

# MachineLogic Python Programming Guide

## Contents

[MachineLogic Python Programming Guide](#)

[1. Introduction](#)

[2. Getting Started](#)

[3. Programming your application from MachineLogic's Python Editor](#)

[4. Operator Interface \(HMI\)](#)

[5. Deploying Custom Operator Interfaces to Python Applications](#)

## MachineLogic Python Programming Guide

This technical document provides guidelines and examples for using MachineLogic's online web editor to control actuators and robots using Vention's Python API. To access the technical reference of our API, [click here](#). For local installation, the python package can also be accessed [here](#)

---

### 1. Introduction

Welcome to the technical documentation for programming your Vention Machine in Python. This document is designed to guide you through the essential steps and best practices for leveraging the full potential of our API within MachineLogic's integrated web development environment. In this documentation, you will learn how to access the web editor, configure your machine, creating Python applications and custom operator interfaces.

Whether you are a seasoned developer or new to our platform, this guide provides a comprehensive resource for creating scalable automation applications within Vention's platform. Vention's python API can

This document is intended to be used by technicians and engineers with some level of familiarity with Python programming and software development. Knowledge in web front-end development is also recommended for the creation of Custom Operator Interfaces.

### 2. Getting Started

#### Step 1: Accessing MachineLogic:

**Prerequisite:** In order to configure and program in MachineLogic, the design must contain at least:

- A MachineMotion Controller
- One of the following component types:
  - [Linear Motion](#)
  - [Rotary Motion](#)
  - [Pneumatics](#)
  - [Material Handling](#)
  - [Control and Motors](#)
  - [Sensors](#)
  - [Robots](#)

See resources [here](#) for help when creating a new design with MachineBuilder.

To begin, select the configuration tab from MachineLogic's landing page:

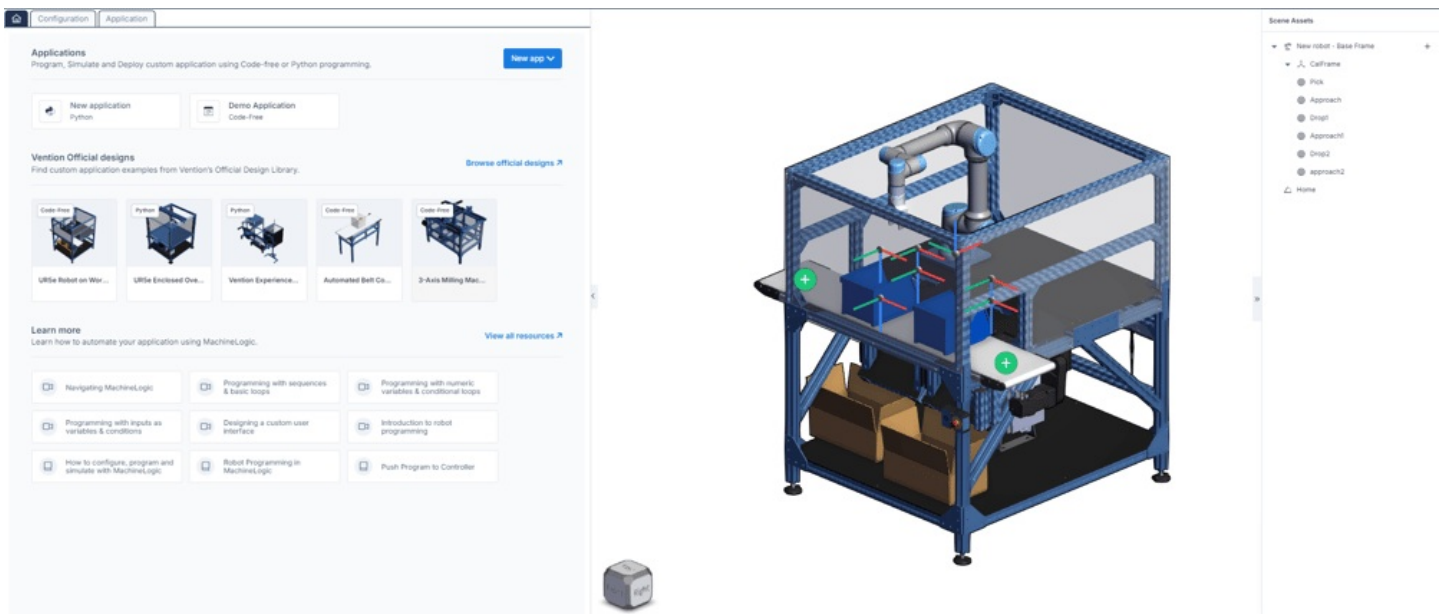


Figure 1: MachineLogic Landing page

**Tip:** MachineLogic's user Interface does not yet adapt to different monitor sizes. For an optimal experience, consider reducing your browser's Zoom setting when using MachineLogic with a smaller monitor such as a laptop screen

## Step 2: Configuration

### Linear/Rotary motion and Material Handling devices

Click the MachineLogic tab to begin configuring the automated equipment in your design.

1. Click the "Add Actuator" to begin your machine configuration
2. In the "Type" dropdown menu, select the actuator you would like to configure from your design.
3. The following fields will be auto-populated based on the parts connected to the selected actuator from the "Type" drop down menu:
  - Actuator name: Friendly name given to the actuator.
  - Motor size: Motor size connected to the actuator.
  - Homing sensor: We will guess which sensor should be the gantry's home position. If you would like to reverse the motor direction, switch the homing sensor with the end-stop sensor.
  - End-stop sensor: We will guess which sensor should be the gantry's end-stop sensor. If you would like to reverse the motor direction, switch the homing sensor with the end-stop sensor.
  - Brake installed: This checkbox represents the presence of a brake on the associated actuator.
  - Gearbox installed: This checkbox represents the presence of a gearbox on the associated actuator.
  - Advanced: Allows you to configure the following fields:
    - Custom Current: A different current value for your motor. The default value will be shown.

Tuning profile: This allows you to tune your step-servo motor using various profiles, to achieve the best performance for your application.

