Code a thoughtful robot

From your first instruction to a robot that checks its own decisions.

Start with Unit 1 and follow the lessons in order. Each takes about 20–30 minutes. You need a browser, a notebook, and curiosity. An adult opens the academy server once. The Python experiments use invented readings and printed action plans; they do not connect to a robot, camera, microphone, or cloud service. You can finish the whole course without robot hardware.

Contents

- 9.3 · Challenge · explore a feedback rule

Who does what?

Draw an input → program → output map and run your first Python line.

Bring this idea with you

No earlier lesson needed. Find the Run Python button and keep a notebook nearby.

1. Understand the idea

Imagine a museum desk robot. A visitor presses a button, its program chooses a greeting, and a speaker plays it. The button is an input; the speaker is an output. A motor is another output. A camera supplies image data, but recognizing an object needs extra processing.

Today the browser is your lab. Python print displays words. Quotation marks surround text; parentheses hold what print receives. Read from top to bottom. The little # comment is a note for humans and does not run.

input	Information given to a system.
output	Something a system produces.
program	Instructions a computer follows.

Predict before you run

Will this code make the physical Reachy speak?

▼ **Compare your prediction**

No. print produces text in this browser. A displayed plan is different from a speaker playing sound.

2. Try it, then change it

1. Press Run Python once. The first load may take a moment. Compare all three lines with your prediction.
2. Change only Hello, explorer! to Welcome to the museum! Keep the quotes and parentheses. Run again.
3. Draw three boxes in your notebook. Put the button, program, and speaker into the correct boxes.

Runnable Python example

```
# Our first observation
print("Input: visitor button")
print("Program: choose a greeting")
print("Output: Hello, explorer!")
```

▼ Expected output for the original example

```
Input: visitor button
Program: choose a greeting
Output: Hello, explorer!
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

If you see SyntaxError, check for a missing closing quote or parenthesis. Use straight quotes, not curly quotation marks from a word processor.

3. Build a mini-challenge

Plan a robot that notices a dark room and displays a message. Print an input, program, and output line.

Check your result

- Your input measures something.
- Your program describes a choice.
- Your output is observable.

▼ Worked solution · try your own first

```
print("Input: light reading")
print("Program: check if the room is dark")
print("Output: Please turn on a lamp")
```

Solution output

```
Input: light reading
Program: check if the room is dark
Output: Please turn on a lamp
```

4. Explain it back

Which is an input: a camera or a speaker? How could you verify speech?

▼ Compare your explanation

The camera is an input. To verify speech, someone must actually hear the approved audio; printed text alone is not enough.

Notebook: My prediction / change / observation / explanation / next test.

Give instructions a robot can follow

Turn a vague job into an ordered, testable plan.

Bring this idea with you

An output is something observable. Python executes lines in order.

1. Understand the idea

"Be helpful" is a goal, but it does not say what to do first. Break it into small behaviors: show a welcome, give one instruction, then say the tour is ready. Write these steps in everyday words first. That is pseudocode.

A robot follows the order you wrote, even if you meant a different order. Numbered output makes that order easy to inspect. In our example the numbers are part of the text; Python is not checking their order for you.

algorithm	An ordered method for solving a problem.
sequence	Steps performed in a particular order.
pseudocode	A plan written in ordinary words before code.

Predict before you run

If you swap the last two lines, will Python repair the order?

▼ **Compare your prediction**

No. It will print Ready to observe before Put your model on the paper.

2. Try it, then change it

1. Run once and read the lines aloud in order.
2. Swap the last two lines and run. Explain why the result is less useful.
3. Put them back. Ask a partner to follow your instructions with a toy, or trace them on paper by yourself.

Runnable Python example

```
print("1. Welcome to the desk")
print("2. Put your model on the paper")
print("3. Ready to observe")
```

▼ **Expected output for the original example**

```
1. Welcome to the desk
2. Put your model on the paper
3. Ready to observe
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

A program can run without an error and still give a bad sequence. Trace each line using a finger; compare with your paper plan.

3. Build a mini-challenge

Write a three-step observation plan that prepares the scene before describing it. No camera is needed.

Check your result

- Preparation happens before observation.
- The final step records what happened.
- A partner can follow the steps without guessing.

▼ **Worked solution · try your own first**

```
print("1. Place a toy on plain paper")
print("2. Observe its shape and color")
print("3. Write two observations")
```

Solution output

```
1. Place a toy on plain paper
2. Observe its shape and color
3. Write two observations
```

4. Explain it back

Why plan before coding?

▼ Compare your explanation

A plan exposes missing steps and wrong order before you spend time fixing Python syntax.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · become a robot detective

Separate a request, an observation, and a conclusion.

Bring this idea with you

A printed instruction is a plan. It is not evidence that a physical action happened.

1. Understand the idea

Your desk robot was asked to greet someone, but the room stayed silent. That observation does not prove the speaker is broken. The wrong file, volume, or connection might also explain it. Record the first missing evidence before deciding what failed.

Use three questions: what did I expect, what did I observe, and what could I check next? Change one thing at a time so you can connect a change to a result. For real heat, unusual noise, red lights, or stuck parts, stop and get an adult. Do not open the robot.

evidence	An observation that supports a claim.
hypothesis	A possible explanation you can investigate.
bug	A problem in a program or its behavior.

Predict before you run

Do these results prove the robot speaker works?

▼ **Compare your prediction**

No. They test text output only. The conclusion must stay within the evidence.

2. Try it, then change it

1. Run the program. Point to the exact evidence supporting its conclusion.
2. Change Observed to no greeting appears. The old conclusion is now unsupported; repair it.
3. Write two possible explanations for silence. Circle one non-invasive check an adult could help with.

Runnable Python example

```
print("Expected: a greeting appears")
print("Observed: a greeting appears")
print("Conclusion: the text output worked")
```

▼ **Expected output for the original example**

```
Expected: a greeting appears
Observed: a greeting appears
Conclusion: the text output worked
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

If you write “broken,” ask which observation proves that exact cause. “Did not respond” is often a more accurate first report.

---

3. Build a mini-challenge

Create a detective report for a pretend movement request that produced no visible movement. Include a next check without declaring a broken motor.

Check your result

- State the request and the observation separately.
- Keep the cause uncertain.
- Choose an adult-assisted check; never force a joint.

▼ **Worked solution · try your own first**

```
print("Requested: a small greeting")
print("Observed: no visible movement")
print("Conclusion: movement is not verified")
print("Next: ask an adult to check the status")
```

Solution output

```
Requested: a small greeting
Observed: no visible movement
Conclusion: movement is not verified
Next: ask an adult to check the status
```

---

4. Explain it back

What evidence would let you mark this unit complete?

▼ **Compare your explanation**

I can label input and output, trace a sequence, and write a report whose conclusion follows from an observation. A checkmark records my review, not a certification.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 0 · inspect a real robot with an adult

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

UNIT 2 / Write a clear program · LESSON 2.1 · 25 MIN

Give useful values a name

Use a variable to change a greeting in one place.

Bring this idea with you

print displays text, and quotes mark literal words.

1. Understand the idea

A variable gives a value a reusable name. robot_name = "Reachy" assigns the text Reachy to robot_name. The equals sign means assign here; it does not ask a question. Choose names that describe the value and use underscores instead of spaces.

Inside quotes, robot_name would just be those ten characters. Without quotes, Python looks up its value. print can take several items separated by commas and puts spaces between them. Assigning a new value changes what later lines read.

variable	A name that refers to a value.
string	A text value, written in quotes.
assignment	Giving a variable a value with =.

Predict before you run

Will changing robot_name on line 3 change the first printed line?

▼ **Compare your prediction**

No. The first line already used Reachy. Assignment affects later reads, not earlier output.

2. Try it, then change it

1. Predict both greetings, then run.
2. Change the first value to Desk buddy. Leave the second assignment alone. Run and check which line changes.
3. Temporarily put quotes around robot_name in the first print. Explain the difference, then restore the code.

Runnable Python example

```
robot_name = "Reachy"
print("Hello from", robot_name)
robot_name = "Museum helper"
print("Today I am", robot_name)
```

▼ **Expected output for the original example**

```
Hello from Reachy
Today I am Museum helper
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

NameError often means a spelling mismatch or using a name before assigning it. robot_name and Robot_name are different names.

