ANSWERS

A13.1 The functional paradigm

Functional programming · A level · OCR H446 1.2.4, AQA 7517 4.12.2.1, Eduqas A500QS 1.4 · about 20 min

BugBotLab

What this lesson is about

Side effects, pure functions and referential transparency, immutability and statelessness, with the robot's side effects kept at the edges.

- A function, in the mathematical sense
- Immutability
- Statelessness
- Where the robot fits

Questions

6 marks in all

1. Which best describes a pure function? [1 mark]

- ☐ **A** Its result depends only on its arguments and it has no side effects
- ☐ **B** It is written without any if statements
- ☐ **C** It never takes more than one argument
- ☐ **D** It is defined inside a class

> **Answer:** A. Pure means the same arguments always give the same result, and calling it changes nothing else.

2. Which of these are side effects of a function? [1 mark]

Tick every answer that is true.

- ☐ **A** Printing a message
- ☐ **B** Changing a global variable
- ☐ **C** Returning a value
- ☐ **D** Reading the distance sensor
- ☐ **E** Working out the square root of its argument

> **Answer:** A, B, D. Returning a value computed only from the arguments is what a function is for; anything else it does, including reading the world or changing it, is a side effect.

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

```
calls = 0

def next_id(name):
    global calls
    calls = calls + 1
    return name + str(calls)

print(next_id("bot"), next_id("bot"))
```

Answer:

```
bot1 bot2
```

The same call gives two different results because each call changes the global calls, so next_id is not pure.

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

```
a = [1, 2]
b = a
a = a + [3]
print(b)
```

Answer:

```
[1, 2]
```

a + [3] builds a new list and makes a refer to it, so b still refers to the original, unchanged list. a.append(3) would have changed the list both names shared.

5. A call to a pure function can always be replaced by the value it returns without changing what the program does. What is this property called? [1 mark]

Answer: referential transparency. Referential transparency follows from having no side effects and no dependence on state.

6. What does statelessness mean in functional programming? [1 mark]

- ☐ **A** There is no program state that changes as the program runs: a name, once given a value, keeps it
- ☐ **B** The program has no variables at all
- ☐ **C** The program cannot store data in files
- ☐ **D** Functions cannot take parameters

Answer: A. Instead of updating variables in a loop, a functional program builds new values, using recursion and higher-order functions.

The task: make it pure

The starter has two impure functions. Rewrite both as pure functions, with no `global` and nothing changed in place: - `count_clear(readings, limit)` : `readings` is a tuple of distances in cm (floats), `limit` a number of cm. It **returns** how many readings are greater than `limit`, and gives the same answer however many times it is called. - `add_reading(log, cm)` : `log` is a tuple of floats and `cm` a float. It **returns** a new tuple with `cm` added at the end, and leaves `log` exactly as it was. The main program (keep it as it is) prints `clear: 2` twice, then `before: (31.0, 51.0, 15.0, 44.0)` and `after: (31.0, 51.0, 15.0, 44.0, 60.0)` . Write the functions in the stateless style: no `global` , no `for` or `while` loop (use recursion, as `total_of` does), and no `append` , `extend` or `+=` anywhere.

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

READINGS = (31.0, 51.0, 15.0, 44.0)

count = 0
def count_clear(readings, limit):
    global count
    for cm in readings:
        if cm > limit:
            count = count + 1
    return count

def add_reading(log, cm):
    log = list(log)
    log.append(cm)
    return tuple(log)

print("clear:", count_clear(READINGS, 40))
print("clear:", count_clear(READINGS, 40))
new_log = add_reading(READINGS, 60.0)
print("before:", READINGS)
print("after:", new_log)
```

The hint students can ask for: A pure function can only use its parameters. For the count, think about the empty tuple first, then how the count for a whole tuple relates to its first item and the count for the rest. For the log, which operator makes a new tuple from two tuples?

A solution

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

READINGS = (31.0, 51.0, 15.0, 44.0)

def count_clear(readings, limit):
    if readings == ():
        return 0
    first = 1 if readings[0] > limit else 0
    return first + count_clear(readings[1:], limit)

def add_reading(log, cm):
    return log + (cm,)
```

```
print("clear:", count_clear(READINGS, 40))
print("clear:", count_clear(READINGS, 40))
new_log = add_reading(READINGS, 60.0)
print("before:", READINGS)
print("after:", new_log)
```

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