Manual Advanced Programming

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

Table of Contents:

Preparation

Before we can get started with programming, we must first prepare your computer to be able to connect to your robots. Whichever programming language you choose, there are still some things to be set up before we can really get started. If you are not sure which programming language is right for you, check our FAQ for some useful tips.

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 and JavaScript. Both programming languages look roughly the same in terms of structure, but there are important differences between the two languages. To clarify which programming language we use during the examples, we use a table with a color. For Python code we use a light yellow color and for JavaScript a light blue / purple background color:

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

```
console.log ("This is JavaScript 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. Install Autobahn and OpenSSL via pip:

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

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

6. Open this file in your favorite Python editor by double clicking on it;
7. 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="Hallo, ik ben succesvol verbonden!")
    session.leave() # Sluit de verbinding met de robot

# Create wamp connection
wamp = Component(
    transports=[{
      "url": "ws://wamp.robotsindeklas.nl",
      "serializers": ["msgpack"]
    }],
    realm="VUL HIER JE REALM IN",
)
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`'.

8. 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. Then find the robot you want to work with and press the hamburger menu. Within the menu, click on the clipboard and the realm is at the bottom of the screen that now appears. An example realm: rie.5e1312363dbf49eed032e123;
9. Run demo.py by clicking Run;
10. 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.

JavaScript installation

We don't need to set up much to get started with JavaScript. With a few simple steps you can get started with programming:

1. Make sure that the robot you want to work with is turned on;
2. Create a new file called demo.html in a convenient place on your computer;
3. Then open this file with, for example, notepad. Note: programs such as Microsoft Word will not work here;
4. In the emptydocument, add the following snippet of code:

```
<script
src="https://cdn.jsdelivr.net/npm/autobahn-browser@19.7.3/autobahn.min.js"></script>
<span>Welkom bij het programmeren van JavaScript met Robots in de Klas</span>
<script>
  let wamp = new autobahn.Connection({
    url: "wss://wamp.robotsindeklas.nl",
    realm: "VUL HIER JE REALM IN",
    protocols: ["wamp.2.msgpack"]
  })

  wamp.onopen = async (session) => {
    await session.call("rie.dialogue.say", [],
          {text: "Hallo, ik ben succesvol verbonden!"})
    wamp.close() // Sluit de verbinding met de robot
  }

  wamp.open()
</script>
```

5. Fill in the piece of text where“ ENTER YOUR REALM HERE ”, the realm of the robot in. You can find this realm when you go to the robot page in portal.robotsindeklas.nl or portal.robotsindezorg.nl. Then find the robot you want to work with and press the hamburger menu. Within the menu, click on the clipboard and the realm is at the bottom of the screen that now appears. An example realm: rie.5e1312363dbf49eed032e123;
6. Now go back to your explorer and right click on demo.html. Then click on 'open with…' and select a browser there;
7. If everything is set up correctly, the browser will open a new tab with the text: 'Welcome to programming JavaScript with Robots in the Classroom' and 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;
8. Tip: if you press F12 while in the browser, you open an extra window in your screen where you can easily test code and see if there are errors in your code.

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):
    # PLAATS HIER JE CODE

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

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

For JavaScript you have the following piece code needed to connect:

```
let wamp = new autobahn.Connection({
    url: "wss://wamp.robotsindeklas.nl",
    realm: "VUL HIER JE REALM IN",
    protocols: ["wamp.2.msgpack"]
})
wamp.onopen = async (session) => {
    // PLAATS HIER JE CODE
}
wamp.open()
```

Once you have a successful connection, thefunction is `main` called for Python and `wamp.onopen` is called for JavaScript.

commands Giving

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 canwith `session` invoke a certain command. We do this byon `session` a `call` executing, 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:

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

And in JavaScript it looks like this:

[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 codes 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 bythe `session`, `subscribe` calling:

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

The next step is to tell the robot which sensor data we are interested in. We do this by telling in the 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 wegive the `subscribe` alsoa function that is always called as soon as new data becomes available.