---

3. Build a mini-challenge

Create a museum variable and use it in two messages. Change the museum name only once to update both.

Check your result

- Assign before using the variable.
- Use the variable without quotes in both print calls.
- Use an invented place name.

▼ **Worked solution · try your own first**

```
museum = "Moon Museum"
print("Welcome to", museum)
print("Enjoy exploring", museum)
```

Solution output

```
Welcome to Moon Museum
Enjoy exploring Moon Museum
```

---

4. Explain it back

What is the difference between "score" and score?

▼ **Compare your explanation**

"score" is literal text. score asks Python for the value assigned to that variable.

Notebook: My prediction / change / observation / explanation / next test.

Calculate before you act

Use arithmetic and units to predict a plan's duration.

Bring this idea with you

Variables store values. Some values are numbers instead of text.

1. Understand the idea

Use numbers without quotes when you want arithmetic. Python uses + for addition, - for subtraction, * for multiplication, and / for division. Parentheses group a calculation. Multiplication happens before addition unless parentheses change the order.

A variable name can remind you of its unit. three moves × two seconds per move = six seconds of planned movement. That excludes connection time and pauses. A duration is not an angle, and a printed calculation does not move Reachy.

integer	A whole number such as 3.
decimal	A number with a fractional part such as 1.5.
unit	The scale of a measurement, such as seconds or degrees.

Predict before you run

What becomes larger if seconds_per_move changes to 3?

▼ **Compare your prediction**

Planned time becomes 9 seconds, and the version with one pause becomes 10 seconds. This says nothing about distance.

2. Try it, then change it

1. Run the original and check both totals.
2. Change seconds_per_move to 3 and compare with your prediction.
3. On paper, calculate 3 * (2 + 1) and 3 * 2 + 1. Explain why the results differ.

Runnable Python example

```
moves = 3
seconds_per_move = 2
planned_seconds = moves * seconds_per_move
print("Planned seconds:", planned_seconds)
print("With one pause:", planned_seconds + 1)
```

▼ **Expected output for the original example**

```
Planned seconds: 6
With one pause: 7
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

"3" is text, while 3 is a number. If a calculation produces repeated characters or a type error, check whether you quoted a number.

---

3. Build a mini-challenge

Plan four gestures taking 1.5 seconds each, with one two-second pause after the entire sequence. Print the total.

Check your result

- Multiply the gesture count by duration.
- Add the pause once.
- Label the result in seconds.

▼ **Worked solution · try your own first**

```
gestures = 4
seconds_per_gesture = 1.5
pause_seconds = 2
print("Total seconds:", gestures * seconds_per_gesture + pause_seconds)
```

Solution output

```
Total seconds: 8.0
```

---

4. Explain it back

Why put seconds in a variable name?

▼ **Compare your explanation**

It makes the meaning clear and helps avoid mixing time with angles or other measurements.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · build a welcome desk

Combine variables, sequence, and arithmetic into a small program.

Bring this idea with you

Assign values first; print them in the order a visitor needs them.

1. Understand the idea

Before building, write requirements someone else could check. “Nice welcome” is hard to measure. “Shows the museum name, a time estimate, and a next instruction in that order” is testable. These requirements become your checklist.

Build one piece, run it, and add the next piece. Try a second set of values to find a hidden hard-coded answer. If your tour duration changes but the estimate does not, you probably printed the answer instead of calculating it.

requirement	A specific thing your solution must do.
test	A check comparing actual and expected behavior.

Predict before you run

What should the estimate be for five stations at two minutes each?

▼ **Compare your prediction**

10 minutes. Only the stations value needs to change.

2. Try it, then change it

1. Write the three requirements from the example before running it.
2. Change stations to 5. Check the greeting, estimate, and final instruction separately.
3. Add a one-minute introduction to the calculation. Keep the station time unchanged.

Runnable Python example

```
museum = "Tiny Worlds"
stations = 3
minutes_per_station = 2
print("Welcome to", museum)
print("Tour minutes:", stations * minutes_per_station)
print("Begin at the model display")
```

▼ Expected output for the original example

```
Welcome to Tiny Worlds
Tour minutes: 6
Begin at the model display
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

If you get 20 minutes, check whether you added the introduction inside the multiplication. It happens once, not once per station.

3. Build a mini-challenge

Create a welcome desk for Robot Gallery: four stations, three minutes per station, and a two-minute introduction.

Check your result

- Print the place name first.
- Calculate a 14-minute total using named values.
- Finish with a clear next instruction.

▼ Worked solution · try your own first

```
museum = "Robot Gallery"
stations = 4
minutes_per_station = 3
intro_minutes = 2
print("Welcome to", museum)
print("Tour minutes:", stations * minutes_per_station + intro_minutes)
print("Start at the welcome sign")
```

Solution output

```
Welcome to Robot Gallery
Tour minutes: 14
Start at the welcome sign
```

4. Explain it back

How do you know the estimate is calculated rather than memorized?

▼ **Compare your explanation**

Change an input and predict a new result. A working calculation updates correctly for both test cases.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 1 • save and run Python on your Mac

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Separate pose from timing

Describe a whole-head pose separately from its duration.

Bring this idea with you

A unit tells you whether a number means seconds, degrees, or something else.

1. Understand the idea

Reachy is a tabletop robot, not a wheeled rover. Its head pose is coordinated by several joints. A five-degree head roll does not mean every motor turns five degrees. Use the prepared robot software to coordinate them; never guess individual motor commands.

A motion plan names both the target and the time. For the same path, more time means lower average speed. It does not make a larger movement safe. Here we print a three-pose storyboard. These words describe a plan and are not SDK commands.

pose	A position and orientation.
actuator	A part, such as a motor, that produces physical action.
duration	How long an action takes.

Predict before you run

Does the printed word Neutral prove the robot reached neutral?

▼ **Compare your prediction**

No. This browser program prints a planned pose. Real movement needs an approved command and observation.

2. Try it, then change it

1. Draw the three poses using simple head sketches. Label each with its time.
2. Run, then change seconds to 3. Check that the poses remain identical.
3. Open the linked Reachy lesson if you want its browser motion rehearsal. Follow that lesson's adult setup before any real movement.

Runnable Python example

```
seconds = 2
print("Neutral for", seconds, "seconds")
print("Small tilt for", seconds, "seconds")
print("Neutral for", seconds, "seconds")
print("Planned seconds:", 3 * seconds)
```

▼ Expected output for the original example

```
Neutral for 2 seconds
Small tilt for 2 seconds
Neutral for 2 seconds
Planned seconds: 6
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

Do not confuse a successful calculation with hardware feedback. Label your evidence “paper,” “browser,” or “real robot.”

3. Build a mini-challenge

Storyboard the same three poses at 1.5 seconds each. Print the planned total and state what stays unchanged.

Check your result

- Keep neutral → small tilt → neutral.
- Label duration in seconds.
- Calculate 4.5 seconds without claiming a real test.

▼ Worked solution · try your own first

```
seconds = 1.5
print("Neutral, small tilt, neutral")
print("Seconds per pose:", seconds)
print("Planned seconds:", 3 * seconds)
print("Same poses; different timing")
```

Solution output

```
Neutral, small tilt, neutral
Seconds per pose: 1.5
Planned seconds: 4.5
Same poses; different timing
```

4. Explain it back

What is different about a head angle and a motor angle?

▼ **Compare your explanation**

A head angle describes the whole-head orientation. Individual joint angles cooperate to produce it and need not be equal.

Notebook: My prediction / change / observation / explanation / next test.

Change one thing at a time

Compare two timing plans using a fair experiment.

Bring this idea with you

Three poses at two seconds each have six seconds of planned movement.

1. Understand the idea

Suppose you want a greeting to seem calmer. You could change its angles, speed, words, and sound all at once, but then you would not know which change mattered. Keep the pose sequence fixed and vary only the duration.

Write your prediction first. Our arithmetic compares planned durations, not stopwatch measurements. A real timer would include delays and human reaction. Keep measured results in a separate column and label a skipped real test “not run.”

comparison	Looking for differences between results.
controlled variable	Something you deliberately keep the same.

Predict before you run

If both the angle and duration change, can this test isolate the effect of duration?

▼ **Compare your prediction**

No. Keep the poses unchanged to compare duration alone.

2. Try it, then change it

1. Make two columns: planned seconds and observed evidence. Do not fill an observation you have not made.
2. Run and compare the computed difference with subtraction on paper.
3. Set second_seconds to 3.0. Predict the difference before running.

Runnable Python example

```
poses = 3
first_seconds = 1.5
```

```
second_seconds = 2.0
first_total = poses * first_seconds
second_total = poses * second_seconds
print("First plan:", first_total)
print("Second plan:", second_total)
print("Extra seconds:", second_total - first_total)
```

▼ Expected output for the original example

```
First plan: 4.5
Second plan: 6.0
Extra seconds: 1.5
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

Use second_total - first_total to describe extra time in the second plan. Reversing the order gives a negative difference with a different meaning.

