

FIRA Air Simulation League

Student Programming Guide — U14 & U19

Learn to Fly with Python, DJITelloPy and AprilTag

Student Edition • Version 1.1 • October 2026

Welcome! Your First Autonomous Drone

In this guide you will learn to control the virtual DJI Tello, just as you would control a physical DJI Tello. We start with takeoff, moving, turning and landing. Then we use the drone camera, recognize AprilTags, and put the movements together into a small autonomous race program. You do not need to write your own network code.

The competition organizer has confirmed that programs written for the real DJI Tello work with the virtual DJI Tello and produce the corresponding flight behavior. Therefore, we will use DJITelloPy, a popular Python library built on the official Tello command interface. For AprilTag recognition, we use the pupil-apriltags detector and OpenCV.

This is a programming lesson, not the competition rulebook. Always follow the official U14 and U19 rules for scoring, attempts, gate order, and landing.

1. Get Ready: What Do You Need?

You need Python 3, the official FIRA Air simulator, and these three Python packages:

CODE — Install libraries

```
python -m pip install djitellopy pupil-apriltags opencv-python  
END CODE
```

What are they?

- DJITelloPy — sends real Tello commands through simple Python methods such as `takeoff()` and `move_forward()`.
- pupil-apriltags — recognizes the black-and-white AprilTags attached to the gates.
- OpenCV (cv2) — handles images and helps draw shapes and labels on the camera view.

You can write your programs in VS Code, PyCharm, or another Python editor. The sample filenames are suggestions, not special filenames required by the simulator.

Your typical program looks like this:

1. Connect to the drone.
2. Take off.
3. Send movement and turning commands.
4. Watch flight information or the camera when needed.
5. Land and end the connection.

For U14, you can begin with a fixed sequence of commands. For U19, computer vision and automatic correction can make your code more adaptable.

2. Lesson One: Your First Flight

Create a Python file called `first_flight.py`. Copy the whole example.

CODE — `first_flight.py`

```
from djitellopy import Tello

drone = Tello()
drone.connect()

print("Battery:", drone.get_battery(), "%")

try:
    drone.takeoff()
    drone.move_forward(100)
    drone.land()
finally:
    drone.end()
END CODE
```

What should happen? The drone takes off, flies 100 centimetres forward and lands. You can see the simulation react to the same commands that a real Tello understands.

Understand each line:

- `Tello()` creates your Python drone controller.
- `connect()` starts communication with the drone.
- `get_battery()` reads the reported battery percentage.
- `takeoff()` starts the automatic takeoff.
- `move_forward(100)` tells the drone to move forward 100 cm — not 100 pixels.
- `land()` requests an automatic landing.
- `end()` closes the DJITelloPy session and handles cleanup.

Practice: Change 100 to 60. Run the program again and compare the distance.

IMPORTANT: Many basic Tello movement commands accept a distance of 20–500 cm. A value such as `move_forward(5)` is not a valid 5 cm movement. Do not fly beyond your course boundaries.

3. Lesson Two: Move in Every Direction

The DJI Tello moves in three dimensions. The values in these examples are in centimetres.

CODE — Movement examples

```
drone.move_forward(80)    # Forward 80 cm
drone.move_back(80)       # Backward 80 cm
drone.move_left(60)       # Left 60 cm
drone.move_right(60)      # Right 60 cm
drone.move_up(50)         # Climb 50 cm
drone.move_down(50)       # Descend 50 cm
```

END CODE

These lines are examples to put inside a connected, airborne Tello program. They are not a complete file by themselves.

The direction is based on where the drone is facing. If you rotate the drone, a later `move_forward()` command travels in its new forward direction.

Practice challenge: Take off, move right 60 cm, forward 80 cm, left 60 cm, then land. Draw the path on paper before flying.

4. Lesson Three: Turn and Change Speed

A yaw turn rotates the drone left or right while it stays approximately in place.

CODE — Turning and speed

```
drone.rotate_clockwise(90)
drone.move_forward(80)
drone.rotate_counter_clockwise(90)
drone.set_speed(30)
```

END CODE

`rotate_clockwise(90)` means a 90-degree clockwise turn, not a movement of 90 cm. `rotate_counter_clockwise(90)` turns the opposite way. `set_speed(30)` requests the default speed in centimetres per second for appropriate movement commands; it does not immediately move the drone.

