INTERACTIVE ROBOTICS

Manual Advanced Programming

This guide will help you get started with programming robots in Python. We wish you a lot of fun coding, and we are very curious about what cool programs you will make!

Table of Contents:

INTERACTIVE ROBOTICS

Preparation

Before we can get started with programming, we must first prepare your computer to be able to connect to your robots. During the preparation and especially afterwards, we will use some terms that need a little more explanation:

- **Realm (environment):** when we talk about a realm, we are talking about a kind of closed environment in which the robot resides. Consider, for example, a street with houses. Each house has its own address. For example, if you want to send a letter to number 17, write the street where he or she lives and the house number on the envelope. In a way, this house number corresponds to the realm. If you want to communicate with the robot, use its house number (realm) to send messages to it.
- **Session:** Through this session the robot can communicate with you and you can give the robot commands.

In this document we show sample codes for Python.

```
print ("This is Python code")
# This line is commented and will not be executed
```

Python installation

When you want to get started with Python, there are a few steps you need to do. These steps may vary slightly depending on the operating system. For this step-by-step plan we assume a Windows operating system:

1. Make sure that the robot you want to work with is turned on;
2. Open a command prompt and enter the command: python. If **Python 3.6 or higher is** already installed, you will see the version number in the text after your command. If Python is not installed yet or you are using an older Python version, you can download Python from the official website: https://www.python.org/downloads/
3. The next step is that we download and install pip on your computer. Pip ensures that we will soon be able to download other Python packages that we need to be able to connect to the robot:
    a. Open a new command prompt check if pip is installed with the following command:

    ```
    pip3 help
    ```

    If pip is already installed, you will get an explanation how you have to use pip and you can go directly to step 3. If the command prompt says that the command is not recognized, you should proceed to step 3.b;

    b. Download the pip installation script: https://bootstrap.pypa.io/get-pip.py;

c. Open a new command prompt and use the command cd to go to the place where you downloaded the get-pip.py file. If it is in your Downloads folder, the command to run is:

```
cd Downloads
```

d. In your command prompt type the following to install pip:

```
python get-pip.py
```

e. If the command prompt says they can't install pip because you don't have the have rights to it. Then perform steps 3.c and 3.d again, but this time start the command prompt as administrator;

f. Check if pip has been installed successfully with the following command:

```
pip3 help
```

If pip has been installed successfully, you will get an explanation how to use pip.

4. To work with python on the robot, you need to install Autobahn and OpenSSL via pip:

```
pip3 install autobahn[twisted,serialization] pyopenssl
```

5. To install extra functions available for alpha-mini run:

```
pip3 install alpha_mini_rug
```

6. Create a new file called demo.py in a convenient place on your computer;

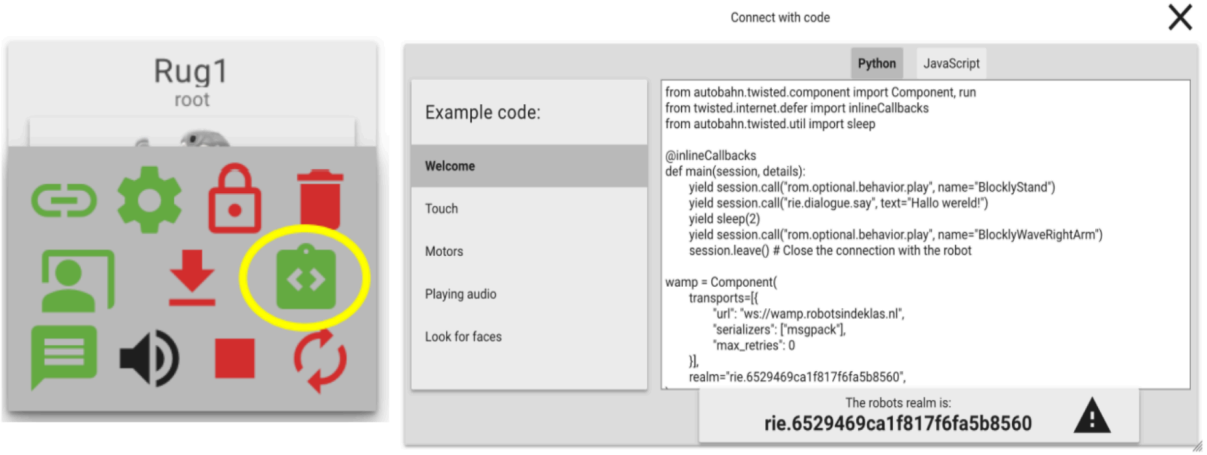
7. Open this file in your favorite Python editor by double clicking on it;
8. In the Python editor, add the following lines:

```
from autobahn.twisted.component import Component,
run
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep

@inlineCallbacks
def main(session, details):
    yield session.call("rie.dialogue.say",
        text="Hello, I am successfully connected!")
    session.leave() # Close the connection with the robot

# Create wamp connection
wamp = Component(
    transports=[{
      "url": "ws://wamp.robotsindeklas.nl",
      "serializers": ["msgpack"]
    }],
    realm="ENTER YOUR REALM HERE",
)
wamp.on_join(main)

if __name__ == "__main__":
    run([wamp])
```

Note: if you have the code above directly copy and paste in your Python editor, then you have to manually add tabs yourself, for example `yield session.call`. This prevents errors such as '`IndentationError: expected an indented block`'.

Note: You may receive a `msgpack` error, in which case you need to install the msgpack package in python (`pip3 install msgpack`).

9. Fill in the realm of the robot in the piece of text that says “ENTER YOUR REALM HERE”. You can find this realm when you go to the robot page in portal.robotsindeklas.nl or portal.robotsindezorg.nl. When the robot is connected to the server, a green check mark will become visible in the top-right corner. Click on its hamburger menu, and a menu with icons will appear as in the picture below. Then, you click on the green <>-icon (circled) and a popup will appear with some example code and the realm at the bottom of the popup screen (see picture below). An example realm: rie.6529469ca1f817f6fa5b8560;



10. Run demo.py by clicking Run;
11. If everything is set up correctly, the robot will say "Hello, I have connected successfully!". If the robot does not say anything, check that the robot is online and that you have copied the realm in its entirety. If the robot still doesn't say anything and you see the error message: `TLS failure: certificate verify failed`, it means that `pyopenssl` is not installed yet. Do step 4 again.

Robot

With our platform it is possible to control robots and design interactions in a smart and easy way. For example, with just a few lines of code, you can have the robot wait until you see someone standing in front of the robot, and then have the robot greet that person. In this chapter we explain step by step how connecting and calling the robot works.

Connecting

You need to perform two steps to communicate with the robot. The first step is to pause the robot so that the default program running on the robot does not interfere with your program. You can pause this program by going to the robots overview and clicking on the hamburger menu. Then click on the pause button and the robot will say 'agent is paused'. Now that this piece of program is paused, you can play your program by starting a connection with the robot. Python requires the following piece of code to connect:

```
from autobahn.twisted.component import Component,
run from twisted.internet.defer import
inlineCallbacks

@inlineCallbacks
def main(session, details):
    # PUT YOUR CODE HERE

# Create wamp connection
wamp = Component(
    transports=[{
       "url": "ws://wamp.robotsindeklas.nl",
       "serializers": ["msgpack"]
    }],
    realm="ENTER YOUR REALM HERE",
)
wamp.on_join(main)

if __name__ == "__main__":
    run([wamp])
```

Once you have a successful connection, the function `main` is called for Python.