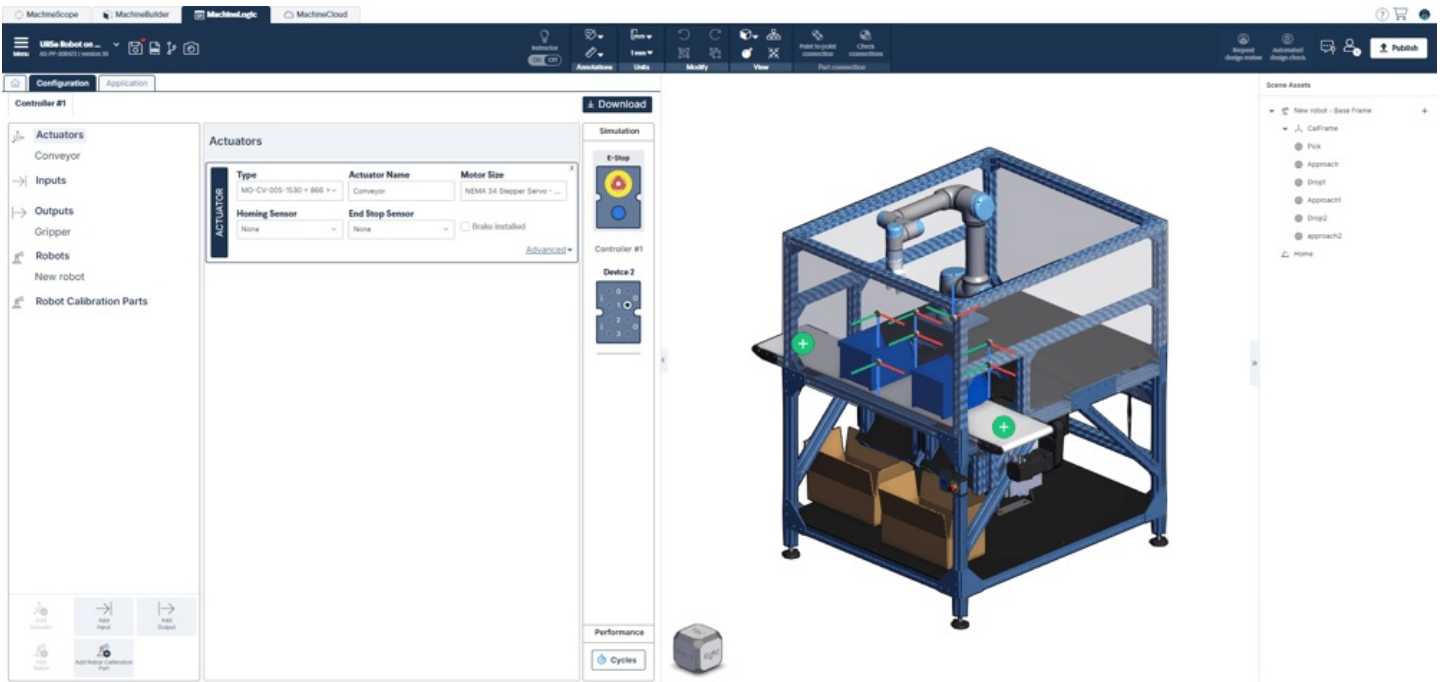


Figure 2: Actuator configuration in MachineLogic

## Pneumatic actuators

Click the MachineLogic tab to begin configuring a pneumatic actuator.

1. Click the “Add Actuator” to begin your machine configuration
2. In the “Type” dropdown menu, select the pneumatic actuator you would like to configure from your design.
  - Actuator name: Friendly name given to the pneumatic actuator.
  - Valve A Push: Represents the output pin from the digital IO module (CE-MD-001-0001) to activate the extended piston position of the pneumatic actuator.
  - Position Sensor (push): This field is optional. Represents the pneumatic actuator position sensor (CE-SN-008-0001) to give the feedback if the pneumatic actuator is in the “push” position. Select the input pin that will be connected to this sensor.
  - Valve B Pull: Represents the output pin from the digital IO module (CE-MD-001-0001) to activate the retracted piston position of the pneumatic actuator.

Position Sensor (push): This field is optional. Represents the pneumatic actuator position sensor (CE-SN-008-0001) to give the feedback if the pneumatic actuator is in the “pull” position. Select the input pin that will be connected to this sensor.

Figure 3: Configuring a pneumatic actuator in MachineLogic

## Inputs and outputs

1. Click the “Add Input” or “Add Output” button
2. Provide the following information:

- Input/Output: Specify IO type
- Name: Friendly name of the associated IO signal
- Device: Device number of the associated module as shown in the simulation pane.
- Pin: Associated Pin used on the corresponding device

Position Sensor (push): This field is optional. Represents the pneumatic actuator position sensor (CE-SN-008-0001) to give the feedback if the pneumatic actuator is in the “pull” position. Select the input pin that will be connected to this sensor.

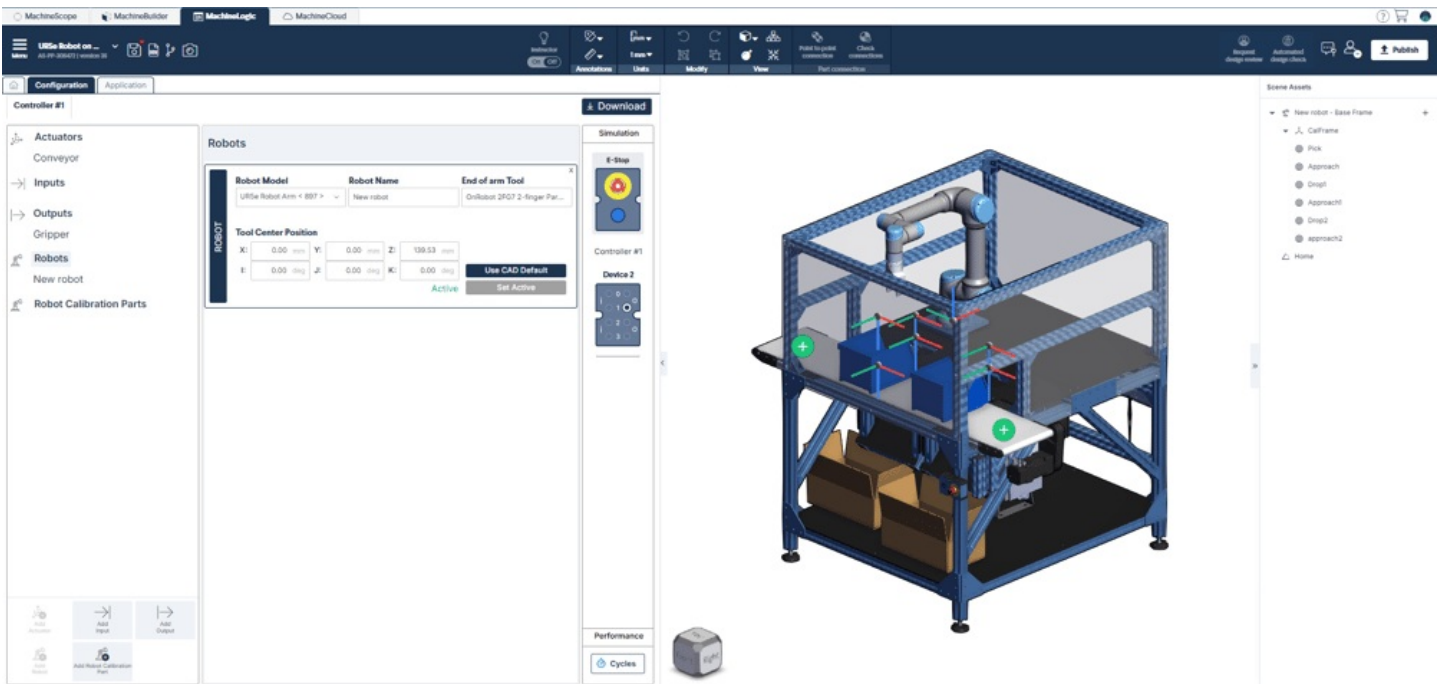


Figure 4: Configuring an I/O in MachineLogic

### 3. Programming your application from MachineLogic’s Python Editor

To create a new Python application, begin by selecting the **New App** button from the MachineLogic landing page:

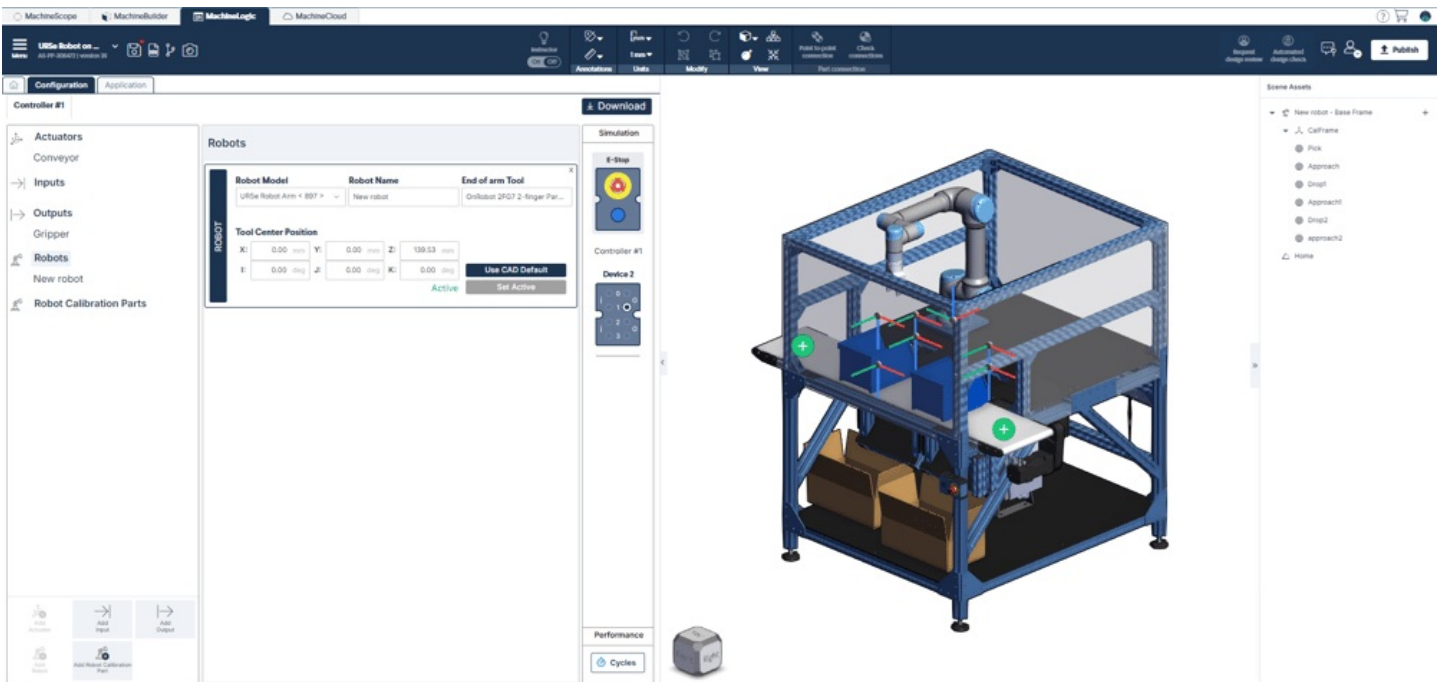


Figure 5: Configuring a Robot in MachineLogic

Every new Python application contains two files by default, **main.py** and **project.json**

## main.py

this file will be executed by default when launching your application. This is where most of the control code for your application should be written. This file is pre-populated with initialization code so you can start programming your application as soon as the application is created.

Here is an example of the initialization code for a new application containing a robot, conveyor and a digital output:

```
from machinelogic import Machine, ActuatorGroup
from machinelogic import MachineException, MachineMotionException, ActuatorGroupException, ActuatorException, RobotException, DigitalOut

### Configuration ###

# The following code has been automatically generated from the configuration.
# If the configuration changes, please update the code below, and ensure that the names match.

machine = Machine()

scene = machine.get_scene()

my_controller__1 = machine.get_machine_motion("Controller #1")

# Configured Actuators
my_conveyor = machine.get_actuator("Conveyor")
# Configured Robots
robot = machine.get_robot("New robot")
# Configured Outputs
my_gripper = machine.get_output("Gripper")

### Program ###

# Start coding here!
# Documentation can be found at https://vention.io/resources/guides/machinelogic-python-programming-514

print("Hello, Vention")
```

Notice how the content of the configuration is automatically populated in the main.py file using commands from the API:

```
# Configured Actuators
my_conveyor = machine.get_actuator("Conveyor")
# Configured Robots
robot = machine.get_robot("New robot")
# Configured Outputs
my_gripper = machine.get_output("Gripper")
```

**Tip:** These commands use the friendly names as defined in the configuration page, any changes made to the configuration prior to the creation of the app will have to be manually added to the main.py file afterwards.

Using a single **main.py** file as the entry point for simulation is the most straightforward way to simulate your python application from the web environment using [Vention's Python API](#). However, specifying additional simulation entry points is possible by configuring the **project.json** file

## project.json

Using the **project.json**, file you can specify what bash commands you want to run when playing the application. This makes it possible to launch multiple processes simultaneously as specified in the **project.json** file, enabling enabling the creation of more scalable and modular Python apps. Complex applications can be broken down into different processes running in parallel.

By default, the **project.json** file specifies the **main.py** file as the single entry point:

```
{
  "processes": {
    "onPlay": [
      {
        "name": "MachineCode App","command": "python3 -u main.py"
      }
    ]
  }
}
```

**Tip:** Adding the `-u` command line option when running python scripts allow for the stdout and stderr streams to be unbuffered, allowing for proper log output in the “LOGS” tab

to specify additional processes, add them to the **project.json** file as shown in the example below:

```
{
  "processes": {
    "onPlay": [
      { "name": "Process 1", "command": "python3 -u Process 1.py" },
      { "name": "Process 2", "command": "python3 -u Process 2.py" },
      { "name": "Server process", "command": "node web_server.js", "port": 3020 },
    ]
  }
}
```

Notice how the example above also contains the port number when launching a web server process. In the example below, pressing the play button will simultaneously launch Process 1.py, Process 2.py and web\_server.js.

For access to python applications example, [click here](#)

## Create a virtual environment for the application (if required) CAD

If your application requires special python packages, you need to create a virtual environment in the root of your application directory. If you do not require any additional packages, the default python environment will be used, which only contains the MachineLogic SDK and its standard dependencies found in requirements.txt of the SDK's package. You can create one by running the executing these commands in the terminal:

```
python3 -m venv venv
```

Then activate it with:

```
source venv/bin/activate
```

Finally, install the required packages:

```
pip install machine-logic-sdk
pip install <other package-name>
```

## 4. Operator Interface (HMI)

To create an Operator Interface (HMI) for your application, you can either use **UI Builder**, which enables the creation of simple operator interfaces using a code-free drag and drop interface, or create a completely custom UI using web front-end development libraries and workflows.

### Creating an Operator interface with UI Builder.

UI builder uses the MQTT protocol to communicate between the automation control code and the Operator interface. To learn more about UI builder, [click here](#).

To link a functionality in the application with an MQTT event outputted by UI builder, the `on_mqtt_event` method must be used. [Click here](#) for more information.

Here is example for using this method when the “Start” topic is received from the Operator Interface:

```
# Functions
def Start_Conveyor(topic, message):
    my_conveyor.move_continuous_async(250, 1000) # speed of 250 mm/s and acceleration of 1000 mm/s^2

### Program ###
machine.on_mqtt_event('Start', Start_Conveyor) # start event received from operator button press
```

In this example, we begin by creating a function containing both a `topic` and `message` argument. We then use the `machine.on_mqtt_event` to link the previously defined function as a callback when the 'Start' event is received

## 5. Deploying Custom Operator Interfaces to Python Applications

Users familiar with front-end software development practices can create completely custom User interfaces from MachineLogic's web editor or any Code editor of their choice. Please follow the guidelines below to ensure custom operator interfaces can function as intended with applications authored using Vention's Python API.

### Building and Deploying a React Front-End App as an Operator Interface

- **Step 1** - Develop and test your React Front-end from the code-editor of your choice
- **Step 2** - Make sure to add the `"homepage": ".",` to the `package.json` file in the root of the React app in order to display the UI correctly on the pendant.

Example:

```
{
  "name": "multi-process-application-ui",
  "homepage": ".",
  "version": "0.1.0",
  "private": true,
  etc ...
}
```

- **Step 3** - Build your React App
- **Step 4** - Bundle your application files and React files in an application folder
- **Step 5** - Create a `/ui` directory within the application folder.
- **Step 6** - In the `/ui` directory, insert the build folder content of your React App, this will be the entry point to your custom operator interface.

**Step 6** - To use MQTT for communication between the python processes and the operator interface, use this file helper to integrate MQTT for both the simulation environment and MachineMotion controller [mqtt-client.TS](#)

**Step 7** - For enabling HTTP communication with a web server for both the simulation and MachineMotion Controller, use the function helpers in this file [Utils.TS](#)

### Deploying a HTML Operator Interface

For non-React User interface use an `index.html` file as the entry point.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <h1>Hello, Vention</h1>
</body>
</html>
```