# Documentation de Programmation Robotique

Consignes traduites



Rev.2
FRM268S9131M

# Table des matières

# Bases de Python

# Standards de base de Python

Cette section explique certains principes de base de la programmation en Python pouvant être utilisés dans l'éditeur d'applications. Il s'agit simplement d'un résumé des standards Python (3.13) les plus importants ; bien entendu, de nombreuses autres fonctions intégrées sont disponibles. Gardez à l'esprit qu'une application robotique n'est pas seulement une application Python ; par conséquent, certaines fonctionnalités Python sont limitées ou ne peuvent pas être utilisées. Pour les utilisateurs Python avancés ou les tutoriels de programmation, nous renvoyons à la documentation officielle en ligne de Python 3.13.

## Commentaires

Les lignes de code suivantes représentent des commentaires qui ne sont pas exécutés lors de l'exécution.

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

## Définitions des variables

Python prend en charge de nombreux types de variables différents et ne fait pas de distinction entre la définition et la déclaration d'une variable.

```
my_boolean = True           # defining "my_boolean" as a Boolean with value True
my_integer = 3              # defining "my_integer" as an integer number with value 3
my_float = 3.1416           # defining "my_float" as a floating-point number 3.1416
my_string = "Hello World"   # defining "my_string" as a string with value "Hello
World"
```

## Conversions de variables

Les variables peuvent facilement être converties d'un type à un autre si les valeurs sont compatibles.

```
x = bool(x)    # converts x to a boolean
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
```

## Structures de données

Python prend en charge quatre types différents de structures de données:

- Une liste contient un ensemble ordonné d'éléments.

```
my_list = list()            # create an empty list
my_list = [1, 2, 3, 4, 5]   # create a list containing some numbers
my_list.append(6)           # add element to existing list
my_list[0] = 7              # sets the first list element to be 7
```

- Un tuple contient un ensemble ordonné mais immuable (non modifiable) d'éléments.

```
my_tuple = tuple()           # create an empty tuple
my_tuple = (1, 2, 3, 4, 5)   # create a tuple containing some numbers
```

- Un ensemble contient une collection non ordonnée d'éléments sans doublons.

```
my_set = set()                          # create an empty set
my_set = {1, 2, 3, 4, 5}                # create a set containing some numbers
my_set.intersection({1, 3, 5, 7, 9})    # intersection of two sets
my_set.union({1, 3, 5, 7, 9})           # union of two sets
```

- Un dictionnaire contient une combinaison non ordonnée de clés et de valeurs.

```
my_dictionary = dict()              # create an empty dictionary
my_dictionary = {"a": 1, "b": 2}    # create a dictionary containing key-value pairs
my_dictionary["a"] = 3              # sets the element "a" to be 3
```

## Bibliothèques et importations

Python fournit un ensemble de modules intégrés (la bibliothèque standard). Les modules suivants sont pris en charge:

- Opérations numériques et génération aléatoire – utiles pour les calculs, les fonctions mathématiques et la génération de valeurs aléatoires

```
import math, random, statistics
```

- Gestion du temps et des dates – travail avec des horodatages, des délais et des valeurs de date et d'heure

```
import time, datetime
```

- Stockage de données et formats de fichiers – lecture et écriture de données structurées telles que les formats JSON ou CSV

```
import json, csv
```

- Traitement de texte – manipulation de chaînes et recherche de motifs à l'aide d'expressions régulières

```
import re, string
```

- Outils d'itération avancés et fonctionnels – boucles efficaces, combinaisons et fonctions d'ordre supérieur

```
import itertools, functools
```

En plus de la bibliothèque standard, les bibliothèques tierces suivantes sont disponibles:

- Calcul numérique – traitement efficace des tableaux et des données numériques

```
import numpy
```

- Mathématiques d'orientation 3D – représentation des quaternions et transformations (liées à la robotique)

```
import pyquaternion
```

- Communication HTTP – envoi de requêtes aux services web et aux API REST

```
import requests
```

- Configuration et sérialisation de données – travail avec des données au format YAML

```
import yaml
```

Les modules standard suivants de Python ne sont pas disponibles dans les applications robot:

- Accès au système d'exploitation (p. ex. `os`, `shutil`, `subprocess`, `sys`)
- Parallélisation personnalisée (p. ex. `multiprocessing`, `threading`)
- Inspection et dépannage personnalisés (p. ex. `inspect`, `traceback`)

De plus, l'installation de bibliothèques tierces personnalisées n'est pas prise en charge dans l'environnement Python du robot.

# Fonctions Python de base

## Arguments de fonction obligatoires et optionnels

Considérons qu'une fonction a été définie avec les arguments suivants:

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

Les arguments de fonction arg1 et arg2 sont des arguments standards qui doivent être fournis lors de l'appel de cette fonction. Les arguments arg3 et arg4 ont une valeur par défaut et ne doivent donc pas nécessairement être fournis lors de l'appel de la fonction. Tous les appels de fonction suivants sont valides:

```
function("a", "b")                                # -> function("a", "b", "x", "y")
function("a", "b", "c")                           # -> function("a", "b", "c", "y")
function("a", "b", "c", "d")                      # -> function("a", "b", "c", "d")
function("a", "b", arg4="d")                      # -> function("a", "b", "x", "d")
function(arg1="a", arg2="b", arg3="c", arg4="d")  # -> function("a", "b", "c", "d")
function(arg3="c", arg2="b", arg1="a")            # -> function("a", "b", "c", "y")
```

## Arguments de fonction décompressés

Certaines fonctions mentionnées dans la documentation du code utilisent des arguments décompressés. Ces arguments sont utiles lorsque le nombre d'arguments d'une fonction peut varier. Considérons les arguments de fonction suivants:

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

Ces fonctions peuvent alors être utilisées de la manière suivante:

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

# Opérateurs Python de base

## Opérateurs de calcul

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

## Opérateurs de comparaison

Ici, x et y peuvent être de n'importe quel type.

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

Ici, x et y sont des nombres.

```
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
```

## Opérateurs logiques

Ici, x et y sont des booléens.

```
x and y   # logical AND returns True if x and y are both True
x or y    # logical OR returns True if either x or y is True
not x     # logical NOT returns True if x is False
```

## Opérateurs bit à bit

Ici, x et y sont des entiers.

```
x & y   # bitwise AND returns an integer resulting from bitwise logical AND of x and y
x | y   # bitwise OR returns an integer resulting from bitwise logical OR of x and y
```

# Programmation conditionnelle

## Condition If–Else

Les conditions `if`, `elif` (else if) et `else` peuvent être combinées selon les règles suivantes:

- Le bloc `if` est exécuté uniquement si sa condition renvoie True.
- Un bloc `elif` est optionnel et s'exécute uniquement si toutes les conditions `if` ou `elif` précédentes renvoient False et que sa propre condition renvoie True.
- Un bloc `else` est optionnel et s'exécute uniquement si toutes les conditions `if` ou `elif` précédentes ont renvoyé False.

Le code général if-elif-else ressemble à ceci:

```
if condition_1:
    # the following code is executed if 'condition_1' is True
    ...
elif condition_2:
    # the following code is executed if 'condition_1' is False
    # and 'condition_2' is True
    ...
elif condition_3:
    # the following code is executed if 'condition_1' and 'condition_2' are False
    # and 'condition_3' is True
    ...
else:
    # the following code is executed if all previous conditions leading up
    # to this 'else' statement are False
    ...
```

**Exemples:**

Vérification si une variable `my_integer` est égale à 5, 10 ou à aucun de ces nombres.

```
if my_integer == 5:
    print("your integer is 5.")
    print("have a nice day.")
elif my_integer == 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.")
```

## Boucle While

Une boucle while est exécutée et répétée selon les règles suivantes:

- Le bloc `while` s'exécute de manière répétée tant que sa condition renvoie True.
- Le mot-clé `continue` est optionnel et peut être utilisé pour passer à l'itération suivante de la boucle, en ignorant le code restant dans le bloc while.
- Le mot-clé `break` est optionnel et peut être utilisé pour sortir de la boucle.
- Un bloc `else` est optionnel (et rarement utilisé) et s'exécute si la condition de la boucle devient False. En d'autres termes, ce bloc n'est pas exécuté si la boucle while se termine par un break.

```
while condition:
    # the following code is executed repeatedly as long as 'condition' is True
    ...
    if condition_1:
        # the 'continue' keyword skips the remaining code of this loop
        # and jump to the next iteration
        continue
    if condition_2:
        # the 'break' keyword breaks out of this loop, no matter the loop condition
        break
else:
    # the following code is executed only if the while loop exits
    # by setting 'condition' to False
    ...
```

**Exemples:**

Affichage de tous les nombres de 1 à 10.

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

Boucle infinie: Les boucles infinies sont utiles lorsque vous voulez que le robot répète une tâche jusqu'à ce que vous l'arrêtiez manuellement, ou que vous sortiez de la boucle avec une condition ou fonction spécifique. Si la boucle s'exécute trop rapidement, il est recommandé d'ajouter des commandes d'attente pour laisser un peu de temps au processeur.

```
import time
while True:
    ...
    time.sleep(0.1)
```

## Boucle For

Une boucle for est exécutée et répétée selon les règles suivantes:

- Le bloc `for` s'exécute de manière répétée, une fois pour chaque élément dans une séquence itérable (par exemple une liste, un set, un dictionnaire ou une chaîne).
- Le mot-clé `continue` est optionnel et peut être utilisé pour passer à l'itération suivante, en ignorant le reste du code dans le bloc for.
- Le mot-clé `break` est optionnel et peut être utilisé pour sortir de la boucle.
- Un bloc `else` est optionnel (et rarement utilisé) et s'exécute après le traitement du dernier élément de la boucle for. En d'autres termes, ce bloc n'est pas exécuté si la boucle for se termine par un break.

```
for item in sequence:
    # the following code is executed once for each 'item' in 'sequence'
    ...
    if condition_1:
        # the 'continue' keyword skips the remaining code of this loop
        # and jump to the next iteration
        continue
    if condition_2:
        # the 'break' keyword breaks out of this loop, no matter the loop condition
        break
else:
    # the following code is executed only if the for loop exits
    # by finding no more 'item' in 'sequence'.
    ...
```

**Exemples:**

Affichage de tous les nombres de 1 à 5.

```
for index in range(5):
    print(index+1)
```

Affichage de mes fruits préférés.

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

Comptage des lettres dans `python`.

```
index = 1
for letter in "python":
    print(f"letter number {index} in 'python' is {letter}")
    index += 1
```

# Fonctions principales

## *class* Core

**Note**

Ceci est une collection de toutes les fonctions principales. Ces fonctions sont directement disponibles et peuvent être appelées dans le code des applications.

**move_joints(*joints*, *joint_position*, *joint_velocity=None*, *joint_acceleration=None*, *relative=False*, *block=True*)**

Déplacer une ou plusieurs articulations du robot vers un ensemble spécifique d'angles articulaires. Ce mouvement synchronise toutes les articulations pour qu'elles atteignent leur position cible en même temps.

**Paramètres:**

- **joints** (*int, str, list*) –
  Les IDs des actionneurs à déplacer. Spécifiez la valeur comme suit:
  - l'ID d'une seule articulation (ex. *6*)
  - une liste d'IDs pour plusieurs articulations (ex. *[1, 4, 6]*)
  - la constante *ALL_KIN_JOINTS* pour déplacer les six articulations
- **joint_position** (*float, list of float*) –
  Les positions angulaires des articulations à déplacer en [°]. Spécifiez la valeur comme suit:
  - un seul nombre pour déplacer toutes les articulations spécifiées à la même position (ex. *-30*)
  - une liste de nombres pour déplacer chaque articulation spécifiée à une position différente (ex. *[90, -45, 30]*)
- **joint_velocity** (*float or None*) – La vitesse maximale en pourcentage de la limite de vitesse des articulations. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *45* %). Si aucune valeur n'est spécifiée, la vitesse globale des articulations est utilisée.
- **joint_acceleration** (*float or None*) – L'accélération maximale en pourcentage de la limite d'accélération des articulations en mouvement. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *60* %). Si aucune valeur n'est spécifiée, l'accélération globale des articulations est utilisée.
- **relative** (*bool*) – Indique si les positions des articulations sont considérées comme un décalage relatif par rapport aux positions actuelles. Par défaut, les valeurs de position absolues sont utilisées.
- **block** (*bool*) – Indique s'il faut attendre la fin de ce mouvement avant de poursuivre l'exécution du programme. Notez que seules les motions synchronisées utilisant les mêmes articulations peuvent être exécutées successivement de manière non bloquante.

**Exemples:**

Déplacement de toutes les articulations du robot synchronisées à 0°, correspondant à la position debout verticale du robot:

```
# these are equivalent
move_joints(ALL_KIN_JOINTS, 0)
move_joints([1, 2, 3, 4, 5, 6], [0, 0, 0, 0, 0, 0])
```

Déplacement de l'articulation 6 de -30° depuis sa position actuelle:

```
move_joints(6, -30, relative=True)
```

Déplacement des articulations 2, 3 et 5 à 20°, -40° et 20°, respectivement. Le mouvement sera exécuté de sorte que chaque articulation termine son déplacement en même temps. Ce mouvement utilisera l'accélération maximale autorisée pour les articulations, mais seulement 50% de la vitesse maximale autorisée.

```
move_joints([2, 3, 5], [20, -40, 20], joint_velocity=50,
joint_acceleration=100)
```

**move_coordinates(*coordinates*, *configuration=None*, *joint_velocity=None*, *joint_acceleration=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *relative=False*, *block=True*)**

Déplacement du TCP (point central de l'outil) du robot vers un ensemble spécifique de coordonnées cartésiennes. Le robot se déplace le long d'une trajectoire optimale en temps, produisant des profils de vitesse trapézoïdaux pour chaque articulation.

**Paramètres:**

- **coordinates** (*Coordinates, list or dict*) –
  Les coordonnées cibles du TCP. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}
- **configuration** (*str or None*) – La configuration de la pose cible. Il s'agit d'une chaîne de "c1" à "c8".
- **joint_velocity** (*float or None*) – La vitesse maximale en pourcentage de la limite de vitesse des articulations. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *45* %). Si aucune valeur n'est spécifiée, la vitesse globale des articulations est utilisée.
- **joint_acceleration** (*float or None*) – L'accélération maximale en pourcentage de la limite d'accélération des articulations en mouvement. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *60* %). Si aucune valeur n'est spécifiée, l'accélération globale des articulations est utilisée.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

- **relative** (*bool*) – Indique si les coordonnées sont considérées comme un décalage relatif par rapport aux coordonnées actuelles. Par défaut, des valeurs de coordonnées absolues sont utilisées.
- **block** (*bool*) – Indique s'il faut attendre que ce mouvement soit terminé avant de continuer l'exécution du programme. Par défaut, cette fonction est bloquante.