```
def aangeraakt(frame):
  print(frame)

session.subscribe(aangeraakt, "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 makingduring `session` a `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 aangeraakt(frame):
  print(frame)

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

And in JavaScript it looks like this:

```
aangeraakt = function(frame){
  console.log(frame)
}
session.subscribe("rom.sensor.touch.stream", aangeraakt)
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 JavaScript, this order is reversed. So first you specify the channel and then the function.

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

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.hand`

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

```
body
| - head
| | - eyes
| | | - right
| | | - left
| | - ears
| | | - right
| | | - left
| - arms
| | - right
| | | - upper
| | | - lower
| | | - hand
| | - left
| | | - upper
| | | - lower
| | | - hand
| - torso
| - legs
| | - right
| | | - upper
| | | - lower
| | | - foot
| | - left
| | | - upper
| | | - lower
| | | - foot
| - tail
```

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 or JavaScript, we definitely recommend copying and playing this code. Do not forget to adjust the realm of the examples, because it is different for every robot.

All code examples are written in Python, but you can find the same code for JavaScript in portal.robotsindeklas.nl. Go to the robot overview and click on the hamburger menu of a robot that is online. Then click on the clipboard there and you will see all these code examples there in Python and in JavaScript.

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 =
"http://icecast-qmusicnl-cdp.triple-it.nl/Qmusic_nl_classics_96.mp
3",
      sync=False)
    print("Het laden van de radio kan even duren")
    print("Druk op je toetsenbord om de muziek te stoppen!")
    input()
    yield session.call("rom.actuator.audio.stop")
    session.leave() # Sluit de verbinding met de 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-kinks

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 `rom.actuator.motor.write` 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

@inlineCallbacks
def main(session, details):
    # Ja-knikken
    yield session.call("rom.actuator.motor.write",
       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() # Sluit de verbinding met de 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 you can react to 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.

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

def aangeraakt(frame):
    if ("body.head.front" in frame["data"] or
            "body.head.middle" in frame["data"] or
            "body.head.rear" in frame["data"]):
        print("Hoofd is aangeraakt!")

@inlineCallbacks
def main(session, details):
    yield session.subscribe(aangeraakt, "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])
```

Faces find and follow

if a robot has a camera sitting on his head, then said to us to find a face and then also follow. So suppose you sit diagonally opposite the robot and you look at the robot, then with the example code below you let the robot 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="Hoi!")
    yield session.call("rie.vision.face.track")
    yield session.call("rie.dialogue.say",
        text="Ik zie je niet meer!")
    yield session.call("rom.optional.behavior.play",
        name="BlocklyCrouch")
    session.leave() # Sluit de verbinding met de 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 & Await

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

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 in Python that you **do not** use the default `time.sleep ()`! Instead of the built `time.sleep`(), then youthe trail `autobahn.twisted.util.` useto 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="Ik wacht nu 20 seconden")

    # Wacht 20 seconden
    yield sleep(20)

    yield session.call("rie.dialogue.say",
        text="Ik ben klaar met wachten!")
```

For JavaScript you can use the default setTimeout () or our sleep function which works the same way as's `sleep` Python, see the sample code below.

```
function slaap(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
async main(session) => {
    await session.call("rie.dialogue.say", [],
        {"text": "Ik wacht nu 20 seconden"})

    // Wacht 20 seconden
    await slaap(20_000)

    await session.call("rie.dialogue.say", [],
        {text: "Ik ben klaar met wachten!"})
}
```

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 connectso that you can get started programming blocks or writing real code. A ROM generally consists of two parts, namely:

- Sensorand: 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
    }
}
```

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 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 subscribed 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.	None

Argument: - ubi: (List <String>, default = [ubi]): A list of UBIs you no longer want to listen to.	
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):
  # Deze functie wordt elke keer aangeroepen wanneer
  # de robot beweegt
  print(frame)

@inlineCallbacks
def main(session, details):
  # Krijg de hoeken van alle motoren op dit moment
  frames = yield session.call("rom.sensor.proprio.read")
  print("Huidige motoren stand:")
  print(frames[0]["data"])
  print("")

  # Hier melden we ons aan voor de proprio data
  # en starten we een stream
  yield session.subscribe(proprio, "rom.sensor.proprio.stream")
  yield session.call("rom.sensor.proprio.stream")

  # Nu laten we de robot wat bewegen
  # zodat we proprio data krijgen
  # van onze 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() # Sluit de verbinding met de robot
```

And in JavaScript:

