Click on a Section to Jump to It

Contents

Introduction

What is uArm Creator Studio

uArm Creator Studio is a Visual Programming Language and IDE, and is an attempt to make robot arms easy and fun to program, with a platform that scales well from beginners all the way to experts. The idea for this software is to gather the community onto one platform where developers can share programs with ease, where beginners don't have to set up a complicated environment, and everyone is guaranteed that their program will work with other uArms.

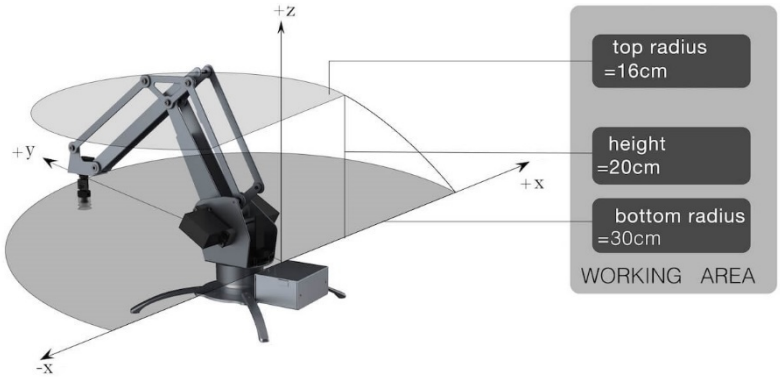
By providing easy to use computer vision (and a way to create vision objects easily), this software is valuable for beginners and advanced users at the same time. Furthermore, it scales incredibly well by allowing users to create custom python scripts that are able to import any library from the standard python library, plus other useful libraries like OpenCV, Numpy, PySerial, and more. Add the fact that the entire project is Open Source, and it should be useful for anybody with any use case.

The other vision for this software is for it to be possible to share programs with ease, so that a community based around creating cool software and sharing it will be possible. At the moment this is a difficult task (especially with computer vision), since everyone has a slightly different robot, webcam, and setup. By having a standard calibration and standard communication protocol and a standard way of teaching objects, it should be possible to program for any robot in any setup, as long as they have performed the appropriate calibrations. This software accomplishes all of these things, and will continue to be worked on in the future with the hope of inspiring people to accomplish robotic feats.

Getting Started

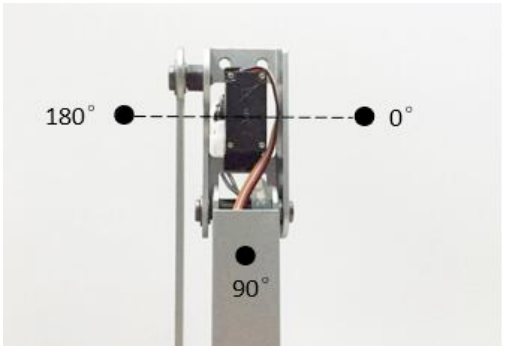
Basics

1) Three Dimensions (x,y,z) Diagram of uArm



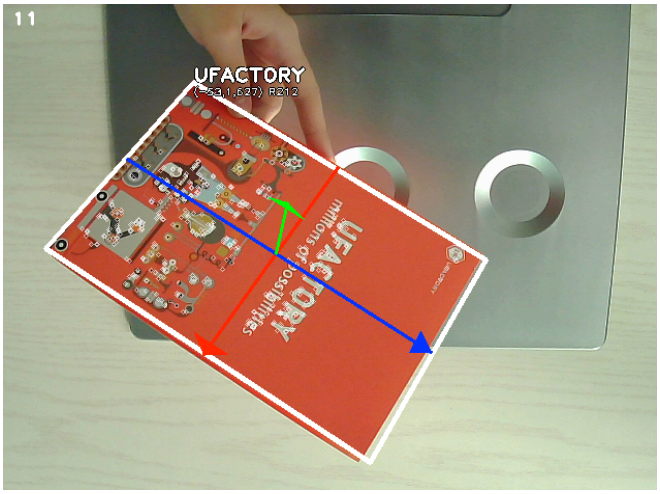
2) Wrist Angle

90°



You may turn the wrist to 0° - 180°

3) Three Dimensions (x,y,z) Diagram of Object



Blue Arrow: X axis / Red Arrow: Y axis / Green Arrow: Z axis

Connecting to Your uArm

Before you connect your uArm with UCS, please make sure:

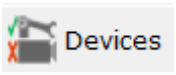
1) Your uArm is powered on and connected with PC.
2) If you are using Windows OS, please install the **Driver** needed to run uArm library. Please follow the uArm - Software Installation Guide
3) You have updated the uArm's **Firmware** to the **latest version**.

Notice:
Please double-click the flash_firmware.exe (within the UCS zip file) to upgrade your firmware before your FIRST USAGE of UCS.

When you first connect to your uArm, you might find some difficulties. Here is a quick guide for some issues that might occur.

First, start by clicking "Devices" on the toolbar. Then, click "Scan for Robots".

Select you have selected the correct port, click "Apply", and wait ten seconds. Sometimes, multiple ports will show up. This can happen when multiple devices that register as COM ports are plugged into the computer, or the computer has something internally plugged in. The way know which port is your robot is to open the Arduino IDE, click "tools", "port", and the port that is shown there will be your robot. Or you can experiment with different ports. Give the program ten seconds each time you try a new port, to see if it connects correctly.



The "Devices" button on the toolbar should now look like this

If you continue to have difficulties, contact UFACTORY support (ucs.support@ufactory.cc), ask for help in UFACTORY Forum, or contact the developer directly through GitHub. This project is open source, so improvements to the source code help everyone.

Programming

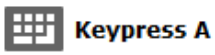
Moving the Robot

uArm Creator Studio is a Visual Programming Language with a heavy emphasis on combining computer vision and robotic arms in a standard platform. It has Python 3.4.4 bundled inside and requires no extra dependencies. It can even run without a robot or a camera, albeit at the cost of using commands that require a robot or camera.

What Are Events?

Programming in this software is **event** based. What this means is that no **Commands** (code) are ever run until an **event** is triggered. An example of an event might be the letter "A" being pressed on your keyboard.

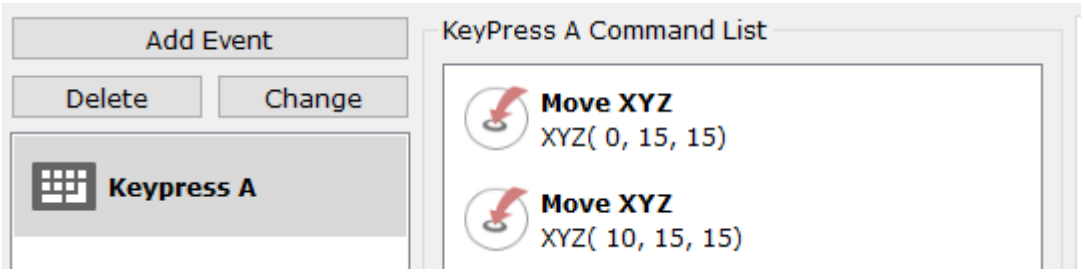
Example:



Click "Add Event", "Keyboard", "Letters", "A". This should create an element like the one shown below.

There should be a white box next to the event list, titled "KeyPress A Command List". Click and drag the icon shown below to this white box.

When this happens, a window will appear. Fill out the parameters x=0, y=15, z=15, then click "Apply" to close the window. Do this one more time, with parameters x=10, y=15, z=15. Your program should look like this now:



Now, press the "Start" button on the window toolbar. If a window pops up telling you that certain requirements are missing, it is possible that you have not yet connected your robot. If this is the case, go to the Connecting to Your Robot section of this manual and follow those steps before continuing here.