**Exemples:**

Les coordonnées cibles peuvent être fournies sous différentes formes. Toutes les commandes spécifiées déplacent le TCP à la position x = 700 mm, y = -125 mm, z = 500 mm et aux angles d'orientation roll = 180°, pitch = 0° et yaw = -90° dans le repère de travail global.

```
move_coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
move_coordinates({"x": 700.0, "y": -125.0, "z": 500.0,
                  "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
move_coordinates(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

Si toutes les coordonnées ne sont pas spécifiées, les coordonnées actuelles du TCP sont utilisées pour les coordonnées restantes. Par exemple, il est possible de ne modifier que la composante x et le roll comme suit:

```
move_coordinates([500.0, None, None, 150.0, None, None])
move_coordinates({"x": 500.0, "roll": 150.0})
```

Dans les exemples ci-dessus, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
move_coordinates({"y": -100.0, "yaw": -60.0},
                 joint_velocity=25, joint_acceleration=100,
                 relative=True, tool_offset=[20, 0, 0, 0, 0, 0])
```

ce qui déplace de -100 mm supplémentaires dans la direction y et -60° dans la direction yaw à 50% de la vitesse maximale et avec l'accélération maximale. De plus, le TCP est décalé de 20mm dans la direction x par l'offset de l'outil.

Il existe plusieurs options pour définir le mouvement dans l'espace de l'outil, l'une d'elles est:

```
move_coordinates(get_reference_coordinates(), tool_offset=[0.0, 0.0, 100.0,
0.0, 0.0, 0.0])
```

qui se déplace le long de la direction z de l'outil.

**move_pose(*pose*, *joint_velocity=None*, *joint_acceleration=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *block=True*)**

Déplacement du TCP (point central de l'outil) du robot vers une pose spécifique. Le robot se déplace le long d'une trajectoire optimale dans le temps, ce qui produit des profils de vitesse trapézoïdaux dans chaque articulation.

**Paramètres:**

- **pose** (*Pose*, *str*, *or int*) –
  La pose cible du TCP. Spécifiez la valeur comme suit:
  - un objet Pose
  - le nom d'une pose prédéfinie (enregistrée dans la base de données)
  - l'ID d'une pose prédéfinie (enregistrée dans la base de données)
- **joint_velocity** (*float or None*) – La vitesse maximale en pourcentage de la limite de vitesse des articulations. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *45* %). Si aucune valeur n'est spécifiée, la vitesse globale des articulations est utilisée.
- **joint_acceleration** (*float or None*) – L'accélération maximale en pourcentage de la limite d'accélération des articulations en mouvement. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *60* %). Si aucune valeur n'est spécifiée, l'accélération globale des articulations est utilisée.
- **tool_offset** (*Coordinates*, *list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates*, *list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates*, *list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates

- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.
- **block** (*bool*) – Indique s'il faut attendre que ce mouvement soit terminé avant de continuer l'exécution du programme. Par défaut, cette fonction est bloquante.

**Exemples:**

Déplacement vers une pose de la base de données:

```
move_pose("fancy_pose")
move_pose(10)   # ID of the pose in the database
```

Une pose personnalisée peut également être spécifiée dans le script en utilisant la fonction `create_pose()`:

```
pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
move_pose(pose)
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
move_pose("final_pose", joint_velocity=45, joint_acceleration=60,
          work_offset=[0, 0, 50, 0, 0, 0], block=False)
```

ce qui déplace le robot vers la « final_pose » de la base de données avec 45% de la vitesse maximale des articulations et 60% de l'accélération maximale des articulations. De plus, un décalage est appliqué au repère de travail, décalant la pose cible de 50 mm dans la direction z.

Il existe plusieurs options pour définir le mouvement dans l'espace de l'outil, l'une d'elles est:

```
move_pose(get_reference_pose(), tool_offset=[0.0, 0.0, 100.0, 0.0, 0.0, 0.0])
```

qui se déplace le long de la direction z de l'outil.

**move_linear(*coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *relative=False*, *block=True*)**

Déplacement du TCP (point central de l'outil) du robot de sa pose actuelle vers une pose cible en ligne droite. Notez que la position et l'orientation sont interpolées linéairement le long de la trajectoire. Ce mouvement ne peut être exécuté que si chaque pose le long de la trajectoire est atteignable.

**Paramètres:**

- **coordinates** (*Coordinates, list, or dict*) –
  Les coordonnées cibles du TCP. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}
- **linear_velocity** (*float or None*) – La vitesse linéaire maximale de l'outil [mm/s] pour toutes les parties mobiles du robot.
- **linear_acceleration** (*float or None*) – L'accélération linéaire maximale de l'outil [mm/s$^2$] pour toutes les parties mobiles du robot.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates

- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.
- **relative** (*bool*) – Indique si les coordonnées sont considérées comme un décalage relatif par rapport aux coordonnées actuelles. Par défaut, des valeurs de coordonnées absolues sont utilisées.
- **block** (*bool*) – Indique s'il faut attendre que ce mouvement soit terminé avant de continuer l'exécution du programme. Par défaut, cette fonction est bloquante.

**Exemples:**

Les coordonnées cibles peuvent être fournies sous différentes formes. Toutes les commandes spécifiées déplacent le TCP linéairement vers la position x = 600 mm, y = -125 mm, z = 500 mm et angles d'orientation roll = 180°, pitch = 0° et yaw = -90° dans le repère de travail global.

```
move_linear([600.0, -125.0, 500.0, 180.0, 0.0, -90.0])
move_linear({"x": 600.0, "y": -125.0, "z": 500.0,
             "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
move_linear(Coordinates([600.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

Si toutes les coordonnées ne sont pas spécifiées, les coordonnées TCP actuelles sont utilisées pour les coordonnées restantes. Par exemple, les composants x et roll peuvent être modifiés comme suit:

```
move_linear([500.0, None, None, 150.0, None, None])
move_linear({"x": 500.0, "roll": 150.0})
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
move_linear({"x": 100.0}, linear_velocity=500, linear_acceleration=1000,
            relative=True, work_frame=[20, 0, 0, 0, 0, 0])
```

ce qui déplace 100 mm supplémentaires dans la direction x avec une vitesse de 500 mm/s et une accélération de 1000 mm/s$^2$. De plus, le repère de travail est remplacé par les coordonnées données.

Il existe plusieurs options pour définir le mouvement dans l'espace de l'outil, l'une d'elles est:

```
move_linear(get_reference_coordinates(), tool_offset=[0.0, 0.0, 100.0, 0.0,
0.0, 0.0])
```

ce qui se déplace linéairement le long de la direction z de l'outil.

**move_circular(*intermediate_position*, *coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *relative=False*, *block=True*)**

Déplacement du TCP (point central de l'outil) du robot depuis sa pose actuelle vers une pose cible le long d'une trajectoire circulaire, en passant par une position intermédiaire. Notez que l'orientation est interpolée linéairement le long de la trajectoire. Ce mouvement ne peut être exécuté que si chaque pose le long de la trajectoire est atteignable.

**Paramètres:**

- **intermediate_position** (*list*) – Une position intermédiaire sur le chemin vers les coordonnées cibles. La position intermédiaire détermine la courbure du mouvement circulaire.
- **coordinates** (*Coordinates, list, or dict*) –
  Les coordonnées cibles du TCP. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}
- **linear_velocity** (*float or None*) – La vitesse linéaire maximale de l'outil [mm/s] pour toutes les parties mobiles du robot.
- **linear_acceleration** (*float*) – L'accélération linéaire maximale de l'outil [$mm/s^2$] pour toutes les parties mobiles du robot.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –

La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:

- un objet Coordinates
- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.
- **relative** (*bool*) – Indique si les coordonnées sont considérées comme un décalage relatif par rapport aux coordonnées actuelles. Par défaut, des valeurs de coordonnées absolues sont utilisées.
- **block** (*bool*) – Indique s'il faut attendre que ce mouvement soit terminé avant de continuer l'exécution du programme. Par défaut, cette fonction est bloquante.

**Exemples:**

Le robot peut être déplacé circulairement vers la position cible x = 0 mm, y = -500 mm, z = 500 mm et angles d'orientation roll = 180°, pitch = 0° et yaw = -90° dans le repère de travail global. Pendant son mouvement, il passera par la position intermédiaire x = 0 mm, y = -500 mm, z = 500 mm.

```
move_circular([0.0, -500.0, 500.0], [-500.0, -125.0, 500.0, 180.0, 0.0,
-90.0])
```

Si toutes les coordonnées ne sont pas spécifiées, les coordonnées TCP actuelles sont utilisées pour les coordonnées restantes.

```
move_circular([0.0, -500.0, None], [-500.0, -125.0, None, None, None, None])
move_circular([0.0, -500.0, None], {"x": -500.0, "y": -125.0})
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
move_circular([500.0, -300.0, 100.0], [-1000.0, 0.0, 0.0, 20.0, -10.0, 0.0],
              linear_velocity=500, linear_acceleration=1000,
              tool_offset={"x": 20}, relative=True)
```

ce qui déplace le TCP vers les coordonnées spécifiées avec une vitesse de 500 mm/s et une accélération de 1000 mm/s². De plus, le TCP est décalé de 20 mm dans la direction x par le décalage de l'outil.

**move_orientation(*tool_orientation*, *angular_velocity=None*, *angular_acceleration=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *relative=False*, *block=True*)**

Déplacement du TCP (point central de l'outil) de manière à ce que seule l'orientation change. Le robot pivote ensuite linéairement selon l'orientation souhaitée de l'outil tout en maintenant sa position constante. Ce mouvement ne peut être exécuté que si chaque pose le long de la trajectoire est atteignable.

**Paramètres:**

- **tool_orientation** (*list*) – Orientation souhaitée de l'outil en angles nautiques (roll, pitch, yaw) [°]
- **angular_velocity** (*float or None*) – Vitesse angulaire maximale de l'outil [°/s] du TCP.
- **angular_acceleration** (*float or None*) – Accélération angulaire maximale de l'outil [°/s²] du TCP.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
    - un objet Coordinates
    - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
    - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.
- **relative** (*bool*) – Indique si les coordonnées sont considérées comme un décalage relatif par rapport aux coordonnées actuelles. Par défaut, des valeurs de coordonnées absolues sont utilisées.
- **block** (*bool*) – Indique s'il faut attendre que ce mouvement soit terminé avant de continuer l'exécution du programme. Par défaut, cette fonction est bloquante.

**Exemples:**

L'orientation cible peut spécifier tous les angles ou seulement certains, en laissant "None" pour les autres. Dans ce cas, l'orientation actuelle est utilisée pour les angles manquants.

```
move_orientation([150.0, -20.0, -60.0])
move_orientation([150.0, None, -60.0])
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
move_orientation([-60, 0, 0], angular_velocity=250, angular_acceleration=3000,
                 tool_frame=[0, 0, 0, 0, 20, 0], relative=True)
```

Ce qui correspond à un mouvement relatif de -60° dans la direction du roulis avec une vitesse de 250°/s et une accélération de 3000°/s². De plus, le repère de l'outil est écrasé avec les coordonnées données.

Il existe plusieurs options pour définir le mouvement dans l'espace de l'outil, l'une d'elles est:

```
move_orientation(get_reference_coordinates().orientation, tool_offset=[0.0,
0.0, 0.0, 10.0, 0.0, 0.0])
```

Qui effectue une rotation autour de la direction z de l'outil.

**move_waypoints(*poses*, *joint_velocity=None*, *joint_acceleration=None*, *blend_radius=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *block=True*)**

Déplace le TCP (point central de l'outil) à travers les points intermédiaires définis, en partant de la pose actuelle du robot.

**Paramètres:**

- **poses** (*list of str, int or Pose*) – Une liste de points intermédiaires.
- **joint_velocity** (*float or None*) – La vitesse maximale en pourcentage de la limite de vitesse des articulations. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *45* %). Si aucune valeur n'est spécifiée, la vitesse globale des articulations est utilisée.
- **joint_acceleration** (*float or None*) – L'accélération maximale en pourcentage de la limite d'accélération des articulations en mouvement. C'est un seul nombre pour limiter toutes les articulations à la même valeur maximale (par ex. *60* %). Si aucune valeur n'est spécifiée, l'accélération globale des articulations est utilisée.
- **blend_radius** (*float or None*) – Rayon de fusion autorisé [mm] définissant l'écart maximal par rapport aux points intermédiaires dans l'espace de l'outil lors de la transition entre les points.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]

- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.
- **block** (*bool*) – Indique s'il faut attendre la fin de ce mouvement avant de poursuivre l'exécution du programme. Notez que seules les motions synchronisées utilisant les mêmes articulations peuvent être exécutées successivement de manière non bloquante.

**Exemples:**

Les points intermédiaires peuvent être donnés sous différentes formes ( `Pose` objet, nom de la pose dans la base de données, ID de la pose dans la base de données):

```
first_waypoint = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
move_waypoints([first_waypoint, "pick_pose", 6])
```

Dans les exemples ci-dessus, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
move_waypoints([first_waypoint, "pick_pose", 6], joint_velocity=25,
               joint_acceleration=100, blend_radius=50)
```

Qui se déplace le long des points intermédiaires en permettant une déviation maximale de 50mm. Il utilise 50% de la vitesse maximale et l'accélération maximale.

**move_segments(*segments*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *block=True*)**

Déplace le TCP (point central de l'outil) selon les segments définis. Il commence à la pose actuelle du robot.

**Paramètres:**

- **segments** (*list*) – Une liste de segments constituant le trajet.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.
- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

- **block** (*bool*) – Indique s'il faut attendre la fin de ce mouvement avant de poursuivre l'exécution du programme. Notez que seules les motions synchronisées utilisant les mêmes articulations peuvent être exécutées successivement de manière non bloquante.

**Exemples:**

Les segments peuvent être définis dans le script en utilisant des objets Segment ( `LinearSegment` , `CircularSegment` , ...):

```
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
pose_seg = PoseSegment("place_2")

move_segments([lin_seg, pose_seg])
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
pose_seg = PoseSegment("place_2")

move_segments([lin_seg, pose_seg], tool_offset=[20, 0, 0, 0, 0, 0],
block=False)
```

Qui exécute les segments en utilisant un léger décalage du TCP défini par l'offset de l'outil. De plus, le mouvement est défini comme non bloquant.

**run_path(*path*, *previous_angles=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*, *block=True*)**

Déplace le TCP (point central de l'outil) selon le chemin spécifié. D'abord, le robot est déplacé à la pose initiale du chemin en utilisant les vitesses et accélérations par défaut. Ensuite, les segments sont exécutés les uns après les autres selon leur spécification.