Giving Commands

After you have successfully connected to the robot, you can give the robot commands to perform certain tasks. For example, you can ask the robot to start waving its right arm. Once you are connected, you have access to a piece of code called `session` . `session` is, as it were, the piece of code we need to communicate with the robot. For example, we can invoke a certain command with `session`. We do this by executing a `session call`, as shown below:

```
session.call(...)
```

When we call this, the robot knows that it has to do something, but it does not yet know what to do. To give the robot instructions on what to do, we also have to fill in the part of the three dots. So first we have to tell the robot what the command is. We also call a command for a robot an RPC [1]. An RPC is a complicated term for a command, so you basically say with it: "Hey robot, I want you to perform a movement".

Now the robot knows: 'Okay, I have been instructed to move, but I don't know yet which movement to perform'. Now it is up to us to tell the robot exactly which movement it should perform. For this we use arguments that tell the robot exactly what we want. For example, for the command to wave the right arm, we give the robot the following command:

[1] Remote Procedure Call: de technische term voor de commando die we willen dat de robot uitvoert.

```
session.call("rom.optional.behavior.play",
             name="BlocklyWaveRightArm")
```

Now the robot has all the information it needs to perform the command and will go with its right arm to wave. In this way you can give the robot commands to perform certain tasks. In the first part of the call you give the assignment (the RPC) and in the second part you clarify your assignment. The second part is not always necessary, there are assignments where the first part is already clear enough for the robot. Just take a look at our Sample Code and try to discover for yourself which commands are immediately clear to the robot and which commands require additional information.

Listening to sensors

Besides being able to give the robot all kinds of commands, we can also start using the sensors that the robot has. Sensors are small, smart components on the robot. For example, a robot can look around with its camera and wait until it sees a face. The camera on the robot is an example of a sensor. Another example of a type of sensor are the touch sensors on the robot. Suppose we would like to know whether the head of the robot has been touched. To find out, we have to take two steps. First of all, we want to have a piece of code that is called every time the robot says new sensor data is available. For this we use the `session` again. We then sign up to let us know that we would like to be kept informed as soon as new sensor data becomes available. We do this by the `session.subscribe` command:

```
session.subscribe(...)
```

The next step is to tell the robot which sensor data we are interested in. We do this by telling `subscribe` which topic is of interest to us. A topic is a kind of TV channel on which only one channel can be broadcast. So we tell the robot we would like to see this transmitter. Then, of course, we also want to do something with this data and for this we give `subscribe` also a function that is always called as soon as new data becomes available.

```
def touched(frame):
  print(frame)

session.subscribe(touched, "rom.sensor.touch.stream")
```

We have now signed up for touch data. So as soon as the head is touched, then the function `touched` is called. We now only have to take one step before we actually receive the information. We have to tell the robot that we are logged in and that we would like to collect the information now. We do this by making a `session.call` that tells the robot: “Hey robot, I would like you to forward all your touch data to your touch channel”. Below is a

complete example of how to do this Python:

```
def touched(frame):
  print(frame)

session.subscribe(touched, "rom.sensor.touch.stream")
session.call("rom.sensor.touch.stream")
```

Pay particular attention to the difference in order of commands in the `session.subscribe`. In Python, you first tell which function to call and then you tell which channel we want to sign up to.

In the appendix you will find a few more extensive examples for Python on how to get started with giving commands and listening to new data.

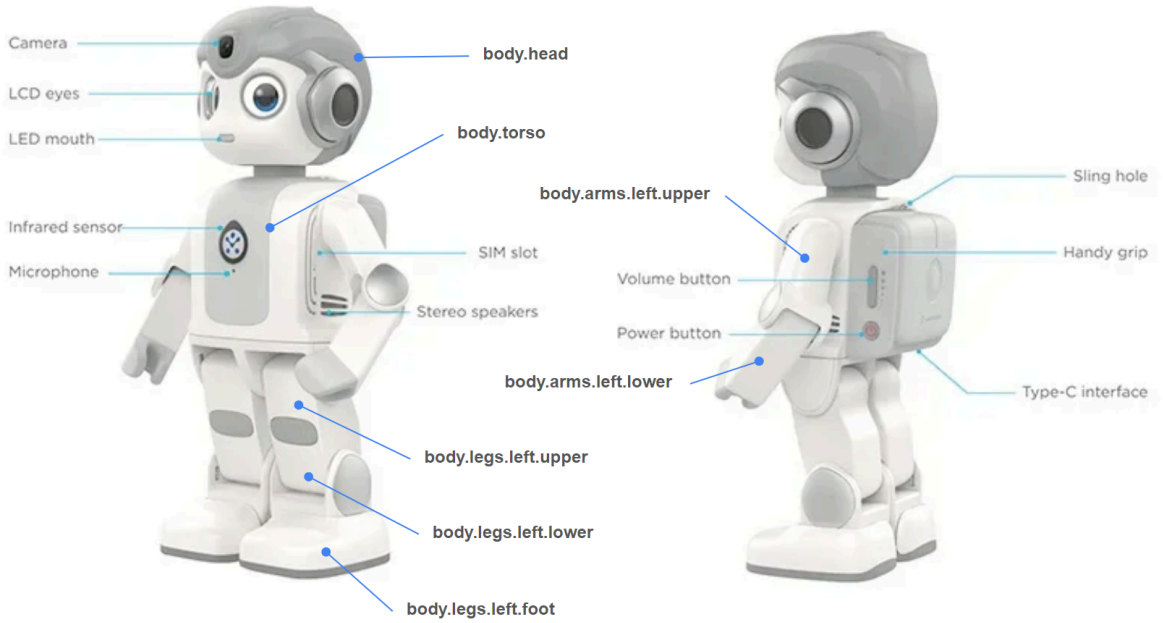
INTERACTIVE ROBOTICS

Body parts (UBI)

In order to be able to use the body parts of the robot, our platform uses a system called UBIs (Uniform Body Identifiers). With these UBIs you can, for example, indicate that you want to move the left arm. UBIs use hierarchical notation and always start with **body**. The subsequent body part is separated with a point. So for example if you want to use the left hand, you use the UBI: `body.arms.left.upper`. See the image below for examples.

The tree structure of body parts (UBI's):

```
body
 | - head
 | - arms
 | | - right
 | | | - upper
 | | | - lower
 | | - left
 | | | - upper
 | | | - lower
 | - torso
 | - legs
 | | - right
 | | | - upper
 | | | - lower
 | | | - foot
 | | - left
 | | | - upper
 | | | - lower
 | | | - foot
```



Sample Code

For inspiration on how to get started programming the robots, we'll give you a few examples in this chapter. We'll start with a simple example of a Christmas radio playing on the robot. The code examples then become increasingly difficult. If this is your first time getting started programming in Python, we definitely recommend copying and playing this code. Do not forget to adjust indentations if necessary and to update the realm of the examples, because it is different for every robot.