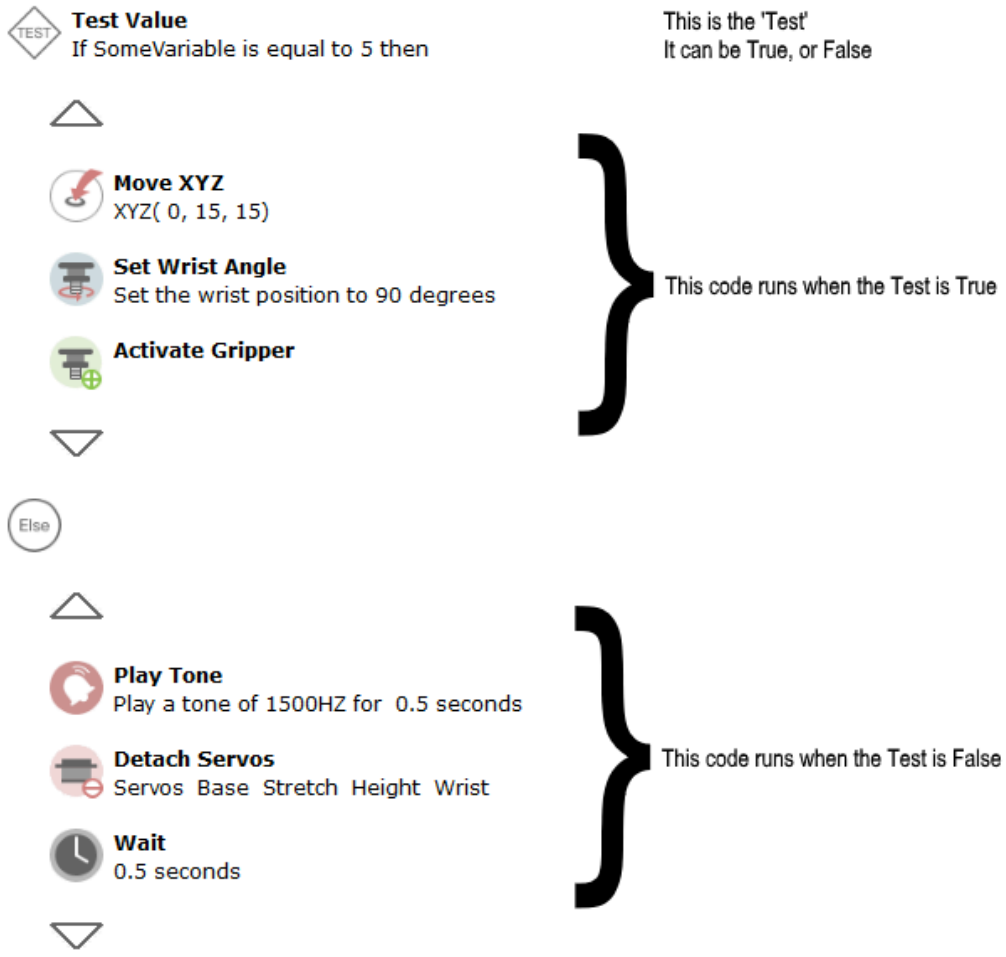
Once the program starts, it won't do anything until you press A You might notice that the Event lights up green when you press A, as does the "Move XYZ" command. This is because uArmCreatorStudio is showing what code is currently running.

If everything works properly, the robot should move to 0, 15, 15, then to 10, 15, 15. If it doesn't, make sure the robot is plugged in, and if it still doesn't work, make absolutely sure you've followed the directions in the Connecting to Your uArm section.