**Paramètres:**

- **path** (*Path, str, or int*) – Le chemin que doit suivre le TCP.
- **previous_angles** (*list or None*) –
  Les angles pour lesquels la solution la plus proche pour la pose initiale sera calculée. Spécifiez la valeur comme suit:
  - None, pour utiliser les angles actuels du robot
  - Une liste d'angles articulaires [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.
- **work_frame** (*Coordinates, list, dict, or None*) –

La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:

- un objet Coordinates
- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

- **block** (*bool*) – Indique s'il faut attendre la fin de ce mouvement avant de poursuivre l'exécution du programme. Notez que seules les motions synchronisées utilisant les mêmes articulations peuvent être exécutées successivement de manière non bloquante.

**Exemples:**

Exécuter un chemin depuis la base de données:

```
run_path("fancy_path")
run_path(10)   # ID of the path in the database
```

Un chemin personnalisé peut également être spécifié dans le script en utilisant la fonction `create_path()`:

```
path = create_path(start_pose, [LinearSegment(...), ...])
run_path(path)
```

Une autre option consiste à utiliser l'objet `Path` pour créer un chemin et l'exécuter:

```
path = Path("start_pose", [])
path.add_linear([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])

run_path(path)
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
run_path("fancy_path", tool_frame=[20, 0, 0, 0, 0, 0], block=False)
```

Qui exécute le « fancy_path » en utilisant un léger décalage du TCP défini par le repère de l'outil. De plus, le mouvement est défini comme non bloquant.

**wait_for_motor(*actuator_ids=None*)**

Attente que un ou plusieurs moteurs d'articulation cessent de bouger.

**Paramètres:**

**actuator_ids** (*None, str, int, list, set*) –

Les ID des actionneurs à attendre. Spécifiez la valeur comme suit:

- l'ID d'une seule articulation (ex. *6*)
- une liste d'IDs pour plusieurs articulations (ex. *[1, 4, 6]*)
- un ensemble de noms d'articulations pour plusieurs articulations (ex. *{« 1 », « 4 », « 6 »}*)
- la constante *ALL_KIN_JOINTS* ou *None* pour lire toutes les articulations

**Exemples:**

Cet exemple montre que le temps pendant lequel un robot se déplace peut être utilisé à des fins personnalisées.

```
# move to the 'start pose'
move_pose("start_pose", block=True)
# start moving towards the 'end pose'
move_pose("end_pose", block=False)
# the idle time can be used e.g. for calculations, observations, etc.
#custom_function()
# wait for the non-blocking motion to end
wait_for_motor()
```

### stop_motion()

Arrêt du mouvement en envoyant un signal d'arrêt à tous les actionneurs connectés.

**Exemples:**

Cet exemple montre que les mouvements non bloquants peuvent être interrompus en utilisant la fonction "stop_motion". Ici, nous voulons arrêter le mouvement si une entrée numérique spécifique change d'état.

```
# move to the 'start pose'
move_pose("start_pose", block=True
# start checking for the value of a digital input
initial_value = read_digital_main_input(1)
# start moving towards the 'end pose'
move_pose("target_pose", block=False)
# stop the motion as soon as the value of the digital input toggles
while is_motor_moving() and read_digital_main_input(1) == initial_value:
    wait(0.1)
stop_motion()
```

### is_motor_moving(*actuator_ids=None*)

Vérifier si un ou plusieurs moteurs sont en mouvement

**Paramètres:**

**actuator_ids** (*None, str, int, list, set*) –

Les ID des actionneurs à vérifier. Spécifiez la valeur comme suit:

- l'ID d'une seule articulation (ex. *6*)
- une liste d'IDs pour plusieurs articulations (ex. *[1, 4, 6]*)
- un ensemble de noms d'articulations pour plusieurs articulations (ex.: *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* ou *None* pour lire toutes les articulations

**Renvoie:**

Indique si les moteurs spécifiés sont en mouvement. Peut être l'un des suivants:

- un booléen, si un seul moteur a été vérifié
- une liste de booléens, si plusieurs moteurs ont été vérifiés

**Type renvoyé:**

bool or list of bool

**Exemples:**

Cet exemple montre que les mouvements non bloquants peuvent être interrompus en utilisant la fonction "stop_motion". Ici, nous voulons arrêter le mouvement si une entrée numérique spécifique change d'état.

```
# move to the 'start pose'
move_pose("start_pose", block=True
# start checking for the value of a digital input
initial_value = read_digital_main_input(1)
# start moving towards the 'end pose'
move_pose("target_pose", block=False)
# stop the motion as soon as the value of the digital input toggles
while is_motor_moving() and read_digital_main_input(1) == initial_value:
    wait(0.1)
stop_motion()
```

### set_global_joint_velocity(*joint_velocity*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `joint_velocity` utilisé dans les fonctions de mouvement point à point. La valeur de la vitesse articulaire est donnée en pourcentage de la vitesse articulaire maximale.

**Paramètres:**

**joint_velocity** (*float*) – La vitesse articulaire globale en [%] de la vitesse articulaire maximale.

**Exemples:**

Appel de deux mouvements point à point consécutifs en utilisant les mêmes limites de vitesse.

```
set_global_joint_velocity(40)
move_pose("pose_1")
move_pose("pose_2")
# ... this is similar to ...
move_pose("pose_1", joint_velocity=40)
move_pose("pose_2", joint_velocity=40)
```

### set_global_joint_acceleration(*joint_acceleration*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `joint_acceleration` utilisé dans les fonctions de mouvement point à point. La valeur d'accélération des articulations est donnée en pourcentage de la limite maximale d'accélération des articulations.

**Paramètres:**

**joint_acceleration** (*float*) – L'accélération globale des articulations en [%] de l'accélération maximale des articulations.

**Exemples:**

Appel de deux mouvements point à point consécutifs en utilisant les mêmes limites d'accélération.

```
set_global_joint_acceleration(80)
move_pose("pose_1")
move_pose("pose_2")
# ... this is similar to ...
move_pose("pose_1", joint_acceleration=80)
move_pose("pose_2", joint_acceleration=80)
```

### set_global_linear_velocity(*linear_velocity*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `linear_velocity` utilisé dans les fonctions de mouvement de l'outil et de planification de trajectoire. La valeur de la vitesse linéaire limite la vitesse linéaire maximale de l'outil et de toutes les parties mobiles du robot.

**Paramètres:**

**linear_velocity** (*float*) – La vitesse linéaire globale en [mm/s].

**Exemples:**

Appel de deux mouvements linéaires consécutifs en utilisant les mêmes limites de vitesse.

```
set_global_linear_velocity(150)
move_linear("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_linear("pose_1", linear_velocity=150)
move_linear("pose_2", linear_velocity=150)
```

### set_global_linear_acceleration(*linear_acceleration*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `linear_acceleration` utilisé dans les fonctions de mouvement de l'outil et de planification de trajectoire. La valeur d'accélération linéaire limite l'accélération linéaire maximale de l'outil et de toutes les parties mobiles du robot.

**Paramètres:**

**linear_acceleration** (*float*) – L'accélération linéaire globale en [mm/s²].

**Exemples:**

Appel de deux mouvements linéaires consécutifs en utilisant les mêmes limites d'accélération.

```
set_global_linear_acceleration(600)
move_linear("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_linear("pose_1", linear_acceleration=600)
move_linear("pose_2", linear_acceleration=600)
```

### set_global_angular_velocity(*angular_velocity*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `angular_velocity` utilisé dans les fonctions de mouvement rotationnel. La valeur de la vitesse angulaire limite la vitesse de rotation de l'outil.

**Paramètres:**

**angular_velocity** (*float*) – La vitesse angulaire globale en [°/s].

**Exemples:**

Appel de deux mouvements rotationnels consécutifs en utilisant les mêmes limites de vitesse.

```
set_global_angular_velocity(20)
move_orientation("pose_1")
move_orientation("pose_2")
# ... this is similar to ...
move_orientation("pose_1", angular_velocity=20)
move_orientation("pose_2", angular_velocity=20)
```

### set_global_angular_acceleration(*angular_acceleration*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `angular_acceleration` utilisé dans les fonctions de mouvement rotationnel. La valeur d'accélération angulaire limite l'accélération rotationnelle de l'outil.

**Paramètres:**

**angular_acceleration** (*float*) – L'accélération angulaire globale en [°/s$^2$].

**Exemples:**

Appel de deux mouvements rotationnels consécutifs en utilisant les mêmes limites d'accélération.

```
set_global_angular_acceleration(45)
move_orientation("pose_1")
move_orientation("pose_2")
# ... this is similar to ...
move_orientation("pose_1", angular_acceleration=45)
move_orientation("pose_2", angular_acceleration=45)
```

### set_global_blend_radius(*blend_radius*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `blend_radius` utilisé dans les fonctions de mouvement et de trajectoire.

**Paramètres:**

**blend_radius** (*float*) – Le rayon de fusion global en [mm].

**Exemples:**

Appel de deux mouvements consécutifs sur les points de passage en utilisant le même rayon de fusion.

```
set_global_blend_radius(100)
move_waypoints(["pose_1, "pose_2"])
# ... this is similar to ...
move_waypoints(["pose_1, "pose_2"], blend_radius=100)
```

### set_global_work_frame(*work_frame*)

Définition d'une nouvelle valeur par défaut globale pour l'argument `work_frame` utilisé dans les fonctions de mouvement et de cinématique. Le cadre de travail est la transformation qui modifie le référentiel du robot, donné par rapport au cadre de base.

**Paramètres:**

**work_frame** (*Coordinates, list, or dict*) –

Le repère de travail global, donné par rapport au repère de base. Spécifiez la valeur comme suit:

- un objet Coordinates
- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

**Exemples:**

Exécution de deux mouvements consécutifs en utilisant le même repère de travail.

```
my_work_frame = Coordinates([450, -300, 0, 0, 0, 90])

set_global_work_frame(my_work_frame)
move_pose("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_pose("pose_1", work_frame=my_work_frame)
move_linear("pose_2", work_frame=my_work_frame)
```

### set_global_tool_frame(*tool_frame*)

Définition d'une nouvelle valeur par défaut globale pour le `tool_frame` utilisé dans les fonctions de mouvement et cinématiques. Le repère de l'outil est la transformation modifiant le TCP du robot, donnée par rapport au repère de la bride.

**Paramètres:**

**tool_frame** (*Coordinates, list, or dict*) –

Le repère d'outil global, donné par rapport au repère de la bride. Spécifiez la valeur comme suit:

- un objet Coordinates
- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

**Exemples:**

Exécution de deux mouvements consécutifs en utilisant le même repère d'outil.

```
my_tool_frame = Coordinates([0, 0, 65, 0, 0, 180])

set_global_tool_frame(my_tool_frame)
move_pose("pose_1")
move_linear("pose_2")
# ... this is similar to ...
move_pose("pose_1", tool_frame=my_tool_frame)
move_linear("pose_2", tool_frame=my_tool_frame)
```

**get_current_pose(*tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

Lecture de la pose actuelle du TCP (point central de l'outil) du robot. Si aucun repère n'est donné, la pose est lue par rapport aux repères globaux.

**Paramètres:**

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

La pose actuelle du TCP du robot.

**Type renvoyé:**

Pose

**Exemples:**

Lecture des propriétés d'une pose de deux manières différentes.

```
pose = get_current_pose()
coordinates = pose.coordinates
configuration = pose.configuration
# ... this is similar to ...
coordinates = get_current_coordinates()
configuration = get_current_configuration()
```

Notez que la pose actuelle est lue à partir des données des capteurs. Les capteurs peuvent fluctuer et ne renvoyer pas toujours des valeurs précises. Par conséquent, l'exemple suivant de déplacement avant-arrière par rapport à la pose actuelle entraînera un décalage inattendu au fil du temps. Dans ce cas, il est préférable d'utiliser la fonction *get_reference_pose*.

```
while True:
    move_pose(get_current_pose(), tool_offset=[0.0, 0.0, 100.0, 0.0, 0.0,
0.0])
    move_pose(get_current_pose(), tool_offset=[0.0, 0.0, -100.0, 0.0, 0.0,
0.0])
```

**get_current_coordinates(*tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

Lecture des coordonnées actuelles du TCP (point central de l'outil) du robot par rapport aux repères de coordonnées donnés. Si aucun repère n'est spécifié, elle renvoie les coordonnées par rapport aux repères globaux.

**Paramètres:**

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

Les coordonnées actuelles du TCP du robot.

**Type renvoyé:**

Coordinates

**Exemples:**

Lecture des coordonnées actuelles du TCP dans les repères de coordonnées globaux.

```
coordinates = get_current_coordinates()
```

Notez que les coordonnées actuelles sont lues à partir des données des capteurs. Les capteurs peuvent fluctuer et ne pas toujours renvoyer des valeurs précises ; par conséquent, l'exemple suivant de mouvement aller-retour par rapport à la pose actuelle entraînera une dérive inattendue au fil du temps. Dans ce cas, il est préférable d'utiliser la fonction *get_reference_coordinates*.

```
while True:
    move_coordinates(get_current_coordinates(), tool_offset=[0.0, 0.0, 100.0,
0.0, 0.0, 0.0])
    move_coordinates(get_current_coordinates(), tool_offset=[0.0, 0.0, -100.0,
0.0, 0.0, 0.0])
```

### get_current_configuration()

Lecture du nom de la configuration de la pose actuelle du robot. Notez que cette configuration est indépendante des repères globaux actuels.

**Renvoie:**

La configuration actuelle (qui est l'une de "c1" à "c8").

**Type renvoyé:**

str

**Exemples:**

Lecture de la configuration de la pose actuelle du robot de deux manières différentes.

```
configuration = get_current_configuration()
# ... this is similar to ...
pose = get_current_pose()
configuration = f"c{pose.configuration + 1}"
```

**get_reference_pose(*tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

Obtention de la pose TCP (point central de l'outil) de référence du robot. Si aucun repère n'est fourni, la pose est renvoyée par rapport aux repères globaux. Notez que cette pose est calculée à partir des angles de référence envoyés aux actionneurs.

**Paramètres:**

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

La pose TCP de référence du robot.

**Type renvoyé:**

Pose

**Exemples:**

Lecture des propriétés d'une pose de référence de deux manières différentes.

```
pose = get_reference_pose()
coordinates = pose.coordinates
configuration = pose.configuration
# ... this is similar to ...
coordinates = get_reference_coordinates()
configuration = get_reference_configuration()
```

**get_reference_coordinates(*tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

Obtention des coordonnées de référence du TCP (point central de l'outil) du robot par rapport aux repères de coordonnées donnés. Si aucun repère n'est spécifié, les coordonnées sont renvoyées par rapport aux repères globaux. Notez que ces coordonnées sont calculées à partir des angles de référence envoyés aux actionneurs.

**Paramètres:**

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]

- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

Les coordonnées TCP de référence du robot.

**Type renvoyé:**

Coordinates

**Exemples:**

Lecture des coordonnées TCP de référence dans les repères de coordonnées globaux.

```
coordinates = get_reference_coordinates()
```

### get_reference_configuration()

Obtention du nom de la configuration de la pose de référence du robot. Notez que cette configuration est indépendante des repères globaux de référence et qu'elle est calculée à partir des angles de référence envoyés aux actionneurs.