Stream music

With part of the actuator modality you can play music on the robot. This way you can make a cool interaction while the robot plays sound from a radio. You start the radio stream by `calling rom.actuator.audio.stream` and in the parameter `url` you tell the robot where to get the music from. In this example it starts for example a Christmas radio station :-). The robot will then load the stream and you can stop it by pressing a button on your keyboard. As soon as you press the keyboard, we tell the robot to stop playing the music by `calling rom.actuator.audio.stop .`

```
from autobahn.twisted.component import Component,
run from twisted.internet.defer import
inlineCallbacks

@inlineCallbacks
def main(session, details):
   yield session.call("rom.actuator.audio.stream",
               url="https://stream.qmusic.nl/qmusic/mp3",
               sync=False
          )

    yield sleep(30)
    print("Loading the radio may take a while")
    print("Press your keyboard to stop the music!")
    input()
    yield sleep(30)
    yield session.call("rom.actuator.audio.stop")
    session.leave() # Close the connection with the robot

# Create wamp connection
wamp = Component(
    transports=[{
       "url": "wss://wamp.robotsindeklas.nl",
       "serializers": ["msgpack"]
    }],
    realm="rie.5e1312363dbf49eed032e123",
)
wamp.on_join(main)

if __name__ == "__main__":
    run([wamp])
```

Yes-nods

With the actuator modality we can send commands to the robot to move its arms and head. The sample code below shows you how to make a robot's head nod yes with a few lines of code. All you have to do is **perform_movement(session, frames)** and tell them what the movement should look like. In this example, the robot only needs to make a nodding movement with its head and the entire movement takes 2400 milliseconds (= 2.4 seconds).

```
from autobahn.twisted.component import Component, run
from twisted.internet.defer import inlineCallbacks
from alpha_mini_rug import perform_movement

@inlineCallbacks
def main(session, details):
    # Nod yes
    perform_movement(session,
       frames=[{"time": 400, "data": {"body.head.pitch": 0.1}},
               {"time": 1200, "data": {"body.head.pitch": -0.1}},
               {"time": 2000, "data": {"body.head.pitch": 0.1}},
               {"time": 2400, "data": {"body.head.pitch": 0.0}}],
       force=True)
    session.leave() # Close the connection with the robot

# Create wamp connection
wamp = Component(
    transports=[{
       "url": "wss://wamp.robotsindeklas.nl",
       "serializers": ["msgpack"]
    }],
    realm="rie.5e1312363dbf49eed032e123",
)
wamp.on_join(main)

if __name__ == "__main__":
    run([wamp])
```

Responding to touch

Besides being able to play Christmas music, we can also wait for someone to press the robot's head. Below is an example of how the robot can react when someone touches the robot's head. The first thing to do is make sure you are subscribed to a topic where all touch data will be sent to. Then you start with `rom.sensor.touch.stream`, a stream of touch data. Every time someone touches one of the touch sensors, the function is `touched` and you can see in that function which sensor has been touched. Depending on your application, you can have the robot say or do something when someone touches the head. In the example below we print that the head has been touched as soon as someone has touched the head sensors. You do not need to include all parts of the head *(front, middle* and *rear*) for the code to work or for the sensor to react, however, it is helpful since the difference between them is very subtle in the AlphaMini, and it increases the surface for the interaction to work.

```
from autobahn.twisted.component import Component, run
from twisted.internet.defer import inlineCallbacks

def touched(frame):
    if ("body.head.front" in frame["data"] or
            "body.head.middle" in frame["data"] or
            "body.head.rear" in frame["data"]):
        print("Head has been touched!")

@inlineCallbacks
def main(session, details):
    yield session.subscribe(touched, "rom.sensor.touch.stream")
    yield session.call("rom.sensor.touch.stream")

# Create wamp connection
wamp = Component(
    transports=[{
        "url": "wss://wamp.robotsindeklas.nl",
        "serializers": ["msgpack"]
    }],
    realm="rie.5e1312363dbf49eed032e123",
)
wamp.on_join(main)

if __name__ == "__main__":
    run([wamp])
```

Find and follow faces

If a robot has a camera, then we can use it to find a face and then also follow the face. So suppose you sit diagonally opposite the robot and you look at the robot, then with the example code below you let the robot first look for your face and as soon as the robot sees you, it will try to follow you with its head.

The first thing we say to the robot is to get it up, and we do that by `calling rom.optional.behavior.play` and saying we want to get up (`name = "BlocklyStand"`). Then the robot gets up and we tell the robot to look for faces. We do this by `calling rie.vision.face.find` . The robot then starts looking around until it sees a face and looks straight at it. When the robot looks straight at you, we greet you by saying something. Then we tell the robot to follow your face until the robot no longer sees you. We do this by `calling rie.vision.face.track` . As soon as we no longer see a face, let the robot say something and then let it sit.

```
from autobahn.twisted.component import Component, run
from twisted.internet.defer import inlineCallbacks

@inlineCallbacks
def main(session, details):
    yield session.call("rom.optional.behavior.play",
        name="BlocklyStand")
    yield session.call("rie.vision.face.find")
    yield session.call("rie.dialogue.say", text="Hi!")
    yield session.call("rie.vision.face.track")
    yield session.call("rie.dialogue.say",
        text="I don't see you anymore!")
    yield session.call("rom.optional.behavior.play",
        name="BlocklyCrouch")
    session.leave() # Close the connection with the robot

# Create wamp connection
wamp = Component(
    transports=[{
       "url": "wss://wamp.robotsindeklas.nl",
       "serializers": ["msgpack"]
    }],
    realm="rie.5e1312363dbf49eed032e123",
)
wamp.on_join(main)

if __name__ == "__main__":
    run([wamp])
```

Important tips

Yield

In the example codes above you have probably seen that we use a lot of `yield` commands. With this you tell your computer that you want to wait for a result or until a certain action has been performed before moving on to the next statement. For example, if we remove the `yield` from `yield session.call("rie.dialogue.say", text = "Hi!")`, we no longer wait for the robot to finish talking, but the code continues executing the lines below immediately. We recommend, especially if this is your first time working with this kind of code, to always use `yield` before `session.call` and `session.subscribe`. **Note:** in functions that use `yield`, for Python you need to place `@inlineCallback` above the function declaration.

Subscribe

Before you open a stream on a modality, you must first have subscribed (subscribe) to this topic. If you reverse the order, so first open the stream and then subscribe, there is a chance that our robot will think that there are no subscriptions and it will automatically close the topic

again.

Wait

If you want to wait a certain amount of time in the code, it is important that you **do not** use the Python standard `time.sleep()`! Instead, you should use `sleep()` function from `autobahn.twisted.util`. An example of how you should implement this can be found below.

```
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep

@inlineCallbacks
def main(session, details):
    yield session.call("rie.dialogue.say",
        text="I'll now wait 20 seconds")

    # Wait 20 seconds
    yield sleep(20)

    yield session.call("rie.dialogue.say",
        text="I'm done waiting!")
```

ROM API

The ROM API is a piece of code that makes all basic functionality of a robot available to us. ROM stands for Robot Operating Module and therefore ensures that the robotto our platform can connect so that you can get started programming blocks or writing real code. A ROM generally consists of two parts, namely:

- Sensors: for retrieving data such as camera images and touch sensors.

