AR4 DEVELOPER DOCUMENTATION

AR4 Python API User Guide

Connect, control, and integrate the AR4 robot controller and Nano I/O board from Python.

ROBOT CONTROL

`ARRobot`

Motion • calibration • Modbus

AUXILIARY I/O

`ARNano`

Gripper • inputs • outputs

REVISED EDITION • JULY 2026

TECHNICAL SOURCE: `ARrobot.py`

About this guide

This guide explains how to use the Python API supplied in `ARrobot.py` to communicate with the AR4 Teensy 4.1 motion controller and the Arduino Nano I/O board. It has been reorganized around common tasks, corrected against the supplied source code, and expanded with error-handling and troubleshooting guidance.

Document	Details
Title	AR4 Python API User Guide
Edition	Revised edition, July 2026
Primary reference	`ARrobot.py` supplied with this guide
Application reference	`AR4.py` supplied with this guide
Python dependency	`pyserial`

> **READ BEFORE COMMANDING MOTION**
> An industrial or collaborative robot can move unexpectedly if a script, coordinate, tool configuration, or calibration state is wrong. Keep the work envelope clear, begin at low speed, verify the robot is calibrated, and remain ready to stop the system. The examples in this guide explain API behavior; they do not replace the AR4 hardware, electrical, or operating safety instructions.

How to use this guide Start with Chapters 1–4 if you are new to the API. Chapters 5–10 explain robot control, calibration, and Modbus. Chapters 11–12 cover the Nano I/O board. The appendices provide compact method signatures, direct JSON commands, and troubleshooting steps.

Conventions Class and method names appear as `ARRobot`, `ARNano`, and `move_linear()`. Literal controller responses appear as `"Done"` or `{"msg":"error"}`. Joint arrays use the order `[J1, J2, J3, J4, J5, J6]`. Cartesian poses use `[x, y, z, rx, ry, rz]`, with translation in millimetres and orientation in degrees.

Contents

1 Introduction and Safety

The API exposes two independent serial devices. `ARRobot` communicates with the Teensy 4.1 motion controller. `ARNano` communicates with the Arduino Nano I/O board. Most motion scripts use `ARRobot`; gripper and auxiliary 5 V I/O tasks use `ARNano`.

1.1 Choose the correct device

Class	Default serial settings	Use it for
`ARRobot`	`115200` baud; timeout `None`	Position queries, calibration, robot motion, spline paths, Modbus, parameter updates, and raw robot-controller commands.
`ARNano`	`9600` baud; timeout `2` s	Servo/gripper commands, 5 V digital inputs, 5 V digital outputs, input waits, and the gripper amperage test.

The devices normally appear as separate COM ports. On Windows, use Device Manager or the AR4 HMI port selector to identify the correct port for each controller.

1.2 Safe commissioning sequence

Before running a new script, confirm that the selected COM port belongs to the intended controller. Power and configure the robot according to the AR4 hardware documentation. Clear the work envelope, verify calibration, start with low motion speeds, and test one move at a time. Check every returned value before continuing to the next motion.

> **SCRIPT STOP VERSUS HARDWARE STOP**
> `robot.stop()` sends a software stop command. Do not treat a Python method as a substitute for the system's required physical emergency-stop or safety circuit.

2 Installation and Project Setup

2.1 Install the dependency

The API uses Python's standard library plus `pyserial`. Install it in the Python environment that will run your script.

```
python -m pip install pyserial
```

2.2 Recommended folder structure

Keep the supplied API module separate from your application code.

```
my_project/
├── main.py
├── defaults.json          # needed only for update_params()
└── ARrobots/
    ├── __init__.py
    └── ARrobot.py
```

Import both device classes from the package:

```
from ARrobots.ARrobot import ARRobot, ARNano
```

For a small test, `ARrobot.py` may be placed beside the script and imported with `from ARrobot import ARRobot, ARNano`.