3. Build a mini-challenge

Compare three poses at 2 seconds with the same poses at 3 seconds. Report both totals and the extra time.

Check your result

- Keep the pose count identical.
- Calculate 6, 9, and 3 seconds.
- Describe one observation you would need before judging whether the real greeting feels calmer.

▼ Worked solution · try your own first

```
poses = 3
first_total = poses * 2
second_total = poses * 3
print("First seconds:", first_total)
print("Second seconds:", second_total)
print("Extra seconds:", second_total - first_total)
```

Solution output

```
First seconds: 6
Second seconds: 9
Extra seconds: 3
```

4. Explain it back

What did the code establish, and what did it not establish?

▼ **Compare your explanation**

It established the planned times. It did not measure real motion or prove that a visitor finds one gesture calmer.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · direct a robot greeting

Build a complete gesture storyboard with a return and an evidence plan.

Bring this idea with you

A useful plan includes order, duration, and a clear end.

1. Understand the idea

A greeting is a small system: introduce yourself, show one prepared gesture, return to the prepared neutral pose, and finish. Decomposition lets you check each part. A return belongs in the plan, but writing it does not guarantee recovery after a physical fault.

Work within the existing approved motion range. Personality can come from timing and words. Keep this challenge as a paper or printed plan; use the linked Reachy mission for its bounded physical example with adult setup.

decomposition	Breaking a larger job into smaller jobs.
constraint	A limit your design must respect.

Predict before you run

Why put the return pose in the storyboard?

▼ **Compare your prediction**

It makes the intended end state explicit, so you can check whether the sequence includes it.

2. Try it, then change it

1. Label the words as printed speech plans, not audible speech.
2. Draw a sleepy version by changing duration only. Write its predicted total.
3. Write what an observer would check: order, small prepared poses, intended return, and any warning signs.

Runnable Python example

```
seconds_per_pose = 2
pause_seconds = 1
print("Say: Welcome, explorer")
print("Plan: neutral, small tilt, neutral")
print("Motion seconds:", 3 * seconds_per_pose)
print("Total with pause:", 3 * seconds_per_pose + pause_seconds)
print("Finish the greeting")
```

▼ **Expected output for the original example**

```
Say: Welcome, explorer
Plan: neutral, small tilt, neutral
Motion seconds: 6
Total with pause: 7
Finish the greeting
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

If the storyboard sounds like live robot control, relabel it. Physical warnings require stopping and an adult, not repeatedly running a return command.

3. Build a mini-challenge

Direct a calm welcome using three prepared poses at three seconds each and one two-second pause. Print a useful ending.

Check your result

- Include the return in the pose sequence.
- Calculate 11 seconds including the pause.
- End with an instruction a visitor can understand.

▼ **Worked solution · try your own first**

```
seconds_per_pose = 3
pause_seconds = 2
print("Welcome to the quiet gallery")
print("Plan: neutral, small tilt, neutral")
print("Planned seconds:", 3 * seconds_per_pose + pause_seconds)
print("Please look at the model display")
```

Solution output

```
Welcome to the quiet gallery
Plan: neutral, small tilt, neutral
Planned seconds: 11
Please look at the model display
```

4. Explain it back

Name three checks before calling a greeting successful.

▼ **Compare your explanation**

The intended order happened, timing matched the test goal, and the observable output was verified. Record the test mode and any missing evidence.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 2 · rehearse a small, approved greeting

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Meet True and False

Represent a two-state input and compare values.

Bring this idea with you

A variable refers to a value. = assigns that value.

1. Understand the idea

A simple bumper input can answer pressed or not pressed. Our model uses a variable named pressed instead of a real sensor. Reachy has no VEX bumper attached in this course. Digital data is broader than yes/no, but today we study a two-state signal.

Python spells the Boolean values True and False with capital first letters and no quotes. The comparison == asks whether two values are equal. It differs from =, which assigns. The result can be stored or printed.

Boolean	A value that is either True or False.
comparison	A question about values that produces True or False.

Predict before you run

What will the last two lines show when pressed is True?

▼ **Compare your prediction**

Is pressed? True and Is released? False. The questions are opposites for a Boolean input.

2. Try it, then change it

1. Run with False and compare the three lines.
2. Change only pressed to True. Predict and rerun.
3. Write “assignment” beside = and “comparison” beside == in your notebook.

Runnable Python example

```
pressed = False
print("Pressed:", pressed)
```

```
print("Is pressed?", pressed == True)
print("Is released?", pressed == False)
```

▼ **Expected output for the original example**

```
Pressed: False
Is pressed? False
Is released? True
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

true is an undefined name in Python; True is the Boolean. A quoted "False" is text, not a Boolean signal.

---

3. Build a mini-challenge

Model a closed door as True. Print whether it is closed and whether it is open.

Check your result

- Use a Boolean, not the string "True".
- Use == for the two questions.
- Expect the two answers to differ.

▼ **Worked solution · try your own first**

```
door_closed = True
print("Closed?", door_closed == True)
print("Open?", door_closed == False)
```

Solution output

```
Closed? True
Open? False
```

---

4. Explain it back

Is pressed = True asking a question?

▼ **Compare your explanation**

No. It assigns True. pressed == True asks whether the value equals True.

Notebook: My prediction / change / observation / explanation / next test.

Choose one path with if / else

Make a visitor prompt respond to a modeled button.

Bring this idea with you

A Boolean is True or False. Comparisons produce Booleans.

1. Understand the idea

if runs its indented block when the condition is true. else runs a different block when it is false. Put a colon after if and else, and indent each block by four spaces. Only one branch of this pair runs.

The unindented final print happens after either choice. Trace your finger down the chosen path. A button not being pressed should produce a useful waiting message; it should not make the program guess that a visitor is ready.

condition	A test used to choose a path.
branch	One possible path through a program.
indentation	Leading spaces that group lines into a block.

Predict before you run

Does the final line run only when pressed is False?

▼ **Compare your prediction**

No. It is outside both indented blocks, so it runs after either branch.

2. Try it, then change it

1. Trace the False path before running.
2. Change pressed to True. Check that you see the welcome and final line, but not the waiting message.
3. Add a second print inside the welcome block. Indent it exactly like the first one.

Runnable Python example

```
pressed = False
if pressed:
    print("Welcome, explorer!")
else:
    print("Press when you are ready")
print("Button check finished")
```

▼ **Expected output for the original example**

```
Press when you are ready
Button check finished
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

IndentationError means the grouping is unclear. Compare the spaces before each print. A colon belongs on the if and else lines.

---

3. Build a mini-challenge

Show “Gallery ready” when ready is True and “Please wait” when it is False. Test both states.

Check your result

- Include both branches.
- Use consistent indentation.
- Write expected output for True and False before testing.

▼ **Worked solution · try your own first**

```
ready = True
if ready:
    print("Gallery ready")
else:
    print("Please wait")
```

Solution output

```
Gallery ready
```

---

4. Explain it back

Why include an else message here?

▼ **Compare your explanation**

It tells an unready visitor what to do next and gives us a visible result for the False test.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · wait for permission

Build a permission gate and test both outcomes.

Bring this idea with you

if / else selects a branch from the current input.

1. Understand the idea

Our museum helper should explain a pretend exhibit only after permission. The gate must come before the action. A message saying “permission denied” is not useful if the action already happened above it.

Changing a Boolean is a model of permission, not actual consent from a person. Real recording lessons require everyone nearby to agree and the adult privacy checks. This challenge only prints messages and captures no personal data.

gate	A condition that must be met before an action.
fallback	A prepared response when the requested action is unavailable.

Predict before you run

What breaks if Describe the pretend exhibit is printed before the if line?

▼ **Compare your prediction**

The gated action would happen even when permission is False. The condition must guard the action itself.

2. Try it, then change it

1. Test False first. Confirm that the action text is absent.
2. Test True. Confirm the description text appears exactly once.
3. Write a two-row test table: permission value, expected action, observed output.

Runnable Python example

```
permission = False
if permission:
    print("Describe the pretend exhibit")
```

```
else:
    print("Permission needed; no description started")
print("Finished this check")
```

▼ **Expected output for the original example**

```
Permission needed; no description started
Finished this check
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Read every line before and after the gate. An action outside the branch can accidentally run regardless of permission.