- Actuators: for controlling the robot. Consider, for example, direct control of the motors or playing sound;

Frames

If you are going to use our programming environment, you will probably see the word 'frame' often. A frame keeps us in a piece of information that describes data in a standard way. This information can tell you, for example, whether someone has pressed one of the main buttons of Nao or you can tell in a frame how the robot must control its motors to nod yes. A single frame consists of a `time` and `data` key. The `time` key indicates the time in milliseconds when something has happened or how long something should last.in the `data` UBI's are storedkey with the associated values. This value can be a sensor value of touch sensors or the angle that the motor should reach.

```
{
    time: time,
    data: {
        ubi1: value,
        ubi2: value,
        ubix: value
    }
}
```

Remote Procedure Calls

Remote Procedure Calls (or RPCs) refers to the specific calls that you can use to read or listen to sensors or control actuators. In essence, they are functions to operate with a sensor or actuator. Examples include:

- `rom.sensor.sight.read`
- `rom.actuator.motor.write`
- `Rom.optional.behavior.play`
- `rie.dialog.say`

Sensors

Sensors are smart, small components on the robot that can tell something about the robot's environment. For example, all robots from Robots in the Classroom have a camera with which they can look around. They also have a microphone with which they can listen. In total, our platform supports four different sensors:

1. Sight - Vision: the camera images of a robot;
2. Hearing - Hearing: the audio from a microphone;
3. Touch - Touch: the touch sensitive sensors on a robot;
4. Proprio - Proprioception: the reading of all the angles that the motors of a robot make.

Each sensor has the same implementation:

RPC	**Returns**
rom.sensor. <modality> .info: request information from a modality	A dictionary containing information about the specific modality.

rom.sensor. <modality> .read: listen for a certain time to all data collected for this modality Arguments: - ubi: (List <String>, default = [ubi]): A list of UBI's where you have the data you want for this read. - time: (Double, default = 0.0): the time for how long you want the read to run in milliseconds, at time = 0.0 a single frame is returned.	The data collected from the specific modality and UBI (s) over the preset time (time).
rom.sensor. <modality> .stream: open a stream on which data is directly posted to the topic `rom.sensor. <modality> .stream` as soon as it is available. It is important that you first subscribe before calling the stream. Argument: - ubi: (List <String>, default = [ubi]): A list of UBIs you want to listen to. If you leave this empty, the modality will send all available data.	None
rom.sensor. <modality> .close: close a stream after you no longer need this data. Argument: - ubi: (List <String>, default = [ubi]): A list of UBIs you no longer want to listen to.	None
rom.sensor. <modality> .sensitivity: the sensitivity indicates when the difference between two data points is large enough to be transmitted. Argument: - sensitivity: (Double, default = None): the sensitivity to which you want to set the modality.	A double of the sensitivity the robot is currently on for the selected modality

An example of how to collect sensor data via the `stream` and `read` RPC in Python:

```
from twisted.internet.defer import inlineCallbacks

def proprio(frame):
  #This function is called every time the robot
  #moves
  print(frame)

@inlineCallbacks
def main(session, details):
  # Get the angles of all engines at this time
  frames = yield session.call("rom.sensor.proprio.read")
  print("Current engines status:")
  print(frames[0]["data"])
  print("")

  # Here we register for the proprio data and start a stream
  yield session.subscribe(proprio,
  "rom.sensor.proprio.stream") yield
  session.call("rom.sensor.proprio.stream")

  # Now we let the robot move a bit so that we get proprio data
  #from our rom.sensor.proprio.stream stream
  yield session.call("rom.optional.behavior.play",
    name="BlocklyStand")
  yield session.call("rom.optional.behavior.play",
    name="BlocklyWaveRightArm")
  yield session.call("rom.optional.behavior.play",
    name="BlocklyCrouch")
  session.leave() # Close the connection with the robot
```

Actuators

Actuators are parts of the robot that allow the robot to do something. Think, for example, of playing music or waving your arms. Our platform supports four standard actuators that each robot has implemented, namely:

1. Motor
2. Audio
3. Behavior

An overview of the actuator modality:

Motor

`rom.actuator.motor.info`

~~`rom.actuator.motor.write`~~

`rom.actuator.motor.stop`

Audio

`rom.actuator.audio.info`

`rom.actuator.audio.volume`

`rom.actuator.audio.play`

`rom.actuator.audio.stream`

`rom.actuator.audio.stop`

Behavior (Behavior)

`rom.optional.behavior.info`

`Rom.optional.behavior.play`

`rom.optional.behavior.stop`

Motor

With the Motor modality, you can directly control the motors and create your own movements. To control the motors we use a coordinate system to determine that, for example, a counterclockwise movement is positive and clockwise is negative. More information about what this coordinate system looks like and how you can get started with it can be found here. A good example of how to create your own movements can be seen in our Sample Code chapter where as an example we let the robot nod yes. Please note that all values for the motors are in **radians**. For a visual description of the motor joints and ranges of value, see the appendix.

RPC	Returns
rom.actuator.motor.info: request the information of the motor modality	A dictionary containing information about the Motor modality.
perform_movement(session, frames, mode="linear", sync=True, force=False): This function performs a movement with the robot's joints. The time of each frame is calculated based on the proportional time of the movement. Arguments: `session` (Component): The session object. `frames` (list): A list of dictionaries with the time and data of the joints to be moved. `mode` (str): The mode of the movement. Choose one of the following: "linear", "last", "none". `sync` (bool): A flag to wait for the robot to execute all motor commands. `force` (bool): A flag to force the joints to get as close as possible to the desired motor position even though the motor cannot reach the exact position due to hardware limitations. If False, you assume the robot can move to the desired position. For extra information, you can also have a look at rom.actuator.motor.write	Does not return anything. Performs the requested move.

rom.actuator.motor.write: directs a motor or several motors to the desired position Arguments: - frames: (List <Frame>): a list containing frames that tell which orientations the motors have to go and how much time to do so to have; - mode: (String, default = 'linear'): With the mode you can tell the robot how to move between the different frames to the next position. The modality supports the following three modes: 'linear', 'last' and 'none'. In the appendix you can find a detailed explanation of what these modes do between the frames. In most cases the default value 'linear' is sufficient; - sync: (Bool, default = True): if True, wait for the robot to execute all motor commands; - force: (Bool, default = False): if True, try to get as close as possible to the desired motor position even though the motor cannot reach the exact position due to hardware limitations. If False, then you assume that the robot can move to the desired position.	None, but if sync = True, you can wait for the RPC until the robot has finished executing the motor commands. This RPC can return an Error if your force is set to False and you try to move to a position that the robot cannot reach. For example, if you want the head to rotate 360 degrees and the head can only rotate 180 degrees, this RPC will give an error at force = False and the robot will go to 180 degrees at force = True.
rom.actuator.motor.stop: stop the running `rom.actuator.motor.write` RPC. Please note that the robot may end up in an unstable position, which could cause it to fall over.	None

Audio

With the Audio modality you can play and stream sounds and music on the robot. For example, you can play a radio station on the robot while the robot is dancing. In our Sample Code chapter we have an example showing how to play a music stream on the robot.

RPC	Returns
rom.actuator.audio.info: request the information of the Audio modality	A dictionary containing information about the Audio modality.
rom.actuator.audio.volume: adjust the volume and / or get the current volume back. Argument: - volume: (Int): the volume is between a value of 0 (no sound) and 100 (maximum volume).	The volume the robot is now at

rom.actuator.audio.play: directly play a raw stereo wave audio file. Arguments: - data: (byte []): the raw stereo wave audio data; - rate: (int, default = 44100): the sample rate of the audio that will be sent; - sync: (Bool, default = True): if True, wait for the sound to finish playing.	None, but if sync = True then you can wait for the RPC until the robot has finished playing the sound
rom.actuator.audio.stream: stream play sound from a web. Arguments: - url: (String): a link to a (web) location from which audio is streamed. Important is that Nao and Pepper only support http streams and the virtual robot only https; - sync: (Bool, default = False): submits True, wait for the stream to finish playing the sound.	None, but if sync = True, then you can wait for the RPC until the robot finishes playing the sound
rom.actuator.audio.stop: stop all audio playing	None

Behavior

With the Behavior (Behavior) modality, it is possible to have the robot perform pre-programmed movements. For example, it is possible to make the robot swing in a simple way with the behavior: BlocklyWaveRightArm.

