
```

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