F&P Personal Robotics

myP Script Functions Manual

Version 1.2.1

(The versioning of this document corresponds to the versioning of the myP software.)

2016-04-06

The information contained in this document is property of F&P Robotics AG and shall not be reproduced in whole or in part without prior written approval of F&P Robotics AG. The information provided herein is subject to changes without notice and should not be constructed as a commitment by F&P Robotics AG. This manual is periodically reviewed and revised. F&P Robotics AG assumes no responsibility for any errors or omissions in this document.

This manual consists of original instructions from F&P Robotics AG and is intended for all users of myP. Before using the functions mentioned in this document together with myP and a P-Arm robotic arm or P-Grip robotic gripper, you must read the corresponding user manuals carefully, paying particular attention to the safety instructions on the P-Arm user manual. Compliance with the instructions in all manuals is mandatory.

Contents

1. Basic Python Programming

Basic Python Standards

commentary

```
# this is a commentary and will not be considered as executable code
"""
This is a multiline commentary
That will also not be executed.
"""
```

variable definitions and declarations

```
myBoolean = True                    # here, myBoolean is defined as a Boolean with value True
myInteger = 3                       # here, myInteger is defined as an integer number with value 3
myFloat = 3.1416                    # here, myFloat is defined as a floating point number 3.1416
myString = "Hello World!"           # here, myString is defined as a string with value "Hello World!"
```

list and dictionary definitions and usage

```
myList = [1, 2, 3, 4, 5]            # create a list containing some numbers
myList[0] = 6                       # sets the first list element to be 6
myDictionary = {"a": 1, "b": 2}     # create a dictionary linking the element "a" and "b"
                                    #    to the values 1 and 2, respectively
myDictionary["a"] = 3               # sets the element "a" to be 3
```

converting variables

```
x = int(x)                          # converts x to an integer number
x = float(x)                        # converts x to a floating point number
x = str(x)                          # converts x to a string
```

Basic Python Functions

Standard and Default Function Arguments

Let's consider that a function has been defined with the following arguments:

```
function(arg1, arg2, arg3 = "x", arg4 = "y")
```

The function arguments **arg1** and **arg2** are standard arguments that need be given when calling this function. The arguments **arg3** and **arg4** have a default value and therefore do not necessarily be given when calling this function. All of the following function calls are valid:

```
function("a", "b")                                  # this will execute function("a", "b", "x", "y")
function("a", "b", "c")                             # this will execute function("a", "b", "c", "y")
function("a", "b", "c", "d")                        # this will execute function("a", "b", "c", "d")
function("a", "b", arg4="d")                        # this will execute function("a", "b", "x", "d")
function(arg1="a", arg2="b", arg3="c", arg4="d")    # this will execute function("a", "b", "c", "d")
function(arg3="c", arg2="b", arg1="a")              # this will execute function("a", "b", "c", "y")
```

Unpacked Function Arguments

A few myP function use unpacked arguments. These arguments are useful if the number of arguments of a function may vary. Let's consider the following functions arguments:

```
function1(*arguments)                               # function with unpacked list argument
function2(**arguments)                              # function with unpacked dictionary argument
```

Then these functions can be used in the following way:

```
function1(value1, value2, ...)                      # function with updacked list argument
function2(arg1=value1, arg2=value2, ...)            # function with unpacked dictionary argument
```

Basic Python Operators

Calculation Operators

```
x = 3 + 5          # addition:          x has now value 8
x = 7 - 2          # subtraction:       x has now value 5
x = 4 * 8          # multiplication:    x has now value 32
x = 54 / 6         # division:          x has now value 9
x = 16 % 7         # modulo:            x has now value 2
x = 2 ** 3         # exponential:       x has now value 8
```

Comparison Operators

Here x and y can be of any type.

```
x == y             # returns true if x is equal to y
x != y             # returns true if x is not equal to y
```

Here x and y are numbers.

```
x > y              # returns true if x is greater than y
x < y              # returns true if x is smaller than y
x >= y             # returns true if x is greater than or equal to y
x <= y             # returns true if x is smaller than or equal to y
```

Logical Operators

Here x and y are Booleans.

```
x and y            # logical AND operator:   returns true if x and y are true
x or y             # logical OR operator:    returns true if either x or y is true
not x              # logical NOT operator:   returns true if x is false
```

Conditional Programming

"if", "elif" (else if) and "else" conditions can be combined according to the following rules:

If Condition

The "if" scope is executed if and only if its condition returns the value True.

```
if condition:
    ...
```

Elif Condition

An "elif" (else if) scope is executed if and only if all previous "if" or "elif" conditions returned the value False and its own condition returns the values True.

```
if firstCondition:
    ...
elif secondCondition:
    ...
elif thirdCondition:
    ...
```

Else Condition

An "else" scope is executed if and only if all previous "if" or "elif" conditions returned the value False.

```
if condition:
    ...
else:
    ...
```

Example: check if the variable 'myInteger' is equal to 5, 10, or neither one of those numbers.

```
if myInteger == 5:
    print("your integer is 5.")
    print("have a nice day.")
elif myInteger == 10:
    print("your integer is 10.")
    print("have a nice day.")
else:
    print("your integer is neither 5 nor 10.")
    print("have a nice day anyway.")
```

Loops

While Loop

A while loop is executed and repeated as long as its condition returns the value True.

```
while condition:
    ...
```

Example: printing all numbers from 1 to 10.

```
i = 1
while(i <= 10):
    print(i)
    i = i + 1
```

For Loop

A for loop iterates over the items of any sequence, such as a list or a string.

```
for item in sequence:
    ...
```

Example: printing all numbers from 1 to 5.

```
for i in range(5):
    print(i)
```

Example: printing my favorite fruits.

```
for fruit in ["apple", "banana", "orange"]:
    print("one of my favorite fruits is" + str(fruit))
```

Example: counting out the letters in 'python'.

```
i = 1
for letter in "python":
    print("letter number " + str(i) + " in 'python' is " + str(letter))
    i = i + 1
```

Infinite Loops

Of special interest to a lot of applications are infinite loops, in order for P-Rob to perform a task repetitively until it is stopped manually using the appropriate myP functions. If using infinite loops, make sure that myP does not iterated through this loop too fast, otherwise myP might not be fast enough to process other functions (like sending a stop command to the robot) in parallel to executing the application.

Example: never ending infinite loop.

```
while(True):
    ...
```

2. Robot Movements

Move Joint

move_joint(actuator_ids, position, velocity = None, acceleration = None, block = True, relative = False**)**

This function moves a joint or multiple joints of the robot to a specific position or set of positions in the joint space, respectively. This function has been designed such that the movement of each joint finishes exactly at the same time.

Parameter	*Types*	*Description*
actuator_ids	Various	This argument contains the IDs of the motors to move. It can be - the ID of a single joint (e.g. **6**), - a list of joint IDs for multiple joints (e.g. **[1,4,6]**), - the global constant **ALL_KIN_JOINTS**, which is equal to the list of all kinematic joints, - the global constant **ALL_ACTUATORS**, which is equal to the list of all actuators including end effectors.
position	Float or List of Floats	This argument contains the angles (in **degrees**) that are assigned to the joints. It can be - a single floating point number that will be assigned to each joint (e.g. **-30**), - a list of floating point numbers that will be assigned to each of the joints specified in **actuator_ids** (e.g. **[90, -45, 30]**).
velocity	Float or List of Floats	This argument contains the angular velocity (in **degrees per second**) that is used during the motion. It can be - a single floating point number that defines the maximum velocity of each joint (e.g. **45**), - a list of floating point numbers that will be assigned to each of the joints specified in **actuator_ids** (e.g. **[45, 25, 30]**). Please note that if you are using this option, the movements of the joints will not necessarily finish at the same time.
acceleration	Float or List of Floats	This argument contains the angular acceleration (in **degrees per second squared**) that are assigned to the joints. It can be - a single floating point number that defines the maximum acceleration of each joint (e.g. **60**), - a list of floating point numbers that will be assigned to each of the joints specified in **actuator_ids** (e.g. **[60, 35, 45]**). Please note that if you are using this option, the movements of the joints will not necessarily finish at the same time.
block	Boolean	If **True**, the next command will be executed once this motion has finished. If **False**, the next command will be executed without waiting for this motion to finish.
relative	Boolean	If **True**, the position values are interpreted as a relative offset to the current joint angles. If **False**, the position values are interpreted as absolute joint angles.