RPC	**Returns**
rom.optional.behavior.info: request the information from the Behavior modality.	A dictionary containing a list of behaviors that can be played.
rom.optional.behavior.play: play a behavior on the robot. Arguments: - name: (String, default = ''): the name of the behavior, you can request a list of behaviors by first `calling rom.optional.behavior.info` ; - sync: (Bool, default = True): if True, this RPC waits for the robot to finish performing its movement.	None, but if sync = True then you can wait for the RPC until the robot has finished executing the movement.

rom.optional.behavior.stop: stop the `rom.optional.behavior.play`. Please note that the robot may end up in an unstable position, which could cause it to fall over.	None

In the appendix you can find a list of all default behaviors installed on the robot.
An example of how to get the robot to get up. Then let them say something while the robot waves. And as a last step, let the robot sit again. In Python:

```
# Have the robot stand up
yield session.call("rom.optional.behavior.play",
    name="BlocklyStand")

# Have the robot say something while it waves
behavior =
    session.call("rom.optional.behavior.play",
    name="BlocklyWaveRightArm")
yield session.call("rie.dialogue.say",
text="Hello!") yield behavior

# Let the robot sit again
yield session.call("rom.optional.behavior.play",
    name='BlocklyCrouch')
```

Data

With the Data modality it is possible to retrieve all information from all implemented modalities in one storage.

The RPC is `rom.data.info` and returns, for example:

```
{
    data: {
        serial: "abcd0123456789",
        model: "alpha-mini",
        rom_version: 11
        firmware: "Alpha_Mini_outEdu-V1.3.0.610"
    },
    sensor: {
        sight: {...},
        hearing: {...},
        touch: {...},
        proprio: {...}
    },
    actuator: {
        light: {},
        audio: {},
        motor: {}
    },
    optional: {...}
}
```

Besides the info of all modalities, you can also request the current status with the RPC:with the Data modality `rom.data.status`. This RPC returns the following:

```
data: {
    battery: 50,
    network: "network_name"
}
```

RIE API

Besides the basic functionalities of the robot, we can also make the robot do advanced things. For example, we can use the advanced functionalities to search for faces and have the robot say something. We call this piece of advanced functionality RIE [2].

[2] Robot Interaction Engine: een geavanceerd stukje code waarmee we de robot slimme dingen mee kunnen laten doen.

An overview of all:

Dialogue:	**Vision :**

`rie.dialogue.say`
`rie.dialogue.say_animated`
`rie.dialogue.config.native_voice`
`rie.dialogue.config.language`
`rie.dialogue.config.profanity`
`rie.dialogue.ask`
`rie.dialogue.stop`
~~`rie.dialogue.stt.info`~~
~~`rie.dialogue.stt.read`~~
~~`rie.dialogue.stt.stream`~~
~~`rie.dialogue.stt.close`~~
`rie.dialogue.keyword.add`
`rie.dialogue.keyword.remove`
`rie.dialogue.keyword.clear`
`rie.dialogue.keyword.stream`
`rie.dialogue.keyword.close`
`rie.dialogue.keyword.read`
	`rie.vision.card.info`
	`rie.vision.card.stream`
	`rie.vision.card.read`
	`rie.vision.card.close`
	`rie.vision.face.info`
	`rie.vision.face.stream`
	`rie.vision.face.read`
	`rie.vision.face.close`
	`rie.vision.face.find`
	`rie.vision.face.track`
	`rie.vision.face.stop`

Dialogue

The dialogue module is designed to allow interaction with a user through speech. For example, through this module we can have the robots say something, but the robot can also listen to the user.

Text-to-Speech (TTS)

With Text-to-Speech (TTS) you can make the robot say something. For example, with TTS you can greet a user when the robot sees someone or give feedback on a question from the user. The TTS module can use two voices depending on whether the robot itself can talk. The first voice is from the robot itself and the second voice is from ReadSpeaker. Some robots, such as the Alpha-Mini, have no voice of their own and only support the voice from ReadSpeaker.

RPC	**Returns**

rie.dialogue.say: makes the robot say something without performing any movements. Arguments: - text (String): the text the robot should say; - lang (String, Default = None): The language in which the robot should say the text. This can be left blank and then the robot uses the last set language. An example of long is 'nl' en 'en'.	None
rie.dialogue.say_animated Have the robot say something while the robot performs movements. Arguments: - text (String): The text the robot should say; - lang (String, Default = None): The language in which the robot should say the text. This can be left blank and then the robot uses the last set language. An example of long is 'nl' en 'en'.	None
rie.dialogue.config.native_voice: set which speech engine is preferred. Argument: - use_native_voice (Boolean): True → the dialog module tries to use the language already installed on the robot, but falls back to ReadSpeaker if the robot does not have a speech engine or does not support the desired language. False → the robot only uses ReadSpeaker voice.	None
rie.dialogue.config.language: set the language of the dialog module. The language can be set successfully when the speech of the robot or ReadSpeaker supports the desired language. If you leave the variable empty for a long time, the RPC immediately returns which language it currently uses. Argument: - lang (String, Default = None): the language code. Supported formats are: 'en' and 'en_uk'. Language codes ending with a dash are not supported.	String of the language code that the dialog module is currently using.

Example of how to make the robot say something in Python, by default, the language is set to Dutch (the example is translated):

```
# Let the robot say something:
yield session.call("rie.dialogue.say",
    text="Hello, good to see you!")

# Only use the ReadSpeaker TTS engine:

yield session.call("rie.dialogue.config.native_voice",
     use_native_voice=False)

#Say something with the ReadSpeaker voice:
yield session.call("rie.dialogue.say",
     text="This is the ReadSpeaker voice!")

# Change the language to English:
yield session.call("rie.dialogue.config.language", lang="en")


# Say something with the ReadSpeaker voice in English:
yield session.call("rie.dialogue.say",
     text="I am now speaking English!")
```

Speech-to-Text (STT) - Beta version

With Speech-to-Text (STT) you can make the robot hear something you say and save it as text. For speech recognition an extra package is used. Install the package with

```
pip3 install SpeechRecognition
```

RPC	**Returns**
Parameters and flags that can be changed and their corresponding default values: **SpeechToText.silence_time** = 1 - parameter set to indicate the relative time of a silent period (e.g. end of a sentence) - the value 1 represents approximately 1 second - when a silent period was recorded, the audio is being processed **SpeechToText.silence_threshold2** = 100 - any sound recorded below this value is considered a region of silence - A good value for this would be in the range of 100 to 500 **SpeechToText.logging** = False - prints all the debugging output - the information is not well structured yet **SpeechToText.new_words** = False - this flag indicates that you are recording and no words needs to be processed yet	