**Renvoie:**

La configuration de référence (qui est l'une de "c1" à "c8").

**Type renvoyé:**

str

**Exemples:**

Lecture de la configuration de la pose de référence du robot de deux manières différentes.

```
configuration = get_reference_configuration()
# ... this is similar to ...
pose = get_reference_pose()
configuration = f"c{pose.configuration + 1}"
```

### create_pose(*coordinates*, *configuration*)

Crée un objet Pose à partir des coordonnées et de la configuration fournies.

**Paramètres:**

- **coordinates** (*Coordinates, list, or dict*) –
  Les coordonnées TCP de la pose. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}
- **configuration** (*str*) – La configuration de la pose. Elle correspond à l'une de "c1" à "c8".

**Renvoie:**

L'objet de pose du robot.

**Type renvoyé:**

Pose

**Exemples:**

Création d'une pose à partir d'une liste de valeurs de coordonnées, dans une configuration spécifique.

```
pose = create_pose([350.0, 0.0, 550.0, 0.0, -30.0, 0.0], "c1")
```

### get_pose(*pose*)

Lecture de l'objet de pose à partir d'une pose prédéfinie enregistrée dans la base de données.

**Paramètres:**

**pose** (*Pose, str, or int*) –

La pose du robot à partir de laquelle lire les coordonnées. Elle peut être l'une des suivantes:

- un objet Pose
- le nom (str) d'une pose enregistrée dans la base de données.
- l'ID (int) d'une pose enregistrée dans la base de données

**Renvoie:**

L'objet pose.

**Type renvoyé:**

Pose

**Exemples:**

Chargement d'une pose depuis la base de données avec le nom « grasping_pose ».

```
pose = get_pose("grasping_pose")
```

### create_path(*initial_pose*, *segments*)

Crée un objet Path à partir de la pose initiale et des segments fournis.

**Paramètres:**

- **initial_pose** (*Pose*) – La pose initiale du robot.
- **segments** (*list*) – Une liste de segments dont se compose la trajectoire. Notez que des segments peuvent également être ajoutés ultérieurement.

**Renvoie:**

L'objet de trajectoire du robot.

**Type renvoyé:**

Path

**Exemples:**

Création d'une trajectoire à partir d'une pose initiale et de plusieurs segments.

```
path = create_path(start_pose, [LinearSegment(...), ...])
```

### get_path(*path*)

Lecture de l'objet trajectoire à partir d'une trajectoire prédéfinie enregistrée dans la base de données.

**Paramètres:**

**path** (*Path, str, or int*) –

La trajectoire du robot à récupérer. Peut être l'un des éléments suivants:

- un objet Path
- le nom (str) d'une trajectoire enregistrée dans la base de données
- l'ID (int) d'une trajectoire enregistrée dans la base de données

**Renvoie:**

L'objet trajectoire.

**Type renvoyé:**

Path

**Exemples:**

Chargement d'une trajectoire depuis la base de données avec le nom « grasping_path ».

```
pose = get_path("grasping_path")
```

**parse_linear_segment(*coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Créer un objet LinearSegment.

**Paramètres:**

- **coordinates** (*list, dict, or Coordinates*) – Coordonnées cibles.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet LinearSegment en définissant tous les paramètres:

```
linear_seg = parse_linear_segment([100, 200, 300, 15, -15, 30],
                                  linear_velocity=500,
linear_acceleration=1000,
                                  blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
linear_seg = parse_linear_segment([100, 200, 300, 15, -15, 30])
```

**parse_circular_segment(*intermediate_position*, *coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Créer un objet CircularSegment.

**Paramètres:**

- **intermediate_position** (*list*) – Une position intermédiaire [x, y, z] sur le chemin vers les coordonnées cibles. La position intermédiaire détermine la courbure du mouvement circulaire.
- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet CircularSegment en définissant tous les paramètres:

```
circular_seg = parse_circular_segment([0, 300, 300], [100, 200, 300, 15, -15,
30],
                                      linear_velocity=500,
linear_acceleration=1000,
                                      blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
circular_seg = parse_circular_segment([0, 300, 300], [100, 200, 300, 15, -15,
30])
```

**parse_orientation_segment(*orientation*, *angular_velocity=None*, *angular_acceleration=None*, *blend_radius=None*)**

Créer un objet OrientationSegment.

**Paramètres:**

- **orientation** (*list*) – Orientation cible. Notez que la position reste constante.
- **angular_velocity** (*float or None*) – Vitesse angulaire souhaitée [°/s]. Si aucune vitesse angulaire n'est définie, la valeur globale sera utilisée.
- **angular_acceleration** (*float or None*) – Accélération angulaire souhaitée [°/s$^2$]. Si aucune accélération angulaire n'est définie, la valeur globale sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Un rayon supérieur à 0 permet au point central de l'outil (TCP) de tourner avant que la position de ce segment ne soit atteinte. Si aucun rayon n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet OrientationSegment en définissant tous les paramètres:

```
orientation_seg = parse_orientation_segment([15, -15, 30],
                                            angular_velocity=250,
angular_acceleration=500,
                                            blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
orientation_seg = parse_orientation_segment([15, -15, 30])
```

**parse_coordinates_segment(*coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Créer un objet CoordinatesSegment.

**Paramètres:**

- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet CoordinatesSegment en définissant tous les paramètres:

```
coordinates_seg = parse_coordinates_segment([100, 200, 300, 15, -15, 30],
                                            linear_velocity=250,
linear_acceleration=500,
                                            blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
coordinates_seg = parse_coordinates_segment([100, 200, 300, 15, -15, 30])
```

**parse_pose_segment(*pose*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Créer un objet PoseSegment.

**Paramètres:**

- **pose** (*Pose*) – Pose cible.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet PoseSegment en définissant tous les paramètres:

```
pose_seg = parse_pose_segment(create_pose([100, 200, 300, 15, -15, 30], "c1"),
                              linear_velocity=250, linear_acceleration=500,
                              blend_radius=100)
```

ou en utilisant les valeurs par défaut et la pose depuis la base de données:

```
pose_seg = parse_pose_segment("target_pose")
```

### get_coordinates(*pose*)

Lecture des coordonnées de la pose spécifiée. Notez que ces coordonnées sont indépendantes des cadres globaux actuels.

**Paramètres:**

**pose** (*Pose, str, or int*) –

La pose du robot à partir de laquelle lire les coordonnées. Elle peut être l'une des suivantes:

- un objet Pose
- le nom (str) d'une pose enregistrée dans la base de données.
- l'ID (int) d'une pose enregistrée dans la base de données

**Renvoie:**

Les coordonnées de cette pose.

**Type renvoyé:**

Coordinates

**Exemples:**

Lecture des coordonnées de la pose précédemment enregistrée dans la base de données sous le nom « grasping_pose ».

```
coordinates = get_coordinates("grasping_pose")
```

Lecture des coordonnées de la pose précédemment enregistrée dans la base de données avec l'ID 3.

```
coordinates = get_coordinates(3)
```

### get_configuration(*pose*)

Lecture de la configuration d'une pose spécifique.

**Paramètres:**

**pose** (*Pose, str, or int*) –

La pose dont on souhaite lire la configuration. Spécifiez la valeur comme suit:

- un objet Pose
- le nom d'une pose prédéfinie (enregistrée dans la base de données)
- l'ID d'une pose prédéfinie (enregistrée dans la base de données)

**Renvoie:**

La configuration de cette pose (qui est l'une de « c1 » à « c8 »).

**Type renvoyé:**

str

**Exemples:**

Lecture de la configuration d'une pose enregistrée dans la base de données sous le nom « grasping_pose ».

```
configuration = get_configuration("grasping_pose")
```

Lecture de la configuration depuis un objet Pose de deux manières différentes.

```
configuration = get_configuration(pose)
# ... this is similar to ...
configuration = f"c{pose.configuration + 1}"
```

### parse_coordinates(*coordinates*)

Créer un objet Coordinates.

**Paramètres:**

**coordinates** (*list, dict, or Coordinates*) –

Un ensemble spécifique de coordonnées, qui peut être l'un des suivants:

- une liste de la forme [x, y, z, roll, pitch, yaw] avec des valeurs de position en [mm] et des valeurs d'orientation en [deg]
- un dictionnaire de la forme {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} avec des valeurs de position en [mm] et des valeurs d'orientation en [deg]
- un objet Coordinates (copiant un autre objet Coordinates)

**Constructeur:**

Créer un objet Coordinates à partir de valeurs de coordonnées.

```
coordinates = parse_coordinates({"x": 100, "y": 200, "z": 300,
                                 "roll": 15, "pitch": -15, "yaw": 30})
coordinates = parse_coordinates([100, 200, 300, 15, -15, 30])
```

**transform_coordinates(*coordinates*, *relative_tool_frame=None*, *relative_work_frame=None*)**

Transformation des coordonnées du robot entre différents repères en utilisant la différence entre les repères relatifs de l'outil et du travail. Si ni le repère d'outil relatif ni le repère de travail relatif ne sont spécifiés, les coordonnées ne sont pas transformées et l'objet de coordonnées inchangé est renvoyé.

**Paramètres:**

- **coordinates** (*Coordinates, list, or dict*) –
  Les coordonnées TCP à transformer. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}
- **relative_tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **relative_work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

Les coordonnées du robot après transformation.

**Type renvoyé:**

Coordinates

**Exemples:**

Transformation d'un ensemble de coordonnées vers un autre repère de travail. Si l'origine du repère de travail correspond exactement aux coordonnées à transformer, la transformation retourne un objet de coordonnées avec des zéros pour toutes les valeurs.

```
coordinates = [150, -250, 600, 15, -60, -15]
transformed_coordinates = transform_coordinates(coordinates,
relative_work_frame=coordinates)
```

### get_grid_points(*grid_name*)

Lecture des positions de tous les points d'une grille spécifique préalablement enregistrée dans la base de données.

**Paramètres:**

**grid_name** (*str*) – Le nom de la grille dont les points doivent être extraits.

**Renvoie:**

La liste des points de la grille, dans l'ordre d'itération défini dans les propriétés de la grille.

**Type renvoyé:**

list

**Exemples:**

Déplacement vers tous les points d'une grille prédéfinie.

```
# defining the orientation with which to approach the grid points
orientation = [0.0, -85.0, 0.0]
# iterate through all grid points
for position in get_grid_points("sample_tray"):
    # move to each of the grid points
    coordinates = position + orientation
    move_coordinates(coordinates)
```

**forward_kinematics(*joint_angles*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

La *cinématique directe* est un concept de la cinématique robotique qui calcule la pose du TCP (point central de l'outil) à partir d'un ensemble donné d'angles articulaires. La fonction inverse est la *cinématique inverse*, qui calcule les angles articulaires à partir de la pose TCP du robot. Les calculs cinématiques sont généralement effectués entre les repères de base et de bride intégrés du robot. L'utilisateur peut toutefois modifier les repères de référence en définissant les repères de travail et d'outil. Par défaut, les repères de travail et d'outil définis globalement sont utilisés.

**Paramètres:**

- **joint_angles** (*list of float*) – La liste des six angles articulaires en [deg].
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.
- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates

- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

L'objet de pose du robot.

**Type renvoyé:**

Pose

**Exemples:**

Dans une singularité, plusieurs ensembles d'angles articulaires conduisent à la même pose TCP. Dans cet exemple, les deux ensembles d'angles aboutissent exactement à la même pose.

```
pose_1 = forward_kinematics([0, 0, 0, 0, 0, 0])
pose_2 = forward_kinematics([45, 0, 0, 45, 0, -90])
```

Lecture de la pose TCP actuelle à l'aide de deux approches différentes.

```
pose = forward_kinematics(read_actuator_position())
# ... this is similar to ...
pose = get_pose()
```

**inverse_kinematics(*pose*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

La *cinématique inverse* est un concept de la cinématique robotique qui calcule un ensemble d'angles articulaires à partir de la pose du TCP (point central de l'outil). La fonction inverse est la *cinématique directe*, qui calcule la pose TCP à partir des angles articulaires du robot. Notez que la *cinématique inverse* possède généralement plusieurs solutions, mais n'en retourne qu'une seule.

**Paramètres:**

- **pose** (*Pose, str, or int*) –
  La pose pour laquelle calculer les angles articulaires correspondants. Spécifiez la valeur comme suit:
  - un objet Pose
  - le nom d'une pose prédéfinie (enregistrée dans la base de données)
  - l'ID d'une pose prédéfinie (enregistrée dans la base de données)
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

La liste des six angles articulaires en [deg].

**Type renvoyé:**

list of float

**Exemples:**

Calcul des angles articulaires d'une pose précédemment enregistrée dans la base de données.

```
inverse_kinematics("grasping_pose")
```

Calcul des angles articulaires permettant d'atteindre un ensemble spécifique de coordonnées dans une configuration donnée.

```
inverse_kinematics(create_pose([350.0, 0.0, 550.0, 0.0, -30.0, 0.0], "c1"))
```

**inverse_kinematics_nearest(*coordinates*, *previous_angles=None*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

Cette fonction calcule la solution de la *cinématique inverse* (voir la fonction `inverse_kinematics` pour plus de détails) qui est la plus proche soit de la pose actuelle du robot, soit d'une pose spécifique.

**Paramètres:**

- **coordinates** (*Coordinates, list, or dict*) –
  Les coordonnées TCP pour lesquelles calculer les angles articulaires correspondants. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

- **previous_angles** (*list or None*) –
  Les angles pour lesquels la solution la plus proche sera calculée. Spécifiez la valeur comme suit:
  - None, pour utiliser les angles actuels du robot
  - Une liste d'angles articulaires [deg]

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates

- une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
- un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

La liste des six angles articulaires en [deg].

**Type renvoyé:**

list of float

**Exemples:**

La cinématique inverse peut être utilisée pour vérifier simplement si un ensemble spécifique de coordonnées renvoie une solution cinématique.

```
inverse_kinematics_nearest([600, 0, 300, 180, -90, 0])
```

Pour obtenir plusieurs solutions pour la même pose, l'instruction de pose précédente peut être modifiée en conséquence. Dans cet exemple, deux solutions pour un robot debout, légèrement déplacé vers l'avant et vers le bas, sont calculées.

```
solution_1 = inverse_kinematics_nearest({"x": 485, "y": 188, "z": 1059,
"roll": 44, "pitch": -45, "yaw": -81},
                                        previous_angles=[-160, -20, -20, 20,
-20, -160])
solution_2 = inverse_kinematics_nearest([600, 0, 300, 180, -90, 0],
                                        previous_angles=[20, 15, 30, 30, 15,
10])
```

**inverse_kinematics_complete(*coordinates*, *tool_offset=None*, *tool_frame=None*, *work_offset=None*, *work_frame=None*)**

Cette fonction calcule la solution de la *cinématique inverse* (voir la fonction `inverse_kinematics` pour plus de détails) qui est la plus proche soit de la pose actuelle du robot, soit d'une pose spécifique.

