ANSWERS

5.6 Project: rescue

Behaviours · Robot club · about 25 min

What this lesson is about

Search, approach, return: camera, servoing and states in one machine.

- The plan, drawn
- Search and approach
- Home
- What you have built
- Where next

Questions

7 marks in all

1. Put the rescue states in the order the robot goes through them. [1 mark]

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

___ `home`
___ `done`
___ `search`
___ `approach`

Answer:

```
search
approach
home
done
```

Find the tag, drive to it, then go back to the start. Each arrow has a condition you can read off the sensors.

2. Which condition moves the machine from search to approach? [1 mark]

- ☐ **A** Tag 9 is in view
- ☐ **B** The robot is closer than 12 cm
- ☐ **C** The robot is near the start
- ☐ **D** The loop has run 300 ticks

Answer: A. As soon as the filtered list for tag 9 is not empty, there is something to approach.

3. Why does the code say `if tags and tags[0][3] < 12` rather than just `if tags[0][3] < 12`? [1 mark]

- ☐ **A** If the list is empty, `tags[0]` would be an error, and `and` stops before reading it
- ☐ **B** `and` makes the test run faster
- ☐ **C** It checks the tag is number 9
- ☐ **D** Without it the robot would never stop

Answer: A. `and` only checks the right side when the left is true, so an empty list is safe.

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

```
def step(state, tag_dist, at_home):
    if state == "search" and tag_dist is not None:
        return "approach"
    if state == "approach" and tag_dist is not None and tag_dist < 12:
        return "home"
    if state == "home" and at_home:
        return "done"
    return state

state = "search"
for tag_dist, at_home in [(None, False), (60, False), (30, False), (11, False), (None,
False), (None, True)]:
    state = step(state, tag_dist, at_home)
    print(state)
```

Answer:

```
search
approach
approach
home
home
done
```

The machine only changes state when its own condition comes true. Losing the tag after it is home does not matter, because home only cares about being near the start.

5. How does the home state know where home is? [1 mark]

- ☐ **A** Positions are relative to the start, so home is (0, 0)
- ☐ **B** It searches for a home tag
- ☐ **C** It reverses every move it made
- ☐ **D** It follows the distance sensor back

Answer: A. `position()` counts from where the robot started, so `go_to(0, 0)` until `near(0, 0)` takes it back.

6. The task starter stops at the marker with `break`. What turns it into the full rescue? [1 mark]

- ☐ **A** Set `state = "home"` instead of breaking, and add an `elif state == "home":` branch
- ☐ **B** Add a second loop after the first one that searches again
- ☐ **C** Change the 12 to 0
- ☐ **D** Call `set_cv("apriltag")` again at the marker

Answer: A. A new stage of the plan is a new state and a new transition, inside the same loop.

7. The rescue ends with `print("home:", position(), "after", tick / 10, "seconds")`. If the loop ended on tick 437, how many seconds does it print? Give the exact number. [1 mark]

Answer: 43.7 (accept within 0.05). Ten ticks a second, so 437 ticks is 43.7 seconds.

The task: rescue

Find marker 9, print `found 9`, drive to it, and return home, without touching the box.

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

# maths: atan2, hypot, sin, cos, radians
import math

def wrapped(h):
    return (h + 180) % 360 - 180

def go_to(x, y, speed=70):
    # where am I?
    px, py = position()
    a = math.radians(wrapped(math.degrees(math.atan2(x - px, y - py)) - heading()))
    # forward, sideways, rotation: -100 to 100 each, until the next command
    drive(speed * math.cos(a), speed * math.sin(a), wrapped(0 - heading()) * 3)

def near(x, y, cm=6):
    # where am I?
    px, py = position()
    return math.hypot(x - px, y - py) < cm

def bearing_of(cx):
    return (cx - 160) * 120 / 320

# camera: the tag detector
set_cv('apriltag')
state = 'search'
# do this 590 times (tick counts from 0)
for tick in range(590):
    tags = [t for t in apriltags() if t[0] == 9]
    if state == 'search':
        if tags:
            print('found 9')
            state = 'approach'
        else:
            # spin clockwise on the spot at 40
            turn_right(40)
    elif state == 'approach':
        if tags and tags[0][3] < 12:
            # leave the loop
            break
        elif tags:
            # forward, sideways, rotation: -100 to 100 each, until the next command
            drive(60, 0, bearing_of(tags[0][1]) * 3)
        else:
            # spin clockwise on the spot at 30
            turn_right(30)
    # pause 0.1 s (the robot keeps doing what it was told)
    wait(0.1)
# all motors off
stop()
```

The hint students can ask for: Find marker 9 with the camera, print `found 9`, drive to it, then come back home, all without touching the box. Search, approach and return are three states.

A solution

```
from bugbot import *
connect()
def wrapped(h):
    return (h + 180) % 360 - 180
import math

def go_to(x, y, speed=70):
    # one tick of sliding straight towards (x, y) (relative to the start) while holding
heading 0
    px, py = position()
    a = math.radians(wrapped(math.degrees(math.atan2(x - px, y - py)) - heading()))
    drive(speed * math.cos(a), speed * math.sin(a), wrapped(0 - heading()) * 3)

def near(x, y, cm=6):
    px, py = position()
    return math.hypot(x - px, y - py) < cm
def bearing_of(cx):
    return (cx - 160) * 120 / 320

set_cv("apriltag")
state = "search"
for tick in range(590):
    tags = [t for t in apriltags() if t[0] == 9]
    if state == "search":
        if tags:
            print("found 9")
            state = "approach"
        else:
            turn_right(40)
    elif state == "approach":
        if tags and tags[0][3] < 12:
            state = "home"
        elif tags:
            drive(60, 0, bearing_of(tags[0][1]) * 3)
        else:
            turn_right(30)
    elif state == "home":
        if near(0, 0, cm=8):
            break
        go_to(0, 0, speed=60)
    wait(0.1)
stop()
```

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