Examples:

```
move_joint(ALL_KIN_JOINTS, 0)
```

Moving all kinematic joints of the robot to 0°, which executed in a calibrated robot causes it to stand up straight.

```
move_joint(6, -30, 45, 60, relative = True)
```

Moving joint 6 by -30° from its current position, with a velocity of 45°/s and acceleration of 60°/s^2.

```
move_joint([2,3,5], [20,-40,20], 35, 50)
```

Moving joints 2, 3 and 5 to 20°, -40° and 20°, respectively. The motion will be performed such that each joint will finish its movement at the same time. The joint that has the biggest angular distance to move, will move with a velocity of 35°/s and acceleration of 50°/s^2.

```
move_joint([1,4,6], [90, -45, 30], [45, 25, 30], [60, 35, 45], block = True)
# ... this is similar to ...
move_joint(1, 90, 45, 60, block = False)
move_joint(4, -45, 25, 35, block = False)
move_joint(6, 30, 30, 45, block = True)
```

Simultaneously moving three joints that will not finish their movements at the same point in time.

Move Tool

```
move_tool(x, y, z, orientation = None, velocity = None, acceleration = None, block = True, relative = None, frame = "base")
```

This function moves the tool center point (TCP) of the robotic arm to a specific target position. Optionally, the target orientation and the motion velocity and acceleration can be chosen. Also, the coordinate frame can be chosen in which to move the robot.

Parameter	*Types*	*Description*
x, y, z	Floats	These values define the three dimensional position of the TCP (in **mm**) expressed in Cartesian coordinates in the world frame, the main coordinate system of the robot. For more details about the coordinate systems, please refer to the **myP User Manual**.
orientation	Float or List of Floats	These values define the orientation of the TCP (in **degrees**). The format of the orientation differs for different types of robots: - P-Rob 1R: the orientation is defined as **[roll, pitch, yaw]** - P-Rob 1U: the orientation is defined as the **angle of the uppermost joint**
velocity	Float	This argument contains the maximum angular velocity (in **degrees per second**) that is used during this motion (e.g. **45**).
acceleration	Float	This argument contains the maximum angular acceleration (in **degrees per second squared**) that is used during this motion (e.g. **60**).
block	Boolean	If **True**, the next command will be executed once this motion has finished. If **False**, the next command will be executed without waiting for this motion to finish.
relative	Boolean	If **True**, the pose values of the TCP are interpreted as relative offsets to the current pose components. If **False**, the pose values are interpreted as absolute values. By default, the robot will move absolute if it moves in base coordinates and relative if it moves in tool coordinates.
frame	String	This value specifies the coordinate frame in which the robot will move. Allowed values are **"base"** and **"tool"**, to move in the robot base coordinates or the end-effector tool coordinates, respectively. By default, the "base" coordinates are chosen. For more details about the coordinate frames, have a look at the **myP User Manual**.

Examples:

```
move_tool(200, 600, 350, [30, -20, -90])
# ... this is similar to ...
move_tool(200, 600, 350, [30, -20, -90], relative = False, frame = "base")
```

Moving the TCP to the position [200mm, 600mm, 350mm] with orientation angles 30°, -20° and -90° for the roll, pitch and yaw angles, respectively.

```
move_tool(800, 0, 300, None, 250, 300)
```

Moving the TCP to the position [800mm, 0mm, 300mm] with a velocity of 250mm/s and acceleration of 300mm/s^2. The orientation is not given and will therefore be chosen by the function.

```
move_tool(20, 0, 0, relative = True, frame = "tool")
# ... this is similar to ...
move_tool(20, 0, 0, frame = "tool")
```

Moving the TCP by 20mm in the positive x-direction in tool coordinates (forward in direction of the gripper fingers).

Move to Pose

```
move_to_pose(pose_name, velocity = None, acceleration = None, block = True)
```

This function moves the robot to a previously saved pose.

Parameter	*Types*	*Description*
pose_name	String	This argument contains the name of the pose to move to (e.g. **"myPose"**, if this name is compatible with the name of a previously saved pose).
velocity	Float	This argument contains the maximum angular velocity (in **degrees per second**) that is used during this motion (e.g. **45**).
acceleration	Float	This argument contains the maximum angular acceleration (in **degrees per second squared**) that is used during this motion (e.g. **60**).
block	Boolean	If **True**, the next command will be executed once this motion has finished. If **False**, the next command will be executed without waiting for this motion to finish.

Examples:

```
move_to_pose("startPose")
```

Moving to a pose named "startPose" with default velocity and acceleration.

```
move_to_pose("endPose", 45, 60, block=False)
```

Moving to a pose named "endPose" with a maximum velocity of 45°/s and maximum acceleration of 60°/s^2 for each joint and awaiting new commands while moving.

Play Path

```
play_path(path_name, velocity = None, acceleration = None)
```

This function plays a path from the database. This can either a simple path (using interpolated trapezoidal movements and online calculation) or an advanced path (using cubic spline, linear and/or circular interpolations, its trajectory need to be precalculated offline). The additional velocity and acceleration input arguments can only be used for a simple path. In order to change the velocity and acceleration of an advanced path, the path editor should be used. Note that in order to execute a path quickly, the robot needs to be located at the very first pose given in the path before executing 'play_path'. Otherwise the robot will first move to the initial path position using the default velocity and acceleration values before running the path.

Parameter	*Types*	*Description*
path_name	String	This argument contains the name of the path to play (e.g. **"myPath"**, if this name is compatible with the name of a previously saved path).
velocity	Float or List of Floats	The velocity of a simple path can be given as a single number or a list of numbers (in **degrees per second**). If a single velocity is given, this value is used as the maximum velocity for all path segments. If a list is entered, the given velocity values correspond to the maximum velocities in each segment of the path.
acceleration	Float or List of Floats	The acceleration of a simple path can be given as a single number or a list of numbers (in **degrees per second squared**). If a single acceleration is given, this value is used as the maximum acceleration for all path segments. If a list is entered, the given acceleration values correspond to the maximum acceleration in each segment of the path.

Examples:

```
play_path("myPath")
```

This example plays a path from the database called "myPath", which can be either a simple or advanced Path. Since velocity and acceleration values are not given specifically, the path will move using the speed values it was saved with.

```
play_path("mySimplePath", velocity=20, acceleration=25)
```

This example plays a simple path with the desired maximum velocity and acceleration values of 20 °/s and 25 °/s^2, respectively.

```
play_path("mySimplePath", velocity=[20, 10, 15], acceleration=[10, 50, 30])
```

This example plays a simple path with different velocity and acceleration values for each path segment. The number of elements in the velocity and acceleration arrays must be equal to the number of path segments. The path "mySimplePath" for example contains four postures and therefore three segments between consecutive postures.

Run Simple Path

```
run_simple_path(path_poses, velocity = None, acceleration = None)
```

This function runs a simple path, which is a path that uses interpolated trapezoidal movements and which is calculated online before its execution. If no velocity or acceleration is passed as argument, a standard value will be chosen. Note that in order to execute a path quickly, the robot needs to be located at the very first pose given in the path before executing 'run_simple_path'. Otherwise the robot will first move to the initial path position using the default velocity and acceleration values before running the path.

Parameter	*Types*	*Description*
path_poses	List of Poses	Contains all the postures of the path. The poses can be in the following formats: different types of robots: - name of pose saved in the database (e.g. **"myPose")** - actuator angles as list with values in degrees (e.g. **[-10, 30, 30, -20, 45, 90]**) - actuator angles as dictionary with values in degrees (e.g. **{"1": -10, "2": 30, "3": 30, "4": -20, ...}**) - posture as list with values in mm and degrees (e.g. **[600, 0, 300, [0, -25, 0]**) - posture as dictionary with values in mm and degrees - (e.g. **{"x": 600, "y": 0, "z": 300, "roll": 0, ...}**)
velocity	Float or List of Floats	The velocity of a simple path can be given as a single number or a list of numbers (in **degrees per second**). If a single velocity is given, this value is used as the maximum velocity for all path segments. If a list is entered, the given velocity values correspond to the maximum velocities in each segment of the path.
acceleration	Float or List of Floats	The acceleration of a simple path can be given as a single number or a list of numbers (in **degrees per second squared**). If a single acceleration is given, this value is used as the maximum acceleration for all path segments. If a list is entered, the given acceleration values correspond to the maximum acceleration in each segment of the path.

Examples:

```
run_simple_path([[750,   0, 400, [180,  0,  0]],
                 [900,   0, 410, [170, 10, 10]],
                 [700, 100, 410, [170, 10, 10]]], velocity=100, acceleration=50)
```

This example runs a simple path through the three given path poses. Due to the interpolated movements, the robot will only stop at the end of the path, but not in intermediate poses. The velocity and acceleration values of 100 °/s and 50 °/s^2, respectively, guarantees that the robot tries to move as fast as possible, but not exceeding these values at any time.

```
run_simple_path(["pose1", "pose2", "pose3", "pose4"],
                velocity=[30, 60, 90], acceleration=[50, 100, 150])
```

This example runs a path through four poses that have been saved as “pose1”, “pose2”, “pose3” and “pose4”. The robot will move fairly slow at first and increase its speed with each new path segment.

```
run_simple_path ([{"1": 20, "2": 20, "3": -20, "4": -20, "5": -20, "6": 20},
                  [-20, -20, 20, 20, 20, -20],
                  "myPose"])
```

This example runs a path that uses poses of different formats.

3. Gripper Functions

Open Gripper

```
open_gripper(position = None, velocity = None, acceleration = None)
```

This function opens the gripper with optional arguments such as opening angle, velocity or acceleration. By default, the gripper will open up to its upper angular limit.

Parameter	*Types*	*Description*
position	Float	This argument contains the opening angle (in **degrees**) of the gripper (e.g. **15**).
velocity	Float	This argument contains the angular velocity (in **degrees per second**) of the gripper (e.g. **200**).
acceleration	Float	This argument contains the angular acceleration (in **degrees per second squared**) of the gripper (e.g. **500**).

Examples:

```
open_gripper()
```

Open gripper to its upper angular limit using default values for velocity and acceleration.

```
open_gripper(15, 200, 500)
```

Open gripper to an opening angle of 15° by moving with a velocity of 200°/s and an acceleration of 500°/s^2.

Close Gripper

```
close_gripper(velocity = None, acceleration = None, current = None)
```

This function closes the gripper with optional arguments for position, velocity and acceleration. By default, the gripper will close until either its lower angular limit is reached or an object is grabbed.

Parameter	*Types*	*Description*
velocity	Float	This argument contains the angular velocity (in **degrees per second**) of the gripper (e.g. **200**).
acceleration	Float	This argument contains the angular acceleration (in **degrees per second squared**) of the gripper (e.g. **500**).
current	Float	This argument specifies the current (in **amperes**) that is applied on the gripper in order to hold an object (e.g. **0.2**).

Examples:

```
close_gripper()
```

Close gripper by using default values for velocity and acceleration to grab an object (if there is one).

```
close_gripper(200, 500)
```

Close gripper and trying to grab an object by moving the fingers with a velocity of 200°/s and an acceleration of 500°/s^2.

```
close_gripper(current = 0.2)
```

Grab an object (if there is one) and hold it with a gripper current of 0.2A.

Read Gripper Angle

```
read_gripper_angle()
```

This function reads the current gripper angle.

Parameter	*Types*	*Description*
return	Float	Returns the angle (in **degrees**) of the gripper.

Example:

```
close_gripper()
angle = read_gripper_angle()
if angle >= 10:
  print("big object has been grabbed")
elif 0 < angle < 10:
  print("small object has been grabbed")
else:
  print("no object has been grabbed")
```

By reading the gripper angle you can e.g. determine the size of an object. This example prints a message showing if a big or small object has been grabbed, by checking if the opening angle is bigger or smaller than 10°.

Read TCP Pose

```
read_tcp_pose()
```

This function reads the current pose of the tool center point (TCP).

Parameter	*Types*	*Description*
return	List	Returns the TCP pose in the format [x, y, z, roll, pitch, yaw], where [x, y, z] (in **mm**) represents the position and [roll, pitch, yaw] (in **degrees**) the orientation of the TCP with respect to the base frame of the robot. For more information about the coordinate frames, please refer to the myP User Manual.

Example:

```
[x, y, z] = read_tcp_pose()[:3]
z += 100
move_tool(x, y, z)
```

This example moves the TCP of the end-effector to a position located 100mm above the current end-effector position.

Recognize Object

```
recognize_object(lesson_name)
```

This function closes the gripper of P-Grip and uses its sensors to recognize an object that has been previously trained in a lesson. The object recognition compares all trained objects and returns the parameters of the object with the

Please note that you have to define multiple objects for P-Grip to recognize an object. You can also save and train P-Grip when it has no object

Parameter	*Types*	*Description*
lesson_name	String	This argument contains the name of the lesson that is used to detect the object (e.g. **"myLesson"**).
return	Dictionary	Returning all the following parameters of the recognized object in a dictionary: - name - weight (in **kg**) - size (as a list of coordinates **[x, y, z]**) - color (as a list of RGB values **[red, green, blue]**) - force (in **N**)

Example:

```
# the objects "smallObject" and "bigObject" have been previously saved in the database
# additionally, "noObject" has also been saved by training the gripper without gripping any object at all
# a lesson "myLesson" has been trained with all three predefined objects
my_object = recognize_object("myLesson")
if my_object["name"] == "smallObject":
   print("the small object has been recognized")
elif my_object["name"] == "bigObject":
   print("the big object has been recognized")
else:
   print("no object has been recognized")
print("the size of the recognized object is: " + str(my_object ["size"]))
```

This example uses a predefined lesson to recognize what object it currently grabs. The program will then print which object has been recognized and the size of this object.

4. User Communication

Text Dialog

```
dialog_text(dialog_text, dialog_title="", dialog_type=0)
```

This function opens a dialog window with an input field for entering a message. This message is then returned.

Parameter	*Types*	*Description*
dialog_text	String	The text displayed in the dialog window.
dialog_title	String	The title displayed in the header of the dialog window.
dialog_type	Integer	The dialog type does not affect the behaviour of this function, but it is simply to change the symbol displayed in the dialog window. - If this argument is **0**, a blue information sign is shown. - If this argument is **1**, a green check mark is shown. - If this argument is **2**, a red cross is shown.
return	String	The entered message is returned as a string. If other data types (e.g. integer, float, list) need to be entered, the string can be converted to the desired type.

Examples:

```
input_message = dialog_text("Please enter a message...", "User Input", 1)
print(input_message)
```

This example asks the user to enter a message. This message is then printed.

```
input_message = dialog_text("Please enter an integer...", "User Input")
input_integer = int(input_message)
```

This example asks the user to enter an integer in an informative window. The input is returned as a string and then converted into an integer.

Yes-No Dialog

```
dialog_yes_no(dialog_text, dialog_title="", dialog_type=0)
```

This function opens a dialog window .

Parameter	*Types*	*Description*
dialog_text	String	The text or question displayed in the dialog window.
dialog_title	String	The title displayed in the header of the dialog window.
dialog_type	Integer	The dialog type does not affect the behaviour of this function, but it is simply to change the symbol displayed in the dialog window. - If this argument is **0**, a blue information sign is shown. - If this argument is **1**, a green check mark is shown. - If this argument is **2**, a red cross is shown.
return	Bool	If the option "Yes" is chosen, True will be returned. If the option "No" is chosen, False will be returned.

Example:

```
good_day = dialog_text("Are you having a good day?", "Question", 1)
if good_day:
   print("I'm glad your having a good day!")
else:
   print("Don't worry, be happy!")
```

This example asks the user a simple question that can be answered with "Yes" or "No" and prints different messages according to the chosen answer.

```
while True:
   ...
   if error:
      error_check = dialog_text("An error occurred! Do you want to continue anyway?", "Custom Error", 2)
      if not error_check:
         break
```

This example executes some custom code (shown as "…") in a infinite while loop. If an error occurres, the user is asked to continue with the next iteration of this loop or to break out of the loop to exit the application.

Print

```
print(message)
```

This function prints a message into the Application Output section of the Application Menu.

Parameter	*Types*	*Description*
message	String	This argument contains the message to print.

Example:

```
print("Hello world!")
```

This example prints "Hello world!" in the output section of the application menu.

Wait

```
wait(seconds)
```

This function waits for time to pass before the next script command is executed.

Parameter	*Types*	*Description*
seconds	Float	This argument specifies how many seconds will be waited before the script continues its execution.

Examples:

```
move_to_pose("pose1")
wait(3)
move_to_pose("pose2")
```

This example moves the robot to two predefined poses and waits for 3 seconds between the two movements.

```
i = 0
while True:
  print(i)
  i += 1
  wait(0.1)
```

This example prints increasing numbers in an infinite while loop. A wait command controls the execution speed of this loop. With

Say

```
say(phrase, language = "gb", gender = "f")
```

This function sends a message to the browser, where it is converted into an audio file and executed. This function requires an active internet connection.

Parameter	*Types*	*Description*
phrase	String	This argument contains the message to say in the desired language (e.g. **"Hello world!"**).
language	String	This argument contains the language code of the phrase to speak (e.g. **"gb"** for UK English, **"de"** for German, **"fr"** for French or **"cn"** for Chinese/Mandarin).
gender	String	This argument contains the gender of the voice that speaks. Some languages are available in male (**"m"**), some in female (**"f"**) and some in both. If a gender is not available, the default voice will be used.

Examples:

```
say("Hello, how are you?", "en")
wait(4)
say("Hallo, wie geht es?", "de")
wait(4)
say("Bonjour, comment allez-vous?", "fr")
wait(4)
say("Salve, come sta? ", "it")
wait(4)
say("你好！你好吗？", "cn")
wait(4)
say("こんにちは、元気ですか？", "jp")
wait(4)
say("Привет, как дела?", "ru")
```

Saying "Hello, how are you?" in different languages.

List of available languages and their corresponding genders

Language	Code	Genders
Afrikaans	af	m
Albanian	sq	m
Arabic	ar	m
Armenian	hy	f
Australian	au	f
Bosnian	bs	m
Catalan	catalonia	m
Chinese / Mandarin	cn	f
Croatian	hr	m
Czech	cz	f / m
Danish	dk / da	f / m
Dutch	nl	f
English (UK)	gb	f / m
English (US)	us	f / m
Esperanto	eo	m
Finnish	fi	f / m
French	fr	f
German	de	f
Greek	gr	f / m
Hatian Creole	ht	f
Hindi / Tamil	hi	f / m
Hungarian	hu	f / m
Icelandic	is	m
Indonesian	id	f

Language	Code	Genders
Italian	it	f
Japanese	jp	f
Korean	kr	f
Latin	va	f / m
Latvian	lv	m
Macedonian	mk	m
Moldavian	md	m
Montenegrin	me	m
Norwegian	no	f / m
Polish	pl	f
Portuguese	br	f
Romanian	ro	m
Russian	ru	f
Serbian	sr	m
Serbo-Croatian	hr	m
Slovak	sk	f / m
Spanish	es	f
Swahili	sw	m
Swedish	sv	f / m
Thai	th	f
Turkish	tr	f
Vietnamese	vi	m
Welsh	cy	m

5. Running Subscripts

Run Script

```
run_script(script_name)
```

This function executes a previously saved script within the running script. The chosen script is executed entirely before the main script continues.

Parameter	*Types*	*Description*
script_name	String	This argument contains the name of the script to run (e.g. **"myScript"**).

Shared Variable

```
shared["variable"] = value
```

In order to share variables between a main script and its subscripts, the user may write to and read from the global dictionary 'shared'. When the main script is executed, the shared variable is always empty.

Example:

Let's consider a simple function like 'z = x + y'. By using the shared variable 'shared', we can implement this function as a subscript as follows:

```
# ... this is the script "myFunction" ...
x = shared["x"]
y = shared["y"]
z = x + y
shared["z"] = z
```

This script simply reads the shared input arguments, executes the desired function and then writes the shared return values.

Now using the function 'run_script', this subscript can be called as follows:

```
# ... this is the main script "mainScript" ...
x = 4
y = 5
shared["x"] = x
shared["y"] = y
run_script("myFunction")
z = shared["z"]
print("the result of 'x + y' is '" + str(z) + "'")
del shared["x"], shared["y"], shared["z"]
```

The main script "mainScript" first writes the shared input arguments of "myFunction", then this subscript is called, and finally the shared return values of "myFunction" are read. Optionally, unused variables can be deleted if they are no longer used.

6. Sensors and I/Os

The sensors in myP are split in two categories, internal sensors that are included in the robot or its end-effectors and external custom sensors. Internal sensors use the CAN bus of P-Rob to transfer data and they cannot be edited. External sensors are connected to the same network as P-Rob is connected to and they can be added, edited or removed. The following external sensors can be used with myP:

- TCP (Transmission Control Protocol) Sockets:
    - If a TCP socket connection is open (given IP address and port), sensor data can be sent through this socket as a byte string. On the myP side, the data arrives as a Python string and can be handled appropriately.
- Modbus:
    - Using an open connection to e.g. a modular WAGO I/O System, digital sensor data can be transferred to myP by connecting to its IP address and giving the size of the bits (multiple of 8) you want to read. On the myP side, the data arrives as a list of bits (0 or 1).
- RFID (Radio-Frequency Identification):
    - Using a RFID reader (e.g. Siemens Simatic RFID System), RFID tags can be detected, read and written using myP. On the myP side, the raw data is handled as XML strings and processed data is returned as Python strings.

Read Sensor Data

```
read_sensor_data(sensor_name)
```

This function reads the data of a sensor connected to the robot (e.g. through an open socket).

Parameter	*Types*	*Description*
sensor_name	String	This argument contains the name of the sensor (e.g. finger sensor **"finger_right_in"** or external sensor **"mySensor"**). The available internal and external sensors are shown in the sensor menu.
return	Various	Returns the data of a sensor. The data format is different for different types of sensors and depends on how this type of sensor is implemented in myP (e.g. finger sensors return integer values, TCP sockets return strings). For further information on the gripper and finger sensors, please refer to the **P-Grip User Manual**)

Examples:

```
open_gripper()
distance_open = read_sensor_data("gripper_distance")
close_gripper()
distance_close = read_sensor_data("gripper_distance")
print("distance sensor measurement if gripper is open: " + str(distance_open))
print("distance sensor measurement if gripper is closed: " + str(data_closed))
```

This example prints the current measurements of the distance sensor of P-Grip in open and closed gripper position.

```
# for this example, a custom sensor "mySensor" has been added to myP
sensor_data = read_sensor_data("mySensor")
```

This example saves the current sensor data of a custom sensor into a variable.

Send Sensor Instruction

```
send_sensor_instruction(sensor_name, command)
```

This function sends instructions to a sensor (e.g. through an open socket). In case of a generic sensor, this instruction will be send as a raw string. The instruction will be send in the next transmission. I case of specialized sensors, such as the optional RFID sensor, also additional parameters can be send.

Parameter	*Types*	*Description*
sensor_name	String	This argument contains the name of the sensor (e.g. external sensor **"mySensor"**) the instruction is sent to. The available external sensors are shown in the sensor menu.
command	String	This argument contains the instruction for the sensor in XML format.

Examples:

```
send_sensor_instruction('mySensor', '<command><reset>1</reset></command>')
```

This example sends an instruction in XML format to the sensor "mySensor".