---

3. Build a mini-challenge

Create a pretend photo-preview gate. With permission False, print “Preview skipped”; otherwise print “Show the supplied toy picture.” Do not access a camera.

Check your result

- Test refusal as well as permission.
- No action text appears after refusal.
- Keep the input explicitly modeled.

▼ **Worked solution · try your own first**

```
permission = False
if permission:
    print("Show the supplied toy picture")
else:
    print("Preview skipped")
```

Solution output

```
Preview skipped
```

---

4. Explain it back

What evidence proves the False branch works?

▼ **Compare your explanation**

A refusal test shows the skipped message and no gated action. Testing only True leaves refusal behavior unchecked.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 6 • trace a local decision rule

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Read a number with its scale

Interpret a numeric input without inventing its meaning.

Bring this idea with you

A two-state input gives two possibilities. Numeric inputs can describe many levels.

1. Understand the idea

Light in the world varies continuously. A computer stores sampled, finite-resolution numbers representing it. Numeric readings and Boolean states need different questions. “How much?” asks for a measurement; “Is it dark?” requires a rule.

We invent a brightness scale from 0 to 100 for this lesson: 0 is the darkest end and 100 the brightest end. It is not lux, a measured camera value, or a built-in Reachy sensor API. Different sensors need their own units, ranges, and calibration.

scale	The range and meaning of a measurement.
sample	One recorded reading at one moment.
threshold	A boundary used to turn a measurement into a choice.

Predict before you run

What is brightness < threshold when both values are 40?

▼ **Compare your prediction**

False. Strictly less than does not include equality.

2. Try it, then change it

1. Run with 35 and label the scale in your notes.
2. Change brightness to 40, then 45. Predict each comparison before running.
3. Write what you would need to know before comparing readings from two different sensors: scale, units, and how readings were taken.

Runnable Python example

```
brightness = 35
threshold = 40
print("Model brightness (0 to 100):", brightness)
print("Below threshold?", brightness < threshold)
print("Distance below threshold:", threshold - brightness)
```

▼ Expected output for the original example

```
Model brightness (0 to 100): 35
Below threshold? True
Distance below threshold: 5
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

If a subtraction is negative, check which value is larger. The sign describes direction relative to the threshold; it is not automatically an error.

3. Build a mini-challenge

On the same invented scale, compare a reading of 62 to a threshold of 60. Print whether it is above the threshold.

Check your result

- Label the invented scale.
- Use > for strictly above.
- Explain why a reading is not automatically a robot action.

▼ Worked solution · try your own first

```
brightness = 62
threshold = 60
print("Model brightness (0 to 100):", brightness)
print("Above threshold?", brightness > threshold)
```

Solution output

```
Model brightness (0 to 100): 62
Above threshold? True
```

4. Explain it back

Can 35 from one device be compared directly with 35 from another?

▼ **Compare your explanation**

Only when their scales and meanings match. Equal numbers with different units may describe different things.

Notebook: My prediction / change / observation / explanation / next test.

Test both sides of a boundary

Create a threshold rule and test just below, at, and above it.

Bring this idea with you

< means strictly less than. Equality needs an explicit decision.

1. Understand the idea

A threshold turns a numeric reading into a choice. The design decision is what happens exactly at the boundary. Our rule uses brightness < 40, so 39 is dark and 40 is not dark. <= 40 would classify 40 differently.

Choose a threshold using observations rather than guessing forever. In a real experiment, hold the camera scene and position fixed, change lighting, and compare readings. Here we use invented values so you can practice the decision precisely.

boundary test	A test at or very close to a rule's cutoff.
less or equal	<= includes the cutoff itself.

Predict before you run

Which values among 39, 40, and 41 trigger more light?

▼ **Compare your prediction**

Only 39 with the < 40 rule.

2. Try it, then change it

1. Write a three-row test table for 39, 40, and 41.
2. Run each value separately and record the exact output.
3. Change < to <= and repeat the value 40. Explain the changed behavior.

Runnable Python example

```
brightness = 40
if brightness < 40:
    print("Suggest more light")
```

```
else:
    print("Light is sufficient in this model")
```

▼ **Expected output for the original example**

```
Light is sufficient in this model
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Do not fix a boundary by changing random input values. First decide which branch should own equality, then choose the comparison symbol.

3. Build a mini-challenge

Create a model warning when brightness is 70 or higher. Test 69, 70, and 71.

Check your result

- Use >= to include 70.
- Provide a non-warning branch.
- Predict all three boundary results.

▼ **Worked solution · try your own first**

```
brightness = 70
if brightness >= 70:
    print("Model warning: bright")
else:
    print("Model brightness acceptable")
```

Solution output

```
Model warning: bright
```

4. Explain it back

Why test 40 instead of only 10 and 90?

▼ **Compare your explanation**

Far-apart tests can both pass while the exact boundary is wrong. Equality reveals what the rule really includes.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · tune the observation desk

Use a small dataset to justify a threshold and describe its limits.

Bring this idea with you

A rule must specify how it treats equality, and measurements need a scale.

1. Understand the idea

Suppose our invented desk dataset gives dim readings 18, 22, 24 and bright readings 68, 72, 75. A threshold of 45 separates these examples. It does not prove that every future reading will be classified correctly. A reading near the cutoff deserves special testing.

Test with a second set after choosing a rule. If you only check the examples used to choose it, you may miss new situations. Flickering light and changing backgrounds can affect real measurements. Record the conditions and avoid claiming a universal threshold.

calibration	Checking readings against known conditions.
noise	Variation in readings that may not represent a useful change.

Predict before you run

Do the six example readings prove that 45 is always the best threshold?

▼ **Compare your prediction**

No. Several thresholds separate that small dataset. New scenes and near-boundary readings still need tests.

2. Try it, then change it

1. Check on paper that 18, 22, and 24 fall below 45, while 68, 72, and 75 do not.
2. Run separate tests with 44, 45, and 46. Record the equality rule.
3. Try a new dim example of 47. Explain how it challenges the current rule rather than silently relabeling the example.

Runnable Python example

```
threshold = 45
brightness = 44
if brightness < threshold:
    print("Model result: dim")
else:
    print("Model result: bright enough")
print("Threshold:", threshold)
```

▼ **Expected output for the original example**

```
Model result: dim
Threshold: 45
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

A wrong classification can come from the threshold, the input, or the assumed scale. Check those separately before changing the rule.

3. Build a mini-challenge

A second invented desk has dim examples 20 and 30, and bright examples 60 and 80.
Choose threshold 50, write its rule, and test 49, 50, and 51.

Check your result

- The four calibration examples fit the rule.
- At 50, the bright-enough branch runs.
- Your report names one untested condition.

▼ **Worked solution · try your own first**

```
threshold = 50
brightness = 50
if brightness < threshold:
    print("Model result: dim")
else:
    print("Model result: bright enough")
```

Solution output

```
Model result: bright enough
```

4. Explain it back

What should your threshold report include?

▼ **Compare your explanation**

The scale, sample conditions, chosen comparison, boundary results, and a limitation such as untested lighting or background changes.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 3 · investigate light and a camera image

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Repeat a fixed number of times

Trace a for loop and predict its exact number of iterations.

Bring this idea with you

Indented lines belong to a block. A variable can change over time.

1. Understand the idea

Copying the same print line many times makes changes awkward. A for loop repeats an indented block for each value in a sequence. range(3) provides 0, 1, and 2: three values, not four. The name attempt takes each value in turn.

The final unindented line runs once after the loop. Numbering attempts with attempt + 1 is useful for people who expect counting to begin at one. Fixed counts give a clear stopping point; begin with small counts when testing.

loop	A structure that repeats a block.
iteration	One trip through a loop.
range	A Python sequence of numbers with an excluded ending value.

Predict before you run

How many times will All checks finished appear?

▼ **Compare your prediction**

Once, because it is outside the loop.

2. Try it, then change it

1. Write the attempt values 0, 1, and 2 on paper. Trace the displayed numbers beside them.
2. Run, then change range(3) to range(2). Predict the exact output.
3. Try range(0). Explain why only the final message remains.

Runnable Python example

```
for attempt in range(3):
    print("Check", attempt + 1)
print("All checks finished")
```

▼ **Expected output for the original example**

```
Check 1
Check 2
Check 3
All checks finished
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

An ending message printed repeatedly is probably indented inside the loop. Count spaces, and trace which lines are repeated.