SpeechToText.do_speech_recognition = True - this flag helps you turn the microphone on and off - if it is set to True, it records; if it set to False, it stops recording and removes the captured sound recording. -	
SpeechToText.loop This function needs to be run for the class to continuously process the speech. It is not needed for the listen split. Continuously processes audio frames if processing is active. This method checks if the processing flag is set to True. If it is, it logs the start of the processor and processes the next audio frame from the to_process_frames list. **Attributes:** **processing (bool):** Flag indicating whether processing is active. **logger (function):** Function to log messages. **process_audio (function):** Function to process audio frames. **to_process_frames (list):** List of audio frames to be processed.	**None**
SpeechToText.listen_continues Captures continuous audio data for speech recognition. This method listens to incoming audio data and processes it for speech recognition. It appends audio frames to a buffer and detects silence to determine when to process the buffered audio frames. Only create one subscriber otherwise the audio processor can not decipher the words. **Args:** **data (dict):** A dictionary containing audio data. The audio data is expected to be in the format data["data"]["body.head"]. **Attributes:** **do_speech_recognition (bool):** A flag indicating whether speech recognition is enabled. **mode_continues (bool):** A flag indicating whether continuous mode is active. **audio_frames (list):** A list to store audio frames. **silence_counter (int):** A counter to track the number of silent audio frames. **silence_threshold2 (int):** A threshold value to determine silence in audio frames. **sample_rate (int):** The sample rate of the audio data. **silence_time (float):** The duration of silence to detect in	

seconds. **processing (bool):** A flag indicating whether audio processing is ongoing. **to_process_frames (list):** A list to store frames to be processed. **Raises:** **Exception:** If there is an error appending audio frames to the buffer.	
SpeechToText.give_me_words Retrieve the list of English words. This method logs the action, sets the `new_words` attribute to False, and returns the list of English words stored in the `english_words` attribute. **Returns:** **list:** A list of English words.	**list: A list of English words.**

Example of how to make the robot hear something in Python, by default, the language is set to English:

```
from autobahn.twisted.component import Component, run
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep
from time import time

import cv2 as cv
import numpy as np
import wave
import os

from alpha_mini_rug.speech_to_text import SpeechToText


audio_processor = SpeechToText() # create an instance of the
class

# changing these values might not be necessary
audio_processor.silence_time = 0.5 # parameter set to indicate
when to stop recording
audio_processor.silence_threshold2 = 100 # any sound recorded
below this value is considered silence

audio_processor.logging = False # set to true if you want to see
all the output


@inlineCallbacks
def STT_continuous(session):
```

```
      info = yield session.call("rom.sensor.hearing.info")
      print(info)

      yield session.call("rom.sensor.hearing.sensitivity", 1650)

      yield session.call("rie.dialogue.config.language",
lang="en")
      yield session.call("rie.dialogue.say", text="Say
something")

      print("listening to audio")
      yield session.subscribe(audio_processor.listen_continues,
"rom.sensor.hearing.stream")
      yield session.call("rom.sensor.hearing.stream")

      while True:
      if not audio_processor.new_words:
            yield sleep(0.5)  # VERY IMPORTANT, OTHERWISE THE
CONNECTION TO THE SERVER MIGHT CRASH
            print("I am recording")
      else:
            word_array = audio_processor.give_me_words()
            print("I'm processing the words")
            print(word_array[-3:])  # print last 3 sentences

      audio_processor.loop()


@inlineCallbacks
def main(session, details):
      # Define the file path
      output_dir = "output"
      output_file = os.path.join(output_dir, "output.wav")

      # Create the directory if it doesn't exist
      os.makedirs(output_dir, exist_ok=True)

      # Create the file if it doesn't exist
      if not os.path.exists(output_file):
      with open(output_file, "wb") as f:
            f.write(b"")
            # Write an empty byte string to create the file

      yield STT_continuous(session)

      session.leave()
```

```
wamp = Component(
      transports=[
      {
            "url": "ws://wamp.robotsindeklas.nl",
            "serializers": ["msgpack"],
            "max_retries": 0,
      }
      ],
      realm="rie.67a1cded85ba37f92bb12d56",
)


wamp.on_join(main)

if __name__ == "__main__":
      run([wamp])
```

Smart question

Another smart trick that we can do with RIE is using a smart question to interact with the user. What you often see happening during interactions is that the robot asks a question, then the user gives an answer and the robot has not heard the user properly. What then happens is that the robot keeps looking at you with a cold look and you as a user do not know whether the robot has not understood you or whether the robot is still processing the text you just said. This creates an uncomfortable situation for a user. What this module does is ask a question to a user and then be smart with the answers that the user says. For example, if the user says, “What did you say?” The robot will say in turn, “I'll repeat the question for you. The question is: ...". Even if the robot was not able to hear the correct answer all at once, the robot will say politely, "Sorry, can you repeat what you just said?" When the smart question module has found an answer match, it sends you the answer back and you can then give a personalized response. See the example below on how to implement the smart question module.

RPC	Returns
rie.dialogue.ask: ask a question in a smart way. Arguments: - question (String): the question the robot should ask; - answers (Dict <List>, default = None): A dictionary that consists of the answer as keys and a list of words that sound like the answer. That list helps our algorithm extract spoken text from audio. An important tip here is to try not to have two answers that sound the same (a homophone like 'we' and 'whey') or are the same but mean something different (a homonym like 'bank'); - max_attempts (Int, default = 4): the maximum amount of attempts the robot will attempt if the user does not understand them; - language (String, Default = None): The language code. Currently only 'nl' en 'en' is supported.	One of the answers of the answers parameter if `rie.dialogue.ask found` a match. In all other cases, it returns None if it could not find a match or the RPC was stopped because someone `rie.dialogue.stop` called.
rie.dialogue.stop: if `rie.dialogue.ask is` active, you can stop the RPC by `calling rie.dialogue.stop .`	None

Example of how to implement the smart question in Python:

```
question = "Which color do you like better, red or blue??"
answers = {"red": ["red", "ret"], "blue": ["blue", "lue"]}

answer = yield session.call("rie.dialogue.ask",
                            question=question,
                            answers=answers)

if answer == "red":
    yield session.call("rie.dialogue.say",
        text="Red is certainly a beautiful colour!")
elif answer == "blue":
    yield session.call("rie.dialogue.say",
        text="I love blue!")
else:
    yield session.call("rie.dialogue.say",
         text="Sorry, but I didn't hear you properly.")
```

Keyword

Keywords are specific words we want to listen to. Consider, for example, a situation where

you ask a closed question, a question to which you know the answer in advance, and you wait until you hear an answer. In that situation, you can use keywords because you know in advance what the answers are.

RPC	**Returns**
rie.dialogue.keyword.info: get the information from the Keyword module.	A dictionary containing information which languages are supported by the Keyword module
rie.dialogue.keyword.add: add a list of new keywords that the robot should listen to. Argument: - keywords: (List <String>, default = None): Add a list of keywords that the robot should listen to.	A list containing the keywords the robot is currently listening to
rie.dialogue.keyword.remove: remove previously added keywords from the list of keywords the robot should listen to Argument: - keywords: (List <String>, default = None): Get a few keywords from the list of what the robot should listen to.	A list containing the keywords that the robot is currently listening to
rie.dialogue.keyword.clear: remove all keywords from the list of keywords the robot should listen to.	None
rie.dialogue.keyword.language: change the language in which we expect the keywords. If the language of the Keyword module is in Dutch, then the module will have great difficulty listening to English words. Argument: - lang: (String, default = None): the language code of the language you want for the Keyword module. Supported formats are: 'en' and 'en_uk'. Language codes ending with a dash are not supported.	The language code of the language the robot is currently using. Can throw a ValueError if you try to set a language that is not supported by the robot
rie.dialogue.keyword.read: listen for a certain time to all data collected for this module. Arguments:	The data collected from the keyword module over the preset time (time).