Get Sensors

```
get_sensors()
```

This function returns a list of all available sensors in myP.

Parameter	*Types*	*Description*
return	List of Dictionaries	Returns a list of sensors dictionaries. Each dictionary contains the following data: - name (string) - type (string) - interval (float) - parameters (dictionary containing e.g. address, port)

Example:

```
available_sensors = get_sensors()
for sensor in available_sensors:
    print(sensor["name"])
```

This example prints the names of all available sensors.

Add Sensor

```
add_sensor(sensor_name, sensor_type, sensor_interval=0.5, sensor_parameters = None)
```

This function adds a new sensor to myP.

Parameter	*Types*	*Description*
sensor_name	String	This argument contains the name of the sensor to add (e.g. **"mySensor"**).
sensor_type	String	This argument contains the type of the sensor to add. Currently available types are: "TCP", "MODBUS", "RFID"
sensor_interval	Float	This argument specifies the time interval (in **seconds**), in which the sensor data is updated. Sensor data should not be read faster than this interval, otherwise all the values read during the same sensor time interval will be equal. Note that the smaller this interval is, the faster a sensor can be read, but the more computation time will be needed to update the sensor data.
sensor_parameters	Dictionary	This argument contains all parameters that are important for establishing its connection. Different sensor types have different parameters: - "TCP": {"address", "port"} - "MODBUS": {"address", "size"} - "RFID": {"address", "port"}

Remove Sensor

```
remove_sensor(sensor_name)
```

This function removes a predefined sensor from myP.

Parameter	*Types*	*Description*
sensor_name	String	This argument contains the name of the sensor to remove (e.g. **"mySensor"**).

Examples:

```
add_sensor("mySensor", "TCP", 0.1, {"address": "192.168.80.201", "port": 6000})
while True:
  sensor_data = read_sensor_data("mySensor")
  print(sensor_data)
  wait(0.1)
```

This example adds a TCP sensor to myP and prints the sensor data with a frequency of 10Hz. In order to reach this frequency, the sensor interval needs to be at least 0.1 seconds.

```
add_sensor("mySensor", "MODBUS", sensor_parameters={"address": "192.168.80.200", "size": 16)
sensor_data = get_sensor_data("mySensor")
remove_sensor("mySensor")
```

This example adds a ModBus sensor to myP, reads its data and removes it again.

Read Actuator Position

```
read_actuator_position(actuator_ids)
```

This function reads the position measurements of one or multiple joints.

Parameter	*Types*	*Description*
actuator_ids	Various	This argument contains the IDs of the joints of whose the position should be measured. It can be - the ID of a single joint (e.g. **6**), - a list of joint IDs for multiple joints (e.g. **[1,4,6]**), - the global constant **ALL_KIN_JOINTS**, which is equal to the list of all kinematic joints, - the global constant **ALL_ACTUATORS**, which is equal to the list of all actuators including end effectors.
return	List of Floats	Returns a list of position measurements (in **degrees**), one for each of the joints specified in **actuator_ids**.

Example:

```
position_data = read_actuator_position(ALL_KIN_JOINTS)
print("joint angles of all kinematic joints: " + str(position_data))
```

This example prints the current angles of all kinematic joints.

Read Actuator Current

```
read_actuator_current(actuator_ids)
```

This function reads the current measurements of one or multiple joints.

Parameter	*Types*	*Description*
actuator_ids	Various	This argument contains the IDs of the joints of which the current should be measured. It can be - the ID of a single joint (e.g. **6**), - a list of joint IDs for multiple joints (e.g. **[1,4,6]**), - the global constant **ALL_KIN_JOINTS**, which is equal to the list of all kinematic joints, - the global constant **ALL_ACTUATORS**, which is equal to the list of all actuators including end effectors.
return	List of Floats	Returns a list of current measurements (in **A**), one for each of the joints specified in **actuator_ids**.

Example:

```
current_data = read_actuator_current(4)
print("current of joint 4: " + str(current _data))
```

This example prints the current velocities of the joints 2 & 3

Read Digital Inputs & Outputs

```
read_digital_inputs(bits = None)
```

```
read_digital_outputs(bits = None)
```

These functions read either one, multiple or all of the 8-bit digital inputs and outputs of P-Rob. Note that both the inputs and outputs can be read, but only the digital outputs can be written.

Parameter	*Types*	*Description*
bits	Various	This argument specifies which bits to read. If specific bits are addressed, **integers between 1 and 8** need to be given, referring to the numbers of these digital inputs, which are also visible at the front of P-Rob. Using different argument formats, you may choose to - read all bits (by default or by using None), - read a single bit (giving an integer, e.g. **3**), - read multiple bits (giving a list of integers, e.g. **[1, 4, 5]**), - read multiple bits (giving a set of integers, e.g. **{7, 8}**).
return	Various	The return format is different for each possible type of the input argument "bits". It can be - an **integer** between 0 and 255 (if "bits" is None), - a **bool** (if "bits" is an integer), - a **list of bools** (if "bits" is a list of integers), - a **dictionary of bools** (if "bits" is a set of integers).

Example:

```
all_inputs = read_digital_inputs()
print(all_inputs)
print(bin(all_inputs))
```

This example reads all digital inputs and prints them as an integer number in decimal form and binary form (e.g. 115 and 01110011, hence the digital inputs {1, 2, 5, 6, 7} are high and the digital inputs {3, 4, 8} are low).

```
print(read_digital_inputs(3))
```

This example reads only the digital input 3 and prints its value (e.g. False, which means the the digital input 3 is low).

```
print(read_digital_inputs([1, 4, 5]))
```

This example reads the digital inputs {1, 4, 5} and prints their values as a list of bools (e.g. [True, False, True], which means that the inputs 1 and 5 are high and the input 4 is low).

```
print(read_digital_outputs({7, 8}))
```

This example reads the digital outputs {7, 8} and prints their values as a dictionary of bools (e.g. {"7": True, "8": False}, which means that the output 7 is high and the output 8 is low).

Write Digital Outputs

```
write_digital_outputs(values, bits = None)
```

These functions write either one, multiple or all of the 8-bit digital outputs of P-Rob. Note that only the outputs can be written, but both the digital inputs and outputs can be read. The input arguments "values" and "bits" need to have compatible formats, given in the list below.

Parameter	*Types*	*Description*
values	Various	This argument specifies the value to write to the digital outputs. If specific bits are addressed, Boolean values need to be written, where **True** refers to a high output voltage and **False** refers to a low output voltage. Using different argument formats, you may choose to write - an **integer** between 0 and 255 (if "bits" is None), - a **bool** (if "bits" is an integer), - a **list of bools** (if "bits" is a list of integers), - a **dictionary of bools** (if "bits" is None).
bits	Various	This argument specifies which bits to write. If specific bits are addressed, **integers between 1 and 8** need to be given, referring to the numbers of these digital outputs, which are also visible at the front of P-Rob. Using different argument formats, you may choose to - write all bits (by default or if "values" is an integer), - write a single bit (giving an integer, e.g. **3**), - write multiple bits (giving a list of integers, e.g. **[1, 4, 5]**), - write multiple bits (giving a set of integers, e.g. **{7, 8}**).

Example:

```
write_digital_outputs(115)
# ... this is similar to ...
write_digital_outputs(0b01110011)
```

This example writes the values 01110011 (or 115 in decimal format) to all digital outputs. Hence, the digital outputs {1, 2, 5, 6, 7} are high and the digital outputs {3, 4, 8} are low).

```
write_digital_outputs(False, 3)
```

This example writes the value False to the digital output 3, which means that the output 3 is set to low.

```
write_digital_outputs([True, False, True], [1, 4, 5])
```

This example writes the values {True, False, True} to the digital outputs {1, 4, 5}, which means that the outputs 1 and 5 are set to high and the output 4 is set to low.

```
write_digital_outputs({"7": True, "8": False})
```

This example writes the values {True, False} to the digital outputs {7, 8}, which means that the output 7 is set to high and the output 8 is set to low.

7. Calibration Functions

Calibrate Joint

```
calibrate_joint(actuator_id, direction)
```

This function calibrates one joint of the robotic arm into a desired direction. This function is useful to write custom calibration scripts, e.g. if the default calibration cannot be executed due to lack of space.

Parameter	*Types*	*Description*
actuator_id	String or Integer	This argument contains the joint ID of the joint to calibrate (e.g. **4**).
direction	Integer	This argument contains the direction of the joint to calibrate. If the direction is a positive number, the joint will be calibrated in its positive rotation direction (e.g. **-1** will cause a wrist joint to rotate around the negative z-axis or an elbow joint to rotate around the negative y-axis).

Finalize Calibration

```
finalize_calibration()
```

This function is needed to end the calibration procedure and update the program status once the calibration is done.

Example:

```
calibrate_joint(4, 1)
calibrate_joint(1, -1)
move_joint([1, 4], 0)
calibrate_joint(3, 1)
calibrate_joint(2, -1)
move_joint([2, 3], 0)
calibrate_joint(5)
finalize_calibration()
```

This example describes the default calibration procedure of a four-axis P-Rob. First, the wrist joints (1 & 4) move into their opposite mechanical stops one after the other and move to their calibrated zero position. Then this procedure is repeated for the elbow joints (2 & 3). After all robot joints are calibrated and the robot stands straight, the gripper (joint 5) is calibrated. Finally the status of the robot is set to 'calibrated' and the robot thereafter is ready to execute further applications.

8. Workspace Boundaries

Workspace boundaries are specific boundaries that the robot cannot touch or enter when moving. In myP, workspace boundaries are implemented as a set of forbidden zones that the robot is not allowed to enter. These zones are cuboid shaped and defined in coordinates given in the "base" frame of the robot. The type of a zone can be either "CUBOID", meaning that the inside of this cuboid is not allowed to be entered, or "INVERTED_CUBOID", meaning that the outside of this cuboid is not allowed to be entered. The robot always checks all the forbidden zones and makes sure that it doesn't move in one of them.

Get Forbidden Zones

```
get_forbidden_zones()
```

This function returns a list of all currently set forbidden zones. Each forbidden zone is cuboid shaped, where some of the boundaries can also be chosen infinitely far away.

Parameter	*Types*	*Description*
return	List of Dictionaries	Returns a list of dictionaries containing all forbidden zones. Each dictionary contains the following data: - name (string) - type (string) - limits (list of floats or None)

Example:

```
forbidden_zones = get_forbidden_zones()
for zone in forbidden_zones:
    print(zone["name"])
```

This example prints the names of all previously defined forbidden zones.

Add Forbidden Zone

```
add_forbidden_zone(zone_name, zone_type, zone_limits)
```

This function adds a new forbidden zone to the workspace boundaries.

Parameter	*Types*	*Description*
zone_name	String	This argument contains the name of the forbidden zone to add (e.g. **"floor"**).
zone_type	String	This argument contains the type of the forbidden zone to add. Currently available types are: - **"CUBOID"**, then the limits define a cuboid area in which the robot is not allowed to enter - **"INVERTED_CUBOID"**, then the limits define a cuboid area, which the robot can never leave
zone_limits	Dictionary containing Floats	This argument contains a dictionary of all the zone limits that the user wants to define, given in "base" coordinates. At least one limit, but not all of them, need to be given per zone. The limits can be addressed using the following keys: ["x_min", "x_max", "y_min", "y_max", "z_min", "z_max"] (e.g. **{"x_min": 400, "z_max": 0}**). If a zone limit is not given, this limit is assumed to be infinitely far away.

Remove Forbidden Zone

```
remove_forbidden_zone(zone_name)
```

This function removes a predefined forbidden zone from the workspace boundaries.

Parameter	*Types*	*Description*
zone_name	String	This argument contains the name of the forbidden zone to remove (e.g. **"floor"**).

Examples:

```
add_forbidden_zone("floor", "CUBOID", {"z_max": 0})
add_forbidden_zone("wall", "CUBOID", {"x_min": 400})
# ... this has the same effect as ...
add_forbidden_zone("floor_and_wall", "INVERTED_CUBOID", {"x_max": 400, "z_min": 0})
```

Let's assume that a robot is mounted on the floor and there is a wall located 40cm from the center of the base of the robot (in x direction). This example provides two possibilities to define both the floor and the wall as zones that are forbidden for the robot to enter.

```
remove_forbidden_zone("floor_and_wall")
add_forbidden_zone("table", "CUBOID", {"x_max": 100, "z_min": -50, "z_max": 0})
```

Now let's assume that we take the robot from the previous example and place it on a table with thickness 5cm, with the edge of the table being 10cm in front of the robot (from the center of its base). This example removes the unused zone and adds a new zone, such that the robot cannot move inside the table it stands on, but still is able to reach below the table. The table edges that are out of reach are neglected and assumed to be infinitely far away.

9. Vision Functions

The currently implemented vision function allow the user to train and detects objects by their contours to a camera mounted at the gripper. The following examples show how to use these functions in combination to detect objects:

Examples:

```
move_to_pose("overview")
contour_training("small_triangle")
raw_data, timestamp = contour_detection("small_triangle")
contours = {"1": 0, "2": 0, "3": 0, "4": 0}
for data in raw_data:
   n = data["contour_number"]
   contours[str(n)] += 1
for n in contours:
   print("contour nr. " + n + "has been detected " + contours[n] + " times")
```

This example first moves the robot to its position overviewing the whole scene and then prepares it for the task of training objects with a contour shape of a small triangle to its camera. First, the robot moves to an initial overview pose, then the contour training is initialized with the name "small_triangle" and finally, the contour training mode is started. The user now can switch through all detected contours by pressing "NEXT" until the small triangle is shown. Then, by pressing "CHOOSE" this shape is selected and by pressing "ID3" this shape is saved with the identification number 3. After training the triangular shape, the contour detection is initialized and objects with the shape of a small triangle are detected. Finally, raw data from the camera is read and the number of contours the camera detected are printed.

```
all_data, timestamp = contour_detection("small_triangle", object_name="tetrahedron")
for data in all_data:
   print("a tetrahedron is located approximately at the position " + str(data))
stop_camera_service()
```

Assuming that the previous example has been executed, this example now checks the current camera view for tetrahedrons (which have previously been defined as objects with the name "tetrahedron" and trained with the contour "small_triangle"). Then the locations of these objects are printed and the camera is stopped since no further images need to be analyzed.

Contour Training

contour_training(contour_name**)**

This function initializes and runs the contour training mode. In order to initialize, the gripper camera needs to see the complete surface, where the objects are placed for detection. Make sure that this surface is horizontal, otherwise the algorithm probably won't work properly. Please note that the camera initialization will take a few seconds to complete. Then, the contour training mode is started in the vision webpage. For more information about the vision webpage, have a look at the section "Vision Frontend" in the User Manual.

Parameter	*Types*	*Description*
contour_name	String	This argument contains the contour name that will be used to save the contours detected during the training mode (e.g. **"small_triangle"**).

Contour Detection

contour_detection(contour_name, fixed_z=None, object_name=None**)**

This function initializes and runs the contour detection mode. In this mode, the user can place objects in front of the camera. Make sure that this surface is perfectly horizontal (given by its fixed z-coordinate). The camera then recognizes up to four objects that match the trained contours and returns their coordinates and other useful data. Please note that, if you call this function for the first time, the initialization of the camera will take a few seconds before the detection starts running. After initialization, this function operates in weak real-time.

Parameter	*Types*	*Description*
contour_name	String	This argument contains the name of the contour to detect during detection mode (e.g. **"small_triangle"**).
fixed_z	Float	This argument specifies the fixed height (in **mm**) of the horizontal plane on which the objects are to be detected. This coordinate is used to determine the distance from the camera to the object. If this argument is given, 'object_name' cannot be given.
object_name	String	If this argument is specified, the size of the object with this name will be queried from the database. This size will then be used to estimate the distance to the object. If this argument is given, 'fixed_z' cannot be given.
return (1st value)	List of Various	This argument contains a list with each element containing the parameters of one detected object. The length of this list is equal to the number of detected objects. The object parameters depend on which input arguments are chosen. - If "fixed_z" is given, the object parameters are given as **[x, y]**, where x and y are given in the robot base coordinate frame. Please note that the z-coordinate is given by the input argument "fixed_z", but the real z-coordinate of the object might be slightly different. - If "object_name" is given, the parameters are given as **[x, y, z]**, where x, y and z are given in the robot base coordinate frame. - If neither "fixed_z" nor "object_name" is given, the object parameters are returned in raw camera data of the form **{"object_center": [x, y], "pick_point": [x, y], "major_axis": [x, y], "minor_axis": [x, y], "contour_number": n}**. The object center and the pick point (point where the object can be grabbed) are given in absolute camera pixel coordinates. The major and minor axes (vectors pointing along the long and short side of the object) are given in relative pixel coordinates. The contour number (1-4) specifies which of the previously trained contours belongs to this object.
return (2nd value)	Integer	This argument contains the timestamp of the returned data.

Stop Camera Service

```
stop_camera_service()
```

Once a vision function has been used and the gripper camera is running, this function stops the camera recordings.

10. Task Planner

Task functions are used to add smart behaviour to an already existing task (e.g. one that has been generated or created using the task generator).

In a first step this requires the user to define the structure of the task behaviour by adding

- **actions** that a task can execute,
- **features** and their task relevant states,
- **dependencies** between those features,
- **digital inputs** to link external input states to feature states,
- **rewards** for executing an action given the feature states and
- **effects** to link actions to feature state changes

to the task. In a second step the user can further improve the task performance by setting

- **probabilities** (conditional probability distributions) of the feature states and
- **initial states** to describe the current situation.

Example:

```
test_task = load_task("water_the_grass")
subtask = get_task("turn_on_sprinkler")
test_task.add_actions([subtask])
test_task.add_features(sprinkler=["on", "off"], grass=["wet", "dry"], rain=["true", "false"])
test_task.add_dependencies(("sprinkler", "grass"), ("rain", "grass"))
test_task.add_digital_inputs(5, "grass", dry=True, wet=False)
test_task.add_rewards("turn_on_sprinkler", reward_value=True,
                      grass="dry", sprinkler="off", rain="false")
test_task.add_effects("turn_on_sprinkler", sprinkler="on")
test_task.set_probabilities(sprinkler = [[0.2], [0.8]],
                            rain = [[0.3], [0.7]],
                            grass = [[0.99, 0.4, 0.6, 0], [0.01, 0.6, 0.4, 1]])
test_task.set_initial_states(sprinkler="on", grass="wet")
play_task(test_task)
save_task(test_task)
```

In the following, each of these functions will be explained in detail.

Get Task

```
get_task(task_name)
```

This function gets a task that was previously generated by the task generator. This task can then be edited using the following functions in this section to increase the intelligence and complexity of this task.

Parameter	*Types*	*Description*
task_name	String	This argument contains the name of the task to edit and play (e.g. **"myTask"**). This name should correspond to a task that was previously saved in the task generator. To see a list of all available tasks, you can enter the Task Generator menu.
return	Task object	Returns a Task instance. This object can be edited using the following functions to create a complex task.

Save Task

```
save_task(task)
```

Once a task was constructed, it can be saved in the database using this function. The name of the task remains the same (the one that was used in the function 'get_task' or 'load_task').

Parameter	*Types*	*Description*
task	Task object	This argument contains the constructed task to save.

Load Task

```
load_task(task_name)
```

Once a task was constructed and saved in the database it can be reloaded using this function. The difference between this function and 'get_task' is that this function should only be called after a task has been previously saved using 'save_task'.

Parameter	*Types*	*Description*
task_name	String	This argument contains the name of the task to edit and play (e.g. **"myTask"**). This name should correspond to a task that was previously saved using 'save_task'.
return	Task object	Returns a Task instance. This object can be edited using the following functions to create a complex task.

Play Task

```
play_task(task, simulation=False, abort_subtasks=True)
```

Once a task is constructed, loaded or saved, it can be played. A task without features will be executed in a fixed sequence as would be done in the task generator. A task with features will choose the best actions at every moment and terminate only when no best action is found. A task has the option to be played in simulation mode and to be aborted if errors occur within the execution of subtasks of this task. By default, the task will be executed with real movements (not in simulation mode) and errors In subtasks will be skipped.

Parameter	*Types*	*Description*
task	String or Task object	This argument can contain the name of a task saved in the database, or a task object after it has been loaded or constructed.
simulation	Boolean	Flag that determines whether the task should be run in simulation mode. In simulation mode, the movements will not be executed.
abort_subtasks	Boolean	Flag that determines whether subtasks should be aborted if an error occurs while in them.

Example:

```
test_task = get_task("water_the_grass")
save_task(test_task)
test_task = load_task("water_the_grass")
play_task(test_task, simulation=True, abort_subtasks=False)
```

This example gets a task that has been previously recorded using the task generator, then saves in into the database as a task object. After saving, the task is loaded again and finally played. Please note that usually only one of the two functions ‘get_task’ and ‘load_task’ is called, depending on whether the task has just been recorded or whether it has already been modified and saved. The ‘load_task’ function in this example could also be skipped, since the variable ‘test_task’ contains the same task before and after loading it.

Add Actions

```
<TASK>.add_actions(actions)
```

This function is used to add additional actions to a task. These actions are specific subtasks that the user wants the robot to perform during this task.

Parameter	*Types*	*Description*
actions	List of Task objects	This argument holds a list of subtasks to be added to this task. Use the function 'get_task' to get a specific predefined subtask.

Example:

```
test_task = get_task("test_task")
subtask_1 = get_task("subtask_1")
subtask_2 = get_task("subtask_2")
subtask_3 = get_task("subtask_3")
test_task.add_actions([subtask_1, subtask_2, subtask_3])
```

This example gets three saved subtasks and adds them to the edited task.

Add Features

```
<TASK>.add_features(**features)
```

In order to construct an intelligent task, features must be added to this task. These could be physical objects that are most relevant to the task (e.g. coffee cups, coffee capsules, coffee machine lever, etc.) or non-physical features (e.g. season, weather, time of day, etc.). In addition their most important states must be specified. For example for a coffee making application it might be helpful to only consider two states for a cup, namely "under the coffee machine"/"on the table" (or "in"/"out" for short).

Parameter	*Types*	*Description*
****features**	Unpacked Dictionary containing Lists of Strings	This arguments needs to have the following form: **feat1=["state1", "state2", ...], feat2=...** The keys of this dictionary are the object names. The values are lists of possible states these objects can be in.

Example:

```
test_task = get_task("test_task")
test_task.add_features(coffee_cup=["in", "out"], coffee_machine_lever=["up", "down"])
```

This example adds two features to the task, namely a coffee cup and a coffee machine lever. They have the simplified states "in"/"out" and "up"/"down", respectively.

Add Dependencies

```
<TASK>.add_dependencies(*dependencies)
```

In addition to features, dependencies between features must be added to the task. For example, in a task containing two features such as the current weather and the current season, a dependency could be inserted to specify that the weather depends on the season. In this example, season is said to be a parent of weather and weather is said to be a child of season.

Parameter	*Types*	*Description*
***dependencies**	Unpacked List of Tuples of Strings	These arguments are tuples containing two depending features. The features are given as strings by their names. The order of the features within a tuple is important, such that the feature depending on the other should always be at the end of the tuple. These arguments can be entered as follows: **("feat1", "feat2"), ("feat3", "feat4"), ...**

Example:

```
test_task = get_task("test task")
test_task.add_features(season=["summer", "winter"], weather=["snow", "sunny"])
test_task.add_dependencies(("season", "weather"))
```

This example adds the features "season" and "weather" to the task. To show that the weather depends on the season a dependency is then added in the form (season, weather) meaning that season is a parent of weather (and weather is a child of season).

Add Digital Inputs

```
<TASK>.add_digital_inputs(input_bit, feature_name, **state_values)
```

This function adds a digital input to a task and specifies, how this input affects the states of a feature. Each time a digital input is read, the specified features change their state according to the state values given in this function. Note that the feature linked to the digital input can only have two states and each state needs to be linked to one of the input values of the digital input.

Parameter	*Types*	*Description*
input_bit	Input	This argument specifies which input bit to read. Each input number (**between 0 and 7**) relates to one of the digital input bits of the robot.
feature_name	String	This argument contains the name of the feature that will be linked to the digital input in question.
****state_values**	Unpacked Dictionary containing Strings	State values are given as a dictionary relating each state to one of the values read by the digital input. These can be entered as follows: **state1=True, state2=False**

Example:

```
test_task = get_task("water_the_grass")
subtask = get_task("turn_on_sprinkler")
test_task.add_actions([subtask])
test_task.add_features(sprinkler=["on", "off"], grass=["wet", "dry"], rain=["true", "false"])
test_task.add_digital_inputs(5, "grass", dry=True, wet=False)
```

In this example a digital input is connected to the digital input bit 5 of the robot to measure the moisture state of the grass. This input is added to the task. When this input is read, the state of the feature “grass” is set to “dry” and “wet” for inputs True and False, respectively.

Add Rewards

```
<TASK>.add_rewards(action_name, do_action=True, reward_value=True, **influences)
```

This function adds a reward for executing (or not executing) an action (or subtask) in given circumstances. These rewards are important for the task behavior, since the task always tries to maximize its reward.

Parameter	*Types*	*Description*
action_name	String	This argument contains the name of an action that was previously added to the task. These can be skills or subtasks.
do_action	Boolean	This is a flag that determines whether the reward is given for executing a task (**True**) or not executing it (**False**).
reward_value	Boolean, Float	This argument contains a numerical reward value. A **positive reward** is good and is given if the execution of this action is desired. A **negative reward** is bad and is given if the execution of this action should be punished. A **reward of zero** means that the execution of this action is neither positive nor negative, we don't really care if this action is executed or not. Reward values can also be entered as Booleans, where **True** corresponds to a reward of +100 and **False** corresponds to a punishment of -100.
****influences**	Unpacked Dictionary containing Strings	Influences describe the particular circumstances in which the reward should be given. These can be entered as follows: **feat1="state1", feat2="state2", ...**

Example:

```
test_task = get_task("water_the_grass")
subtask = get_task("turn_on_sprinkler")
test_task.add_actions([subtask])
test_task.add_features(sprinkler=["on", "off"], grass=["wet", "dry"], rain=["true", "false"])
test_task.add_rewards("turn_on_sprinkler", reward_value=True,
                      grass="dry", sprinkler="off", rain="false")
```

In this example a task called "water_the_grass" is created together with a subtask called "turn_on_sprinkler". Again, the features "sprinkler", "grass" and "rain" are added to this task. Finally, a reward of +100 is given to the execution of "turn_on_sprinkler" if the grass is dry, the sprinkler is turned off and it is not raining.

Add Effects

```
<TASK>.add_effects(action_name, **effects)
```

This function provides a way to influence features by defining action effects. These effects update the states of features if the specified action is executed successfully. Note that sensors are read continuously and may therefore overwrite state changes performed by this function.

Parameter	*Types*	*Description*
action_name	String	This argument contains the name of an action that was previously added to the task. These can be skills or subtasks.
****effects**	Unpacked Dictionary of Strings	The effects are given as a dictionary relating features to the states they will have after a successful action. These can be entered as follows: **feat1="state1", feat2="state2", ...**

Example:

```
test_task = get_task("water_the_grass")
subtask = get_task("turn_on_sprinkler")
test_task.add_actions([subtask])
test_task.add_features(sprinkler=["on", "off"], grass=["wet", "dry"], rain=["true", "false"])
test_task.add_effects("turn_on_sprinkler", sprinkler="on")
```

In this example an action effect is added to the action "turn on sprinkler". Once this action is successfully executed, the sprinkler will change its state to "on".

Set Probabilities

```
<TASK>.set_probabilities(**probabilities)
```

This function adds conditional probability distributions (CPDs) to a task. In order to add these CPDs, dependencies between features need to be already defined using 'add_dependencies'. To further describe the dependencies between two features, each feature receives a conditional probability table. This table describes the probabilities of features being in specific states given all possible states of their parents. Note that this function does not add CPDs, but it overwrites CPDs that have been set previously.

Parameter	*Types*	*Description*
****probabilities**	Unpacked Dictionary containing Lists of Floats	This argument contains one or multiple CPDs given as two-dimensional arrays (or tables) with probability entries between 0 and 1 (which is equal to 0% and 100%, respectively). The number of columns of this table is defined by the number of states of the associated features (e.g. 2 for a feature with states "on"/"off"). The number of rows of this table is determined by multiplying the number of states of all parents of this features. In each column, the probability entries must add up to 1 (which is equal to 100%). The CPDs can be entered as follows: **feat1 = [[p11, p12, ...], [p21, p22, ...], ...],** **feat2 = ...** where the numbers in **p<r><c>** stands for the table entry at row **<r>** and column **<c>**.

Example:

```
test_task = get_task("test_task")
test_task.add_features(sprinkler = ["on", "off"], grass = ["wet", "dry"], rain = ["true", "false"])
test_task.add_dependencies(("sprinkler", "grass"), ("rain", "grass"))
test_task.set_probabilities(sprinkler = [[0.2], [0.8]], rain = [[0.3], [0.7]],
                            grass = [[0.99, 0.4, 0.6, 0], [0.01, 0.6, 0.4, 1]])
```

This example adds features called "sprinkler", "grass" and "rain" to the task. Since the state of the grass depends on the states of sprinkler and rain, dependencies between sprinkler and grass as well as rain and grass are added. Finally, the following three CPDs are added:

sprinkler "on"	0.2
sprinkler "off"	0.8

rain "true"	0.3
rain "false"	0.7

	rain "true"		rain "false"	
	sprinkler "on"	sprinkler "off"	sprinkler "on"	sprinkler "off"
grass "wet"	0.99	0.4	0.6	0
grass "dry"	0.01	0.6	0.4	1

The numbers in the tables for sprinkler and rain describe the current probabilities of these states (e.g. the current probability that it is not raining is 70%). The numbers given in the grass table must be interpreted as conditional probabilities (e.g. the probability of the grass being

wet in case of falling rain and active sprinkler is 99%). These conditional probabilities need to be determined (or calculated) for all possible combinations of the sprinkler and rain states before this probability table can be added to the task.

For the order of the probabilities within the defined tables, the order in which the feature states and dependencies have been added is very important. The rain states have been defined as ["true", "false"], which means that the corresponding column [30%, 70%] in the sprinkler table also need to be defined in the same order. The grass dependencies have been defined as [("sprinkler", "grass"), ("rain", "grass")], which means that the corresponding rows within the probability tables first consider a change of the sprinkler state and then a change of the rain state (this also means that the parent of the first added dependency always corresponds to the feature mentioned in the last header row of the table, the second dependency corresponds to the second last header row, etc.).

Set Initial States

```
<TASK>.set_initial_states(**evidences)
```

This function defines the evidences of features of a task. Before running the task, evidences of features can be defined. This is the only task function that can be used once the task has been built using 'build'. Note that this function does not add evidences, but it overwrites previously defined evidences.

Parameter	*Types*	*Description*
****evidences**	Unpacked Dictionary or Strings	This argument contains the evidences for some of the features. These can be entered as follows: **feat1="evidence1", feat2="evidence2", ...**

Example:

```
test_task = get_task("water_the_grass")
subtask = get_task("turn_on_sprinkler")
test_task.add_actions([subtask])
test_task.add_features(sprinkler=["on", "off"], grass=["wet", "dry"], rain=["true", "false"])
test_task.set_initial_states(sprinkler="on", grass="wet")
```

This example creates the features "sprinkler", "grass" and "rain". In this specific application it is known that at the beginning the grass is "wet" and the sprinkler is "on". The evidence of the rain is not initially given.