```
proprio = (event) => {
    // Deze functie wordt elke keer aangeroepen wanneer
    // de robot beweegt
    frame = event[0]
    console.log(frame)
}

async function main(session) {
    // Krijg de hoeken van alle motoren op dit moment
    frames = await session.call("rom.sensor.proprio.read")
    console.log("Huidige motoren stand:")
    console.log(frames[0]["data"])
    console.log("\n") // Print een lege regel

    // Hier melden we ons aan voor de proprio data
    // en starten we een stream
    await session.subscribe("rom.sensor.proprio.stream", proprio)
    await session.call("rom.sensor.proprio.stream")

    // Nu laten we de robot wat bewegen
    // zodat we proprio data krijgen
    // van onze rom.sensor.proprio.stream stream
    await session.call("rom.optional.behavior.play", [],
                       {"name": "BlocklyStand"})
    await session.call("rom.optional.behavior.play", [],
                       {"name": "BlocklyWaveRightArm"})
    await session.call("rom.optional.behavior.play", [],
                       {"name": "BlocklyCrouch"})
    session.leave() // Sluit de verbinding met de robot
}
```

Actuators

Actuators are parts of the robot that allow the robot to to do. 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. Light
4. Behavior

An overview of allin the actuator modality:

Motor

```
RPCsrom.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
```

Light

```
rom.actuator.light.info
rom.actuator.light. write
rom.actuator.light.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 here found. 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** .

RPC	**Returns**
rom.actuator.motor.info: request the information of the motor modality	A dictionary containing information about the Motor modality.
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 adetailed 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

Light

With the light modality we can change the color of lights on a robot. With the Nao, for example, we can adjust the eye colors via this module. For example, we can change the eye color to green if the answer to a question is correct. The underlying technology for changing the colors is very similar to how you control motors, only now you send color information instead of angles.

RPC	**Returns**
rom.actuator.light.info: request the information of the light modality.	A dictionary containing information about the light modality.
rom.actuator.light.write: set the color of the lights of the robot, for example the eye color of the Nao robot. Arguments: - frames: (List <Frame>, default = [ubi]): a list containing frames that tell which colors the lights should change to and how much time they have to do so. A color is defined as [Red, Green, Blue] and can have a value between 0 and 255; - mode: (String, default = 'linear'): With the mode you can tell the robot how to move between frames to the next color. The ROM supports the following three modes: 'linear', 'last' and 'none'. See the appendix for a further explanation of fashion; - sync: (Bool, default = True): if True, wait until the robot has executed all light commands; - force: (Bool, default = False): if True, then your values above and below the limits of 0 and 255 will be converted to the min and max value. If False, you will get an error if you send a value below 0 and above 255.	None, but if sync = True, you can wait for the RPC until the robot has finished performing the change lights. This RPC can return an Error if your force is set to False and you try to set a color that does not exist such as [500,0,0] or [125,0]. In the first case you are over the limit of 255 and in the second case the light modality does not know what color that is, because there is still one value missing.
rom.actuator.light.stop: stop the `rom.actuator.light.write`	None

An example of how to adjust the robot's eye colors in Python:

```
# Verander de oogkleuren van groen, naar rood, naar blauw
yield session.call("rom.actuator.light.write",
  mode = "linear",
  frames=[{"time": 1000, "data": {"body.head.eyes": [0,255,0]}},
          {"time": 2000, "data": {"body.head.eyes": [255,0,0]}},
          {"time": 3000, "data": {"body.head.eyes": [0,0,255]}}])
```

And in JavaScript:

```
// Verander de oogkleuren van groen, naar rood, naar blauw
await session.call("rom.actuator.light.write", [],
  {"frames": [
    {"time": 1000, "data": {"body.head.eyes": [0,255,0]}},
    {"time": 2000, "data": {"body.head.eyes": [255,0,0]}},
    {"time": 3000, "data": {"body.head.eyes": [0,0,255]}}
  ],
  "mode": "linear"})
```

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:

```
# Laat de robot opstaan
yield session.call("rom.optional.behavior.play",
    name="BlocklyStand")

# Laat de robot wat zeggen terwijl die zwaait
behavior = session.call("rom.optional.behavior.play",
    name="BlocklyWaveRightArm")
yield session.call("rie.dialogue.say", text="Hallo!")
yield behavior

# Laat de robot weer zitten
yield session.call("rom.optional.behavior.play",
    name='BlocklyCrouch')
```

In JavaScript:

```
// Laat de robot opstaan
await session.call("rom.optional.behavior.play", [],
    {"name": "BlocklyStand"})

// Laat de robot wat zeggen terwijl die zwaait
behavior = session.call("rom.optional.behavior.play", [],
    {"name": "BlocklyWaveRightArm"})
await session.call("rie.dialogue.say", [], {"text":"Hallo!"})
await behavior

//Laat de robot weer zitten
await 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",
        type:   "nao",
        version:"v5"
    },
    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: {
        ssid:       "naonet",
        strength:   60
    }
}
```

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].