- ubi: (List <String>, default = [ubi]): listen for a certain time to all data collected for this module; - time: (Double, default = 0.0): the time for how long you want the read to run in milliseconds, at time = 0.0 a single frame is returned.	
rie.dialogue.keyword.stream: Open a stream where data will be directly posted to the topic `rie.dialogue.keyword.stream` as soon as it is available. It is important that you first subscribed before calling the stream. Argument: - ubi: (List <String>, default = [ubi]): a list of UBI's for which you want the data. Usually it is sufficient to omit this parameter.	None
rie.dialogue.keyword.close: close a stream after you no longer need this data. Argument: - ubi: (List <String>, default = [ubi]): a list of ubi's that you no longer want to be subscribed to.	None

Example of how to use the keyword module in Python:

```
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep

@inlineCallbacks
def main(session, details):
 global sess
 sess = session

 def on_keyword(frame):
 global sess
 if ("certainty" in frame["data"]["body"] and
      frame["data"]["body"]["certainty"] > 0.45):
    sess.call("rie.dialogue.say", text= "Yes")

 yield session.call("rie.dialogue.say",
                    text="Say red, blue or green")
 yield session.call("rie.dialogue.keyword.add",
                    keywords=["red", "blue", "green"])
 yield session.subscribe(on_keyword,
                    "rie.dialogue.keyword.stream")
 yield session.call("rie.dialogue.keyword.stream")
```

```
#Please wait 20 seconds before we close the keyword stream
yield sleep(20)

yield session.call("rie.dialogue.keyword.close")
yield session.call("rie.dialogue.keyword.clear")
session.leave() # Close the connection with the robot
```

Vision

With the vision module, we unleash smart algorithms on the camera images of the robot. This allows us, for example, to recognize cards from camera images, and we can also detect and track faces.

Card detection

With the camera images of the robot we can recognize all kinds of cards. To be able to ensure that all robots can recognize the same types of cards, we use markers called Aruco. These markers are comparable to its more famous little brother QR code.

We are currently using 6x6 Aruco Markers and these can be generated from the following website: http://chev.me/arucogen/.

RPC	**Returns**
rie.vision.card.info: request the information from the Card detection module.	A dictionary containing information about the Card detection module.
rie.vision.card.read: listen to all data collected for this module for a specified time. Arguments: - ubi: (List <String>, default = ['body']): a list of UBI's for which you want the data. Usually it is sufficient to omit this parameter; - time: (Int, default = 0): the time this RPC card should record detection in a frame. This time is in milliseconds. If you specify time = 0, this RPC will wait until a new ticket is detected.	The data collected from the Card detection module over the preset time (time). If time = 0, you get one frame back containing the detected card number

rie.vision.card.stream: opens a stream on the topic `rie.vision.card.stream` on which the detected card numbers are immediately placed as soon as a card number is detected . It is important here that you first subscribe before calling the stream. Arguments: - ubi: (List <String>, default = [ubi]): a list of UBI's for which you want the data. Usually it is sufficient to omit this parameter. - time: (Double, default = 0.0): the time for how long you want the read to run in milliseconds, at time = 0.0 a single frame is returned.	None
rie.vision.card.close: close a stream after you no longer need this data. Argument: - ubi: (List <String>, default = [ubi]): a list of ubi's that you no longer want to be subscribed to.	None

An example of using rie.vision.card. * In Python:

```
from twisted.internet.defer import inlineCallbacks

def on_card(frame):
  # This function is called every time the robot sees a card
  print(frame)

@inlineCallbacks
def main(session, details):
  # Wait until we see a card
  frames = yield session.call("rie.vision.card.read")
  print(frames[0])

  # Here we subscribe to the card data and start a stream

  yield session.subscribe(on_card, "rie.vision.card.stream")

  yield session.call("rie.vision.card.stream")
```

Face detection and tracking

Another advanced functionality we can do with camera images is to detect and track faces. For example, we can determine on camera images where a face is located in space and move the head of the robot towards it. Please note that this concerns face detection and not face recognition. So the robot does not know who is in front of him, it only knows that

someone is in front of him.

RPC	Returns
rie.vision.face.info: get the information from the Face Detection module.	A dictionary containing information about the Face detection module.
rie.vision.face.stream: opens a stream on the topic `rie.vision.face.stream` that publishes the amount of faces the robot 'sees'. It is important here that you first subscribe before calling the stream. Argument: - ubi: (List <String>, default = [ubi]): a list of UBI's for which you want the data. Usually it is sufficient to omit this parameter.	None
rie.vision.face.read: listen to all data collected for this module for a specified time. Arguments: - ubi: (List <String>, default = [ubi]): a list of ubi's. Usually it is sufficient to omit this parameter; - time: (Double, default = 0.0): the time for how long you want the read to run in milliseconds, at time = 0.0 a single frame is returned.	The data collected from the Face detection module over the preset time (time).
rie.vision.face.close: close a stream after you no longer need this data. Argument: - ubi: (List <String>, default = [ubi]): a list of ubi's that you no longer want to be subscribed to.	None
rie.vision.face.find: Make the robot move its head to see if the robot sees a face. Argument: - active: (Bool, default = True): if True, the robot will actively search for a face. This means that the robot moves its head until it finds a face. When active = False, the robot will only look ahead until it finds a face.	None
rie.vision.face.track: once we have found a face with rie.vision.find, we can follow (= track) the face by calling this RPC. This RPC continues to follow the face until it no	None

longer sees it. Arguments: - track_time: (Int, default = 0): The time in milliseconds that you want to track the face. If you use the default value of 0, the RPC will keep following the face until it no longer sees it. - lost_time: (Int, default = 4000): The time in milliseconds that we can miss a face before determining that we no longer see a face.	
rie.vision.face.stop: stop the `rie.vision.face.find` and `rie.vision.face.track` RPC's	None

An example of how to use rie.vision.face. * In Python:

```
from twisted.internet.defer import inlineCallbacks

def on_face(frame):
  # This function is called every time the number of faces
  #the robot sees changes
  print(frame)

@inlineCallbacks
def main(session, details):
  # Wait until we see at least 1 face
  frames = yield session.call("rie.vision.face.read")
  print(frames[0])

  # Here we subscribe to the face data and start a stream

  yield session.subscribe(on_face, "rie.vision.face.stream")
  yield session.call("rie.vision.face.stream")
```

Virtual Robot

In order to work with the Virtual Robot, you will have to start this robot first by pressing the power button in the right-hand corner. Once the Virtual Robot is connected and ready to play, you can program the robot. This works the same as with the real robot and will be explained below.
Once you have written your Python code correctly, you can run this from your computer, and the virtual robot will execute the code you've written.
Be aware, though, that while the robot can perform all your called behaviours, it cannot sense the way the real robot can.
The virtual robot can perform all the default behaviours found in attachment B. It can also perform text-to-speech, as shown in the examples. It cannot perform manually programmed

movements and it does not have access to the camera or microphone, so it cannot perform face or speech-to-text recognition.

Appendix

A. Standard behaviours

B. Motor Joints and range of motion

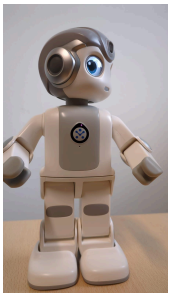
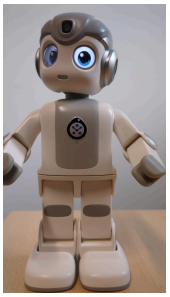
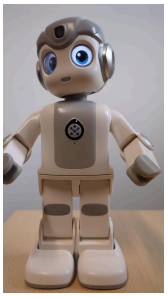
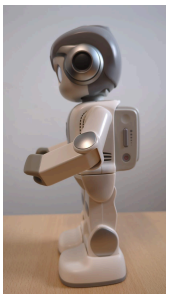
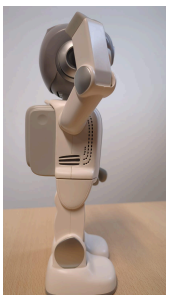
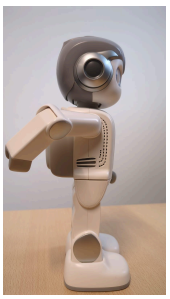
A. Standard behaviors

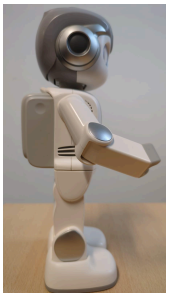
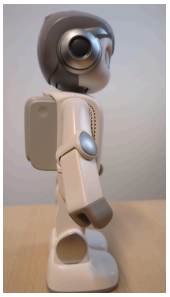
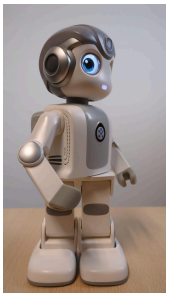
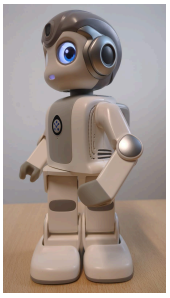
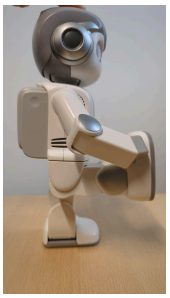
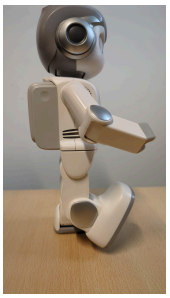
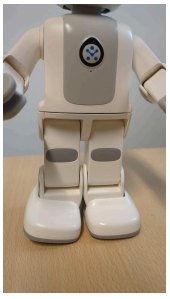
List of standard installed behaviors:

BlocklyStand	BlocklyMoveForward	BlocklyTurnLeft
BlocklyTurnRight	BlocklyTurnAround	BlocklyTouchHead
BlocklyWaveRightArm	BlocklyTouchKnees	BlocklyTouchToes
BlocklyDuck	BlocklyArmsForward	BlocklyLeftArmForward
BlocklyRightArmForward	BlocklyArmsBackward	BlocklyLeftArmUp
BlocklyRightArmUp	BlocklyLeftArmSide	BlocklyTouchShoulders
BlocklyArmsBackward	BlocklyRightArmSide	BlocklySitDown
BlocklyDab	BlocklyBow	BlocklyFreeWalk
BlocklyRobotDance	BlocklyDabLong	BlocklySneeze
BlocklyHappyBirthday	BlocklyGangnamStyle	BlocklyMacarena
BlocklyThriller	BlocklyFlamenco	BlocklySafeStand
BlocklySaxophone	BlocklyCrouch	BlocklyShrug
BlocklyYouAndMe	BlocklyInviteRight	BlocklyLookAtChild
BlocklyLookingUp	BlocklyFearUp	BlocklyApplause
BlocklyDiscoDance	BlocklyVacuum	BlocklyStarWarsStory
BlocklyJingleBells	BlocklyEvolutionOfDance	BlocklyHaka
BlocklyLullaby	BlocklyCaravanPalace	BlocklyAlorsOnDanse
BlocklyCircleOfLife	BlocklyIfYourHappy	BlocklyChickenDance

INTERACTIVE ROBOTICS

B. Motor joints and ranges of motion

Motor joint	Min value	Max value	Minimum time to perform an entire movement (left -> right) (ms)
`body.head.yaw`	-0.874 	0.874 	600
`body.head.roll`	-0.174 	0.174 	400
`body.head.pitch`	-0.174 	0.174 	400
`body.arms.right.upper.pitch`	-2.59 	1.59 	1600
`body.arms.right.lower.roll`	-1.74	6.50e-04	700

			
body.arms.left.upper.pitch	-2.59	1.59	1600
body.arms.left.lower.roll	-1.74	6.50e-04	700
body.torso.yaw	-0.874 	0.874 	1000
body.legs.right.upper.pitch	-1.73 	1.73 	1000
body.legs.right.lower.pitch	-1.50 	1.50 	800
body.legs.right.foot.roll	-0.849 	0.249 	800

`body.legs.left.upper.pitch`	-1.73	1.73	1000
`body.legs.left.lower.pitch`	-1.50	1.50	800
`body.legs.left.foot.roll`	-0.849	0.249	800

C. Extra functions

To use these 'extra functions', the `alpha_mini_rug` package needs to be installed using '`pip3 install alpha_mini_rug`'.

Function	**Returns**
aruco_detect_markers(*frame*): detects all the aruco cards of size 6x6-250 in the given frame Arguments: `frame` (dictionary): The frame dictionary from the robot's camera stream	`(corners)` **list :** The corners of the detected markers `(ids)` **list :** The ids of the detected markers
show_camera_stream(*frame*): Display the robot's camera stream Arguments**:** `frame` (dictionary): The frame dictionary from the robot's camera stream	Does not return anything but displays a window with the camera stream.
detect_face_in_frame(frame): Function to detect a face in a frame using OpenCV's Haar Cascade classifier Arguments: `frame` (dictionary): The frame dictionary from the robot's camera stream	(top_left, bottom_right) **tuple**: The coordinates of the detected face in the frame.

perform_movement(session, frames, mode="linear", sync=True, force=False): This function performs a movement with the robot's joints. The time of each frame is calculated based on the proportional time of the movement. Arguments: `session` (Component): The session object. `frames` (list): A list of dictionaries with the time and data of the joints to be moved. `mode` (str): The mode of the movement. Choose one of the following: "linear", "last", "none". `sync` (bool): A flag to wait for the robot to execute all motor commands. `force` (bool): A flag to force the joints to get as close as possible to the desired motor position even though the motor cannot reach the exact position due to hardware limitations. If False, you assume the robot can move to the desired position. For extra information, you can also have a look at rom.actuator.motor.write	Does not return anything. Performs the requested move.

D. Actuator modes

When controlling actuator modalities, you may send frames that do not contain all ubi's or even empty frames. To tell the actuator what to do with these types of frames, you can give a mode. With the mode you can tell the robot how to move between the different frames to the next position. The modality supports the following three modes: 'linear', 'last' and 'none'. We recommend using ‘linear’ in all cases.

- **None**:

Go pass at the exact moment as fast as possible to this frame and do not fill in empty frames. Please note that this will lead to unstable behavior with the motor modality.

- **Linear**:

Empty frames are filled with the values between the filled-in frames. Between frames the transition will be gradual.

- **Last**:

Fill empty frames with the last specified value, with gradual transitions between frames. This mode works best for data recorded with the proprio modality.