**Paramètres:**

- **coordinates** (*Coordinates, list, or dict*) –
  Les coordonnées TCP pour lesquelles calculer les angles articulaires correspondants. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer le TCP du robot, donnée par rapport au cadre de l'outil. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage de l'outil, de sorte que le TCP se trouve à l'origine du cadre de l'outil.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier le TCP du robot, donnée par rapport au cadre de bride. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre d'outil global est appliqué. Si le cadre d'outil contient uniquement des coordonnées nulles, il coïncide avec le cadre de bride.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformation pour déplacer la référence du robot, donnée par rapport au cadre de travail. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, il n'y a aucun décalage du travail, de sorte que la référence se trouve à l'origine du cadre de travail.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformation pour modifier la référence du robot, donnée par rapport au cadre de base. Spécifiez la valeur comme suit:
  - un objet Coordinates
  - une liste de coordonnées cartésiennes au format [x, y, z, roll, pitch, yaw]
  - un dictionnaire de coordonnées cartésiennes au format {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}

  Par défaut, le cadre de travail global est appliqué. Si le cadre de travail contient uniquement des coordonnées nulles, il coïncide avec le cadre de base.

**Renvoie:**

Les listes de six angles articulaires en [deg] et leurs configurations au format {"c1": [...], "c2": [...], ...}

**Type renvoyé:**

dict

**Exemples:**

Dans cet exemple, le robot se déplace vers tous les ensembles d'angles articulaires qui atteignent les mêmes coordonnées TCP.

```
solutions = inverse_kinematics_complete([600, 0, 300, 180, -90, 0])
for joint_angles in solutions.values():
    move_joints([1, 2, 3, 4, 5, 6], joint_angles)
```

Si nous connaissons un ensemble d'angles articulaires atteignant une pose souhaitée, nous pouvons ainsi calculer toutes les autres solutions qui atteignent la même pose mais avec des configurations articulaires différentes.

```
pose = forward_kinematics([0, 90, -90, 0, 0, 0])
solutions = inverse_kinematics_complete(pose.coordinates)
```

### wait(*seconds*)

Attente et blocage actif de l'exécution du code pendant une durée définie.

**Paramètres:**

**seconds** (*float*) – Durée d'attente en [s].

**Exemples:**

Cet exemple déplace le robot vers des poses prédéfinies et attend 3 [s] entre les mouvements.

```
move_to_pose("pose_1")
wait(3)
move_to_pose("pose_2")
```

Cet exemple répète indéfiniment une partie de votre code. Pour limiter la vitesse d'exécution, le code attend 100 [ms] à la fin de chaque itération.

```
while True:
    # enter your code here
    wait(0.1)
```

Cet exemple est similaire au précédent, mais au lieu d'attendre un temps constant, le temps d'attente est choisi dynamiquement afin d'exécuter votre code à un intervalle constant chaque seconde.

```
import time
cycle_time = 1.0
time_start = time.perf_counter()
while True:
    # enter your code here
    time_now = time.perf_counter()
    elapsed_duration = time_now - time_start
    wait(cycle_time - elapsed_duration)
```

**run_script(*script_name*, *arguments=None*)**

Exécution d'une application (bloquante) depuis l'application principale.

**Paramètres:**

- **script_name** (*str*) – Nom de l'application à exécuter. Si l'application se trouve dans un dossier, utilisez le chemin complet de l'application (ex. *folder/subfolder/app*).
- **arguments** (*dict*) – Arguments transmis à l'application à exécuter. Vous pouvez les lire dans l'application appelée à l'aide de la variable intégrée *script_arguments*.

**Exemples:**

Supposons que nous ayons une application séparée qui mesure la température ambiante. Cet exemple appelle cette application de manière répétée, une fois par minute.

```
cycle_time = 60.0
time_measurement = time.perf_counter()
run_script("measurements/measure_temperature")
while True:
    if time.perf_counter() - time_measurement > cycle_time:
        run_script("measurements/measure_temperature")
        time_measurement += cycle_time
    else:
        wait(0.1)
```

### start_parallel_script(*script_name*)

Exécution d'une application (non bloquante) en parallèle avec l'application principale. Notez que l'application principale continue immédiatement et que toutes les applications parallèles s'arrêtent lorsque l'application principale se termine.

**Paramètres:**

**script_name** (*str*) – Nom de l'application parallèle à démarrer. Si l'application est dans un dossier, utilisez le chemin complet de l'application (ex. *folder/subfolder/app*).

**Exemples:**

Cet exemple déclenche une application parallèle pour surveiller un capteur de température afin de s'assurer que le robot fonctionne uniquement dans une plage de température spécifique. L'application principale s'exécute en continu et s'arrête si le critère de température opérationnelle n'est plus rempli.

```
start_parallel_script("measurements/measure_temperature_continuously")
set_global_variable("temperature_value") = 20
while -10 < get_global_variable("temperature_value") < 40:
    move_to_pose("pose_1")
    move_to_pose("pose_2")
    move_to_pose("pose_3")
    move_to_pose("pose_4")
print("Aborting the program, because the measured temperature is outside of
the operational temperature range.)
```

### stop_application()

Arrêt des applications actuellement en cours d'exécution. Cela inclut l'application principale et toutes les applications déclenchées par celle-ci. Notez qu'une application en *background* continue de fonctionner.

**Lève:**

**ScriptInterrupt** – Provoqué dans l'application principale si cette fonction est exécutée depuis une application lancée avec la fonction *run_script*.

**Exemples:**

Cet exemple de code peut être utilisé dans une application *background* pour mettre temporairement en pause et reprendre manuellement les applications principales en cours d'exécution.

```
pause_application()
answer = dialog_choice("Do you want to continue?", ["Continue", "Abort"],
title="Application Interrupt", dialog_value="Abort")
if answer == "Continue":
    resume_application()
else:
    stop_application()
```

### pause_application()

Met en pause les applications en cours d'exécution. Cela inclut l'application principale et toutes les applications déclenchées par celle-ci. Notez qu'une application *background* en cours continue de fonctionner.

**Exemples:**

Consultez l'exemple combiné de *stop_application*, qui utilise cette fonction.

**resume_application()**

Reprise des applications actuellement en pause. Comme cette fonction ne peut être exécutée qu'en état *paused*, elle ne peut être utilisée que depuis une application *background*.

**Exemples:**

Consultez l'exemple combiné de *stop_application*, qui utilise cette fonction.

**stop_and_release()**

Arrête les applications en cours d'exécution et libère toutes les articulations (activation du mode de compensation de gravité). Notez que l'utilisateur doit maintenir manuellement toutes les articulations (désactivation du mode de compensation de gravité) pour pouvoir continuer à exécuter d'autres applications par la suite.

**is_status(*status*)**

Vérifie si le statut du programme correspond au statut donné.

**Paramètres:**

**status** (*str or list of str*) –

Le statut du programme à vérifier peut être un ou plusieurs des suivants:

- ”ready” (le robot est inactif et attend des commandes)
- ”processing” (le robot traite des données, ce qui peut prendre un certain temps)
- ”hand_guidance_control” (le robot est en mode de contrôle manuel)
- ”running” (le robot exécute une application ou un mouvement)
- ”paused” (le robot met en pause une application ou un mouvement)
- ”emergency_stop” (un arrêt d’urgence a été déclenché)
- ”normal_stop” (un arrêt normal a été déclenché)
- ”protective_stop” (un arrêt de protection a été déclenché)
- ”error” (une erreur ou une violation de sécurité s’est produite)
- ”position_limit” (une violation de position limitée sécurisée s’est produite)
- ”safeguard” (une interruption de sécurité a été déclenchée)
- ”collision” (une interruption de collision a été déclenchée)
- ”proceed” (le robot continue après une interruption)
- ”stopping” (le robot arrête son programme)

**Renvoie:**

Indique si le statut du programme correspond au statut donné (ou à l’un des statuts donnés)

**Type renvoyé:**

bool

**dialog_choice(*text*, *options*, *title=''*, *dialog_type=0*, *dialog_value=None*, *mandatory=False*)**

Création d'une boîte de dialogue avec des boutons. Cette boîte bloque l'exécution du programme jusqu'à ce que l'utilisateur réponde avec l'une des options présentées.

**Paramètres:**

- **text** (*str*) – Le message (ou la question) présenté à l'utilisateur.
- **options** (*list of str*) – Les options pouvant être choisies, correspondant aux étiquettes de chaque bouton.
- **title** (*str*) – Le titre de la boîte de dialogue.
- **dialog_type** (*int*) –
  Le type de dialogue à présenter à l'utilisateur. Cela peut être l'un des suivants:
  - 0: information / couleur bleue
  - 1: succès / couleur verte
  - 2: avertissement / couleur orange
  - 3: erreur / couleur rouge
- **dialog_value** (*str*) – La valeur par défaut à retourner si le dialogue n'est pas répondu.
- **mandatory** (*bool*) – Indique si le dialogue doit être répondu pour que le flux du programme continue. Si défini sur *True*, le dialogue ne peut pas être ignoré, par exemple en fermant la fenêtre pop-up, et il bloque jusqu'à ce qu'une réponse soit reçue.

**Renvoie:**

L'option sélectionnée, qui est l'étiquette du bouton appuyé par l'utilisateur.

**Type renvoyé:**

str

**Exemples:**

Cet exemple montre comment créer une fenêtre pop-up informative simple contenant un bouton « Continuer ». En appuyant sur le bouton, le programme continuera à exécuter son code.

```
dialog_choice("This is just an informative message.", ["Continue"],
title="Information", dialog_type=0)
```

**dialog_dropdown(*text*, *options*, *title=''*, *dialog_type=0*, *dialog_value=None*, *mandatory=False*)**

Créer un dialogue avec des options présentées dans une liste déroulante. Le dialogue bloque le flux du programme jusqu'à ce que l'utilisateur sélectionne et confirme l'une des options.

**Paramètres:**

- **text** (*str*) – Le message (ou la question) présenté à l'utilisateur.
- **options** (*list of str*) – Les options pouvant être choisies, qui sont les étiquettes des options déroulantes.
- **title** (*str*) – Le titre de la boîte de dialogue.
- **dialog_type** (*int*) –
  Le type de dialogue à présenter à l'utilisateur. Cela peut être l'un des suivants:
  - 0: information / couleur bleue
  - 1: succès / couleur verte
  - 2: avertissement / couleur orange
  - 3: erreur / couleur rouge
- **dialog_value** (*str*) – La valeur par défaut à retourner si le dialogue n'est pas répondu.
- **mandatory** (*bool*) – Indique si le dialogue doit être répondu pour que le flux du programme continue. Si défini sur *True*, le dialogue ne peut pas être ignoré, par exemple en fermant la fenêtre pop-up, et il bloque jusqu'à ce qu'une réponse soit reçue.

**Renvoie:**

L'option sélectionnée, qui est l'étiquette de l'option déroulante sélectionnée.

**Type renvoyé:**

str

**Exemples:**

Cet exemple montre comment créer une fenêtre pop-up d'avertissement et permet à l'utilisateur de choisir parmi plusieurs options pour influencer le flux du programme.

```
answer = dialog_dropdown("High temperature detected. You may want to reduce
the robot's speed.",
                         options=["Reduce by 20%", "Reduce by 10%", "Do not
reduce"],
                         title="High Temperature Warning", dialog_type=2,
dialog_value="Do not reduce")
if answer == "Reduce by 20%":
    velocity *= 0.8
elif answer == "Reduce by 10%":
    velocity *= 0.9
```

**dialog_text(*text*, *title*='', *dialog_type=0*, *dialog_value=None*, *mandatory=False*)**

Créer un dialogue avec un champ de texte. Le dialogue bloque le flux du programme jusqu'à ce que l'utilisateur saisisse un texte et confirme le dialogue.

**Paramètres:**

- **text** (*str*) – Le message (ou la question) présenté à l'utilisateur.
- **title** (*str*) – Le titre de la boîte de dialogue.
- **dialog_type** (*int*) –
  Le type de dialogue à présenter à l'utilisateur. Cela peut être l'un des suivants:
  - 0: information / couleur bleue
  - 1: succès / couleur verte
  - 2: avertissement / couleur orange
  - 3: erreur / couleur rouge
- **dialog_value** (*str*) – La valeur par défaut à retourner si le dialogue n'est pas répondu.
- **mandatory** (*bool*) – Indique si le dialogue doit être répondu pour que le flux du programme continue. Si défini sur *True*, le dialogue ne peut pas être ignoré, par exemple en fermant la fenêtre pop-up, et il bloque jusqu'à ce qu'une réponse soit reçue.

**Renvoie:**

Le texte saisi dans le champ de texte.

**Type renvoyé:**

str

**Exemples:**

Cet exemple montre comment créer une fenêtre pop-up simple avec un champ de texte pour saisir votre nom. Après que l'utilisateur ait saisi son nom, il peut être utilisé dans le programme, par exemple pour le journal de bord.

```
name = dialog_text("Please enter your name:", title="Welcome!")
log_message("Registered user: {name}")
```

**dialog_yes_no(*text*, *title*='', *dialog_type*=*0*, *dialog_value*=*None*, *mandatory*=*False*)**

Créer un dialogue avec des boutons *Oui* et *Non*. Le dialogue bloque le flux du programme jusqu'à ce que l'utilisateur appuie sur l'un de ces boutons.

**Paramètres:**

- **text** (*str*) – Le message (ou la question) présenté à l'utilisateur.
- **title** (*str*) – Le titre de la boîte de dialogue.
- **dialog_type** (*int*) –
  Le type de dialogue à présenter à l'utilisateur. Cela peut être l'un des suivants:
  - 0: information / couleur bleue
  - 1: succès / couleur verte
  - 2: avertissement / couleur orange
  - 3: erreur / couleur rouge
- **dialog_value** (*str*) – La valeur par défaut à retourner si le dialogue n'est pas répondu.
- **mandatory** (*bool*) – Indique si le dialogue doit être répondu pour que le flux du programme continue. Si défini sur *True*, le dialogue ne peut pas être ignoré, par exemple en fermant la fenêtre pop-up, et il bloque jusqu'à ce qu'une réponse soit reçue.

**Renvoie:**

Vrai si l'utilisateur sélectionne « oui », Faux si l'utilisateur sélectionne « non ».

**Type renvoyé:**

bool

**Exemples:**

Cet exemple utilise un simple dialogue de confirmation pour permettre à l'utilisateur de choisir s'il souhaite continuer ou arrêter l'application.

```
if not dialog_yes_no("An error occurred. Do you want to continue anyway?",
title="Error", dialog_type=3):
    stop_application()
```

### print(**args*)

Imprimer un message dans la sortie de l'application. Notez que cette fonction est similaire à la fonction intégrée *print* de Python. Elle prend en charge plusieurs arguments et différents types d'arguments. Les arguments sont automatiquement convertis en chaînes et concaténés avec un séparateur d'espaces.