2.3 Find available serial ports

A short `pyserial` command can list ports detected by the operating system.

```
python -m serial.tools.list_ports
```

> **TIP**
> If the robot and Nano are both connected, record which COM port belongs to each device. Their baud rates and command sets are different.

3 Connect and Verify Communication

3.1 Robot-controller quick start

Change `COM4` to the Teensy controller's port. `connect()` returns `True` after the port is open; configuration and serial-port errors are raised as exceptions.

```
from ARrobots.ARrobot import ARRobot

robot = ARRobot(port="COM4", timeout=0.25)

try:
    robot.connect()
    print("Controller:", robot.hello())
    print("Position:", robot.get_position())
finally:
    robot.disconnect()
```

A finite serial timeout is useful while developing because it prevents a single serial read from blocking forever if the controller is silent. Dedicated motion methods still wait for their expected completion response, up to their method-level limit.

3.2 Nano quick start

Change `COM5` to the Nano I/O board's port. `connect()` waits approximately two seconds after opening the port so the board can reset.

```python
from ARrobots.ARrobot import ARNano

nano = ARNano(port="COM5")

try:
    nano.connect()
    print("Nano:", nano.hello())
    nano.gripper_open()
finally:
    nano.gripper_detach()
    nano.disconnect()
```

NO PORT SELECTED

`ARRobot.connect()` raises `ValueError("No COM port selected")` when its port is empty. `ARNano.connect()` raises `ValueError("No serial port specified")`. A missing, busy, or invalid operating-system port may raise a `pyserial` exception.

4 Responses, Errors, and Timeouts

4.1 Two response styles

The API deliberately provides both parsed and raw response methods. Dedicated robot methods such as `get_position()` and `move_linear()` normally return dictionaries. Modbus helpers and generic pass-through commands return raw strings. Nano commands parse JSON when possible and otherwise return their raw string.

Method family	Typical return	Notes
Robot position and motion	`dict` or `None`	Waits for `"robot_pos"`; motion methods also accept `"error"`. Returns `None` when their wait limit expires.
`ARRobot.read_json()`	`dict` or `None`	Non-JSON text becomes `{"msg":"raw","data":...}`.
`ARRobot.command()`	`str`	Returns one raw decoded line; raises `TimeoutError` after its timeout.
Robot Modbus helpers	`str` or `None`	Returns one raw response line.
`ARNano.command()`	`dict` or `str`	Parses JSON when possible; otherwise returns the text response.
`ARNano.input_read()`	`bool`, `dict`, or `str`	Converts `"T"` to `True` and `"F"` to `False`.

4.2 Position and error dictionaries

A typical position response has joint and Cartesian data:

```
{
    "msg": "robot_pos",
    "j": [0.0, 0.0, 0.0, 0.0, 90.0, 0.0],
    "pose": [315.8, 0.0, 445.7, 0.0, 135.0, 0.0],
    "speed_violation": 0
}
```

An error response normally identifies itself with `"msg": "error"`. Additional fields are firmware-dependent.

```
{
    "msg": "error",
    "data": "error message"
}
```

4.3 A reusable response check

```
def require_robot_success(response):
    if response is None:
        raise TimeoutError("No completion response from the robot")
    if response.get("msg") == "error":
        raise RuntimeError(f"Robot error: {response}")
    return response
```

SERIAL TIMEOUT BEHAVIOR

`ARRobot` defaults to `timeout=None`, which makes the underlying serial read blocking. For interactive development, a small finite timeout can make loss of communication easier to recover from. `ARNano` defaults to a two-second per-read timeout, but its `read_response()` loop keeps reading until it receives non-empty text.

5 Positions, Poses, and Motion Settings

5.1 Joint values and Cartesian poses

Robot joint lists contain at least six values. `move_joints()` uses the first six values and the API sends zeros for J7–J9. A pose contains exactly six values in the order `[x, y, z, rx, ry, rz]`.

