A Tutorial on Programming the Thymio Robot in Python

Version 0.1

Moti Ben-Ari

http://www.weizmann.ac.il/sci-tea/benari/

Contents

1 Introduction

The Thymio II robot can be programmed in graphical languages for novices (VPL, VPL3, Blockly, Scratch) and in textual languages for those with experience in programming (Aseba, Python). This document is a tutorial on programming the Thymio II robot using Python. The subset of the Python language supported by the Thymio is sufficient to implement advanced algorithms such as those in [1]. Complete references on programming the Thymio are [2, 3, 4].

The document assumes the following knowledge and skills:

- Familiarity with the Thymio that you can obtain by exploring the pre-programmed behaviors and by constructing programs in one of the graphical languages such as VPL.
- The ability to write elementary programs in Python.

Python programs can be written using the Thonny integrated development environment (IDE).[1] Thonny is recommended because it is similar to other IDEs that you may have used.

The Python programs are processed as shown in Figure 1.

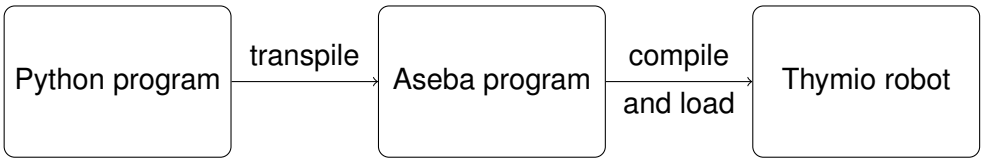


Figure 1: The architecture of the Thymio Python environment

A *transpiler* translates the Python program into a program in the Aseba language which was the original programming language for the Thymio. Fortunately, this step is transparent and you don't have to learn Aseba.

The Thymio Suite compiles and loads programs into the Thymio. You will already have used Thymio Suite to run programs written in VPL. For Python you need to run Thymio Suite initially but no further interaction is needed.

Section 2 explains how to install the software for programming the Thymio and Section 3 explains how to run a program. Section 4 is an extended example that demonstrates Python programming for the Thymio. Section 5 lists the constructs of Python that are supported by the Thymio and Section 6 describes the events, variables and functions used to interface with the Thymio. Section 7 shows how to display the values of Thymio variables.

Acknowledgment I am deeply indebted to Yves Piguet for his untiring help.

[1] The Jupyter IDE can also be used to develop Python programs for the Thymio.

2 Installation

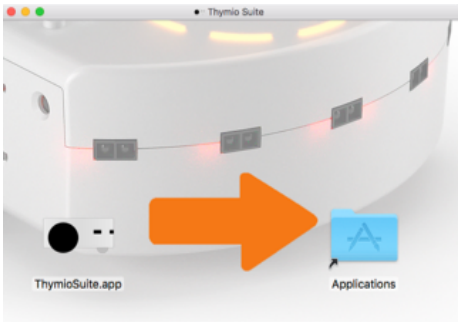
Download and install the Thymio Suite:

`https://www.thymio.org/download-thymio-suite-redirect/`

If you have written graphical programs for the Thymio the software will already be installed.

Windows Run the `exe` installation file.

MacOS Open the `dmg` file, then drag and drop the ThymioSuite.app icon onto the Applications icon in the same window:



Linux Follow the instructions at `https://www.thymio.org/linux-installation/`.

Download and install the Thonny environtment:[2]

`https://thonny.org/`

When you run it for the first time select the menu item Tools/Manage plug-ins. In the search bar write `tdmclient-ty` and select Search on PyPI. In the search results select `tdmclient-ty` which will install the plugin. Select Close when the installation is finished.

3 Running a Python program

1. Attach the Thymio to the computer with a USB cable or a wireless dongle. Turn on the Thymio by touching the center button for several seconds.
2. Run Thymio Suite which will connect the Thymio software with the computer. You can select the icon Aseba Studio to verify that the Thymio is recognized although this is not necessary.

[2]Thonny bundles a Python interpreter in its installation. Should you wish to use a different interpreter (at least version 3.6), after installing Thonny select the menu item Run/Select interpreter and then Alternative Python 3 interpreter or virtual environment.