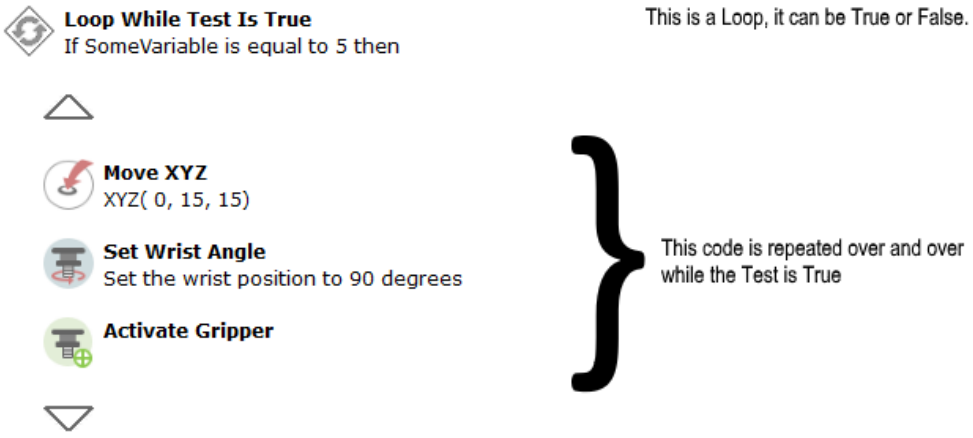
Program Flow (Using Tests and Loops)

uArm Creator Studio uses a model for program flow that has the basic components of other programming languages, but is meant to be easier to use since there's very little syntax to learn. Here is the structure for an "If" statement in this language:

Test Structures (If statements)



Loop Structures (While Loops)



Commands

Basic Commands

Move XYZ

Set the robots position. The X, Y, and Z coordinates are measured in centimeters, where 0, 0, 0 is the robots base, the Y axis is the distance from the base, the X axis is the left/right, and Z is height. If you set the movement to 'relative', it will add the XYZ you give the command to the robots current XYZ, then move there.

If you do not want to set one of the robots axis (keep it at the robots current value for that axis), simply leave it empty. For example, put y and z empty and x to 5 will set the robots x position to 5 while keeping the current Y and Z the same.

Requirements
Robot

Parameters	
X	Each of these parameters can be a number, variable, expression, or left empty.
Y	
Z	
Relative	If checked, the robot will move to a position currentX + X, currentY + Y, and currentZ + Z.

Python Equivalent

```
robot.setPos(x=0, y=15, z=15, relative=False)

#####   OR   #####

params = {"x":"0", "y":"15", "z":"15","relative": False}
MoveXYZCommand(env, interpreter, params).run()
```

Set Speed

This tool sets the speed of the robot for any move commands that are done after this. For example, if you set the speed to 20, then do two Move XYZ commands, the robot will move to those locations with a speed of 20 cm/s. The default robot speed is 10 cm/s.

Requirements

None

Parameters	
Speed	This is the speed of the robot, in centimeters per second.

Python Equivalent

```
robot.setSpeed(10)

#####   OR   #####

SpeedCommand(env, interpreter, {"speed": "10"}).run()
```

Set Wrist Angle

This command sets the angle of the robots 4th axis, the wrist.

Requirements

Robot

Parameters	
Angle	Between 0 and 180. Can be a Number, variable, or expression.
Relative	Move wrist to current position + angle parameter.

Python Equivalent

```
robot.setServoAngles(servo3=90)

#####   OR   #####

params = {"angle": "90", "relative": False}
MoveWristCommand(env, interpreter, params).run()
```

Play Motion Recording

This will play back a 'motion recording' at a playback speed of your choosing. To create robot motion recordings, simply click on 'Resources' on the toolbar and add a new recording.

Requirements

Robot

Movement Recording

Parameters	
Recording	Choose an already made robot recording.
Playback Speed	1.0 will play the recording at the normal speed. X2 will play twice as fast, and .5 will play twice as slow.
Reversed	This will play the recording backwards.

Python Equivalent

```
params = {"objectID":"NAME","speed": "1.0","reversed":False}
MotionRecordingCommand(env, interpreter, params).run()
```

Attach Servos

Re-engage certain servos on the robot. This will 'stiffen' the servos, and they will resist movement.