Data	Order	Meaning
Joint list	`[J1, J2, J3, J4, J5, J6]`	Joint angles in degrees.
Cartesian pose	`[x, y, z, rx, ry, rz]`	Position in millimetres followed by orientation in degrees.
Wrist option	`"A"` by default	Passed to Cartesian path commands as the wrist configuration.

5.2 Speed, acceleration, and deceleration

The wrapper sends `spd_type: "percent"`. The `speed`, `acc`, and `dec` arguments therefore represent percentage-style command values as expected by the controller firmware. The wrapper does not clamp or validate these values; use settings appropriate for the robot and task.

VALIDATE EVERY TARGET
The Python wrapper does not verify that a Cartesian target is reachable or that a path is collision-free. Begin with low speed, confirm the active tool and frame configuration, and test new targets one at a time.

6 Query Robot State

6.1 Controller information

`hello()` sends `{"cmd":"hello"}` and returns the next line through `read_json()`.

```
info = robot.hello()
print(info)
```

6.2 Full position, joints, and pose

```
position = robot.get_position()
if position is not None:
    print("Joints:", position.get("j"))
    print("Pose:", position.get("pose"))
```

Convenience methods extract the same arrays:

```
joints = robot.get_joints()
pose = robot.get_pose()
print(joints)
print(pose)
```

`get_joints()` and `get_pose()` return `None` when `get_position()` returns no response.

7 Joint and Cartesian Motion

7.1 Move to explicit joint angles

```
response = robot.move_joint_angles(
    0, 0, 0, 0, 90, 0,
    speed=15,
    acc=10,
    dec=10,
)
require_robot_success(response)
```

The method sends the six supplied angles and appends zeros for J7–J9. It waits for either a position response or an error response.

7.2 Move using a joint list

`move_joints()` requires at least six values and raises `ValueError` if the list is shorter.

```
joints = robot.get_joints()
if joints is None:
    raise RuntimeError("Could not read joint positions")

new_joints = joints.copy()
new_joints[0] += 5
response = robot.move_joints(new_joints, speed=10)
require_robot_success(response)
```

7.3 Joint-interpolated Cartesian move

`move_cartesian()` sends the controller's `move_j` command. The robot follows a joint-interpolated route to the target pose; this is different from a straight-line tool path.

```
response = robot.move_cartesian(
    x=315,
    y=0,
    z=450,
    rx=0,
    ry=135,
    rz=0,
    speed=15,
    wrist="A",
)
require_robot_success(response)
```

7.4 Move using a pose list

```
pose = robot.get_pose()
if pose is None:
    raise RuntimeError("Could not read the current pose")

new_pose = pose.copy()
new_pose[2] += 25
response = robot.move_pose(new_pose, speed=10)
require_robot_success(response)
```

The caller should provide exactly six pose values. `move_pose()` indexes positions 0–5 and delegates to `move_cartesian()`.

7.5 Stop command

```
robot.stop()
```

`stop()` transmits `{"cmd":"stop"}` and returns `None`; it does not wait for an acknowledgement.

8 Linear, Arc, Circle, and Spline Motion

8.1 Straight-line move

`move_linear()` sends the controller's `move_l` command and includes a `rounding` value for path blending.

```
response = robot.move_linear(
    315, 0, 450,
    0, 135, 0,
    speed=12,
    acc=10,
    dec=10,
    rounding=0,
)
require_robot_success(response)
```

8.2 Arc move

An arc uses the current robot position as its start. `mid` must contain a complete six-value midpoint pose; `end` must contain three XYZ coordinates. Invalid lengths raise `ValueError`.

```
response = robot.move_arc(
    mid=[300, 50, 400, 0, 135, 0],
    end=[350, 0, 400],
    speed=12,
)
require_robot_success(response)
```

8.3 Circular path

`center`, `plane`, and `orientation` must each contain exactly three values.

```
response = robot.move_circle(
    center=[315, 0, 400],
    plane=[315, 50, 400],
    orientation=[0, 135, 0],
    speed=12,
)
require_robot_success(response)
```

8.4 Spline/lookahead queue

Spline mode queues consecutive linear moves for lookahead execution. During spline mode, each `move_linear()` call returns `None` immediately because the move is queued. `spline_end()` waits for the final position or error response.

```
robot.spline_start()
try:
    robot.move_linear(300, 0, 400, 0, 135, 0, speed=12)
    robot.move_linear(350, 0, 400, 0, 135, 0, speed=12)
    robot.move_linear(350, 50, 400, 0, 135, 0, speed=12)
finally:
    response = robot.spline_end()

require_robot_success(response)
```

END SPLINE MODE CLEANLY

After `spline_start()`, make sure the script calls `spline_end()`. A `try` / `finally` structure prevents the API object from remaining in its local spline state if an exception occurs while queuing points.

9 Calibration and Robot Parameters

9.1 Calibration

`calibrate()` accepts three lists containing 6–9 values. Shorter or longer lists raise `ValueError`; missing J7–J9 entries are padded with zeros. With no arguments, stage 1 calibrates J1–J6, stage 2 is disabled, and all offsets are zero.

```
response = robot.calibrate()
require_robot_success(response)
```

Calibrate only J1 in stage 1:

```
response = robot.calibrate(
    stage1=[1, 0, 0, 0, 0, 0]
)
require_robot_success(response)
```

Use a second stage only when its list contains a non-zero entry:

```
response = robot.calibrate(
    stage1=[1, 1, 1, 1, 1, 1],
    stage2=[0, 1, 1, 0, 0, 0],
    offsets=[0, 0, 0, 0, 0, 0],
)
require_robot_success(response)
```

CALIBRATION CAUSES MOTION

Confirm the machine is ready for calibration and the work area is clear before calling `calibrate()`. Use calibration selections and offsets that match the installed robot configuration.

9.2 Local wait helper

`wait_time(seconds)` calls Python's `time.sleep(seconds)`. It pauses the script locally and sends no command to either controller.

```
robot.wait_time(2)
```

9.3 Update controller parameters

`update_params(config_file="defaults.json")` loads a JSON configuration, builds the parameter payload expected by the firmware, and sends `{"cmd":"update_params"}`. It returns `"PARAMETERS_UPDATED"` only when the controller returns that exact text; otherwise it returns `"UPDATE FAILED: ..."`.

```
result = robot.update_params("defaults.json")
if result != "PARAMETERS_UPDATED":
    raise RuntimeError(result)
```

The file search order is: an existing explicit absolute path; the folder containing `ARrobot.py`; its parent folder; and the current working directory. A missing file raises `FileNotFoundError`. Missing configuration keys, malformed JSON, or invalid numeric fields also raise Python exceptions.

USE A MATCHING CONFIGURATION FILE

The parameter builder expects the complete key set used by the AR4 defaults file, including tool-frame values, motor and calibration directions, switches, limits, step/degree values, encoder and microstep values, and DH parameters. Use a configuration file generated for the installed AR4 setup.

10 Modbus Commands

The Modbus helpers send JSON to the Teensy motion controller and return one raw text response. Numeric read values may therefore arrive as strings. Read and write calls do not parse the response into Python integers.

10.1 Read and write helpers

Method	Default arguments	Returns
`modbus_read_coil()`	`slave_id=1, address=0, count=1`	`str` or `None`
`modbus_read_discrete_input()`	`slave_id=1, address=0, count=1`	`str` or `None`
`modbus_read_holding_register()`	`slave_id=1, address=0, count=1`	`str` or `None`
`modbus_read_input_register()`	`slave_id=1, address=0, count=1`	`str` or `None`
`modbus_write_coil()`	`slave_id=1, address=0, value=0`	`str` or `None`
`modbus_write_register()`	`slave_id=1, address=0, value=0`	`str` or `None`

```
coil = robot.modbus_read_coil(
    slave_id=7,
    address=3072,
    count=1,
)
print("Coil:", coil)

result = robot.modbus_write_register(
    slave_id=7,
    address=1088,
    value=55,
)
print("Write:", result)
```

10.2 Wait helpers

Method	Purpose
`wait_modbus_coil(slave_id=1, address=0, expected=1, timeout=10)`	Wait until a coil equals the expected value.
`wait_modbus_input(slave_id=1, address=0, expected=1, timeout=10)`	Wait until a discrete input equals the expected value.
`wait_modbus_holding_register(slave_id=1, address=0, expected=0, timeout=10)`	Wait until a holding register equals the expected value.

Firmware responses commonly include `"Done"`, `"Timeout"`, or `"Modbus Error"`. Treat these as literal strings.

```
result = robot.wait_modbus_coil(
    slave_id=7,
    address=3072,
    expected=1,
    timeout=10,
)

if result != "Done":
    raise RuntimeError(f"Modbus wait failed: {result}")
```

11 Nano Servo, Gripper, and I/O

`ARNano` communicates with the auxiliary Arduino Nano I/O board. Its default constructor is `ARNano(port=None, baudrate=9600, timeout=2)`. The Nano normally uses a different COM port from the Teensy motion controller.

11.1 Controller information and generic commands

```
print(nano.hello())
print(nano.command({"cmd": "hello"}))
```

`ARNano.command()` accepts a dictionary or a JSON string. It returns a parsed dictionary when the response is valid JSON; otherwise it returns the raw response string.

11.2 Servo and gripper

```
nano.servo_write(num=0, pos=45)
nano.gripper_open()    # servo 0, position 45
nano.gripper_close()   # servo 0, position 0
nano.gripper_detach()  # disable gripper servo PWM
```

The convenience gripper positions are defined by the API source. Use `servo_write()` for direct servo commands when appropriate for the installed mechanism.

11.3 Digital input and output

```
input_state = nano.input_read(2)
print("Input 2:", input_state)

nano.set_output(8, True)
nano.set_output(8, False)
```

`set_output()` converts its state argument with `int(bool(state))`, so falsy values become `0` and truthy values become `1`. `input_read()` returns Python `True` for a Nano response of `"T"` and `False` for `"F"`; any other parsed or raw response is returned unchanged.

11.4 Wait for an input

```
result = nano.wait_input(
    pin=2,
    state=1,
    timeout=10,
)
print(result)
```

The timeout is sent to the Nano firmware. The Python-side `read_response()` continues reading until it receives non-empty text.

11.5 Gripper amperage test and stop

```
amps = nano.test_gripper_amps()
print("Measured value:", amps)

nano.stop()
```

`test_gripper_amps()` converts the response to `float` when possible; otherwise it returns the raw response. `nano.stop()` sends the Nano `stop` command and returns its parsed or raw response.

12 Reliable Script Patterns

12.1 Always disconnect

Use `try` / `finally` so the serial port closes even if a command raises an exception.

```
from ARrobots.ARrobot import ARRobot

robot = ARRobot(port="COM4", timeout=0.25)

try:
    robot.connect()
    response = robot.move_cartesian(
        315, 0, 450,
        0, 135, 0,
        speed=10,
    )
    require_robot_success(response)
finally:
    robot.disconnect()
```

12.2 Handle expected exception categories

Exception	Typical cause	Response
`ValueError`	Missing port or invalid list length.	Correct the caller’s configuration or arguments.
`RuntimeError`	Low-level send/read while disconnected; application-level error you raise after checking a response.	Connect first or stop the program safely.

Exception	Typical cause	Response
`TimeoutError`	`ARRobot.command()` received no non-empty line within its command timeout.	Stop the sequence, verify controller state and port, then recover deliberately.
`FileNotFoundError`	`update_params()` could not find its configuration file.	Use a valid absolute path or supported search location.
`serial.SerialException`	Port missing, busy, disconnected, or otherwise unavailable.	Verify cable, device, operating-system port, and exclusive access.

12.3 Combined robot and gripper example

```python
from ARrobots.ARrobot import ARRobot, ARNano
import time

robot = ARRobot(port="COM4", timeout=0.25)
nano = ARNano(port="COM5")

try:
    robot.connect()
    nano.connect()

    print(robot.hello())
    print(nano.hello())

    move = robot.move_cartesian(
        315, 0, 450,
        0, 135, 0,
        speed=10,
    )
    require_robot_success(move)

    nano.gripper_open()
    time.sleep(1)
    nano.gripper_close()
finally:
    if nano.is_connected():
        try:
            nano.gripper_detach()
        finally:
            nano.disconnect()
    robot.disconnect()
```

KEEP DEVICE RESPONSIBILITIES SEPARATE

Use the Teensy port for robot motion and Modbus. Use the Nano port for servo, gripper, and 5 V I/O. Keeping the objects and variable names distinct makes multi-device scripts easier to review and troubleshoot.

13 Advanced Pass-Through Commands

13.1 Robot JSON pass-through

`ARRobot.command(command_dict, timeout=120)` is useful when the firmware supports a JSON command that does not yet have a dedicated wrapper. The argument can be a dictionary or a JSON string. The method serializes the dictionary, waits for one non-empty line, and returns that line as a raw string.

```
response_text = robot.command({
    "cmd": "get_position"
})
print(response_text)
```

If no response arrives before the command timeout, the method raises `TimeoutError`. Calls are protected by an internal lock so two threads do not execute `command()` simultaneously on the same object; other dedicated methods are not covered by that lock.

13.2 Raw legacy command

`command_raw(command_text)` writes the supplied text directly, adds a newline when needed, and returns one raw response. Use it only for legacy text commands that are not JSON.

```
response = robot.command_raw("S")
print(response)
```

DO NOT MIX RESPONSE CONSUMERS

A serial response can be read only once. Avoid calling low-level `send()`, `read_line()`, `read_json()`, and high-level methods concurrently on the same device. Otherwise, one part of the program may consume a response intended for another command.

14 Appendix A — `ARRobot` Method Reference

Signature	Returns	Purpose
`ARRobot(port=None, baudrate=115200, timeout=None)`	object	Create a robot-controller client.
`connect()`	`bool`	Open the configured serial port.
`is_connected()`	`bool`	Report whether the serial port is open.
`disconnect()`	`None`	Close the serial port when open.
`send(command)`	`None`	Send raw text, adding a newline if absent.
`read_line()`	`str`	Read, decode, strip, and return one line.
`send_json(command_dict)`	`None`	Serialize a dictionary and send it.

Signature	Returns	Purpose
`read_json()`	`dict` or `None`	Parse one line; wrap non-JSON text as a `raw` dictionary.
`read_response()`	`str` or `None`	Read one raw line.
`hello()`	`dict` or `None`	Request controller information.
`get_position()`	`dict` or `None`	Wait for a `robot_pos` message.
`get_joints()`	`list` or `None`	Return the `j` array from a position response.
`get_pose()`	`list` or `None`	Return the `pose` array from a position response.
`move_joint_angles(j1, j2, j3, j4, j5, j6, speed=25, acc=10, dec=10)`	`dict` or `None`	Move to explicit joint angles.
`move_joints(joints, speed=25, acc=10, dec=10)`	`dict` or `None`	Move using the first six list values.
`move_cartesian(x, y, z, rx, ry, rz, speed=25, acc=10, dec=10, wrist="A")`	`dict` or `None`	Joint-interpolated move to a pose.
`move_pose(pose, speed=25, acc=10, dec=10, wrist="A")`	`dict` or `None`	Pose-list wrapper for `move_cartesian()`.
`move_linear(x, y, z, rx, ry, rz, speed=25, acc=10, dec=10, rounding=0, wrist="A")`	`dict` or `None`	Straight-line Cartesian move; returns `None` while queuing a spline.
`move_arc(mid, end, speed=25, acc=10, dec=10, ramp=50, wrist="A")`	`dict` or `None`	Move through an arc.
`move_circle(center, plane, orientation, speed=25, acc=10, dec=10, ramp=50, wrist="A")`	`dict` or `None`	Move in a circular path.
`stop()`	`None`	Send robot stop without reading a response.
`spline_start()`	`None`	Enable spline queue mode and notify the controller.
`spline_end()`	`dict` or `None`	End spline mode and wait for final status.
`calibrate(stage1=None, stage2=None, offsets=None)`	`dict` or `None`	Perform one- or two-stage calibration.
`wait_time(seconds)`	`None`	Pause the Python script locally.
`update_params(config_file="defaults.json")`	`str`	Load and send robot parameters.
`command(command_dict, timeout=120)`	`str`	Send arbitrary JSON and return one raw response; raises `TimeoutError`.
`command_raw(command_text)`	`str` or `None`	Send a legacy raw-text command.

14.1 Modbus methods

Signature	Returns	Operation
`modbus_read_coil(slave_id=1, address=0, count=1)`	`str` / `None`	Read coils.
`modbus_read_discrete_input(slave_id=1, address=0, count=1)`	`str` / `None`	Read discrete inputs.
`modbus_read_holding_register(slave_id=1, address=0, count=1)`	`str` / `None`	Read holding registers.
`modbus_read_input_register(slave_id=1, address=0, count=1)`	`str` / `None`	Read input registers.
`modbus_write_coil(slave_id=1, address=0, value=0)`	`str` / `None`	Write a coil.
`modbus_write_register(slave_id=1, address=0, value=0)`	`str` / `None`	Write a holding register.
`wait_modbus_coil(slave_id=1, address=0, expected=1, timeout=10)`	`str` / `None`	Wait for a coil.
`wait_modbus_input(slave_id=1, address=0, expected=1, timeout=10)`	`str` / `None`	Wait for a discrete input.
`wait_modbus_holding_register(slave_id=1, address=0, expected=0, timeout=10)`	`str` / `None`	Wait for a holding register.

15 Appendix B — `ARNano` Method Reference

Signature	Returns	Purpose
`ARNano(port=None, baudrate=9600, timeout=2)`	object	Create a Nano I/O client.
`connect(port=None)`	`bool`	Open the serial port, optionally replacing the stored port.
`disconnect()`	`None`	Close the serial port when open.
`is_connected()`	`bool`	Report whether the serial port is open.
`send_json(command_dict)`	`None`	Serialize and send a newline-terminated command.
`read_response()`	`str`	Keep reading until non-empty text arrives.
`read_json()`	`dict` or `str`	Parse JSON when possible; otherwise return text.
`command(command_dict)`	`dict` or `str`	Send an arbitrary Nano JSON command.
`hello()`	`dict` or `str`	Request board information.
`servo_write(num, pos)`	`dict` or `str`	Write an integer servo channel and position.
`gripper_open()`	`dict` or `str`	Set servo 0 to position 45.
`gripper_close()`	`dict` or `str`	Set servo 0 to position 0.
`gripper_detach()`	`dict` or `str`	Disable gripper servo PWM.

Signature	Returns	Purpose
`input_read(pin)`	`bool`, `dict`, or `str`	Read a digital input; convert `T`/`F` to booleans.
`set_output(pin, state)`	`dict` or `str`	Write a normalized 0/1 digital output state.
`wait_input(pin, state, timeout)`	`dict` or `str`	Ask firmware to wait for an input state.
`test_gripper_amps()`	`float` or `str`	Run the amperage test and convert numeric text.
`stop()`	`dict` or `str`	Send Nano stop and return its response.

16 Appendix C — Direct JSON Serial Commands

The wrapper is a convenience layer over newline-delimited JSON. Each command is one JSON object followed by a newline character. The Teensy robot controller and Nano I/O board are separate serial devices and should be opened with their own port and baud rate.

16.1 Teensy robot controller

Default serial rate: `115200` baud.

```
{"cmd":"hello"}
{"cmd":"get_position"}
{"cmd":"move_joints","j":[0,0,0,0,90,0,0,0,0],"spd_type":"percent","spd":15,"acc":10,"dec":10}
{"cmd":"move_j","pose":[315,0,450,0,135,0],"spd_type":"percent","spd":15,"acc":10,"dec":10,"w":"A"}
{"cmd":"move_l","pose":[315,0,450,0,135,0],"ext":
[0,0,0],"spd_type":"percent","spd":12,"acc":10,"dec":10,"rounding":0,"w":"A"}
{"cmd":"calibrate","calibrate":[1,1,1,1,1,1,0,0,0],"offsets":[0,0,0,0,0,0,0,0,0]}
{"cmd":"stop"}
```

16.2 Nano I/O board

Default serial rate: `9600` baud.

```
{"cmd":"hello"}
{"cmd":"servo","num":0,"pos":45}
{"cmd":"gripper_detach"}
{"cmd":"set_output","pin":8,"state":1}
{"cmd":"input_read","pin":2}
{"cmd":"wait_input","pin":2,"state":1,"timeout":10}
{"cmd":"test_gripper_amps"}
{"cmd":"stop"}
```

RESPONSE FRAMING

The Python API reads one line at a time. A direct client in another language should likewise send one newline-terminated command and read complete newline-terminated responses. Exact response fields depend on the controller firmware and command.

17 Appendix D — Troubleshooting

Symptom	Likely cause	Recommended check
`ModuleNotFoundError: serial`	`pyserial` is not installed in the active environment.	Run `python -m pip install pyserial` with the same Python interpreter used by the script.
`ValueError` on `connect()`	No serial port was supplied.	Set `port="COMx"` or pass a port to `ARNano.connect(port=...)`.
Port cannot be opened	Wrong port, cable/device disconnected, port already in use, or permissions issue.	Close other applications using the port, verify Device Manager, and reconnect the device.
`RuntimeError: Robot is not connected`	A low-level robot method was called before `connect()` or after disconnect.	Connect first and keep calls inside the protected `try` block.
`RuntimeError: Nano not connected`	A Nano send/read method was called without an open port.	Call `nano.connect()` and verify `nano.is_connected()`.
`None` from a robot motion method	No expected `robot_pos` or `error` response arrived before the method limit; or the move was queued in spline mode.	Check `robot.spline_mode`, serial connection, firmware status, and controller output.
`TimeoutError` from `robot.command()`	No non-empty line arrived before the command timeout.	Verify the command name, controller port, controller state, and firmware response behavior.
Nano call appears to wait indefinitely	`read_response()` loops until it receives non-empty text.	Verify the Nano firmware understands the command and that the correct 9600-baud port is open.
Unexpected raw text	The response was not valid JSON.	For robot `read_json()`, inspect the `raw` dictionary. For Nano commands, inspect the returned string.
A later command receives the wrong response	Multiple readers or mixed low-level/high-level calls consumed serial lines out of order.	Use one command path at a time and avoid concurrent reads on the same serial object.
`update_params()` cannot find the file	The file is outside the supported search locations or the path is wrong.	Pass an existing absolute path or place the file beside `ARrobot.py`, in its parent folder, or in the working directory.

17.1 Final checklist

Before running a motion script, confirm the correct Teensy and Nano ports, calibration state, tool and frame setup, clear work envelope, low initial speed, and a tested stop procedure. During execution, check every response and stop the sequence on `None`, `"error"`, `"Timeout"`, `"Modbus Error"`, or an exception. At shutdown, detach the gripper servo when appropriate and close both serial ports.

Technical behavior in this edition was verified against the supplied `ARrobot.py` source and cross-checked against usage in the supplied `AR4.py` application. Firmware revisions may add commands or alter response fields; use `command()` for controlled testing of commands not yet wrapped by a dedicated method.