ANSWERS

F6.7 Project: the fail-safe controller

BugBotLab

Robust programs · GCSE · about 25 min

What this lesson is about

A command program that authenticates, validates every command, and is tested against a plan.

- The brief
- Decompose it
- Step 1: validate one command
- Step 2: log in
- Step 3: the main loop
- Test plan for the whole program

Questions

4 marks in all

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

```
def parse(command):
    parts = command.strip().lower().split()
    if len(parts) != 2 or parts[0] not in ["forward", "back"]:
        return None
    if not parts[1].isdigit() or not 1 <= int(parts[1]) <= 50:
        return None
    return (parts[0], int(parts[1]))

print(parse("Forward 20"), parse("forward 60"), parse(""))
```

Answer:

```
('forward', 20) None None
```

lower() accepts Forward; 60 is out of range; an empty command has no parts.

2. Why put all the command checks in one function, parse? [1 mark]

- ☐ **A** Every command goes through the same checks, and they can be tested on their own
- ☐ **B** Functions run faster
- ☐ **C** Python requires validation in a function
- ☐ **D** It avoids using a loop

Answer: A. One place for validation means nothing reaches the robot unchecked.

3. In the controller's test plan, an empty command is which kind of test data? [1 mark]

- ☐ **A** Erroneous
- ☐ **B** Normal
- ☐ **C** Boundary
- ☐ **D** Valid

Answer: A. It is the wrong kind of input altogether; the controller must refuse it without crashing.

4. Which comes first in the controller, and why? [1 mark]

- ☐ **A** Authentication, so no commands are accepted from someone who is not allowed
- ☐ **B** Validation, so the password is sensible
- ☐ **C** Testing, so the robot is safe
- ☐ **D** The main loop

> **Answer:** A. Check who is giving commands before checking any command.

The task: the fail-safe controller

Build the controller from the brief with `operators = {"ada": "robot42"}`. Print `access denied` for a wrong login, `welcome <name>` for a right one, `invalid command` for every command it refuses, and `bye` on quit. The task types: `ada` and `oops`, then `ada` and `robot42`, then the commands `forward 20`, `fly 10`, `right 500`, an empty line, `right 15`, `beep` and `quit`.

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

operators = {"ada": "robot42"}
MOVES = ["forward", "back", "left", "right"]
```

The hint students can ask for: Three parts: a login that allows a limited number of tries, a parser that splits a command and refuses anything it does not recognise, and a main loop that acts on what the parser gives back. Refusing safely is the point.

A solution

```
from bugbot import *
connect()
operators = {"ada": "robot42"}
MOVES = ["forward", "back", "left", "right"]

def login(tries=3):
    for attempt in range(tries):
        name = input("Username? ")
        password = input("Password? ")
        if name in operators and operators[name] == password:
            return name
        print("access denied")
    return None

def parse(command):
    parts = command.strip().lower().split()
    if len(parts) == 1 and parts[0] in ["beep", "quit"]:
        return (parts[0], 0)
    if len(parts) != 2 or parts[0] not in MOVES:
        return None
    if not parts[1].isdigit() or not 1 <= int(parts[1]) <= 50:
        return None
    return (parts[0], int(parts[1]))

user = login()
if user:
    print("welcome", user)
    while True:
        result = parse(input("Command? "))
        if result is None:
            print("invalid command")
            tone(200, 0.3)
        elif result[0] == "quit":
            print("bye")
            break
        elif result[0] == "beep":
            tone(880, 0.3)
```

```
        elif result[0] == "forward":
            forward(50, distance=result[1])
        elif result[0] == "back":
            backward(50, distance=result[1])
        elif result[0] == "left":
            left(50, distance=result[1])
        elif result[0] == "right":
            right(50, distance=result[1])
```

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