An overview of all:

Dialogue:

```
RPCsrie.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.keyword.add
rie.dialogue.keyword.remove
rie.dialogue.keyword.clear
rie.dialogue.keyword.stream
rie.dialogue.keyword.close
rie.dialogue.keyword.read
```

Vision :

```
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
```

Cloud modules:

```
rie.cloud_modules.ready
```

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

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_animatedHave: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	None

the desired language. False → the robot only uses ReadSpeaker voice.	
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:

```
# Laat de robot iets zeggen:
yield session.call("rie.dialogue.say",
    text="Hallo, goed je te zien!")

# Maak alleen gebruik van de ReadSpeaker TTS-engine:
yield session.call("rie.dialogue.config.native_voice",
    use_native_voice=False)

# Zeg wat met de ReadSpeaker stem:
yield session.call("rie.dialogue.say",
     text="Dit is de ReadSpeaker stem!")

# Verander de taal naar Engels:
yield session.call("rie.dialogue.config.language", lang="en")


# Zeg iets met de ReadSpeaker stem in het Engels:
yield session.call("rie.dialogue.say",
     text="I am now speaking English!")
```

In JavaScript:

```
// Laat de robot iets zeggen:
await session.call("rie.dialogue.say", [],
    {"text":"Hallo, goed je te zien!"})

// Maak alleen gebruik van de ReadSpeaker TTS-engine:
await session.call("rie.dialogue.config.native_voice", [],
    {"use_native_voice": false})

// Zeg wat met de ReadSpeaker stem:
await session.call("rie.dialogue.say", [],
    {"text":"Dit is de ReadSpeaker stem!"})

// Verander de taal naar Engels:
await session.call("rie.dialogue.config.language", [],
    {"lang": "en"})

// Zeg iets met de ReadSpeaker stem in het Engels:
await session.call("rie.dialogue.say", [],
    {"text": "I am now speaking English!"})

// En weer terug naar Nederlands
await session.call("rie.dialogue.config.language", [],
    {"lang": "nl"})
```

Smart question

Another smart trick that wewith RIE can dois a smart ask a question of someone who interacts 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 = "Welke kleur vind je mooier, rood of blauw?"
answers = {"rood": ["rood", "rot"], "blauw": ["blauw", "lauw"]}

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

if answer == "rood":
    yield session.call("rie.dialogue.say",
        text="Rood is zeker een mooie kleur!")
elif answer == "blauw":
    yield session.call("rie.dialogue.say",
        text="Ik ben gek op blauw!")
else:
    yield session.call("rie.dialogue.say",
        text="Sorry, maar ik heb je niet goed verstaan.")
```

And in JavaScript:

```
let question = "Welke kleur vind je mooier, rood, of blauw?"
let answers = {
  "rood": ["rood", "rot"],
  "blauw": ["blauw", "lauw"]
}

let answer = await session.call("rie.dialogue.ask", [],
    {"question":question, "answers": answers})

if (answer == "rood") {
  await session.call("rie.dialogue.say", [],
      {"text":"Rood is zeker een mooie kleur!"})
} else if (answer == "blauw") {
  await session.call("rie.dialogue.say", [],
      {"text":"Ik ben gek op blauw!"})
} else {
  await session.call("rie.dialogue.say", [],
      {"text":"Sorry, maar ik heb je niet goed verstaan."})
}
```

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.	The data collected from the keyword module over the preset time (time).

Arguments: - 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= "Ja")

    yield session.call("rie.dialogue.say",
        text="Zeg rood, blauw of groen")
    yield session.call("rie.dialogue.keyword.add",
        keywords=["rood", "blauw", "groen"])
    yield session.subscribe(on_keyword,
        "rie.dialogue.keyword.stream")
    yield session.call("rie.dialogue.keyword.stream")

    # Wacht 20 seconden voordat we de keyword stream sluiten
    yield sleep(20)

    yield session.call("rie.dialogue.keyword.close")
    yield session.call("rie.dialogue.keyword.clear")
    session.leave() # Sluit de verbinding met de robot
```