---

3. Build a mini-challenge

Print four numbered observation reminders, then a single end message.

Check your result

- Use one loop instead of four copied lines.
- Show reminder numbers 1 through 4.
- Print the ending once.

▼ **Worked solution · try your own first**

```
for observation in range(4):
    print("Observation", observation + 1)
print("Notebook ready")
```

Solution output

```
Observation 1
Observation 2
Observation 3
Observation 4
Notebook ready
```

---

4. Explain it back

Does range(4) include 4?

▼ **Compare your explanation**

No. It supplies 0, 1, 2, and 3, which still makes four iterations.

Notebook: My prediction / change / observation / explanation / next test.

Name a behavior and reuse it

Write a function with a parameter and call it more than once.

Bring this idea with you

A variable names a value. A loop repeats a block.

1. Understand the idea

A function packages a behavior behind a useful name. def show_label(label): defines a function. Its indented block does not run until you call it. The parameter label receives the argument from each call.

This resembles a reusable block with an input in a visual programming language. Keep the function small enough to explain. Here the same display behavior handles two different exhibits; if the label format changes, edit one function body.

function	A named block you can call.
parameter	A name receiving an input when a function is called.
argument	The actual value passed to a function.

Predict before you run

If you remove both calls but keep the definition, what is printed?

▼ **Compare your prediction**

Nothing. Defining a function does not call it.

2. Try it, then change it

1. Find the definition and two calls, then run.
2. Change the second argument to Paper rover. Notice that only the second exhibit name changes.
3. Change the instruction inside the function. Confirm both calls use the new instruction.

Runnable Python example

```
def show_label(label):
    print("Exhibit:", label)
    print("Please observe carefully")

show_label("Moon rock model")
show_label("Tiny robot")
```

▼ Expected output for the original example

```
Exhibit: Moon rock model
Please observe carefully
Exhibit: Tiny robot
Please observe carefully
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

A function call needs parentheses. show_label by itself refers to the function but does not run it. Missing an argument produces a TypeError.

3. Build a mini-challenge

Define announce(station) to print “Visit station” and its input. Call it inside a loop for station numbers 1 through 3.

Check your result

- Define the function before calling it.
- Pass number + 1 as the argument.
- Produce exactly three station messages.

▼ Worked solution · try your own first

```
def announce(station):
    print("Visit station", station)

for number in range(3):
    announce(number + 1)
```

Solution output

```
Visit station 1
Visit station 2
Visit station 3
```

4. Explain it back

What is the parameter in def announce(station)?

▼ **Compare your explanation**

station is the parameter. In announce(2), the number 2 is the argument supplied for that call.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · search and stop

Scan supplied readings, stop at a match, and handle no match.

Bring this idea with you

A loop repeats. A condition selects. They can work together.

1. Understand the idea

A list stores several readings in order. for brightness in readings examines them one at a time. If a reading reaches our target, break stops the loop. Later readings will not be checked. The flag found remembers whether a match occurred.

A real sensor might never produce the value you hoped for. Always plan what happens if the search fails. This exercise consumes a finite list; real hardware needs an adult-designed timeout and stop policy. Avoid an unbounded while True loop.

list	An ordered collection written with square brackets.
break	A statement that exits the nearest loop immediately.
not	An operator that reverses a Boolean.

Predict before you run

Will the program print Checked: 80?

▼ Compare your prediction

No. The reading 60 satisfies the condition and break ends the loop.

2. Try it, then change it

1. Trace which list items are visited, then run.
2. Test [10, 20, 30]. Confirm the no-target message appears.
3. Test an empty list [] and then [60]. Explain why both finish without waiting forever.

Runnable Python example

```
readings = [20, 35, 60, 80]
found = False
for brightness in readings:
    print("Checked:", brightness)
    if brightness >= 60:
        found = True
        print("Target found")
        break
if not found:
    print("No target; search ended")
```

▼ **Expected output for the original example**

```
Checked: 20
Checked: 35
Checked: 60
Target found
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Reset found to False before the loop. Put the no-target check after the loop; inside it, you would announce failure before checking all readings.

---

3. Build a mini-challenge

Search [15, 25, 35] for a brightness of at least 50. Print a fallback when no reading matches. Then test a match at the final reading.

Check your result

- The original list ends with a no-target message.
- A matching value stops the search.
- An empty list also reaches the fallback.

▼ **Worked solution · try your own first**

```
readings = [15, 25, 35]
found = False
for brightness in readings:
    if brightness >= 50:
        found = True
        print("Target found")
        break
if not found:
    print("No target; search ended")
```

Solution output

```
No target; search ended
```

4. Explain it back

Why test an empty list and a no-match list?

▼ **Compare your explanation**

They verify that the program finishes and reports missing evidence instead of assuming a successful search.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

3D + Python lab · rehearse a virtual robot plan

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Choose from several actions

Use if / elif / else to route a command to one known response.

Bring this idea with you

if / else chooses between two paths. == compares values.

1. Understand the idea

A visitor might type hello, help, or something unexpected. if / elif / else checks conditions in order and runs only the first matching branch. The final else catches inputs you did not recognize. It should explain what is allowed.

String methods transform text. .strip() removes surrounding whitespace, and .lower() makes letters lowercase. The transformed text is a new value, so store it. Normalization makes " HELLO " match "hello" without accepting arbitrary instructions.

elif	An additional condition checked if earlier conditions were false.
normalization	Putting input into a consistent form before comparing it.

Predict before you run

What happens for an empty string ""?

▼ Compare your prediction

Neither known command matches, so the unknown-command fallback runs.

2. Try it, then change it

1. Run the original command. Explain why spaces and capitals no longer matter.
2. Test help, dance, and an empty string separately. Write expected and actual results.
3. Keep the action choices as prepared text. A visitor's words must not become Python code or motor angles.

Runnable Python example

```
command = " HELLO "
command = command.strip().lower()
if command == "hello":
    print("Welcome, explorer")
elif command == "help":
    print("Try hello or help")
else:
    print("Unknown command; no action")
```

▼ **Expected output for the original example**

```
Welcome, explorer
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Several separate if statements can behave differently from one if / elif chain. Trace which else belongs to which if.

3. Build a mini-challenge

Add an info command that prints a short exhibit fact while preserving the fallback. Test INFO with surrounding spaces.

Check your result

- Normalize the input.
- Use an elif for info.
- Unknown input still produces no action.

▼ **Worked solution · try your own first**

```
command = " INFO "
command = command.strip().lower()
if command == "hello":
    print("Welcome, explorer")
elif command == "info":
    print("This exhibit is a model robot")
else:
    print("Unknown command; no action")
```

Solution output

```
This exhibit is a model robot
```

4. Explain it back

Why not treat an unknown word as a new action?

▼ **Compare your explanation**

The program has no tested meaning for it. A known fallback keeps its possible responses predictable.

Notebook: My prediction / change / observation / explanation / next test.

Check readiness and permission

Combine conditions and give a stop request priority.

Bring this idea with you

A gate belongs before the action it protects.

1. Understand the idea

A useful action may require both a prepared system and permission. ready and permission is True only when both are True. Try all four combinations to make sure one True value cannot open the gate on its own.

Some rules take priority. A stop request should be considered before an ordinary action. An if / elif chain expresses this order. This is a printed policy model; physical emergency stopping belongs to the adult-approved robot setup.

and	True only when both conditions are true.
or	True when at least one condition is true.
priority	The order in which competing rules are considered.

Predict before you run

If all three inputs are True, which message appears?

▼ **Compare your prediction**

Stop requested; no action. The first matching branch has priority.

2. Try it, then change it

1. Test the four ready / permission combinations with stop_requested False.
2. Set all three inputs True. Check that the stop branch wins.
3. On paper compare ready and permission with ready or permission. Explain why or is wrong for this gate.

Runnable Python example

```
stop_requested = False
ready = True
permission = False
if stop_requested:
    print("Stop requested; no action")
elif ready and permission:
    print("Show the prepared exhibit")
else:
    print("Wait for readiness and permission")
```

▼ **Expected output for the original example**

```
Wait for readiness and permission
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Do not replace and with or until you have written a truth table. Read the sentence aloud: “both required” means and; “either warning” means or.

3. Build a mini-challenge

Model a warning when either heat_warning or stuck_warning is True. Otherwise print that no warning is reported. This model does not detect real faults.

Check your result

- Use or because either warning matters.
- Test False/False and each warning alone.
- The warning asks for adult help without suggesting repair.

▼ **Worked solution · try your own first**

```
heat_warning = False
stuck_warning = True
if heat_warning or stuck_warning:
    print("Warning: stop and get an adult")
else:
    print("No warning reported in this model")
```

Solution output

```
Warning: stop and get an adult
```

4. Explain it back

Why test simultaneous stop and permission?

▼ **Compare your explanation**

It reveals priority. Each rule can work alone while their combination still chooses the wrong branch.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · build a decision test bench

Put a decision in a function and test several cases.

Bring this idea with you

Functions package a behavior; lists and loops let us test multiple inputs.

1. Understand the idea

A function can return an action name instead of printing immediately. This separates deciding from displaying the result. return ends the current function call. The caller can then print, compare, or record that returned value.

Keep a test set containing ordinary, unknown, and empty input. Run it again after a change. This is a simple regression habit: the new feature should work and the old cases should still behave correctly. These tests establish only the cases you ran.

return	Sends a value back to the caller and ends that function call.
regression test	A repeated check that a change did not break expected behavior.

Predict before you run

Does return "greet" print anything by itself?

▼ **Compare your prediction**

No. It gives the value to the caller. The outer print displays it.

2. Try it, then change it

1. Trace each command through the function and predict all four actions.
2. Add a HELP branch returning explain. Test HELP as well as the original inputs.
3. Write an expected-action column before rerunning. Look for any old case that changed accidentally.

Runnable Python example

```
def choose(command):
    command = command.strip().lower()
```

```
    if command == "hello":
        return "greet"
    elif command == "stop":
        return "stop"
    else:
        return "wait"

for command in ["hello", " STOP ", "dance", ""]:
    print("Action:", choose(command))
```

▼ **Expected output for the original example**

```
Action: greet
Action: stop
Action: wait
Action: wait
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

A function that reaches the end without return gives None. If you see None in your output, check every possible path.

---

3. Build a mini-challenge

Write choose(command) for hello → greet, info → explain, and everything else → wait.
Test all three paths.

Check your result

- Return action names instead of executing actions.
- Normalize text.
- The unknown command reaches wait.

▼ **Worked solution · try your own first**

```
def choose(command):
    command = command.strip().lower()
    if command == "hello":
        return "greet"
    elif command == "info":
        return "explain"
    else:
        return "wait"

for command in [" HELLO ", "info", "jump"]:
    print(choose(command))
```

Solution output

```
greet
explain
wait
```

4. Explain it back

Why return an action name rather than run arbitrary input?

▼ **Compare your explanation**

It separates the policy from the output and limits the result to known responses we can test.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 6 · compare with the prepared gesture policy

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Plan the helper before building

Write requirements, decompose the system, and test a decision function.

Bring this idea with you

You can use variables, comparisons, branches, loops, functions, and returned values.

1. Understand the idea

Your project is a museum desk helper using invented inputs: a command, permission, and brightness from 0 to 100. Required behavior: stop wins; otherwise refusal waits; hello greets; inspect asks for more light below 40 and describes a supplied model at 40 or above; unknown input waits.

Split the job into input examples, a decision function, and a display loop. First build the smallest part: deciding whether the model scene has enough light. We return one of two prepared messages. No image recognition, recording, or cloud request is needed.

interface	The inputs and outputs a part promises to use.
acceptance test	A concrete check that a requirement has been met.

Predict before you run

Which input checks the equality decision?

▼ **Compare your prediction**

40. It must return describe supplied model under the stated requirement.

2. Try it, then change it

1. Draw inputs → decision function → printed action. Label every input's type and scale.
2. Write expected actions for brightness 39, 40, and 41, then test them.
3. Make a paper table for stop, refusal, greeting, dim inspection, bright inspection, and unknown command. The next lesson combines them.

Runnable Python example

```
def inspect_scene(brightness):
    if brightness < 40:
        return "add light"
    else:
        return "describe supplied model"

print(inspect_scene(20))
print(inspect_scene(40))
```

▼ **Expected output for the original example**

```
add light
describe supplied model
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Check invalid values before classifying them. Otherwise -1 is accepted as ordinary darkness instead of an out-of-range clue.

---

3. Build a mini-challenge

Expand the scene helper to reject readings outside 0–100 before classifying dim or bright. Assume the input is a number.

Check your result

- -1 and 101 return check reading.
- 39 returns add light.
- 40 returns describe supplied model.

▼ **Worked solution · try your own first**

```
def inspect_scene(brightness):
    if brightness < 0 or brightness > 100:
        return "check reading"
    elif brightness < 40:
        return "add light"
    else:
        return "describe supplied model"

for brightness in [-1, 39, 40, 101]:
    print(inspect_scene(brightness))
```

Solution output

```
check reading
add light
describe supplied model
check reading
```

4. Explain it back

What makes these requirements testable?

▼ **Compare your explanation**

Each names an input condition and a specific expected output, including equality, refusal, and unknown input.

Notebook: My prediction / change / observation / explanation / next test.

Build the complete decision system

Combine priority, permission, and inspection into one function.

Bring this idea with you

return ends a function call. Earlier conditions can take priority.

1. Understand the idea

The decide function below has three parameters. Call arguments must match their order: command, permission, brightness. First normalize the command. Then check stop, refusal, hello, and inspect. The inspect branch contains a nested if for the light reading.

Readiness for this model means the inputs have the agreed types: command is text, permission is Boolean, and brightness is numeric. The inspect branch also rejects numbers outside the scale. This small model does not claim to validate every possible Python object or operate real hardware.

integration	Connecting parts and checking that they work together.
test case	A set of inputs and its expected result.

Predict before you run

Why can these separate if statements still return only one action?

▼ **Compare your prediction**

Each selected path returns immediately. That ends this function call before later conditions run.

2. Try it, then change it

1. Trace each example call using the required priority order.
2. Add calls for hello, unknown, inspect at 40, and inspect at 101. Predict all four.
3. Try " STOP " with permission False. Verify that normalization and stop priority work together.

Runnable Python example

```
def decide(command, permission, brightness):
    command = command.strip().lower()
    if command == "stop":
        return "stop"
    if not permission:
        return "wait for permission"
    if command == "hello":
        return "greet"
    if command == "inspect":
        if brightness < 0 or brightness > 100:
            return "check reading"
        if brightness < 40:
            return "add light"
        return "describe supplied model"
    return "unknown; wait"

print(decide("inspect", True, 25))
print(decide("inspect", False, 70))
print(decide("stop", False, 70))
```

▼ **Expected output for the original example**

```
add light
wait for permission
stop
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

If refusal still reaches inspection, check whether the permission branch returns or merely prints. Printing does not end the function.

---

3. Build a mini-challenge

Keep the full decision function. Replace its final three print calls with a four-case demonstration: greeting, dim inspection, bright inspection, and refusal.

Check your result

- The action sequence is greet, add light, describe supplied model, wait for permission.
- Use the same function for all cases.
- Explain why refusal prevents inspection.

▼ **Worked solution · try your own first**

```
def decide(command, permission, brightness):
    command = command.strip().lower()
```

```
    if command == "stop":
        return "stop"
    if not permission:
        return "wait for permission"
    if command == "hello":
        return "greet"
    if command == "inspect":
        if brightness < 0 or brightness > 100:
            return "check reading"
        if brightness < 40:
            return "add light"
        return "describe supplied model"
    return "unknown; wait"

print(decide("hello", True, 70))
print(decide("inspect", True, 20))
print(decide("inspect", True, 40))
print(decide("inspect", False, 70))
```

Solution output

```
greet
add light
describe supplied model
wait for permission
```

---

4. Explain it back

Which part of this program understands an image?

▼ **Compare your explanation**

None. Brightness is supplied numeric data, and describe supplied model is a prepared action name. Image understanding would be another system with different tests.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · test, improve, explain

Run an acceptance suite, repair a defect, and present evidence.

Bring this idea with you

Test ordinary and unexpected inputs, and compare actual with expected behavior.

1. Understand the idea

A test case can bundle command, permission, brightness, and expected output in parentheses. The loop unpacks these four items into four names. actual == expected prints True for agreement. False is a useful clue, not something to hide.

Our final suite checks stop priority, refusal, a greeting, an unknown input, both sides of the threshold, equality, and invalid readings. Present what passed and one limitation: this is a deterministic model using supplied data, not proof of real sensing, speech, motion, or every possible input.

tuple	An ordered group of values; here it bundles one test case.
unpacking	Assigning a group's values to separate names.
rubric	Explicit criteria for judging a project.

Predict before you run

Which test fails if you replace brightness < 40 with brightness <= 40?

▼ **Compare your prediction**

The equality test at 40 fails. Its actual result becomes add light instead of describe supplied model.

2. Try it, then change it

1. Run the suite and read each result; do not count output lines as passes without checking True.
2. Introduce the <= defect, identify its failing test, then repair it and rerun.
3. Present your diagram, one failed-and-fixed test, and a limitation. Invite a partner to suggest an extra case, or add an empty command yourself.

Runnable Python example

```
def decide(command, permission, brightness):
    command = command.strip().lower()
    if command == "stop":
        return "stop"
    if not permission:
        return "wait for permission"
    if command == "hello":
        return "greet"
    if command == "inspect":
        if brightness < 0 or brightness > 100:
            return "check reading"
        if brightness < 40:
            return "add light"
        return "describe supplied model"
    return "unknown; wait"

cases = [
    (" STOP ", False, 70, "stop"),
    ("inspect", False, 70, "wait for permission"),
    ("hello", True, 70, "greet"),
    ("dance", True, 70, "unknown; wait"),
    ("inspect", True, 39, "add light"),
    ("inspect", True, 40, "describe supplied model"),
    ("inspect", True, 41, "describe supplied model"),
    ("inspect", True, -1, "check reading"),
    ("inspect", True, 101, "check reading"),
]
for command, permission, brightness, expected in cases:
    actual = decide(command, permission, brightness)
    print(actual == expected, command, brightness, actual)
```

▼ Expected output for the original example

```
True  STOP  70 stop
True inspect 70 wait for permission
True hello 70 greet
True dance 70 unknown; wait
True inspect 39 add light
True inspect 40 describe supplied model
True inspect 41 describe supplied model
True inspect -1 check reading
True inspect 101 check reading
```

Your edited program may have different output. Predict that result separately.

▼ Stuck? Read a debugging clue

When a test fails, compare the input, actual output, and expected output. Change the program only if the expectation still matches your written requirement.

3. Build a mini-challenge

Write a small boundary test for the brightness decision. Compare actual messages with expected messages for 39, 40, and 41, then explain the repair you made in the full helper.

Check your result

- All three comparisons report True after repair.
- Describe why the equality defect happened.
- Show requirements, code, test evidence, and one limitation in your project log.

▼ Worked solution · try your own first

```
def light_action(brightness):
    if brightness < 40:
        return "add light"
    return "describe supplied model"

for brightness, expected in [(39, "add light"), (40, "describe supplied model"), (41, "describe supplied
model")]:
    print(brightness, light_action(brightness) == expected)
```

Solution output

```
39 True
40 True
41 True
```

4. Explain it back

What counts as a strong capstone demonstration?

▼ Compare your explanation

A clear system diagram, working required branches, evidence from the acceptance cases, one explained repair, and an honest limitation. Use the rubric in the teaching guide.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 8 • extend into the prepared robot-friend project

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

Use a fresh reading each time

Explain the difference between a one-time and a repeated decision.

Bring this idea with you

A loop can visit each supplied reading and apply an if / else rule.

1. Understand the idea

A one-time decision answers “what should happen now?” Repeated decisions recheck as conditions change. For each reading, the loop should use that new value. Reusing the first reading inside the loop would make later choices ignore the changing scene.

Real continuous control samples repeatedly over time; it is not infinitely fast. Our finite list compresses a sequence of moments into a trace. It models decision changes, not a live camera stream or a controller connected to motors.

feedback	Using a measured result to inform the next decision.
stale data	An old reading used as though it described the current situation.

Predict before you run

Would checking only the first reading describe the whole sequence?

▼ **Compare your prediction**

No. Later readings cross the threshold, so they need new decisions.

2. Try it, then change it

1. Make a table with one row per reading and predict its branch.
2. Run and compare every row.
3. Change the readings to [50, 50, 30]. Explain why repeated values can legitimately produce repeated actions.

Runnable Python example

```
readings = [30, 50, 35, 60]
for brightness in readings:
```

```
    if brightness < 40:
        print(brightness, "suggest more light")
    else:
        print(brightness, "ready to observe")
print("End of supplied readings")
```

▼ **Expected output for the original example**

```
30 suggest more light
50 ready to observe
35 suggest more light
60 ready to observe
End of supplied readings
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

If every output is identical unexpectedly, check whether you compare a fixed number instead of brightness from the current iteration.

---

3. Build a mini-challenge

Trace the sequence [55, 20, 70] with a threshold of 50, printing dim or ready for each reading.

Check your result

- Use the loop's current value.
- Make exactly three decisions.
- Finish when the list ends.

▼ **Worked solution · try your own first**

```
for brightness in [55, 20, 70]:
    if brightness < 50:
        print("dim")
    else:
        print("ready")
```

Solution output

```
ready
dim
ready
```

---

4. Explain it back

Why is this trace not a live feedback controller?

▼ **Compare your explanation**

It uses a fixed list, with no real sensor, timed sampling, actuator, or measurement of an action's effects. It teaches repeated decisions.

Notebook: My prediction / change / observation / explanation / next test.

Keep a noisy signal from flickering

Use remembered state and two thresholds to stabilize a decision.

Bring this idea with you

One threshold can make a choice switch whenever readings cross it.

1. Understand the idea

With a single cutoff at 40, readings 39, 41, 39, 41 repeatedly change the decision. If small changes are just noise, this can make messages flicker. We can keep a state called lamp_on and choose separate boundaries.

Our modeled lamp turns on below 35 and off above 45. Between those boundaries, including 35 and 45, it keeps its previous state. This is hysteresis. It trades some responsiveness for stability; thresholds should match the intended use.

state	A remembered condition used by later decisions.
hysteresis	Using different thresholds for switching on and switching off.

Predict before you run

Why does 40 keep the lamp off after the reading 46?

▼ **Compare your prediction**

It is inside the middle band, so neither assignment runs. The previous False state remains.

2. Try it, then change it

1. Trace the state after every reading. Circle the rows where it changes.
2. Test [34, 35, 45, 46]. Explain both equality cases.
3. Compare with a single threshold at 40 on paper. Which sequence causes unnecessary switching?

Runnable Python example

```
lamp_on = False
for brightness in [30, 39, 41, 46, 40, 34]:
```

```
    if brightness < 35:
        lamp_on = True
    elif brightness > 45:
        lamp_on = False
    print(brightness, "lamp on:", lamp_on)
```

▼ **Expected output for the original example**

```
30 lamp on: True
39 lamp on: True
41 lamp on: True
46 lamp on: False
40 lamp on: False
34 lamp on: True
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Do not put lamp_on = False inside the loop before the conditions. That would erase the memory needed in the middle band.

---

3. Build a mini-challenge

Use on-below-30 and off-above-50 thresholds for readings [25, 40, 55, 40]. Begin with the lamp off.

Check your result

- States are True, True, False, False.
- The middle-band reading remembers the previous state.
- The code models a lamp without wiring or controlling one.

▼ **Worked solution · try your own first**

```
lamp_on = False
for brightness in [25, 40, 55, 40]:
    if brightness < 30:
        lamp_on = True
    elif brightness > 50:
        lamp_on = False
    print(lamp_on)
```

Solution output

```
True
True
False
False
```

4. Explain it back

What does a wider middle band change?

▼ **Compare your explanation**

It reduces switching but may delay a useful response. The best band depends on the signal and the task.

Notebook: My prediction / change / observation / explanation / next test.

Challenge · explore a feedback rule

Calculate signed error and a bounded correction, then explain the model's limits.

Bring this idea with you

Feedback compares a target with a measurement. Functions can return a calculated value.

1. Understand the idea

Imagine keeping a marker at position 50 on a screen. If it is at 30, error = 50 - 30 = 20. A positive correction points toward larger screen coordinates. At position 70, the error is negative, so correction points the other way.

Our supplied rule is correction = gain × error. min(10, correction) keeps the smaller value, capping the positive end. max(-10, that result) keeps the larger value, capping the negative end. Together they limit it to -10 through 10 screen units per model step. With gain 0.5, the correction is half the error until the limit applies. You only need arithmetic to use this rule; deriving controller mathematics is a later topic. This is not a Reachy motor controller.

error	Target minus measured value.
gain	A multiplier controlling the size of a correction.
clamp	Limit a value to a minimum and maximum.

Predict before you run

Why does error 30 give correction 10 instead of 15?

▼ **Compare your prediction**

The gain gives 15, but the upper clamp limits the correction to 10.

2. Try it, then change it

1. Calculate error and correction for each supplied measurement, then run.
2. Try gain 0.25 and compare the corrections. Smaller gain reduces the requested correction for the same error.

3. For the challenge, use an ideal screen model where position += correction means add the correction to position. Trace four steps and compare gains 0.5 and 1.0. Real mechanisms may have lag or noise that this model omits.

Runnable Python example

```
target = 50
gain = 0.5
for measured in [20, 40, 50, 60, 80]:
    error = target - measured
    correction = max(-10, min(10, gain * error))
    print("Measured:", measured, "error:", error, "correction:", correction)
```

▼ **Expected output for the original example**

```
Measured: 20 error: 30 correction: 10
Measured: 40 error: 10 correction: 5.0
Measured: 50 error: 0 correction: 0.0
Measured: 60 error: -10 correction: -5.0
Measured: 80 error: -30 correction: -10
```

Your edited program may have different output. Predict that result separately.

▼ **Stuck? Read a debugging clue**

Reversing target - measured reverses the correction direction. If the ideal position moves away from the target, check the sign first.

---

3. Build a mini-challenge

Complete four steps of an ideal screen-position model starting at 20, targeting 50, with gain 0.5 and corrections limited to ±10. Explain the change in error.

Check your result

- Positions become 30, 40, 45.0, and 47.5.
- Corrections never exceed the limits.
- State why the result does not prove safe or stable physical robot control.

▼ **Worked solution · try your own first**

```
position = 20
target = 50
gain = 0.5
for step in range(4):
    error = target - position
    correction = max(-10, min(10, gain * error))
```

```
    position += correction
    print("Position:", position)
```

Solution output

```
Position: 30
Position: 40
Position: 45.0
Position: 47.5
```

4. Explain it back

What can you now explain about a thoughtful robot?

▼ **Compare your explanation**

It follows a plan, interprets inputs using explicit rules, handles missing permission and unknown commands, repeats with a stopping condition, and uses new measurements to improve its next decision. My model evidence does not establish physical robot behavior.

Notebook: My prediction / change / observation / explanation / next test.

TAKE THE IDEA FURTHER

Reachy Mission 2 • compare with the servo feedback model

Optional connection. Continue this coding path without hardware, or follow the linked lesson's setup and safety steps.

FOR LEARNERS & GROWN-UPS

Teaching guide & course notes

Setup and pacing

An adult runs npm run build, then npm run serve from the academy folder and opens the printed local address. Open Coding course. Python loads from the bundled course files; no account or API key is needed. The browser must support WebAssembly and workers. If Python cannot start, trace the supplied code and expected output on paper, then ask an adult to check the local server.

Use one 25-minute lesson per session: 27 sessions, about 11¼ hours at this pace (9–13½ hours at 20–30 minutes each). At three sessions per week, allow nine weeks. Unit 8 may use extra sessions for an original project. Take breaks; unfinished challenges can continue next time.

How to teach or learn independently

Read the outcome, recall the previous skill, predict, then run. The guided change is practice; solve the mini-challenge before opening its worked solution. The last lesson in each unit brings its skills together. A learner can use a paper trace instead of running Python, but should label that evidence accurately.

In pairs, one learner edits while the other predicts and checks; swap roles halfway through. If learning alone, read the expected result aloud before pressing Run. For support, change one value and trace each line. For extension, add a new test case and justify it rather than adding unbounded hardware actions.

Evidence and completion

Each lesson has a prediction answer, expected Python output, a mini-challenge with acceptance checks and a worked solution, and an explain-it-back answer. Notes and code drafts save in this browser when storage is available. Download your notebook for a portable copy. Browser storage is not an account backup.

A lesson checkmark means the learner reviewed their evidence. It is not an automatic assessment of mastery. Before continuing, ask the learner to explain one decision without reading the answer. Revisit a prerequisite if that explanation is unclear. No academy progress is converted to knowledge-atlas mastery.

Capstone rubric

Score each criterion 0 (not yet demonstrated), 1 (demonstrated with help), or 2 (independently explained and demonstrated): clear requirements and input/output diagram; correct priority and permission handling; correct threshold and invalid-reading behavior; meaningful ordinary, boundary, and fallback tests; an explained improvement with honest limitations. Maximum 10. Use the scores to choose the next practice, not as a certification.

For readiness, require evidence for every criterion and revisit any zero. A learner can present a notebook, code, and actual test output. A scripted model is a valid capstone for this course; physical motion, live sensing, speech, and cloud services are optional extensions with separate setup and checks.

Connect the academy courses

Recommended sequence: complete this coding course, then use Reachy Missions 0–8 to apply the ideas to the prepared robot. Unit links offer optional earlier connections. Spark Lab is a parallel electronics and physics course; the 3D + Python lab is a practice space. The knowledge atlas is a wider curriculum reference whose existing-source audit does not yet include this new coding course.

The CMU reference uses VEX hardware and visual blocks. Here console sequences replace a robot-brain display, Reachy pose storyboards replace a drivetrain, supplied Boolean/numeric data replaces VEX sensors, Python functions replace reusable blocks, a museum-helper capstone combines the skills, and bounded screen models teach repeated decisions. No VEX driving, claw, or line-sensor capability is implied for Reachy.

Safe transfers and troubleshooting

Browser programs print plans and compute with supplied data. For real hardware, use only the existing Reachy lesson's prepared code after adult setup. Stop for unusual noise, heat, red lights, or binding; do not force joints or repair wiring. Camera, microphone, and cloud lessons retain their own consent gates. Never paste passwords, keys, private images, or personal recordings into exercises or notes.

SyntaxError: check quotes, parentheses, and colons. NameError: check spelling and assignment order. IndentationError: use four consistent spaces per block. TypeError: check the input type and function arguments. Wrong output with no error: trace the chosen branch, loop count, units, and boundary case. A computation that does not finish can be stopped; the browser runner also enforces a time limit.

Reference & adaptation

An independent Reachy Mini course informed by the public CMU / CS2N VEX IQ (2nd Gen) curriculum structure, reviewed September 13, 2026. The nine-unit sequence is mapped below. Explanations, Python programs, desk-based challenges, and assessments here are original. We reviewed the public overview and outline, not sign-in-only lesson media or assessments. This is not a CMU or VEX course, certification, or endorsement.

CMU Robotics Academy · curriculum overview

CS2N · public unit and lesson outline

Public CMU unit structure → this academy course

Reference topic	Reachy / Python unit
Getting Started	1. Meet your robot team
Programming the Brain	2. Write a clear program
Robot Movement	3. Plan careful movement
Digital Sensors	4. Ask yes-or-no questions
Analog Sensors	5. Make sense of measurements
Loops	6. Repeat with a purpose
Discrete Decisions	7. Make dependable decisions
Capstone: Subterranean Challenge	8. Build a museum helper
Continuous Decisions	9. Keep checking and adapting