**Paramètres:**

**args** (*tuple of various*) – Les arguments à imprimer.

**Exemples:**

Imprimer la valeur d'une variable dans un style bien formaté. Notez que le premier appel utilise la conversion et la concaténation automatiques, tandis que le second appel utilise des f-strings avec le même résultat.

```
print("Variable value:", var)
print(f"Variable value: {var}")
```

**show_notification(*message_text*, *message_title*='*Notification*')**

Afficher un message de notification personnalisé dans l'interface. Cela n'a aucun effet sur le déroulement du programme. L'utilisateur peut spécifier dans le menu des paramètres si les NOTIFICATIONS s'estompent ou restent affichées.

**Paramètres:**

- **message_text** (*str*) – Le texte de notification à afficher dans l'interface.
- **message_title** (*str*) – Le titre de la notification contextuelle.

### raise_error(*message*)

Déclencher un message d'erreur personnalisé dans l'application en cours d'exécution. L'erreur est de type `ApplicationError` et peut être interceptée dans l'application. Si elle n'est pas interceptée, l'application en cours est arrêtée. L'erreur est également enregistrée dans le fichier `error.log`, qui fait partie du système de journalisation. Les journaux peuvent être téléchargés via l'interface en appuyant sur `CTRL + SHIFT + L`.

**Paramètres:**

**message** (*str*) – Le message d'erreur à déclencher dans l'application.

**Lève:**

**ApplicationError** – Contenant le message personnalisé.

**Exemples:**

Une application en arrière-plan surveille les entrées numériques et vérifie si une entrée d'erreur spécifique est déclenchée. Si une entrée d'erreur est déclenchée, cette erreur est levée dans l'application.

```
# keeping track of the previous error flag value
error_status = False
# do forever
while True:
    # read the value of the error flag (True or False)
    error_flag = read_digital_main_input("error_flag")
    # if the error flag switches from False to True, raise an application
error
    if error_flag:
        if not error_status:
            raise_error("The error input was raised!")
        error_status = True
    else:
        error_status = False
    wait(0.1)
```

Cette erreur d'application peut ensuite être interceptée et traitée par l'application principale qui contrôle le robot.

```
try:
    my_main_program()
except ApplicationError as error:
    ...
```

### raise_warning(*message*)

Afficher une boîte de message d'avertissement personnalisée dans l'interface. Cela n'a aucun effet sur le déroulement du programme. Le comportement des avertissements (disparition ou affichage persistant) peut être défini par l'utilisateur dans le menu des paramètres.

**Paramètres:**

**message** (*str*) – Le message d'avertissement à afficher dans l'interface.

**Exemples:**

Supposons qu'un capteur de température soit connecté à une entrée analogique. Cet exemple affiche un avertissement dans l'interface si la valeur dépasse un certain seuil, et le réinitialise si la valeur redescend en dessous de ce seuil.

```
# flag to store whether a warning was raised
high_temperature = False
# do forever
while True:
    # check temperature value
    temperature_value = read_analog_main_input("temperature_sensor")
    # if value is too high, raise a warning
    if temperature_value > 5.2:
        if not high_temperature:
            raise_warning("High temperature detected! Slow down the robot to
prevent overheating.")
        high_temperature = True
    # if value recovers, reset the flag, such that a warning can be raised
again later on
    elif temperature_value < 5.0:
        high_temperature = False
    wait(5)
```

### log_message(*message*, *logger='custom'*, *include_timestamp=True*)

Enregistrer un message dans un fichier journal personnalisé.

**Paramètres:**

- **message** (*str*) – Le message personnalisé à enregistrer.
- **logger** (*str*) – Le nom du fichier journal dans lequel enregistrer les messages. Le fichier journal se trouve à l'emplacement `custom/<logger>.log` dans le dossier des journaux. Le nom de fichier par défaut est `custom/custom.log`. Les journaux peuvent être téléchargés via l'interface en appuyant sur `CTRL + SHIFT + L`.
- **include_timestamp** (*bool*) – Indique si un horodatage est ajouté au message enregistré.

**Exemples:**

Enregistrer la valeur d'une mesure de capteur dans le fichier *custom/measurements.log*.

```
value = read_analog_main_input("my_sensor")
log_message(f"Sensor value: {value}", logger="measurements")
```

**clear_log()**

Effacer et supprimer tous les fichiers journaux personnalisés.

### set_shared_variable(*name*, *value*)

Définir la valeur d'une variable partagée.

Notez que les *variables partagées* n'existent que tant que l'application principale est en cours d'exécution. Après l'arrêt de l'application principale, toutes les *variables partagées* sont automatiquement supprimées. Si vous avez besoin que des variables conservent leur valeur pendant l'exécution de l'application ou même après le redémarrage du robot, utilisez plutôt des *variables globales*.

**Paramètres:**

- **name** (*str*) – Le nom de la variable à définir.
- **value** (*various*) – La nouvelle valeur de cette variable.

**Exemples:**

Cet exemple montre comment une application parallèle peut être arrêtée par l'application principale à l'aide de variables partagées. L'application parallèle surveille simplement un indicateur de terminaison et s'arrête dès que cet indicateur est défini à *True*. De plus, la variable partagée est supprimée une fois qu'elle n'a plus d'effet. Étant donné que les variables partagées sont effacées lorsque l'application principale se termine, cela n'est pas nécessaire mais constitue une bonne pratique.

```
while not get_shared_variable("terminate_parallel_application"):
    ...
    wait(0.1)
remove_shared_variable("terminate_parallel_application")
```

L'application principale s'assure que l'indicateur de terminaison existe, lance l'application parallèle, puis définit finalement cet indicateur sur *True* afin de finaliser l'application parallèle.

```
set_shared_variable("terminate_parallel_application", False)
start_parallel_script("my_parallel_application")
...
set_shared_variable("terminate_parallel_application", True)
...
```

**get_shared_variable(*name=None*)**

Lire la valeur d'une variable partagée spécifique. Par défaut, cette fonction lit les valeurs de toutes les variables partagées.

Notez que les *variables partagées* n'existent que tant que l'application principale est en cours d'exécution. Après l'arrêt de l'application principale, toutes les *variables partagées* sont automatiquement supprimées. Si vous avez besoin que des variables conservent leur valeur pendant l'exécution de l'application ou même après le redémarrage du robot, utilisez plutôt des *variables globales*.

**Paramètres:**

**name** (*str (or None)*) – Nom de la variable à retourner (ou None si toutes les variables doivent être retournées).

**Renvoie:**

Valeur de la variable à retourner (ou dictionnaire contenant toutes les variables).

**Type renvoyé:**

various (or dict)

**Exemples:**

Consultez les exemples de *set_shared_variable*, qui combinent l'utilisation de toutes les fonctions de *variables partagées*.

**remove_shared_variable(*name*)**

Supprimer une variable partagée. Après sa suppression, sa valeur n'est plus accessible.

Notez que les *variables partagées* n'existent que tant que l'application principale est en cours d'exécution. Après l'arrêt de l'application principale, toutes les *variables partagées* sont automatiquement supprimées. Si vous avez besoin que des variables conservent leur valeur pendant l'exécution de l'application ou même après le redémarrage du robot, utilisez plutôt des *variables globales*.

**Paramètres:**

**name** (*str*) – nom de la variable à supprimer

**Exemples:**

Consultez les exemples de *set_shared_variable*, qui combinent l'utilisation de toutes les fonctions de *variables partagées*.

### set_global_variable(*name*, *value*)

Définition de la valeur d'une variable globale.

Notez que les modifications apportées aux *variables globales* sont permanentes et persistent même après l'arrêt et le redémarrage du robot. Il est donc fortement recommandé de supprimer les *variables globales* lorsqu'elles ne servent plus à rien. Si vous avez besoin de variables uniquement pendant l'exécution d'une seule application, utilisez plutôt des *variables partagées*.

**Paramètres:**

- **name** (*str*) – Le nom de la variable à définir.
- **value** (*various*) – La nouvelle valeur de cette variable.

**Exemples:**

Cet exemple montre comment une application répétitive compte le nombre de ses itérations réussies. Cela est réalisé en initialisant un compteur global lors de la première exécution, puis en incrémentant ce compteur à la fin de chaque itération de la boucle.

```
if "counter" not in get_global_variable():
    set_global_variable("counter", 0)
while True:
    ...
    counter = get_global_variable("counter")
    set_global_variable("counter", counter + 1)
```

**get_global_variable(*name*=*None*)**

Lecture de la valeur d'une variable globale spécifique. Par défaut, cette fonction lit les valeurs de toutes les variables globales.

Notez que les modifications apportées aux *variables globales* sont permanentes et persistent même après l'arrêt et le redémarrage du robot. Il est donc fortement recommandé de supprimer les *variables globales* lorsqu'elles ne servent plus à rien. Si vous avez besoin de variables uniquement pendant l'exécution d'une seule application, utilisez plutôt des *variables partagées*.

**Paramètres:**

**name** (*str (or None)*) – Nom de la variable à retourner (ou None si toutes les variables doivent être retournées).

**Renvoie:**

Valeur de la variable à retourner (ou dictionnaire contenant toutes les variables).

**Type renvoyé:**

various (or dict)

**Exemples:**

Consultez les exemples de *set_global_variable*, qui combinent l'utilisation des fonctions liées aux *variables globales*.

### remove_global_variable(*name*)

Suppression d'une variable globale. Après sa suppression, sa valeur ne peut plus être consultée.

Notez que les modifications apportées aux *variables globales* sont permanentes et persistent même après l'arrêt et le redémarrage du robot. Il est donc fortement recommandé de supprimer les *variables globales* lorsqu'elles ne servent plus à rien. Si vous avez besoin de variables uniquement pendant l'exécution d'une seule application, utilisez plutôt des *variables partagées*.

**Paramètres:**

**name** (*str*) – nom de la variable à supprimer

**Exemples:**

Cet exemple supprime toutes les variables globales.

```
for name in get_global_variable().keys():
    remove_global_variable(name)
```

### set_state_variable(*name*, *value*)

Définition de la valeur d'une variable d'état.

Notez que les *variables d'état* sont définies individuellement pour chaque application et ne sont utilisées que dans les applications principales. Elles peuvent être configurées et personnalisées dans l'onglet *variables* de l'éditeur d'application. Chaque *variable d'état* possède un type spécifique et n'accepte que des valeurs de ce type. L'objectif principal des *variables d'état* est que leur valeur soit affichée en temps réel dans l'"*Interface du Panneau* afin d'observer et de contrôler l'application en cours d'exécution.

**Paramètres:**

- **name** (*str*) – Le nom de la variable à définir.
- **value** (*various*) – La nouvelle valeur de cette variable.

**Exemples:**

Supposons que, pour cette application, une variable d'état en lecture seule de type *percent* nommée *progress* et une variable d'état modifiable dynamiquement de type *boolean* nommée *logging* aient été définies. La variable *progress* a pour but de stocker l'avancement actuel du programme en pourcentage. Le drapeau *logging* indique si la mise à jour de l'avancement doit être enregistrée. Notez que l'utilisateur peut modifier la valeur de *logging* à tout moment, ce qui adapte dynamiquement le programme.

```
number_of_iterations = 128
for i in range(number_of_iterations):
    progress = i / number_of_iterations * 100
    set_state_variable("progress", progress)
    if get_state_variable("logging"):
        log_message(f"The current progress is {progress}%.")
    ...
set_state_variable("progress", 100)
    if get_state_variable("logging"):
        log_message(f"The current progress is 100%. The program successfully
completed.")
```

**get_state_variable(*name*=*None*)**

Lecture de la valeur d'une variable d'état spécifique. Par défaut, cette fonction lit les valeurs de toutes les variables d'état.

Notez que les *variables d'état* sont définies individuellement pour chaque application et ne sont utilisées que dans les applications principales. Elles peuvent être configurées et personnalisées dans l'onglet *variables* de l'éditeur d'application. Chaque *variable d'état* possède un type spécifique et n'accepte que des valeurs de ce type. L'objectif principal des *variables d'état* est que leur valeur soit affichée en temps réel dans l'"*Interface du Panneau* afin d'observer et de contrôler l'application en cours d'exécution.

**Paramètres:**

**name** (*str (or None)*) – Nom de la variable à retourner (ou None si toutes les variables doivent être retournées).

**Renvoie:**

Valeur de la variable à retourner (ou dictionnaire contenant toutes les variables).

**Type renvoyé:**

various (or dict)

**Exemples:**

Consultez les exemples de *set_state_variable*, qui combinent l'utilisation des fonctions liées aux *variables d'état*.

### read_actuator_position(*actuator_ids*=*None*)

Lecture de la position angulaire d'un, de plusieurs ou de tous les actionneurs.

**Paramètres:**

**actuator_ids** (*None, str, int, list, set*) –

Les identifiants des actionneurs à lire. Spécifiez la valeur comme suit:

- l'ID d'une seule articulation (ex. *6*)
- une liste d'IDs pour plusieurs articulations (ex. *[1, 4, 6]*)
- un ensemble de noms d'articulations pour plusieurs articulations (ex.: *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* ou *None* pour lire toutes les articulations

**Renvoie:**

La position angulaire des actionneurs demandés (en [deg]), qui peut être:

- une valeur de position d'une seule articulation → float
- une liste de valeurs de position d'articulations → liste de float
- une correspondance entre les ID d'articulations et leurs valeurs de position → dictionnaire de float

**Type renvoyé:**

float, or list of float, or dict of float

**Exemples:**

Affichage des positions actuelles des actionneurs des articulations du poignet.

```
print(read_actuator_position([1, 4, 6]))
```

### read_actuator_velocity(*actuator_ids=None*)

Lecture de la vitesse angulaire d'un, de plusieurs ou de tous les actionneurs.

**Paramètres:**

**actuator_ids** (*None, str, int, list, set*) –

Les identifiants des actionneurs à lire. Spécifiez la valeur comme suit:

- l'ID d'une seule articulation (ex. *6*)
- une liste d'IDs pour plusieurs articulations (ex. *[1, 4, 6]*)
- un ensemble de noms d'articulations pour plusieurs articulations (ex.: *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* ou *None* pour lire toutes les articulations

**Renvoie:**

La vitesse angulaire des actionneurs demandés (en °/s), qui peut être:

- une valeur de vitesse d'une seule articulation -> float
- une liste de valeurs de vitesse d'articulations -> liste de float
- un mapping entre les IDs des articulations et leurs valeurs de vitesse -> dictionnaire de float

**Type renvoyé:**

float, or list of float, or dict of float

**Exemples:**

Surveillance de la vitesse pendant un mouvement afin de déterminer l'actionneur le plus rapide.

```
move_to_pose("target", block=False)
max_velocities = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}
while is_motor_moving():
    velocities = read_actuator_velocities({1, 2, 3, 4, 5, 6})
    for joint in range(1, 7):
        max_velocities[joint] = max([max_velocities[joint],
abs(velocities[joint])])
fastest_joint = max(max_velocities, key=max_velocities.get)
print(f"The fastest joint is {fastest_joint} with a maximum velocity of
{max_velocities[fastest_joint]}°/s")
```

### get_linear_velocity()

Obtention de la vitesse linéaire du TCP (point central de l'outil) du robot.