The SDK commonly supports turning through 1–360 degrees and a speed setting in the documented range of 10–100 cm/s. Your program should still check the result of each command and the actual field clearance.

Practice challenge: Write a program that goes forward, turns right, goes forward again, turns left, and lands. Compare the final drone heading to the starting heading.

5. Lesson Four: Build a Complete Autonomous Route

A basic flight plan is simply a sequence of commands. U14 teams are allowed to use this method. It is useful for learning coordinates, distances, turns and altitude.

CODE — `simple_route.py`

```
from djitellopy import Tello

drone = Tello()
drone.connect()

try:
    drone.takeoff()
    drone.move_up(50)
    drone.move_forward(100)
    drone.rotate_clockwise(90)
    drone.move_forward(80)
    drone.rotate_counter_clockwise(90)
    drone.move_forward(80)
```

```

        drone.land()
    finally:
        drone.end()
END CODE

```

What should happen? The drone climbs, flies forward, makes a right turn, continues, turns back to its original heading, moves ahead and lands.

This is a PRACTICE ROUTE, not the official racecourse. You must measure and adjust your own commands for the actual gates and landing pad. Finishing the commands does not automatically mean your drone has passed every gate correctly.

Why can a programmed route fail? Turns and movements can produce small deviations. A command may also be refused if it is too short, too long, or unsafe. A more advanced program watches the camera or flight state and corrects those deviations.

Try it: Change one movement distance, then explain why the drone's final position changes. Keep a simple notebook of distances and turns you tested.

6. Lesson Five: Read Drone Information

The Tello can report useful flight values. DJITelloPy gives you simple methods for many of them.

CODE — Read flight data

```

from djitellopy import Tello

drone = Tello()
drone.connect()

try:
    print("Battery:", drone.get_battery(), "%")
    print("Height:", drone.get_height(), "cm")
    print("Yaw:", drone.get_yaw(), "degrees")
    print("Flight time:", drone.get_flight_time(), "s")
finally:
    drone.end()
END CODE

```

Height tells you roughly how high the drone reports it is. Yaw describes its heading. Battery and flight time help you understand the current flight session. Flight-state measurements are useful but not perfectly precise; do not assume they reveal the drone's hidden coordinates inside the simulator.

Practice: Run `get_height()` before and after an upward movement. Run `get_yaw()` before and after a turn. If a heading changes from $+179^\circ$ to -179° , that does not necessarily mean the drone suddenly made a large turn.

The simulator and its scoring system are separate: your program controls the aircraft, but the official referee decides whether a gate crossing or landing counts.

7. Lesson Six: Use the Tello Camera

The drone's front camera helps you see gates, markers and the landing area. DJITelloPy receives video frames for you; you do not need to decode UDP video packets manually.

CODE — camera.py

```
from djitellopy import Tello
import cv2

drone = Tello()
drone.connect()
drone.streamon()

try:
    camera = drone.get_frame_read()

    while True:
        frame = camera.frame
        if frame is None:
            continue

        cv2.imshow("Tello Camera", frame)

        if cv2.waitKey(1) & 0xFF == ord("q"):
            break
finally:
    drone.end()
    cv2.destroyAllWindows()
```

END CODE

What should happen? A window shows the front-camera image. Press Q to close the view.

streamon() enables the camera stream. get_frame_read() provides access to the latest received frame. frame contains an ordinary image that OpenCV can process.

In the example simulator screenshot, the Tello camera view is displayed at 640×480 pixels. Do not assume every future simulator version or camera setting will have that same resolution; a real program should read frame.shape.

Try it: Hold the drone still near a gate and observe the tag below the opening. Move closer and see how the apparent size of the gate changes.

A camera image gives you what the virtual drone can see, not a perfect map of the entire arena.

8. Lesson Seven: What Is an AprilTag?

An AprilTag is a small black-and-white square with a coded pattern. It is similar in purpose to a QR code, but is designed to help robots detect a marker and identify it reliably in a camera image.

The detector can find:

- The tag's ID — a number encoded in the black-and-white pattern.
- The four corners — where the square appears in the image.
- The center — the tag's position in image pixels.
- Optionally, an estimate of its 3D pose if camera calibration and physical tag size are known.