Requirements

Robot

Parameters	
Base	The servo on the center of the robot that controls the rotation of the arm.
Stretch	The left servo (if you are facing the back of the robot arm).
Height	The right servo (if you are facing the back of the arm).
Wrist	The servo on the end effector of the robot, if there is one.

Python Equivalent

```
robot.setActiveServos(servo0=True,
    servo1=True,
    servo2=True,
    servo3=True)

#####   OR   #####

params = {"servo0":True,"servo1":True,"servo2":True,"servo3": True}
AttachCommand(env, interpreter, params).run()
```

Detach Servos

Detach certain servos on the robot. This will disengage the chosen servos, meaning that they will no longer resist movement, and the robot will be free to move around.

Requirements

Robot

Parameters

Same as Attach Servos

Python Equivalent

```
robot.setActiveServos(servo0=False, servo1= False, servo2= False,
servo3= False)

#####   OR   #####

params = {"servo0": True,"servo1":True,"servo2":True,"servo3": True}
DetachCommand(env, interpreter, params).run()
```

Activate Gripper

Turn on the robots pump. This can be used to pick up objects.

Requirements
Robot

Python Equivalent

```
robot.setGripper(True)

#####   OR   #####

GripCommand(env, interpreter).run()
```

Deactivate Gripper

Turn off the robots pump. This can be used to drop picked-up objects

Requirements
Robot

Python Equivalent

```
robot.setGripper(False)

#####   OR   #####

DropCommand(env, interpreter).run()
```

Wait

This command will wait for a certain amount of time. Time is measured in seconds.

Requirements

None

Parameters	
Time	This can be a number, variable, or expression, representing seconds.

Python Equivalent

```
sleep(TIME)  # Sleeps for three seconds, as an example

#####   OR   #####

WaitCommand(env, interpreter, {"time": "TIME"}).run()
```

Play Tone

This tool uses the robots buzzer to play a tone at a certain frequency for a certain amount of time.

Requirements

Robot

Parameters	
Frequency	The frequency of the tone, in Hertz (Hz).
Duration	How long the tone should play for, in seconds.
Wait	If this is unchecked, the tone will start playing, but the panel will continue.

Python Equivalent

```
robot.setBuzzer(1500, 1)  # Turns buzzer on at 1500 Hz for 1 second

#####   OR   #####

params = {"frequency": "1500", "time": "1", "waitForBuzzer": True}
BuzzerCommand(env, interpreter, params).run()
```

Vision Commands

Calibrate Camera/Robot Position

If you want to use the commands below:

 **Move Relative to Object**

 **Set Wrist Relative to Object**

 **Pick Up Object**

 **Test Angle of Object**

Please **Calibrate Camera/Robot Position** first:
Click “Calibrate”, then “Calibrate Camera/Robot Position” and follow the instruction to calibrate.

There are vision commands you can use **WITHOUT** the Camera/Robot Position Calibration:

 **Test If Object Seen**

 **Test If Object Inside Region**

Move Relative To Object

This tool uses computer vision to recognize an object of your choice, and position the robot directly relative to this object's XYZ location.

Example:
If XYZ = 0,0,0, the robot will move directly onto the object.
If XYZ = 0,0,5 the robot will move 5 cm above the object.
If XYZ = 3,0,5, the robot will move 3 centimeters to the right of the object and 5 cm above the object.

If you don't want to set one of the robots axis, simply leave it empty. For example, put y and z empty and x to 5 will set the robots x position to the objects x position,+ 5cm while keeping the current Y and Z the same.

Requirements
Robot
Camera
Vision Object
Camera/Robot Calibration

Parameters	
X	Each of these parameters can be a number, variable, expression, or left empty.
Y	
Z	
Object	The object that will be tracked and the robot will move relative to.