**Renvoie:**

La vitesse linéaire de l'outil en [mm/s].

**Type renvoyé:**

list

**Exemples:**

Obtenir et afficher la vitesse linéaire actuelle.

```
print(get_linear_velocity())
```

### get_angular_velocity()

Obtention de la vitesse angulaire du TCP (point central de l'outil) du robot.

**Renvoie:**

La vitesse angulaire de l'outil en [deg/s].

**Type renvoyé:**

list

**Exemples:**

Obtenir et afficher la vitesse angulaire actuelle.

```
print(get_angular_velocity())
```

**read_actuator_current(*actuator_ids=None*)**

Lecture du courant appliqué sur un, plusieurs ou tous les actionneurs.

**Paramètres:**

**actuator_ids** (*None, str, int, list, set*) –

Les identifiants des actionneurs à lire. Spécifiez la valeur comme suit:

- l'ID d'une seule articulation (ex. *6*)
- une liste d'IDs pour plusieurs articulations (ex. *[1, 4, 6]*)
- un ensemble de noms d'articulations pour plusieurs articulations (ex.: *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* ou *None* pour lire toutes les articulations

**Renvoie:**

Les valeurs de courant des actionneurs demandés (en A), qui peuvent être:

- une valeur de courant d'une seule articulation -> float
- une liste de valeurs de courant d'articulations -> liste de float
- un mapping entre les IDs des articulations et leurs valeurs de courant -> dictionnaire de float

**Type renvoyé:**

float, or list of float, or dict of float

**Exemples:**

Affichage des courants des actionneurs du robot à une position donnée deux fois, une fois avant et une fois après la prise d'un objet.

```
move_to_pose("measure_pose")
print(f"Currents without payload: {read_actuator_current()}")
move_to_pose("pick_pose")
tool_pick()
move_to_pose("measure_pose")
print(f"Currents with payload: {read_actuator_current()}")
```

### tool_pick()

Activation de la fonction de prise de l'outil pour saisir un objet. Cela peut entraîner le serrage pour les outils mécaniques ou l'application d'une aspiration pour les outils pneumatiques.

**Exemples:**

Activer la fonction de prise de l'outil.

```
tool_pick()
```

### tool_place()

Activation de la fonction de dépose de l'outil pour relâcher un objet. Cela peut entraîner l'ouverture d'une pince pour les outils mécaniques ou l'arrêt de l'aspiration pour les outils pneumatiques.

**Exemples:**

Activer la fonction de dépose de l'outil.

```
tool_place()
```

**set_tool_payload(*payload*, *center_of_mass=None*, *inertia=None*)**

Définir dynamiquement la charge utile de l'outil pendant le fonctionnement du robot.

**Paramètres:**

- **payload** (*float*) – la charge utile de l'outil (poids + objets saisis) en [kg]
- **center_of_mass** (*list of float*) – centre de masse [x, y, z] par rapport au cadre de la bride en [m]
- **inertia** (*list of float*) – matrice d'inertie convertie en liste de longueur 9 [kg $m^2$]

**Exemples:**

Modifier la charge utile lors de la prise et du dépôt d'objets d'une position à une autre.

```
tool_mass = 1.3
object_mass = 0.7

while True:
    move_pose("pick_pose")
    tool_pick()
    set_tool_payload(tool_mass + object_mass)

    move_pose("place_pose")
    tool_place()
    set_tool_payload(tool_mass)
```

### read_digital_main_input(*identifiers*)

Lire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Digital Main Inputs".

Les deux états des E/S numériques sont:

- *HIGH*, représenté par la valeur booléenne *True*
- *LOW*, représenté par la valeur booléenne *False*

**Paramètres:**

**identifiers** (*int, str, list (of int or str), set (of int or str)*) –

Spécifie quelles E/S lire. Les identificateurs sont soit les numéros des E/S, soit les noms personnalisés uniques pouvant être attribués à ces E/S. Cela peut être l'une des options suivantes:

- un seul numéro (int) ou nom personnalisé (str) -> retourne un bool
- une liste de numéros (int) ou de noms personnalisés (str), ou mixte -> retourne une liste de bool
- un ensemble de numéros (int) ou de noms personnalisés (str), ou mixte -> retourne un dictionnaire avec des valeurs booléennes et les identifiants comme clés

**Renvoie:**

Les valeurs booléennes des E/S numériques demandées (HIGH = True, LOW = False)

**Type renvoyé:**

bool, list, dict

**Exemples:**

Supposons que le nom "trigger" ait été assigné à l'une des entrées numériques. Ce bloc de code attend que ce déclencheur passe à HIGH avant de poursuivre l'exécution.

```
while not read_digital_main_input("trigger"):
    wait(0.1)
```

Lire les 8 premières E/S numériques et convertir la liste binaire de booléens en un nombre entier entre 0 et 255.

```
bool_list = read_digital_main_input(list(range(1, 9)))
int_representation = int(''.join([str(int(bit)) for bit in bool_list]), 2)
```

**read_digital_tool_input(*identifiers*)**

Lire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Digital Tool Inputs".

Pour plus de détails, reportez-vous à "read_digital_main_input", dont l'utilisation est similaire.

### read_analog_main_input(*identifiers*)

Lire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Analog Main Inputs".

Les E/S analogiques peuvent être en mode tension ou en mode courant. Les valeurs sont appliquées et lues par pas de 0.1, ce qui signifie que la valeur d'E/S 55 représente la valeur réelle de tension de 5.5 V ou la valeur de courant de 5.5 mA. La plage de valeurs des E/S analogiques est:

- Entre 0.0 et 10.0 V (en mode tension)
- Entre 4.0 et 20.0 mA (en mode courant)

**Paramètres:**

**identifiers** (*int, str, list (of int or str), set (of int or str)*) –

Spécifie quelles E/S lire. Les identificateurs sont soit les numéros des E/S, soit les noms personnalisés uniques pouvant être attribués à ces E/S. Cela peut être l'une des options suivantes:

- un seul numéro (int) ou nom personnalisé (str) -> retourne un bool
- une liste de numéros (int) ou de noms personnalisés (str), ou mixte -> retourne une liste de bool
- un ensemble de numéros (int) ou de noms personnalisés (str), ou mixte -> retourne un dictionnaire avec des valeurs booléennes et les identifiants comme clés

**Renvoie:**

Les valeurs numériques des E/S analogiques demandées

**Type renvoyé:**

float, list, dict

**Exemples:**

Supposons que le nom "temperature" ait été attribué à l'une des entrées analogiques en mode courant. Un capteur de température a été raccordé à cette entrée, et il a été déterminé qu'une valeur de retour de 10.6 mA représente un seuil critique de température à ne pas dépasser. Exécutez ensuite le programme suivant en parallèle de votre application principale, lequel interrompt le programme si le seuil critique est atteint.

```
threshold = 10.6
while True:
    value = read_analog_main_input("temperature")
    if value >= threshold:
        raise_error("High temperature detected -> aborting the program")
    wait(0.1)
```

**read_analog_tool_io(*identifiers*)**

Lire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Analog Tool I/O".

Pour plus de détails, reportez-vous à "read_analog_main_input", dont l'utilisation est similaire.

### write_digital_main_output(*identifiers*, *values*)

Écrire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Digital Main Output".

Les deux états des E/S numériques sont:

- *HIGH*, représenté par la valeur booléenne *True*
- *LOW*, représenté par la valeur booléenne *False*

**Paramètres:**

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) –
  Spécifie quelles E/S écrire. Les identificateurs sont soit les numéros d'E/S, soit les noms personnalisés uniques pouvant être attribués à ces E/S. Cela peut être l'une des options suivantes :
  - un seul numéro (int) ou nom personnalisé (str) -> *values* est un bool
  - une liste de numéros (int) ou noms personnalisés (str), ou mixte -> *values* est une liste de bool
  - un ensemble de numéros (int) ou noms personnalisés (str), ou mixte -> *values* est un dictionnaire avec des valeurs booléennes et les identifiants comme clés
- **values** (*bool, list (of bool), dict (of bool)*) –
  Les valeurs booléennes des E/S numériques à écrire (HIGH = True, LOW = False). Selon les *identifiers*, cela peut être l'un des suivants:
  - un seul bool -> si *identifiers* est une seule E/S
  - une liste de booléens -> si *identifiers* est une liste d'E/S
  - un dictionnaire avec des valeurs booléennes et des identifiants comme clés -> si *identifiers* est un ensemble d'E/S

**Exemples:**

Supposons que le nom "trigger" ait été assigné à l'une des sorties digitales. Ce code envoie une impulsion à cette sortie digitale en la mettant à HIGH pendant 100 ms, puis la remet à LOW.

```
write_digital_main_output("trigger", True)
wait(0.1)
write_digital_main_output("trigger", False)
```

Écrire un nombre entier 8 bits (entre 0 et 255) dans les 8 premières E/S digitales, en convertissant d'abord le nombre en liste de booléens, puis en écrivant les 8 valeurs simultanément.

```
number = 123
bool_list = [bool(int(bit)) for bit in f"{number:08b}"]
write_digital_main_output(list(range(1, 9)), bool_list)
```

**write_digital_tool_output(*identifiers*, *values*)**

Écrire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Digital Tool Output".

Pour plus de détails, reportez-vous à "write_digital_main_output", dont l'utilisation est similaire.

### write_analog_main_output(*identifiers*, *values*)

Écrire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Analog Main Outputs".

Les E/S analogiques peuvent être en mode tension ou en mode courant. Les valeurs sont appliquées et lues par pas de 0.1, ce qui signifie que la valeur d'E/S 55 représente la valeur réelle de tension de 5.5 V ou la valeur de courant de 5.5 mA. La plage de valeurs des E/S analogiques est:

- Entre 0.0 et 10.0 V (en mode tension)
- Entre 4.0 et 20.0 mA (en mode courant)

**Paramètres:**

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) –
  Spécifie quelles E/S écrire. Les identificateurs sont soit les numéros d'E/S, soit les noms personnalisés uniques pouvant être attribués à ces E/S. Cela peut être l'une des options suivantes :
  - un seul numéro (int) ou nom personnalisé (str) -> *values* est un float
  - une liste de numéros (int) ou noms personnalisés (str), ou mixte -> *values* est une liste de float
  - un ensemble de numéros (int) ou noms personnalisés (str), ou mixte -> *values* est un dictionnaire avec des valeurs float et les identifiants comme clés
- **values** (*float, list (of float), dict (of float)*) –
  Les valeurs numériques des E/S analogiques à écrire. Selon les *identifiers*, cela peut être l'une des options suivantes:
  - un float unique -> si *identifiers* est un seul I/O
  - une liste de float -> si *identifiers* est une liste d'I/O
  - un dict avec des valeurs float et les identificateurs comme clés -> si *identifiers* est un ensemble d'I/O

**Exemples:**

Supposons que les valeurs RGB d'une LED contrôlable soient directement connectées à trois E/S analogiques portant les noms personnalisés « red », « green » et « blue », de sorte que la plage de valeurs maximale [0, 255] corresponde directement à la plage de tension maximale [0.0 V, 10.0 V]. Cet exemple écrit ces valeurs simultanément.

```
if color == "orange":
    color_code = [255, 188, 32]
elif color == "green":
    color_code = [31, 226, 0]
elif color == "purple":
    color_code = [170, 22, 218]
else:
    raise_error("Unknown color")
write_analog_main_output(["red", "green", "blue"], [c/2.55 for c in
color_code])
```

**write_analog_tool_io(*identifiers*, *values*)**

Écrire la valeur d'une, plusieurs ou de toutes les E/S de la catégorie "Analog Tool I/O".

Pour plus de détails, reportez-vous à "write_analog_main_output", dont l'utilisation est similaire.

### wait_for_digital_main_input(*identifier*, *value*)

Attente qu'une E/S de la catégorie "Digital Main Inputs" lise une valeur spécifique.

Les deux états des E/S numériques sont:

- *HIGH*, représenté par la valeur booléenne *True*
- *LOW*, représenté par la valeur booléenne *False*

**Paramètres:**

- **identifier** (*int, str*) – Spécifie quelle E/S attendre. L'identifiant peut être le numéro de l'E/S (int) ou le nom personnalisé unique (str) qui peut être assigné à cette E/S.
- **value** (*bool*) – La valeur à attendre.

**Exemples:**

Supposons que le nom "trigger" ait été assigné à l'une des entrées numériques. Ce bloc de code attend que ce déclencheur passe à HIGH avant de poursuivre l'exécution.

```
wait_for_digital_main_input("trigger", True)
```

Dans de rares cas, vous pouvez vouloir mettre un programme en pause et souhaitez que la fonction d'attente se termine si l'E/S est déclenchée. Cette fonction d'attente ne peut pas être utilisée pour cette approche, mais l'exemple suivant le fait exactement:

Ajoutez ce code dans une application en arrière-plan. Il surveille la valeur d'une E/S spécifique et définit un drapeau si la valeur spécifiée est atteinte pendant la surveillance.

```
# writing the 'trigger' value into a global variable for further processing
set_global_variable("triggered_flag", False)
while True:
    if not get_global_variable("triggered_flag"):
        set_global_variable("triggered_flag",
read_digital_main_input("trigger"))
    wait(0.01)
```

Et ajoutez ce code dans l'application principale qui déclenche le moniteur en arrière-plan et attend que le drapeau soit défini par le moniteur.

```
# waiting for the 'trigger' value to be written into a global variable
set_global_variable ("triggered_flag", False)
while not get_global_variable("triggered_flag"):
    wait(0.01)
```

**wait_for_digital_tool_input(*identifier*, *value*)**

Attente qu'une E/S de la catégorie "Digital Tool Inputs" lise une valeur spécifique.

Pour plus de détails, reportez-vous à "wait_for_digital_main_input", dont l'utilisation est similaire.

**wait_for_analog_main_input(*identifier*, *value_range*)**

Attente qu'une E/S de la catégorie "Analog Main Inputs" lise une valeur spécifique.