The IDs are useful because the drone can recognize a specific tag instead of confusing it with another patterned object. AprilTags can help identify gates or estimate how the drone is positioned relative to a marker.

Which AprilTag type is used here?

AprilTags have different families, such as tag16h5, tag25h9 and tag36h11. The physical FIRA Air rulebook recommends the AprilTag 36h11 family. For the examples in this guide, we therefore use tag36h11.

IMPORTANT: The markers in our simulator are fixed and provided by the organizers; students do not select, replace, upload or move them. The organizer should confirm that the installed markers are actually of the tag36h11 family, and publish the mapping from tag IDs to gates, before this document is treated as the final simulator-specific camera reference. The appearance of a black-and-white square alone is not enough to prove its family.

An AprilTag is not a Tello EDU Mission Pad. You do not need Mission Pad SDK commands such as `mon` or `mdirection` to detect the fixed tags; you detect them in the camera image using a Python library.

9. Lesson Eight: Detect an AprilTag with Python

For this lesson we use `pupil-apriltags`, a ready-made tag recognition library. OpenCV converts the camera image into grayscale and draws the result.

CODE — `find_apriltag.py`

```
from djitellopy import Tello
from pupil_apriltags import Detector
import cv2

drone = Tello()
detector = Detector(families="tag36h11")

drone.connect()
drone.streamon()

try:
    camera = drone.get_frame_read()

    while True:
        frame = camera.frame
        if frame is None:
            continue

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        tags = detector.detect(gray)

        for tag in tags:
            x, y = map(int, tag.center)
            print("AprilTag ID:", tag.tag_id)
            cv2.circle(frame, (x, y), 5, (0, 255, 0), -1)
            cv2.putText(
                frame, str(tag.tag_id), (x + 10, y),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7,
                (0, 255, 0), 2
            )

        cv2.imshow("AprilTag Detector", frame)

        if cv2.waitKey(1) & 0xFF == ord("q"):
            break

finally:
    drone.end()
```

```
cv2.destroyAllWindows()
END CODE
```

What should happen? When the camera sees a matching AprilTag, a green dot and its ID appear on the image. The printed ID comes from the tag, not from the gate number guessed by the program.

What is happening inside the code?

- `Detector(families="tag36h11")` creates a detector for that tag family.
- `cv2.cvtColor(..., COLOR_BGR2GRAY)` makes the image grayscale.
- `detector.detect(gray)` returns all detected markers in this frame.
- `tag.tag_id` gives the detected numeric ID.
- `tag.center` gives its location in the current image.

Practice: Move the camera left or right and observe how the tag center changes. Try viewing a tag from farther away and see whether detection becomes less reliable.

If tags are not found, check the actual tag family, camera sharpness, visible tag size and lighting. Do not assume a tag is missing simply because your detection code does not see it.

Remember: this example recognizes tags; it does not fly through gates or control the drone.

10. Lesson Nine: Use What the Camera Sees

Suppose the camera frame is 640 pixels wide and an AprilTag's center is at $x=160$. The image center is $x=320$; therefore the tag is to the left in the image.

You can calculate a simple normalized image error:

CODE — image_error.py

```
def horizontal_error(tag_x, image_width):
    center_x = image_width / 2
    return (tag_x - center_x) / center_x

print(horizontal_error(160, 640))    # -0.5
print(horizontal_error(320, 640))    #  0.0
print(horizontal_error(480, 640))    #  0.5
END CODE
```

A negative value means the tag is left of the image center; a positive value means it is right. This can help your program decide whether it should make a small correction.

BUT DO NOT FLY TOWARD THE APRILTAG CENTER. On the FIRA Air gates, the tag is mounted below the opening. If the drone aims at the tag's center, it may hit the lower gate frame or floor. Your program must aim through the actual OPEN SPACE of the gate, with enough clearance for the whole drone.

A beginner can use a measured sequence of movements to cross the gate. A more advanced team can combine colored-gate detection, AprilTag identification and small corrections from the camera.

Example logic, not executable code:

1. Identify the next gate.
2. Look for its AprilTag or colored frame.
3. Estimate the center of the open space above the tag.

4. Move toward the opening in small safe steps.
5. Cross the complete gate without touching the frame.
6. Continue to the next gate and eventually land.

The scorer decides whether each gate was crossed correctly. AprilTag detection by itself does NOT earn gate points.