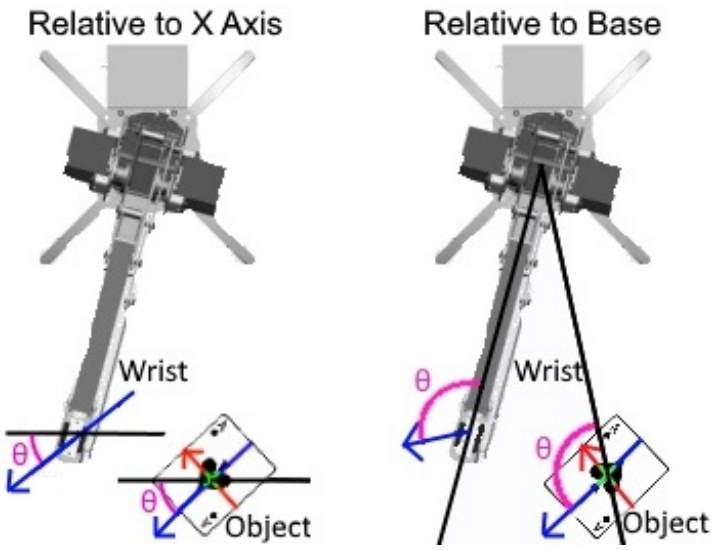
Python Equivalent
params = {"objectID": "OBJECT NAME", "x": "0", "y": "0", "z": "0"}
MoveRelativeToObjectCommand(env, interpreter, params).run()

Set Wrist Relative To Object

This tool will set the angle of the wrist servo on the robot to match the angle of a Vision Object, plus an angle of your choice. There are two ways to set the angle, shown below.

1) Relative to the robots X Axis (shown below, left). This means that robots wrist will be set to the angle that the Vision Object's Blue arrow makes against the robots x axis.

2) Relative to the robots Base. This means that the wrist will be set to the angle between the object and the base of the servo. This is useful, because you can run this command, then move the robot over the object, and the wrist will be aligned with the objects blue arrow.



Requirements

Robot
Camera
Vision Object
Camera/Robot Calibration

Parameters	
Object	The object that will be tracked and whose rotation will be used to set the wrist angle.
Angle	The "angle" offset from the objects actual angle. So if the object is 90 degrees rotated, and the "Angle" parameter is 30, then the wrist will be set to 120.
Relative To	Relative to X Axis (explained above). Relative to Robot Base (explained above).

Python Equivalent

```
params = {"objectID":"OBJECT NAME","angle":"0", "relToBase": False}
MoveWristRelativeToObjectCommand(env, interpreter, params).run()
```

Pick Up Object

This tool uses computer vision to recognize an object of your choice, and attempt to pick up the chosen object. It follows a specific pattern of movements: Move over the object (using an estimated depth position), then move down one centimeter at a time while checking the robots position to make sure it's still on top of the last known position of the object.

Once the robots Tip Sensor is activated, the robot will assume it hit either the floor or the object, and move up. It will test its location before moving up to make sure it was on top of the object.

IMPORTANT FOR ADVANCED USERS: The Pick Up Object command acts like a "Test" command. It returns True if the pickup was successfull, and it returns False if there was an error in the pickup (such as loss of tracking or somethign). This means that Pick Up Object can be used like a "Test" command, and can have blocks of code after it and an else statement, so you can run certain code if the object was successfully picked up, and different code if it was not picked up correctly.

Requirements

Robot
Camera
Vision Object
Camera/Robot Calibration

Parameters	
Object	This is the object that will be picked up.

Python Equivalent

```
params = {"objectID":"OBJECT NAME"}
PickUpObjectCommand(env, interpreter, params).run()
```

Test If Object Seen

This command will allow code in blocked brackets below it to run IF the specified object has been recognized, with parameters that allow you to check how accurately the recognition was, and how long ago to check.

Since the vision system keeps a history of the last 60 frames of tracked objects (approximately two seconds, with most webcams), it can look back through history to check if an object was recognized in those last 60 frames. You can tell it to look at only the most recent frame by having the "When" slider all the way to the left. You can check if the object was seen in the last ~two seconds by putting the slider all the way to the right.

Requirements

Camera

Vision Object

Parameters	
Object	The camera will search for this vision object.
Confidence	If "High", then the test will return true only if its *very* sure that the object was seen. If "Low", it will return true regardless of confidence level.
When	If you want to check the last X frames (the last, say, two seconds) to see if the object was seen.
NOT	Invert the result.

Python Equivalent

```
params = {"objectID": "OBJECT NAME", "age": 0, "confidence": 0,
"not": False}
test = TestObjectSeenCommand(env, interpreter, params)
if test.run():
    print("The Test Returned True!")

#####   OR   #####

trackable = resources.getObject("Object Name")
vision.addTarget(trackable)

# Somewhere later on in the script put this, so that vision has time
to recognize the object
tracked = vision.searchTrackedHistory(trackable=trackable, maxAge=0,
minPoints=30)
if tracked is not None:
    print("The Object Was Seen!")
```

Test If Object Inside Region

This command will allow code in blocked brackets below it to run IF the specified object has been recognized and the objects location in a particular location.

You click and drag on the camera stream to specify the region that the object must be in for the test to return true.

Requirements

Camera

Vision Object

Parameters	
Object	The camera will check the location of this vision object
Part	Here you define which part of the object must enter the selected region in order for the test to return True. "Center" means the middle of the object. "All" means every corner of the object. "Any" means any corner of the object.
Selection	Click and drag on the screen to make a rectangle. If the vision object enters that rectangle while the program is running, the test will return true.
NOT	Invert the result.

Python Equivalent

```
params = {"objectID": "NAME",
          "location":[[0, 0],[100,100]],
          "part": "any",
          "not": False}

test = TestObjectLocationCommand(env, interpreter, params)
if test.run():
    print("The Test Returned True!")
```

Test Angle of Object

This command will allow code in blocked brackets below it to run IF the object's rotation is between two angles. The angles are measured from the robots positive X axis, counter clockwise. The positive X axis is 0 degrees, the positive Y axis is 90 degrees, the negative X axis is 180 degrees, and so on.

Requirements

Robot
Camera
Vision Object
Camera/Robot Calibration

Parameters	
Object	This is the vision object whose rotation will be tested.
Start/End Angles	If the object is between these two angles, the test will return true.
NOT	Invert the result

Python Equivalent

```
params = {"objectID": "NAME","start": "0", "end":"90", "not": False}

test = TestObjectAngleCommand(env, interpreter, params)
if test.run():
    print("Objects rotation is between 0, 90degrees from X axis!")
```

Logic Commands

Set Variable

This command can create a variable or set an existing variable to a value or an expression. Variables can be used to store values, and then used in any textbox in the GUI. For example, you could create a variable XPOSITION and set it to 3, then use a Move XYZ command, where the x parameters is XPOSITION. You could make a variable TIMEDELAY and make it 5, then use it inside of a wait command.

Variables created with set variable can also be used in the Run Python Code command, and vice versa. You can set a variable to an expression, and that expression can contain other variables. Essentially, Set Variable works the same as assigning variables in python.

In fact, you can use any python type with this command. For example, create a variable HOMEPOSITION and set it to [0, 15, 20], then make a Move XYZ command where the X, Y, and Z parameters are HOMEPOSITION[0], HOMEPOSITION[1], HOMEPOSITION[2] respectively. This will work, because you can use python types anywhere in the program. This might be too advanced for some users, so when in doubt, just use a number.

Requirements

None

Parameters	
Variable	This must be a name with no spaces, and it has to start with a letter. It can only use letters and numbers. If you create an invalid variable name, it will be rejected when you close the prompt.
Expression	A valid python expression- a number, variable, equation, or function.

Python Equivalent

```
VARIABLENAME = EXPRESSION
```

Test Value

This will allow/disallow code to run that is in blocked brackets below it IF the test is true.

Requirements

None

Parameters	
Expression	A valid python expression- a number, variable, equation, or function.
Test	This will test the first expression against the second expression.

Python Equivalent

```
if EXPRESSION == EXPRESSION:
    print("Code runs here!")

# Other operators: >, <, !=
```

Loop While Test Is True

Repeat this section of commands while a certain test returns true. You can choose what type of test will be used.

This will essentially loop commands in blocked code, run a test, and if the test is true it will loop again. It does this forever until the test returns false (or if the program ends).

Requirements

Depends on the test you choose.

Parameters	
Test	You can select a different test here, and the code will loop as long as that test is true.

Python Equivalent

```
while EXPRESSION:
    print("Running Code!")
```

Else

If an IF command returns False, you can put an else command after the first two blocks, and the code inside of the ELSE block will run instead.

Exit Current Event

When the code reaches this point, the program will not process the rest of this event. Instead, it will go to back to checking events. This is useful if you have an IF statement and don't want the code to continue unless the IF statement is false.

End Task

When the code reaches this point, the program will end. If this command is run when the task is being run inside of another task, it will end the currently running task and return to the parent task.

Start Block

This is sort of like an opening "bracket" in other programming languages. It marks the start of a section of code that belongs to an IF command, Loop command, or Else command that is directly before the start block.

End Block

This is sort of like an closing "bracket" in other programming languages. It marks the end of a section of code that belongs to an IF command, Loop command, or Else command. There must be a start block command somewhere ahead of the end block for it to be a "section" of code.

Function Commands

Run Python Code

This command will execute any python code inside of it- without needing python installed on your computer. You can click "Show Documentation" to see much much more information about variables that are available in the command.

You can import any python library from the standard library, use python as you would usually expect it to work, and more. This function is intended for advanced users, so I will describe things in-depth here.

The namespace of this command is important to note. If you create a variable, function, or class in one "Run Python Code" command, you will be able to use it in any other "Run Python Code" command elsewhere in the program. Essentially, they have the same scope. This is useful because you can create a bunch of variables/Functions/Classes in the Initialization Event then use them elsewhere.

Furthermore, there are certain built-in variables that can be used. This is described in-depth inside of the "Show Documentation" button in the script command.

Requirements
Depends on the code you run

Parameters	
Script	This is where you type your python code in. It will not allow you to press "Apply" if you have invalid syntax anywhere in your code. It will tell you the issue on the bottom left of the window.

Run Task

This tool will run another task file and run it inside of this task, until the "End Task" command is called within the task, then it will return to the currently running task. All tasks are preloaded when script is launched, so if a child class runs a parent class, an error will be returned.

Requirements

An already-made *.task file

Parameters	
Task	Select a task file that you have already made and tested.
Share Variables	This is an important parameter. Leave it unchecked if you want the opened task to be a "clean slate" that works without any input. Check this if you want to share all of the variables/functions/classes that have been made in this current task with the task that is being opened. Any variables changed in the opened task will also change in the parent task. This is a way to communicate between tasks, or maintain state.

Python Equivalent

```
params = {"filename": "VALID FILENAME", "shareScope": False}
RunTaskCommand(env, interpreter, params).run()

'''
 Be careful when setting the filename. If the name has \t or \n
inside it, it will be interpreted # incorrectly. Use two slashes \\
on every slash in the filename to avoid this problem.
'''
```

Run Function

This will run a custom function that the user defines in the Resources menu. If the function has arguments, the user will be prompted to fill out the arguments

Requirements

Whatever requirements the commands inside the function require.

Parameters	
Function	Here you choose the function you wish to run.
Arguments	Since functions might have custom arguments, you will have to know what each argument means and how it is used.

Python Equivalent

```
params ={"objectID":"NAME","arguments":{"arg1":"val", "arg2":"val"}}
RunFunctionCommand(env, interpreter, params).run()

# If the function has no arguments, then set "arguments" to {}
```