3. Run Thonny. By default three panes are shown: (i) the central pane with tabs for each open source file, (ii) the Shell pane below where printed output will be displayed, (iii) the Assistant pane on the right displays hints on compilation errors and warnings.

4. The Thonny interface is standard with commands for opening and closing files, editing, searching, replacing, and so on. You can run and debug a regular Python program using the commands in the Run menu.

5. The menu Thymio contains four commands for running Python programs on the Thymio:

   (a) Run: Transpile and run a Python program. Interact with the Thymio as you would if it were programmed in VPL. Output from **`print`** statements will appear in the Shell.

   (b) Transpile program: Transpile a Python program. If transpiler does not find any errors, the Aseba program that results will be displayed in the Shell. You do not need to examine it.

   (c) Stop: Stop the running of your program in the Thymio. This may not stop the motors so be sure to include a way of stopping them like the button event handler ‡14-30 in the example below.

   (d) Unlock: See [4].

4 An extended example

We implement a *Braitenberg vehicle* displaying the *dogged* behavior [1, Chapter 3]. Several versions of the program, each adding some feature, will be given. The programs are split into sections and each is explained separately.

Notation: ‡n refers to line n of the program.

The initial program

Dogged behavior

```
1  """
2    Braitenberg vehicle: dogged behavior
3
4    When the robot detects an object in front it moves backwards
5    When the robot detects an object in back it moves forwards
6
7    The robot is turned on and off using the center button
8      When turned on, initially the robot moves forwards
9      When turned off, the robot stops
10 """
11
```

```
12 # State: False = off, True = on
13 state = False
```

The program starts with comments describing its behavior. A global variable `state` is "declared" at ‡13. In Python variables are not declared but come into being when they are assigned a value.

```
14 # Event handler for the center button
15 @onevent
16 def button_center():
17     # Global declarations
18     global state
19     global motor_left_target, motor_right_target
20
21     # The variable button_center is False if the button is released
22     if not button_center:
23         if state:
24             state = False
25             motor_left_target = 0
26             motor_right_target = 0
27         else: # not state
28             state = True
29             motor_left_target = 200
30             motor_right_target = 200
```

Most programming for the Thymio is done within *event handlers* which are run when the associated event occurs. An event handler is syntactically a function *without* parameters, prefixed with the *decorator* `@onevent` ‡15.

The center button starts and stops the program. The event handler `button_center()` ‡16 is called whenever the button is touched (`button_center` receives the value `False`) or released (`button_center` receives the value `True`). The event handler uses three predefined variables: `motor_left_target` and `motor_right_target` which are used to set the power of the motors, and `button_center` ‡22 which contains data associated with the event.

When the button is released `state` is toggled off to on or on to off, and the motors powers are set either to move forwards ‡29-30 or to stop ‡25-26.

Since both `state` and the motor powers are global variables that are modified, they must be declared **`global`**. The variable `button_center` is only read so it need not be declared **`global`**.

```
31 # Event handler for proximity sensors
32 @onevent
33 def prox():
34     # Global declarations
35     global motor_left_target, motor_right_target
36 
37     # If state is off don't do anything
38     if not state: return
39 
40     # If front central sensor detects an object
41     # set motors to move backwards
42     if prox_horizontal[2] > 2000:
43         motor_left_target = -200
44         motor_right_target = -200
45     # If both back sensors detect an object
46     # set motors to move forwards
47     elif prox_horizontal[5] > 2000 and \
48          prox_horizontal[6] > 2000:
49         motor_left_target = 200
50         motor_right_target = 200
```

The behavior is implemented in an event handler that reads the front center horizontal proximity sensor and the two rear horizontal proximity sensors. Nothing is done if the Thymio is off ‡38.

The values of the sensors are stored in `prox_horizontal` which holds seven values: values detected by the five front sensors and the values detected by the two rear sensors (Figure 2). If an object is detected in front of the Thymio ‡42, the motors are reversed from forwards to backwards; if the object is behind the Thymio ‡47-48, the motors are reversed from backwards to forwards.

Recall that to continue a line in Python ‡47 you must use the backslash character. Alternatively, parentheses can be used:

```
    elif (prox_horizontal[5] > 2000 and
          prox_horizontal[6] > 2000):
```

Displaying the state in the top LEDs

Let us modify the program so that the top LEDs will display the state of the program, not just the value of `state` but also if the Thymio is moving forwards or backwards. The color displayed by the top LEDs can be set by calling a function with three arguments giving the color as a combination of (red, green, blue) each in the range 0 to 32. Add the following function calls after the numbered line in the above program:

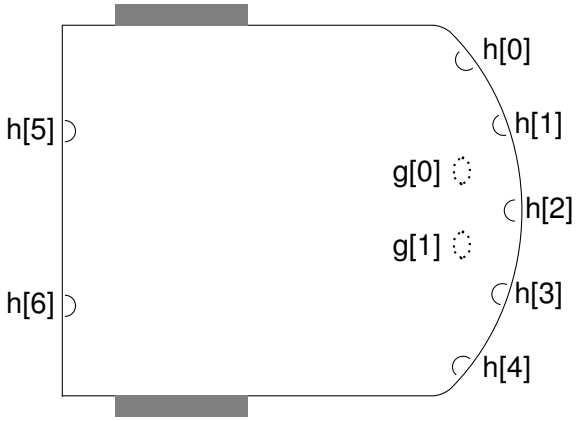


Figure 2: The sensors of the Thymio robot: h=horizontal, g=ground

```
nf_leds_top(32,32,32)    # White, line 24
nf_leds_top(0,0,0)       # Off, line 28
nf_leds_top(0,32,0)      # Green, line 42
nf_leds_top(0,0,32)      # Blue, line 48
```

Defining a function

Instead of writing calls to the Thymio function inline, let us place a single call in a function so that we can change the display operation for all uses in one place:

```
# Function for setting the top LEDs
def top(r,g,b):
    nf_leds_top(r,g,b)
```

Replace all calls to `nf_leds_top(r,g,b)` by `top(r,g,b)`.

The circle LEDs

The 8 circle LEDs that surround the buttons are extremely useful when programming the Thymio, because they can display small numbers 0 to 8 directly without encoding in colors.[3] Function `nf_leds_circle` takes 8 parameters, one for each LED in the circle, giving the intensity of that LED from 0 to 32. Since it is hard to distinguish levels of intensity, it is recommended to turn each LED either off (0) or on (32). The following function will display the four motion states in the first four circle LEDs:

[3]Larger numbers can be display in binary encoding.

```
def circle(motion):
    if motion == 0: nf_leds_circle(0,0,0,0,0,0,0,0)
    if motion == 1: nf_leds_circle(32,0,0,0,0,0,0,0)
    if motion == 2: nf_leds_circle(32,32,0,0,0,0,0,0)
    if motion == 3: nf_leds_circle(32,32,32,0,0,0,0,0)
```

The function call is `circle(m)` where the argument `m` is a value between 0 and 3.

A clever function

Instead of a sequence of **`if`**-statements we can compute whether a LED is turned off or on:

```
def circle(motion):
    nf_leds_circle(
        32*(motion % 4 > 0),
        32*(motion % 4 > 1),
        32*(motion % 4 > 2),
        0,0,0,0,0)
```

The call to `nf_leds_circle` will turn on the first three LEDs:

```
0 0 0
32 0 0
32 32 0
32 32 32
```

5 The Python subset supported by the Thymio

The only data type is **`int`** which are 16-bit integers in the range $(-32768, 32767)$.

Lists of integers can be accessed as complete lists or as individual elements:

```
a = [0,1,2,3] # a is a list of integers
print(a[2])   # Select an individual element
b = a         # Assign an array
print(len(b)) # len is supported
b = a[0:2]    # Slices are not supported
print(b)      # Expressions with lists not are supported
```

The **`str`** type and the string library functions are not supported but a string can be printed:

```
speed = ...
print('Speed = ', speed)
```

Strings can be used for documentation ‡1-10.

The **bool** type is supported.

The undefined value None is not supported.

Both local and global variables are supported, but since the event handlers do not have parameters you will use global variables more often than in a regular Python program. Do not forget to declare a variable as **global** if you change its value inside a function including event handlers.

Arithmetic operations (a+b), assignment operations (a=b, a+=b), comparisons (a!=b), bitwise operations (a|b), and the functions **abs** and **len** are supported. The division operation is integer division with truncation (a//b).

All the control statements are supported with the following exceptions:

- **for** loops are limited to a range of integers (possibly with two or three arguments). The following program prints 10 and then 20:

```
a = [0,10,20,30]
for i in range(1,3):
    print(a[i])
```

- The **break** and **continue** statements are not supported.

Functions can be defined and called but only with positional arguments that are scalars.

6 Interfacing with the Thymio

There are a large number of events, variables and functions that are predefined and used to interface with the Thymio robot. This section will explain some frequently used ones. You can look up the rest of them in the Thymio documentation [3].

6.1 Events

Table 1 shows events that cause an event handler to be called.[4]

6.2 Variables

Table 2 shows predefined variables. They are all global.

[4]Hz, an abbreviation for *hertz*, is the number of times per second that the events occurs. For 20 Hz the event occurs every $1/20 = 0.05$ seconds or 50 milliseconds.

Event	When occurs

`button_center`
`button_forward`
`button_backward`
`button_left`
`button_right`
	pressed or released

`prox` `acc`	periodically 20 Hz periodically 16 Hz

`timer0`
`timer1`
`sound_finished`
	timer 0 expired
	timer 1 expired
	sound duration expired

Table 1: Events

6.3 Predefined functions

Table 3 shows predefined functions. They are prefixed with the string `nf_`, an abbreviation for *native function*. The LED functions take arguments whose values are between 0 and 32. The top and bottom LEDs display colors. The arguments for primary and secondary colors are:

Color	Red	Green	Blue
White	32	32	32
Red	32	0	0
Green	0	32	0
Blue	0	0	32
Yellow	32	32	0
Magenta	32	0	32
Cyan	0	32	32
(off)	0	0	0

Other colors can be specified using values other than 0 and 32.

The LEDs can also be set by assigning lists to variables:

```
leds_top    = [r, g, b]
leds_circle = [i0, i1, i2, i3, i4, i5, i6, i7]
```

6.4 Math functions

Table 4 lists predefined mathematical functions. Many of them have a version prefixed with `nf_` which can be used on lists (arrays) of integers [2]. Arguments are 16-bit signed integers as are the returned values.

Variable	Meaning	Values
`button_center`	State of the center button	True = touched
`button_forward`	State of the forward button	False = released
`button_backward`	State of the backward button	
`button_left`	State of the left button	
`button_right`	State of the right button	
`prox_horizontal`	Light reflected to the horizontal sensors An array with 7 elements	$\sim (0, 4000)$
`prox_ ground_delta`	Light reflected to the ground sensors An array with 2 elements	$\sim (0, 400)$
`acc`	Acceleration along an axis	$(-32, 32)$
`acc[0]`	x-axis (roll)	left is $+$, right is $-$
`acc[1]`	y-axis (pitch)	back is $+$, front is $-$
`acc[2]`	z-axis	down is $+$, up is $-$
`motor_ left_target`	Set power of left motor	$(-500, 500)$
`motor_ right_target`	Set power of left motor	
`timer_period[0]`	Set period of timer 0	Milliseconds
`timer_period[1]`	Set period of timer 1	

Table 2: Predefined variables

6.5 Calculating with integers

Suppose that you have set the variable `motor` to a power level and wish to reduce it by 25%. The computation in ‡2 results in an error because floating-point numbers are not supported. The computation in ‡3 is not correct because it first performs the integer division resulting in 0 and therefore `motor1` will also receive the value 0. The correct way of performing the calculation is shown in ‡4. By first performing the multiplication, we obtain 1350, which, when divided by 4, results in 337, very close to the exact value 337.5.

16-bit integer computations

```
1 motor = 450
2 motor1 = motor * 0.75
3 motor1 = motor1 * (3//4)
4 motor1 = (motor*3)//4
5 motor1 = math_muldiv(motor,3,4)
```

Be careful not to overflow 16-bit computations. $12000 \cdot 3 = 36000$ is too large to fit in a 16-bit integer. The value stored represents the negative integer -22760. The math function `math_muldiv` ‡5 performs the intermediate computation `a*b` using 32-bit integers without overflow.

Function	Action
`nf_leds_top(r,g,b)`	Set top LEDs to (r, g, b) color
`nf_leds_bottom_left(r,g,b)`	Set bottom left LEDs to (r, g, b) color
`nf_leds_bottom_right(r,g,b)`	Set bottom right LEDs to (r, g, b) color
`nf_leds_circle(` `i0,i1,i2,i3,i4,i5,i6,i7)`	Set circle LEDs $n = 0..7$ to intensity i_n
`nf_sound_freq(f,d)`	Plays sound of f hertz for $d/60$ seconds $d = 0$ continuous sound $d = -1$ stop sound

Table 3: Predefined functions

Function	Returned value
`math_rand()`	Random
`math_muldiv(a,b,c)`	$(a * b)//c$
`math_sqrt(a)`	$\sqrt{a}$
`math_sin(a)`	$\sin a$
`math_cos(a)`	$\cos a$

Table 4: Math functions

The arguments of the trigonometric functions are 16-bit integers representing radians $(-\pi, \pi)$:

Radians	$-\pi$	$-\pi/2$	$-\pi/4$	0	$\pi/4$	$\pi/2$	π
Integer	-32768	-16384	-8192	0	8192	16384	32767

The results of the trigonometric functions are 16-bit integers representing $(-1.0, 1.0)$:

Radians	$-\pi$	$-\pi/2$	$-\pi/4$	0	$\pi/4$	$\pi/2$	π
sin(rad)	0	-1.0	$-\sqrt{2}/2$	3	$\sqrt{2}/2$	1.0	0
sin(int)	0	-32767	-23171	0	23171	32767	0
cos(rad)	-1.0	0	$\sqrt{2}/2$	1.0	$\sqrt{2}/2$	0	-1.0
cos(int)	-32767	0	23171	32767	23171	0	-32767

An important advantage of programming the Thymio in Python is that you can develop and test computations offline using the "regular" Python interpreter. For example, suppose that you want to specify a scaling factor for the computation of distance from motor power and time. The following Python statements can be run without a robot (use the commands from the Thonny menu Run):

```
motor   = 300 # motor power
time    = 100 # time between samples in milliseconds
samples = 6   # number of samples
scale   = 600 # set to a value that gives distance in cm

print(motor * time)
print((motor * time) // scale)
print(((motor * time) // scale) * samples)
```

The computations are performed one-by-one to check for overflow.

7 Displaying the variables

It is often helpful to display the values of Thymio's variables explicitly. For example, you may want to observe the precise values returned by the sensors in order to calibrate an algorithm. All the variables can be displayed by running the program `tdmclient.tools.gui` [4]. Normally you will just want to examine a few variables such as the sensors. This can be done by running a small program which prints the values of selected variables:

Displaying variables

```
timer_period[1] = 500

def print_horizontal():
    print('prox_horizontal',
          prox_horizontal[0], prox_horizontal[1],
          prox_horizontal[2], prox_horizontal[3],
          prox_horizontal[4],
          prox_horizontal[5], prox_horizontal[6])
def print_ground():
    print('prox_ground_delta',
          prox_ground_delta[0], prox_ground_delta[1])
def print_accelerometers():
    print('acc: roll = ', acc[0],
          'pitch = ', acc[1],
          'vertical = ', acc[2])

@onevent
def timer1():
    print_horizontal()
    print_ground()
    # print_accelerometers()
```

Set one of the timers to a duration (in ‡1 it is set to 500ms which is 0.5s) so that it is easy to observe the values as they are printed. Define functions ‡3-15 to print the variables you have selected. Finally, a timer event handler ‡17-21 calls these functions at the specified interval. If you only need to display some of these variables you can comment-out the others as in ‡21.

You can add these statements to your own programs assuming that you use only one timer.

References

[1] Mordechai Ben-Ari and Francesco Mondada. *Elements of Robotics*. Springer, 2013. http://www.springer.com/gp/book/9783319625331.

[2] Michael Bonani. Native functions standard library. http://wiki.thymio.org/en:asebastdnative.

[3] Michael Bonani. Programming interface. http://wiki.thymio.org/en:thymioapi.

[4] Yves Piguet. tdmclient project description. https://pypi.org/project/tdmclient/.