And in JavaScript:

```
var sess = null
// Gebruik await slaap(s) om s seconden te wachten
let slaap = s=>new Promise(r=>setTimeout(r, s*1000))

async function on_keyword (event) {
  let frame = event[0]
  if (frame["data"]["body"]["certainty"] > 0.45) {
    await sess.call("rie.dialogue.say", [], {"text": "Ja"})
  }
}

async function main(session, event) {
  sess = session
  await session.call("rie.dialogue.say", [],
      {"text": "Zeg rood, blauw of groen"})

  await session.call("rie.dialogue.keyword.add", [],
      {"keywords": ["rood", "blauw", "groen"]})

  await session.subscribe("rie.dialogue.keyword.stream",
                          on_keyword)
  await session.call("rie.dialogue.keyword.stream")

  // Wacht 20 seconden voordat we de keyword stream sluiten
  await slaap(20)

  await session.call("rie.dialogue.keyword.close")
  await session.call("rie.dialogue.keyword.clear")
  wamp.close() // Sluit de verbinding met de 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 canall kinds of cards recognize. For example, a Nao robot can recognize Naomarks and respond to it. Unfortunately, other types of robots cannot recognize the Naomarks. 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. This way all robots can still recognize cards with a marker and as a bonus Nao still supports the Naomarks.

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.	None

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.	
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):
  # Deze functie wordt elke keer aangeroepen wanneer
  # de robot een kaartje ziet
  print(frame)

@inlineCallbacks
def main(session, details):
  # Wacht totdat we een kaartje zien
  frames = yield session.call("rie.vision.card.read")
  print(frames[0])

  # Hier abonneren we ons op de card data en starten we een stream
  yield session.subscribe(on_card, "rie.vision.card.stream")
  yield session.call("rie.vision.card.stream")
```

And in JavaScript:

```
function on_card(event) {
    // Deze functie wordt elke keer aangeroepen wanneer
    // de robot een kaartje ziet
    let frame = event[0]
    console.log(frame)
}

async function main(session, event) {
    // Wacht totdat we een kaartje zien
    let frames = await session.call("rie.vision.card.read")
    console.log(frames[0])

    // Hier abonneren we ons op de card data en starten we een stream
    await session.subscribe("rie.vision.card.stream", on_card)
    await 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.	None

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.	
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 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.	None
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):
  # Deze functie wordt elke keer aangeroepen wanneer het aantal
  # gezichten dat de robot ziet verandert
  print(frame)

@inlineCallbacks
def main(session, details):
  # Wacht totdat we op zijn minst 1 gezicht zien
  frames = yield session.call("rie.vision.face.read")
  print(frames[0])

  # Hier abonneren we ons op de face data en starten we een stream
  yield session.subscribe(on_face, "rie.vision.face.stream")
  yield session.call("rie.vision.face.stream")
```

And in JavaScript:

```
function on_face(event) {
  // Deze functie wordt elke keer aangeroepen wanneer het
  // aantal gezichten dat de robot ziet verandert
  frame = event[0]
  console.log(frame)
}

async function main(session, event) {
  // Wacht totdat we op zijn minst 1 gezicht zien
  let frames = await session.call("rie.vision.face.read")
  console.log(frames[0])

  // Hier abonneren we op de face data en starten we een stream
  await session.subscribe("rie.vision.face.stream", on_face)
  await session.call("rie.vision.face.stream")
}
```

Cloud Modules

The cloud modules is the collective name for the above-mentioned advanced functionalities . It ensures that as soon as the robot wakes up, the dialogue module and vision module is started. If you make a program yourself that should run immediately when the robot starts, then it is good to check whether the cloud modules are ready for use. You do this by checking if the following RPC already exists:

RPC	Returns
rie.cloud_modules.readyRPCs: this RPC is registered after all otherof the cloud module have been registered.	None

An example of how you can wait in your program until the cloud modules are completely ready for use. In Python:

```
from autobahn.twisted.util import sleep
from twisted.internet.defer import inlineCallbacks
from autobahn.wamp.exception import ApplicationError

@inlineCallbacks
def main(session, details):
    print("Waiting for cloud modules")
    ready = False
    while not ready:
        try:
            yield session.call("rie.cloud_modules.ready")
            ready = True
        except ApplicationError:
            print("Cloud modules are not ready yet")
            yield sleep(0.25)

    print("Cloud modules are ready to go!")
```

And in JavaScript:

```
// Gebruik await slaap(s) om s seconden te wachten
let slaap = s=>new Promise(r=>setTimeout(r, s*1000))

async function main (session, details) {
    console.log("Waiting for cloud modules")
    let ready = false
    while (!ready) {
        try {
               await session.call("rie.cloud_modules.ready")
               ready = true
        } catch (error) {
               console.log("Cloud modules are not ready yet")
               await slaap(0.25)
        }
    }
    console.log("Cloud modules are ready to go!")
}
```

FAQ

Which programming language do you recommend?
If this is your first time getting started with programming, we recommend starting with Python or JavaScript. Both languages are easy entry languages that you can quickly master. In a way, the languages are quite similar, but there are some differences in how you run the code and how you define functions, for example. To give you a better idea of both languages, we have made a comparison table below with the advantages of each language:

Python**:**

+ Python code is known for being very readable;
+ Python is very popular among programmers;
+ The language was invented by a Dutchman :-)

JavaScript**:**

+ JavaScript is faster to set up than Python;
+ JavaScript is beginner friendly, although it can sometimes be difficult to get errors (bugs) out of your code;
+ There are more programmers worldwide who program in JavaScript than in Python.

If you have experience in another language, you can probably also start programming in that language. What you should pay attention to is that your language also supports a WAMP connection. For example C ++ you can use Autobahn-cpp.

When I try to start my program I keep getting a No Such Realm error
Check if the realm matches the realm of the robot. You do this by going to the robots page and clicking on the hamburger menu of the robot you want to work with. Then click there on the clipboard and the realm of the robot will appear at the bottom of the screen that now appears. Check if you have copied the full realm name. An example of a full realm name: `rie.5e1312363dbf49eed032e123`

Changelog

The software for the robots is constantly being improved so that they can interact even better. In order to do this, new RPCs will be added to the platform in the future. But it can also happen that arguments of existing RPCs are changed or that some RPCs are no longer supported. With this chapter we will keep you informed of the changes we are making.

V1.0

- First version of documentation ROM and RIE specifications;

V1.1

- Code example of Keywords has been updated. The subscribe referred to a non-existent function;

V1.2

- JavaScript examples have been added to the ROM and RIE api chapters.
- In the attachment a list of default installed behaviors is added.

Attachment

A. Session examples
B. Actuator modes
C. Default behaviors

A. Session examples

To better understand the session object how it works and how to use it your program, we have included two examples below. These examples show how you can perform the various actions that you can do with the session object in Python and JavaScript. You will notice that there are small differences between Python and JavaScript that are important to know if you are working with both languages.

The examples below start withthe `submittingridk.example.hello` topic, see step `1)`. Then in the next step, step 2), we publish a post on the topic `ridk.example.hello`. Because in step 1the function `weonevent` boundto the topic `ridk.example.hallo`, this function is now called because we publish data on this topic.

In the next step, step 3), we register a new RPC called `ridk.example.add` and every time someone calls this RPC we run the code in thefunction `addition` . You will also see this when we go to step 4), because we call the RPC `ridk.example. addition` and we ask the RPC to add the numbers 2 and 3 together. We wait for an answer using `yield` and `await` and we store the answer in the variable `answer`. At the end of the program, we print the answer to make sure we get the correct answer.

Python Code:

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

@inlineCallbacks
def main(session, details):

    # 1. Abonneer je op het topic ridk.voorbeeld.hallo
    def onevent(bericht):
        print("Ontvangen bericht: " + bericht)

    yield session.subscribe(onevent, "ridk.voorbeeld.hallo")

    # 2. Publiceer text op het topic ridk.voorbeeld.hallo
    yield session.publish("ridk.voorbeeld.hallo",
            "Dit is een publish bericht",
            options=PublishOptions(exclude_me=False))

    # 3. Registreer een optellen RPC
    def optellen(getallen):
        return getallen[0] + getallen[1]

    yield session.register(optellen, "ridk.voorbeeld.optellen")

    # 4. Roep de geregistreerde optellen RPC aan
    antwoord = yield session.call("ridk.voorbeeld.optellen",
                        getallen=[2,3])

    print("Het antwoord is: " + str(antwoord))

    session.leave()

# Create wamp conn
wamp = Component(
    transports=[{
        "url": "wss://wampdev.robotsindeklas.nl",
        "serializers": ["msgpack"],
        "max_retries": 0
    }],
    realm="rie.5e1312363dbf49eed032e123",
)
wamp.on_join(main)

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

JavaScript code:

```
let wamp = new autobahn.Connection({
    url: "wss://wamp.robotsindeklas.nl",
    realm: "rie.5e1312363dbf49eed032e123",
    protocols: ["wamp.2.msgpack"]
})

wamp.onopen = async function (session) {

   // 1. Abonneer je op het topic ridk.voorbeeld.hallo
   function onevent(berichten) {
     console.log("Ontvangen bericht: " + berichten[0])
   }
   session.subscribe("ridk.voorbeeld.hallo", onevent)

   // 2. Publiceer text op het topic ridk.voorbeeld.hallo
   session.publish("ridk.voorbeeld.hallo",
                   ["Dit is een publish bericht"],
                   {},
                   {exclude_me: false})

   // 3. Registreer een optellen RPC
   function optellen(argumenten) {
     return argumenten[0] + argumenten[1]
   }
   await session.register("ridk.voorbeeld.optellen", optellen)

   // 4. Roep de geregistreerde optellen RPC aan
   let antwoord = await session.call("ridk.voorbeeld.optellen",
                             [2, 3])

   console.log("Het antwoord is: " + antwoord)

   session.leave()
}

wamp.open()
```

B. Actuator modes

When controlling actuator modalities you may send frames that do not contain all ubi's or even empty frames. This is useful if, for example, you want to change color independently and at different times in the left and right eyes. Or if you want to play back data recorded by the proprio mode.

To tell the actuator what to do with these types of frames, you can give a mode. We give the example here with the eyes and explain this with the three modes. The data we send will look like this:

```
[{"time": 1000, "data": {"body.head.eyes.left":  [0,0,0]}},      // off
{ "time": 2000, "data": {"body.head.eyes.right": [0,0,0]}},      // off
{ "time": 3000, "data": {}},
{ "time": 4000, "data": {"body.head.eyes.left":  [255,0,0]}},    // red
{ "time": 5000, "data": {"body.head.eyes.right": [0,255,0]}},    // green
{ "time": 6000, "data": {"body.head.eyes.left":  [255,255,255]}}]// white
```

- **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.

C. Standard behaviors

List of standard installed behaviors:

BlocklyRightHandOpen	BlocklyStand	BlocklyBothHandsOpen
BlocklyLeftHandClosed	BlocklyRightHandClosed	BlocklyBothHandsClosed
BlocklyWaveRightArm	BlocklyMoveForward	BlocklyTurnLeft
BlocklyTurnRight	BlocklyTurnAround	BlocklyTouchHead
BlocklyTouchShoulders	BlocklyTouchKnees	BlocklyTouchToes
BlocklyDuck	BlocklyArmsForward	BlocklyLeftArmForward
BlocklyRightArmForward	BlocklyArmsUp	BlocklyLeftArmUp
BlocklyRightArmUp	BlocklyLeftArmSide	BlocklyRightArmSide
BlocklyArmsBackward	BlocklySprinkle	BlocklySitDown
BlocklyDab	BlocklyBow	BlocklyFreeWalk
BlocklyTaiChiChuan	BlocklyDabLong	BlocklySneeze
BlocklyHappyBirthday	BlocklyGangnamStyle	BlocklyMacarena
BlocklyHappy	BlocklyLeftHandOpen	BlocklySafeStand
BlocklySaxophone	BlocklyCrouch	BlocklyShrug
BlocklyYouAndMe	BlocklyInviteRight	BlocklyLookAtChild
BlocklyLookingUp	BlocklyFearUp	BlocklyApplause
BlocklyDiscoDance	BlocklyVacuum	BlocklyStarWars
BlocklyPride	BlocklyFollowMe	