11. Lesson Ten: Put Everything Together for the Race

U14 — TWO IDENTICAL LANES

Two virtual Tello drones can fly side by side, each on its own identical track. Each team controls only its own assigned drone. The U14 course is simpler than U19, but gate sizes and drone behavior match the physical race. Pre-programmed flight sequences are permitted and advanced image processing is optional.

U19 — THE FULL AUTONOMOUS COURSE

U19 uses the race layout based on the physical FIRA Air rules. Teams pass gates in order, avoid obstacles and complete the required landing. In rounds with several landing pads, the official target may be selected after takeoff. A camera-based strategy may help you respond to that selection, but you must use only information available through the permitted interface.

HOW TO WRITE YOUR FIRST RACE PROGRAM

Start with a successful takeoff and landing. Add the first gate's movements, test them, and then add the second gate. Continue until you have a complete route. Keep a list of all movements and compare the result with the drone camera and its height/yaw readings.

A well-organized program usually has a start section, a planned path or gate-search section, checks to handle unexpected results, and a landing section.

Do not assume that completing a Python script means completing the competition track. A collision, wrong gate order or landing on the wrong pad still counts as a mistake under the competition rules.

12. Five Quick Practice Challenges

Challenge 1 — Take off, fly forward 100 cm, and land.

Challenge 2 — Take off, change height by 50 cm, and return safely.

Challenge 3 — Turn 90 degrees, then fly forward; explain why the travel direction changed.

Challenge 4 — Display the Tello camera and print its current image size.

Challenge 5 — Detect a fixed AprilTag, print its ID, and mark its center on the camera view.

Advanced optional task — Build a program that adjusts its position in response to what the camera sees, while aiming at the gate opening rather than the tag itself.

13. Common Problems

The drone does not connect: Check that the official simulator is running and exposing the same Tello connection that a physical drone would. Close other programs trying to control the same drone.

The drone will not take off: Check that `connect()` succeeded and that you are not already in flight. Read error messages rather than repeating takeoff blindly.

The movement command fails: The requested distance might be outside the documented range, or the simulator may reject the action in the current state.

The drone turns the wrong way: clockwise and counter-clockwise are opposite; a new heading changes later movement directions.

The camera is black: Confirm `streamon()` was called and video frames are being received.

No AprilTag is detected: Confirm the installed tag family, focus, size, image quality and whether the marker is actually in view.

The drone hits the gate: Seeing the AprilTag is not enough. The tag is beneath the valid opening; aim at the opening and allow clearance.

Code works in practice but not in the race: Read the official field map, gate order, session rules and landing target. Manual flight commands are not allowed during official autonomous attempts.

Appendix A. Useful DJITelloPy Methods

Connection: `Tello()`, `connect()`, `end()`

Flight: `takeoff()`, `land()`

Movement: `move_forward(cm)`, `move_back(cm)`, `move_left(cm)`, `move_right(cm)`, `move_up(cm)`, `move_down(cm)`

Rotation: `rotate_clockwise(degrees)`, `rotate_counter_clockwise(degrees)`

Speed: `set_speed(cm_per_s)`

Flight information: `get_battery()`, `get_height()`, `get_yaw()`, `get_flight_time()`

Camera: `streamon()`, `get_frame_read()`, `streamoff()`

Additional advanced control: `send_rc_control(left_right, forward_back, up_down, yaw)` — continuous control channels, not distances. Use it only after you understand the drone's response to bounded corrections.

Avoid `emergency()` during normal trials: it is an emergency motor-stop command, not a safe landing method.

You can find the complete official command set in the Ryze Tello SDK documentation. DJITelloPy uses those commands through Python methods.

Appendix B. References

DJITelloPy — API and examples:

<https://djitellopy.readthedocs.io/en/latest/tello/>

DJITelloPy — official project repository:

<https://github.com/damiafuentes/DJITelloPy>

pupil-apriltags — detector installation and usage:

<https://pypi.org/project/pupil-apriltags/>

Ryze Robotics — Tello downloads and SDK manuals:

<https://www.ryzerobotics.com/tello/downloads>

AprilRobotics — AprilTag project and tag families:

<https://github.com/AprilRobotics/apriltag>

End of Student Programming Guide — FIRA Air Simulation League