Les E/S analogiques peuvent être en mode tension ou en mode courant. Les valeurs sont appliquées et lues par pas de 0.1, ce qui signifie que la valeur d'E/S 55 représente la valeur réelle de tension de 5.5 V ou la valeur de courant de 5.5 mA. La plage de valeurs des E/S analogiques est:

- Entre 0.0 et 10.0 V (en mode tension)
- Entre 4.0 et 20.0 mA (en mode courant)

**Paramètres:**

- **identifier** (*int, str*) – Spécifie quelle E/S attendre. L'identifiant peut être le numéro de l'E/S (int) ou le nom personnalisé unique (str) qui peut être assigné à cette E/S.
- **value_range** (*list of float*) –
  En attente que cette E/S atteigne une valeur comprise dans cette plage. Il s'agit de l'un des éléments suivants:
  - à la fois une limite inférieure et une limite supérieure (p. ex. [5.5, 7.0])
  - uniquement une limite inférieure (p. ex. [5.5, None])
  - uniquement une limite supérieure (p. ex. [None, 7.0])

**Exemples:**

Considérons une entrée/sortie analogique en mode tension. Cet exemple attend que la valeur de la tension dépasse 8.0 V, ce qui est représenté par la valeur 80 (par pas de 0.1 V).

```
wait_for_analog_main_input(6, [80, None])
```

**wait_for_analog_tool_io(*identifier*, *value_range*)**

Attente qu'une E/S de la catégorie "Analog Tool I/O" lise une valeur spécifique.

Pour plus de détails, reportez-vous à "wait_for_analog_main_input", dont l'utilisation est similaire.

**get_runtime_position_offsets()**

Cette fonction renvoie les décalages de position du robot.

**Renvoie:**

décalages de position du robot

**Type renvoyé:**

list of float

### read_button_state(*channel=None*)

Lecture de l'état actuel (pressé ou non) d'un ou de tous les boutons personnalisables de l'interface du robot. Notez que les boutons avec des fonctions spécifiques assignées ne peuvent pas être lus en temps réel, car ils sont utilisés autrement.

Ces boutons sont situés sur l'articulation la plus haute du bras du robot.

**Paramètres:**

**channel** (*int or None*) – L'identifiant du bouton, qui est un nombre compris entre 1 et le nombre total de boutons. Par défaut, les états de tous les boutons personnalisables sont lus.

**Renvoie:**

L'état actuel du bouton demandé en tant que booléen, ou l'état actuel de chaque bouton, regroupé dans un dictionnaire de la forme {1: bool, 2: bool, ...}.

**Type renvoyé:**

bool or dict of bool

**Exemples:**

Cet exemple affiche un message chaque fois que l'état d'un bouton personnalisable change.

```
button_states = read_button_state()
while True:
    for channel, value in read_button_state().items():
        if not button_states[channel] and value:
            print(f"Button {channel} was pressed.")
        elif button_states[channel] and not value:
            print(f"Button {channel} was released.")
        button_states[channel] = value
    wait(0.1)
```

Exécutez cet exemple comme une application en arrière-plan. Il permet de mettre en pause et de reprendre l'application principale en cours à l'aide des boutons 1 et 2.

```
pausing = False
while True:
    if read_button_state(1) and not pausing:
        pause_application()
        pausing = True
    elif read_button_state(2) and pausing:
        resume_application()
        pausing = False
    wait(0.1)
```

### create_tcp_client(*ip*, *port*)

Création d'un client socket TCP pour transmettre des messages. Le client tente continuellement de se connecter au serveur à l'adresse spécifiée. Les messages sont encodés avec le standard *UTF-8*. De plus, les messages sont séparés par le caractère de fin de texte *0x03* ( `\x03` en Python).

**Paramètres:**

- **ip** (*str*) – L'adresse IP du serveur (par exemple, `"192.168.80.2"` )
- **port** (*int*) – Le numéro de port par lequel le serveur est accessible (ex. `60001` ). Les ports sont limités à la plage entre 60000 et 61000.

**Renvoie:**

L'objet client de socket TCP, utilisé pour envoyer et recevoir des messages.

**Type renvoyé:**

Socket

**Exemples:**

Cet exemple effectue une poignée de main (handshake) entre un client socket TCP dans l'application et un serveur socket TCP sur le même réseau. Tout d'abord, la connexion est établie. Ensuite, le client attend que le serveur envoie un message initial. À la réception, le client répond avec son propre message, puis ferme finalement la connexion.

```
client = create_tcp_client("192.168.80.2", 60001)
print(client.receive())
client.send("I am your client")
client.close()
```

### create_tcp_server(*port*)

Création d'un serveur de socket TCP pouvant gérer jusqu'à 20 clients. Les messages sont encodés avec le standard *UTF-8*. De plus, les messages sont séparés par le caractère de fin de texte *0x03* ( `\x03` en Python).

**Paramètres:**

**port** (*int*) – Le numéro de port par lequel ce serveur peut être accessible aux clients (ex. `60002`). Les ports sont limités à la plage entre 60000 et 61000.

**Renvoie:**

L'objet serveur de socket TCP, utilisé pour envoyer et recevoir des messages.

**Type renvoyé:**

Socket

**Exemples:**

Cet exemple effectue une poignée de main (handshake) entre un serveur socket TCP dans l'application et le premier client qui s'y connecte. Tout d'abord, le serveur ouvre la connexion sur un port spécifique et attend qu'un client se connecte. Ensuite, le serveur attend que le client envoie un message initial. À la réception, le serveur répond avec son propre message, puis ferme la connexion avec tous les clients connectés.

```
server = create_tcp_server(60002)
print(server.receive()
server.send("I am your server")
server.close()
```

### create_udp_socket(*port*)

Création d'une socket UDP pour transmettre des messages. Les messages sont encodés avec le standard *UTF-8*. De plus, les messages sont séparés par le caractère de fin de texte *0x03* ( `\x03` en Python).

**Paramètres:**

**port** (*int*) – Le numéro de port par lequel cette socket peut être accessible (ex. `60003`). Les ports sont limités à la plage entre 60000 et 61000.

**Renvoie:**

L'objet socket UDP, utilisé pour envoyer et recevoir des messages.

**Type renvoyé:**

Socket

**Exemples:**

Cet exemple effectue une poignée de main (handshake) entre une socket UDP dans l'application et la première connexion qui s'ouvre avec elle. Tout d'abord, la socket ouvre la connexion sur un port spécifique et attend qu'une contrepartie se connecte. Ensuite, la socket attend que la contrepartie envoie un message initial. À la réception, la socket répond avec son propre message, puis ferme la connexion.

```
udp_socket = create_udp_socket(60003)
print(udp_socket.receive()
udp_socket.send("I am your socket")
udp_socket.close()
```

### send_message(*message*)

Envoi d'un message avec la clé "script_send" à une connexion TCP ouverte.

**Paramètres:**

**message** (*str*) – Le message à envoyer.

**Exemples:**

Envoi d'un message via TCP à tout appareil pouvant l'écouter:

```
send_message("Hello World!")
```

Côté appareil connecté, ces données sont reçues dans le format suivant:

```
{"script_send": "Hello World!"}
```

### receive_message(*timeout*=*None*)

Réception d'un message depuis une connexion TCP ouverte en appelant l'action "script_receive".

**Paramètres:**

**timeout** (*None or float*) – Le temps maximum d'attente pour un message. Si non spécifié, attente indéfinie.

**Exemples:**

Un appareil externe se connecte au robot via TCP et envoie les données suivantes:

```
{"bridge": "core", "action": "script_receive", "message": "Hello World!"}
```

Côté robot, cette fonction peut être utilisée pour recevoir et afficher le message:

```
message = receive_message(timeout=3.0)
print(message)
```

### clear_messages()

Effacement de la file de messages TCP des messages entrants précédents qui n'ont peut-être pas été lus.

**Exemples:**

Avant d'attendre un nouveau message, il est parfois judicieux de s'assurer que les anciens messages sont rejetés.

```
clear_message()
message = receive_message()
```

# *class* Coordinates

L'objet Coordinates définit une transformation mathématique pouvant être utilisée, par exemple, comme coordonnées cibles dans les fonctions de mouvement, ou pour définir des repères de coordonnées.

## __init__(*coordinates*)

Créer un objet Coordinates, soit avec des valeurs de coordonnées spécifiques, soit contenant les coordonnées actuelles du robot.

**Paramètres:**

**coordinates** (*list, dict, or Coordinates*) –

Un ensemble spécifique de coordonnées, qui peut être l'un des suivants:

- une liste de la forme [x, y, z, roll, pitch, yaw] avec des valeurs de position en [mm] et des valeurs d'orientation en [deg]
- un dictionnaire de la forme {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} avec des valeurs de position en [mm] et des valeurs d'orientation en [deg]
- un objet Coordinates (copiant un autre objet Coordinates)

**Constructeur:**

Créer un objet Coordinates à partir de valeurs de coordonnées.

```
coordinates = Coordinates({"x": 100, "y": 200, "z": 300,
                           "roll": 15, "pitch": -15, "yaw": 30})
coordinates = Coordinates([100, 200, 300, 15, -15, 30])
```

Créer un objet Coordinates qui contient les coordonnées actuelles du robot en utilisant la fonction `get_current_coordinates`.

```
current_coordinates = get_current_coordinates()
```

***property* position**

La partie position de l'objet Coordinates.

**Renvoie:**

La position sous la forme [x, y, z] en [mm].

**Type renvoyé:**

list of float

***property* orientation**

La partie orientation de l'objet Coordinates.

**Renvoie:**

L'orientation sous la forme [roll, pitch, yaw] en [deg].

**Type renvoyé:**

list of float

**to_list()**

Convertir l'objet Coordinates en une liste de valeurs de coordonnées sous la forme [x, y, z, roll, pitch, yaw], où les valeurs de position sont en [mm] et les valeurs de rotation en [deg].

**Renvoie:**

Valeurs de coordonnées au format liste.

**Type renvoyé:**

list of float

**Exemples:**

Cette fonction peut être utilisée lorsque les coordonnées sont nécessaires sous forme de liste:

```
coord = Coordinates({"x": 700.0, "y": -125.0, "z": 500.0,
                     "roll": 180.0, "pitch": 0.0, "yaw": -90.0})

# convert to list
coord_list = coord.to_list()
```

### to_dict()

Convertir l'objet Coordinates en un dictionnaire de valeurs de coordonnées sous la forme {« x »: x, « y »: y, « z »: z, « roll »: roll, « pitch »: pitch, « yaw »: yaw}, où les valeurs de position sont en [mm] et les valeurs de rotation en [deg].

**Renvoie:**

Valeurs de coordonnées au format dictionnaire.

**Type renvoyé:**

dict of float

**Exemples:**

Cette fonction peut être utilisée lorsque les coordonnées sont nécessaires sous forme de dictionnaire:

```
coord = Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])

# convert to dict
coord_dict = coord.to_dict()
```

### inverted()

Calcule la transformation inverse de cet objet Coordinates.

**Renvoie:**

inverse de l'objet de transformation

**Type renvoyé:**

Transform

**Exemples:**

```
c = Coordinates([100, 200, 300, 15, -15, 30])

# create the inverse transform
c_inv = c.inverted()

# the inverse transform applied to the original transform
# this results in the identity transform
assert c * c_inv == Coordinates([0, 0, 0, 0, 0, 0])
assert c_inv * c == Coordinates([0, 0, 0, 0, 0, 0])
```

## *class* Pose

L'objet Pose définit une pose du robot, composée de ses coordonnées (position et orientation) et de sa configuration (le même ensemble de coordonnées pouvant être atteint avec plusieurs configurations).

### __init__(*pose*)

Créer un objet Pose, soit en chargeant une pose enregistrée par nom ou ID, soit à partir de la pose actuelle du robot.

**Paramètres:**

**pose** (*str, int, or Pose*) –

Référence à la pose à charger, qui peut être l'une des suivantes:

- le nom (str) d'une pose enregistrée dans la base de données.
- l'ID (int) d'une pose enregistrée dans la base de données
- un objet Pose (copiant un autre objet Pose)

**Constructeur:**

Charger une pose depuis la base de données.

```
pose = Pose("start_pose")
```

Créer un objet Pose contenant la pose actuelle du robot en utilisant la fonction `get_current_pose`.

```
current_pose = get_current_pose()
```

Créer un objet Pose contenant les coordonnées et la configuration spécifiées en utilisant la fonction `create_pose`.

```
pose = create_pose(coordinates, configuration)
```

***property* coordinates**

Les coordonnées de cette pose, qui est un objet Coordinates contenant les valeurs de position et d'orientation.

**Renvoie:**

Les coordonnées de cette pose.

**Type renvoyé:**

Coordinates

***property* configuration**

Étant donné qu'un ensemble de coordonnées de pose peut être atteint avec jusqu'à 8 configurations possibles des articulations du robot, la configuration est définie comme un nombre (de 0 à 7) ou une chaîne (de "c1" à "c8") pour définir de manière unique la pose.

**Renvoie:**

L'indice de la configuration de la pose (de 0 à 7).

**Type renvoyé:**

int

## *class* LinearSegment

L'objet LinearSegment décrit un segment de trajectoire linéaire du TCP (point central de l'outil) vers les coordonnées souhaitées. Il inclut également des informations sur la vitesse linéaire souhaitée, l'accélération linéaire et le rayon de mixage au début du segment. Ce rayon de mixage permet une déviation de la trajectoire linéaire pour assurer une transition fluide entre les segments.

### __init__(*linear_segment*)

Créer un objet LinearSegment.

**Paramètres:**

- **coordinates** (*list, dict, or Coordinates*) – Coordonnées cibles.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de mixage au début du segment [mm]. Si aucun rayon de mixage n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet LinearSegment en définissant tous les paramètres:

```
linear_seg = LinearSegment([100, 200, 300, 15, -15, 30],
                           linear_velocity=500, linear_acceleration=1000,
                           blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
linear_seg = LinearSegment([100, 200, 300, 15, -15, 30])
```

***property* coordinates**

Les coordonnées cibles de ce segment, qui est un objet Coordinates contenant les valeurs de position et d'orientation.

**Renvoie:**

Les coordonnées cibles de ce segment.

**Type renvoyé:**

Coordinates

***property* blend_radius**

Le rayon de mixage au début de ce segment.

**Renvoie:**

Le rayon de mixage initial.

**Type renvoyé:**

float

***property* linear_acceleration**

L'accélération linéaire souhaitée pour ce segment. Si elle est None, la valeur globale d'accélération linéaire est utilisée.

**Renvoie:**

L'accélération linéaire souhaitée.

**Type renvoyé:**

float or None

***property* linear_velocity**

La vitesse linéaire souhaitée pour ce segment. Si elle est None, la valeur globale de vitesse linéaire est utilisée.

**Renvoie:**

La vitesse linéaire souhaitée.

**Type renvoyé:**

float or None

## *class* CircularSegment

L'objet CircularSegment décrit un mouvement circulaire du TCP (point central de l'outil) à travers une position intermédiaire jusqu'aux coordonnées souhaitées. Notez que l'orientation de cette pose intermédiaire est définie par l'algorithme. Il inclut également des informations sur la vitesse linéaire souhaitée, l'accélération linéaire et le rayon de mixage au début du segment. Ce rayon de mixage permet une déviation du chemin circulaire pour assurer une transition fluide entre les segments.

### __init__(*circular_segment*)

Créer un objet CircularSegment.

**Paramètres:**

- **intermediate_position** (*list*) – