3006ICT Robotics and Computer Vision

PRACTICAL PRACTICE — PART 1

Webots Practical Guide

Webots R2025a • Python • e-puck

This is the practical Webots guide you should use before starting this week's workshop and the group project.

It focuses on the Webots features and operations you will need: opening worlds, understanding the interface, connecting Python, inspecting devices, running and resetting simulations, reading sensors, controlling motors, viewing camera data, using project files, and debugging controllers.

1. Webots

Webots is a robotics simulator. It provides a virtual robot, a simulated environment, sensors and actuators, and a controller program that interacts with them. For this course, you write the controller in Python. Webots R2025a supports Python controllers, and its GUI provides a 3D view, Scene Tree, text editor and Console.

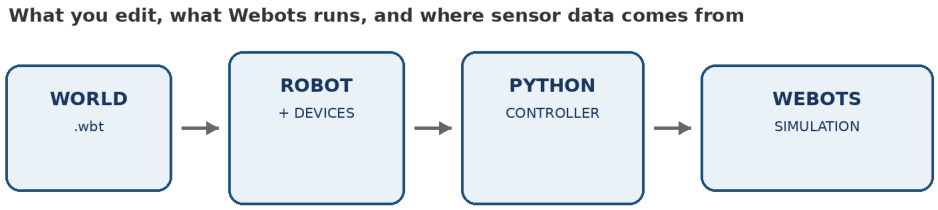

Figure 1. The relationship between the Webots world, robot devices and your Python controller.

You see	What it is	What you normally do with it
World (.wbt)	The simulation environment: robot, obstacles, stations, targets and other objects.	Open the supplied world; inspect it; normally do not redesign the assessment environment.
Robot	The simulated e-puck and its hardware devices.	Select/inspect devices and control them through Python.
Controller	Your Python program.	Read sensors → process data → decide → command motors.
Scene Tree	Hierarchical description of the world and its nodes.	Inspect objects, devices and fields; understand how the world is structured.
3D View	Visual view of the simulation.	Observe motion, geometry, collisions and overall behaviour.
Console	Controller output and Webots messages.	Read errors, print sensor values and debug behaviour.

Webots does not replace the robot controller. Webots supplies the simulated world and hardware interfaces; your controller supplies the intelligence. If the robot behaves incorrectly, ask first: is the problem in the world/device setup, the sensor data, the decision logic, or the actuator command?

2. First-time setup

2.1 Install and launch Webots

1. Install Webots R2025a, the version specified for this week's workshop and group project.
2. Launch Webots once and confirm that the application opens.
3. Keep the supplied course materials in their original folder structure.
4. Do not begin changing the world or controller until the supplied example runs.

5. A working baseline gives you a reference point. If you modify several things before the first successful run, later errors become much harder to diagnose.

2.2 Connect Webots to your Conda Python environment

This week's workshop uses the same Conda environment used in earlier labs. First verify that the required packages are available.

Run in your terminal.

```
conda activate <environment_name>

python -c "import cv2, numpy, matplotlib; print('Python environment OK')"

python -c "import sys; print(sys.executable)"
```

Copy the complete path printed by sys.executable. In Webots, configure the Python command in Preferences → General. On Windows/Linux the workshop uses Tools → Preferences → General → Python command; on macOS use Webots → Preferences → General → Python command. Restart Webots afterwards.

If you see a Python/package error: Do not immediately change your code. First check which Python executable Webots is using. A controller that works in your terminal but fails in Webots often indicates an environment mismatch.

2.3 Verify the baseline world

1. Open the supplied world: week8_lab_materials/worlds/week8_navigation.wbt.
2. Run the simulation.
3. Confirm that the starter controller starts without device-name errors.
4. Check the Console output: the starter controller should report, e.g., the simulation timestep, camera resolution and ps0-ps7.
5. Identify the e-puck, wooden-box obstacles and red target in the 3D View.

Do not continue until the baseline world runs correctly. Your first goal is not to solve navigation; it is to establish a known-good Webots setup.

3. Learn the Webots interface

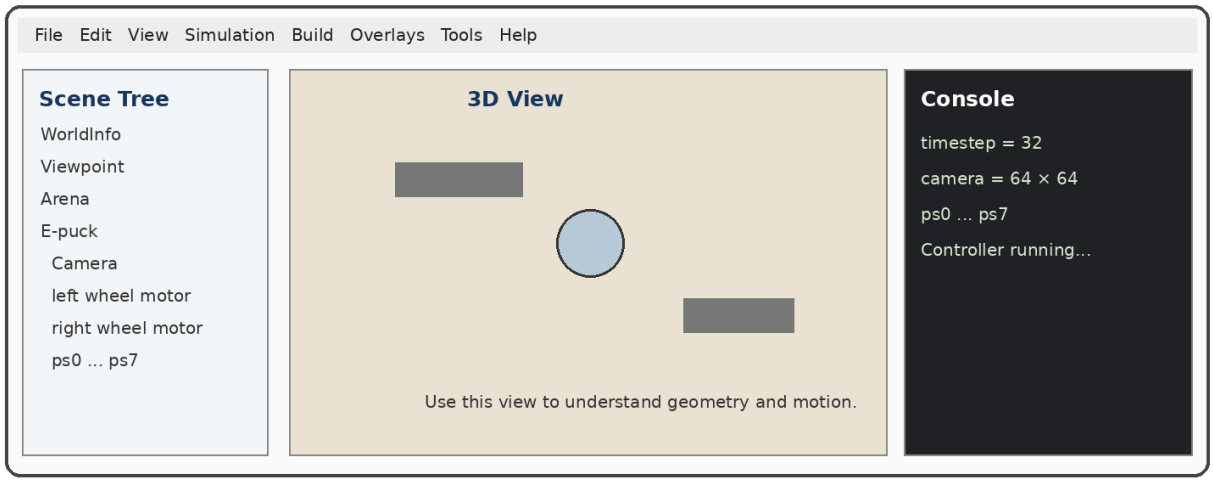

Figure 2. The four areas you will use most: Scene Tree, 3D View, controller/editor and Console.

3.1 3D View

Use the 3D View to understand what the robot is physically doing. Do not rely only on printed numbers.

Action	Use
Left mouse + drag	Rotate the viewpoint.
Right mouse + drag	Move/pan the viewpoint.
Ctrl + mouse wheel	Zoom in/out.
View → Change View → Top View	Useful for seeing the arena layout and navigation geometry.

Engineering use: When the robot fails, first look at the world from above. Ask whether the observed trajectory matches the controller's intended behaviour.

3.2 Scene Tree

The Scene Tree is the hierarchical representation of the current Webots world. It is especially useful when you need to understand what objects and devices exist. Webots documents the Scene Tree as the place where the nodes and fields describing the simulated world are represented.

- Expand the e-puck node and inspect its devices.
- Use exact device names from the world when calling robot.getDevice(...).
- Select an object when you need to understand its position, orientation or other fields.
- Use the Scene Tree for inspection first; do not casually edit supplied assessment worlds.

3.3 Console

The Console is your first debugging tool. Use print() deliberately to expose what the controller is seeing and deciding.

Useful development logging.

```
print("timestep:", timestep)
print("camera:", camera.getWidth(), "x", camera.getHeight())
print("ps0:", ps[0].getValue())
print("target:", target_found)
print("action:", action)
```

Good debugging output: Print information that answers a question. Avoid printing thousands of values every simulation step unless you are deliberately inspecting a signal.

4. Running, pausing, resetting and inspecting simulations

4.1 Run → observe → pause → inspect

For development, avoid letting a faulty controller run indefinitely. A productive cycle is:

1. Run the simulation for a short period.
2. Observe the robot and Console.
3. Pause when the interesting behaviour occurs.
4. Inspect sensor values or intermediate results.
5. Change one thing.
6. Reset and repeat the same test.

This is much faster than repeatedly running the entire mission without knowing why it failed.

4.2 Reset versus restart

Operation	Purpose	When to use
Pause	Temporarily stop simulation time.	Inspect behaviour or sensor output.
Reset/restart simulation	Return the simulation to its initial state.	Repeat a controlled experiment from the same starting condition.
Restart after code/environment change	Start a fresh controller run.	After changing controller code or Python configuration.
Assessment run	Official autonomous run under the project rules.	Do not treat it like a development experiment.

Important for the group project: The assessment rules are stricter than normal development. Once an assessment mission starts, manual steering, keyboard driving, pausing, restarting/resetting and mission-specific controller/configuration edits are not permitted except where the instructor authorises a genuine technical restart.

5. Understand the Webots project files

A Webots project normally separates worlds from controllers. The supplied group-project package adds maps, configuration and target resources. Webots' standard project hierarchy includes a worlds directory and a controllers directory; the controller associated with a Robot node is found from the controller name.

Conceptual project structure.

```
3006ICT_group_project_webots/
├── worlds/
│   ├── training_start_A.wbt
│   ├── ...
├── controllers/
│   └── <controller_name>/
├── maps/
│   ├── occupancy_grid.npy
│   └── occupancy_grid.csv
├── config/
│   ├── project_config.json
│   └── assessment_mission.json
└── targets/
    ├── target_reference.png
    └── ...
```

Keep the supplied folder structure unchanged where possible. The group-project specification requires the submitted solution to be reproducible using reasonable relative paths rather than machine-specific absolute paths.

What students should / should not edit

File/resource	Development use	Assessment caution
.wbt world	Open and run supplied worlds; make copies for development if needed.	Do not modify official assessment geometry/assets.
controller.py	Main place where you develop your solution.	Must work unchanged across missions.
maps/	Read the supplied occupancy grid for navigation if useful.	Do not replace it with hidden ground truth.
project_config.json	Read supplied station/position information where appropriate.	Do not use metadata to infer target-to-station assignment.
assessment_mission.json	Controller reads the required target identity.	Do not edit it after the assessment mission is revealed.
targets/	Use supplied reference images for visual target development.	The correct station must still be determined from RGB camera observations.

6. The Python controller

A Webots controller is a program that repeatedly interacts with simulated devices. The most important pattern is not a particular API call; it is the control loop.

Minimal controller structure.

```
from controller import Robot

robot = Robot()
timestep = int(robot.getBasicTimeStep())

while robot.step(timestep) != -1:
    # Read sensors
    # Process observations
    # Decide
    # Command actuators
    pass
```

Why robot.step() matters: The controller needs to advance with the Webots simulation. Put sensor reading, processing and actuator decisions inside the repeated loop so the robot responds to changing observations.

Get a device

```
camera = robot.getDevice("camera")
left_motor = robot.getDevice("left wheel motor")
right_motor = robot.getDevice("right wheel motor")
ps0 = robot.getDevice("ps0")
```

Device names are exact. If the name in the world is different from the string in getDevice(), your controller will not access the intended device.

7. Motors: make the e-puck move safely

The e-puck uses two independently driven wheels. This is differential-drive control. This week's workshop asks you to configure the motors for velocity control and create a reusable set_speed(left, right) function.

Basic motor setup.

```
left_motor = robot.getDevice("left wheel motor")
right_motor = robot.getDevice("right wheel motor")

left_motor.setPosition(float("inf"))
right_motor.setPosition(float("inf"))

def set_speed(left, right):
    left_motor.setVelocity(left)
    right_motor.setVelocity(right)
```

Command relationship	Expected behaviour
left = right > 0	Move approximately straight.
left < right	Turn left.

left > right	Turn right.
left < 0, right > 0	Rotate approximately in place.
left = right = 0	Stop.

Always have STOP: A robot controller should always have a deliberate stop behaviour. It is useful during normal operation, target loss, obstacle emergencies and debugging.

8. Camera: read images and turn them into useful information

This week's workshop uses the e-puck RGB camera. The immediate task is to detect a red target, but the same camera interface becomes central to the group project, where the RGB camera should identify which observation station contains the required target.

8.1 Enable and inspect the camera

Core camera access.

```
camera = robot.getDevice("camera")
camera.enable(timestep)

width = camera.getWidth()
height = camera.getHeight()

image = camera.getImage()
```

First inspect the image. Do not immediately build a complicated detector. Confirm that the buffer is valid, the dimensions are what you expect, and the image conversion is correct.

8.2 Camera data → OpenCV

This week's workshop uses the camera buffer and converts it into an OpenCV image before HSV processing. The important engineering sequence is:

```
Webots camera buffer
        ↓
OpenCV image
        ↓
Pre-processing
        ↓
Detection / segmentation
        ↓
Useful measurement
        ↓
Decision
```

For the workshop, the useful measurement is the target centroid (u_c, v_c).

For the group project, your useful measurement may be a target match, detection score, image location, or another representation chosen by your group.

8.3 HSV segmentation

Workshop red-target segmentation.

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

mask1 = cv2.inRange(hsv, (0, 100, 80), (10, 255, 255))
mask2 = cv2.inRange(hsv, (170, 100, 80), (180, 255, 255))
mask = mask1 | mask2
```

Stage	Question
Raw image	Can I actually see the target?
HSV conversion	Is the image represented in a useful colour space?
Mask	Which pixels are being treated as target pixels?
Centroid	Where is the detected target in the image?
Decision	What should the robot do with that information?

Important limitation: Colour segmentation is deliberately simple for the workshop. It can fail when appearance, lighting or background changes. The group project is more demanding: it contains realistic target images and non-target distractors, so your group should design a more appropriate visual identification method.

9. Proximity sensors: local safety

The official e-puck in this week's world has ps0-ps7. The workshop uses ps0-ps2 as the right/front-right group and ps5-ps7 as the left/front-left group. The supplied workshop notes that the IR reading becomes larger when an obstacle is very close.

Read all eight proximity sensors.

```
ps = [robot.getDevice(f"ps{i}") for i in range(8)]

for sensor in ps:
    sensor.enable(timestep)

values = [sensor.getValue() for sensor in ps]
```

Do not guess the threshold. The workshop suggests starting around 80, but asks you to print actual readings and adjust the threshold experimentally.

Behaviour priority: SAFETY > TARGET FOLLOWING > SEARCH

This priority becomes important in the group project. A robot that has visually detected a target but is about to hit

B1-B5 or a station structure should avoid the collision first.

10. GPS and InertialUnit: useful project information

The group-project e-puck additionally provides GPS and an InertialUnit. The project states that these can be used for localisation and navigation. They do not reveal which observation station contains the requested target.

Device/resource	Use in project	Do not use it for
GPS	Current robot position; localisation and navigation.	Inferring which station contains the target.
InertialUnit	Robot orientation information.	Inferring target identity/location.
Occupancy grid	Map-based navigation and planning.	Replacing RGB-camera target identification.
Station coordinates	Navigating to a selected station's designated observation position.	Choosing the target station before visual identification.

Think in information boundaries: The project gives you some information directly and requires you to obtain other information visually. Keep those boundaries explicit in your design: the camera determines the target station; map/pose resources can then support navigation.

11. Top View and navigation debugging

For navigation, a top-down view is often more informative than the default perspective. The group-project instructions explicitly note that you may need View → Change View → Top View.

1. Switch to Top View.
2. Identify the robot start, barriers and observation stations.
3. Run the robot slowly during development.
4. Observe whether its actual trajectory matches the intended route.
5. If it deviates, inspect both the planned waypoint and the local steering command.

Global path ≠ motor command: An A* path is a sequence of grid cells or waypoints. The robot still needs a local controller to turn those waypoints into safe wheel-speed commands.

12. Webots workflow for the group project

A practical model, not a solution.

```
MISSION TARGET
      ↓
RGB CAMERA OBSERVATIONS
      ↓
TARGET / STATION IDENTIFICATION
      ↓
SELECT DESIGNATED OBSERVATION POSITION
      ↓
GLOBAL ROUTE (optional)
      ↓
LOCAL CONTROL + PROXIMITY SAFETY
      ↓
ARRIVE AND STOP
```

Phase	What you should verify
1. Start	Correct world, controller and mission input are loaded.
2. Observe	Camera provides usable visual evidence.
3. Identify	The station decision is based on RGB observations, not hidden simulator information.
4. Navigate	The selected observation position is used as the navigation goal.
5. Control	Wheel commands are responsive and stable.
6. Safety	Proximity sensing can override normal navigation.
7. Arrive	Robot stops at the required observation position.
8. Repeat	The same controller works across different starts and target/station arrangements.

Note: Project success requires the same submitted controller to operate across missions, with no hard-coded target-to-station mapping and no simulator ground-truth recognition.

Appendix

A practical Webots debugging method

Diagnose from the bottom up

Layer	Check
Webots setup	Does the correct world open? Does the controller start?
Python environment	Can Webots import the packages you need?
Device access	Does getDevice() return the intended device?
Raw sensor data	Are camera/proximity/GPS/IMU values sensible?
Intermediate result	Is the mask/detection/error/path correct?
Decision	Does the code choose the behaviour you intended?
Actuation	Are the wheel commands correct and within safe limits?
System behaviour	Does the robot actually do what the decision implies?

Typical failures

Symptom	First thing to check
Python import error	Webots Python command / environment.
Device not found	Exact device name in Scene Tree/world.
Robot does not move	Motor position mode, velocity command and control loop.
Robot turns the wrong way	Sign of image error and left/right wheel relationship.
Target not detected	Raw camera image → colour/appearance → mask/detection.
Target detector works but robot is unsafe	Proximity sensor readings and behaviour priority.
A* path crosses obstacles	Grid interpretation, neighbour expansion and obstacle rejection.
Works once but fails elsewhere	Test assumptions: start position, target identity, distractors and route.
Project uses absolute paths	Replace with paths relative to the project root.

The golden debugging rule: Do not change everything at once. Find the lowest layer that is wrong, fix it, and then test upward.

Webots + Python quick reference

Task	Typical code / action
Create robot	from controller import Robot
Get timestep	timestep = int(robot.getBasicTimeStep())
Main loop	while robot.step(timestep) != -1:
Get device	robot.getDevice("device_name")
Enable sensor	sensor.enable(timestep)
Read distance sensor	sensor.getValue()
Get camera size	camera.getWidth(), camera.getHeight()
Get camera image	camera.getImage()
Set wheel velocity	motor.setVelocity(value)
Velocity mode	motor.setPosition(float("inf"))
Print debug information	print(...)
Top view	View → Change View → Top View

Before you start the Week 8 workshop

☐ Webots R2025a opens successfully.
☐ My Conda environment contains cv2, NumPy and Matplotlib.
☐ I know the exact Python executable path.
☐ Webots is configured to use that Python executable.
☐ The supplied week8_navigation.wbt world opens.
☐ The starter controller runs without device-name errors.
☐ I can identify the e-puck, camera, wheel motors and ps0-ps7.
☐ I know how to rotate, pan, zoom and switch to Top View.
☐ I know where to read controller output in the Console.
☐ I understand the basic sense → process → decide → act loop.

Before you start the group project

☐ I can run a Webots Python controller independently.
☐ I can control the e-puck's two wheels.
☐ I can read and process camera images.
☐ I can read proximity sensors and implement a safety response.
☐ I understand how the occupancy grid can support planning.
☐ I understand that RGB-camera evidence must identify the target station.
☐ I understand the difference between global planning and local control.
☐ I know how to inspect the supplied project files without changing their intended structure.
☐ I can debug one layer at a time.

Further reference

Use the official Webots R2025a documentation when you need an API detail or a feature that is not covered here.

- Webots R2025a User Guide — Introduction
- Webots R2025a — Using Python
- Webots R2025a — User Interface
- Webots R2025a — Scene Tree
- Webots R2025a — Reference Manual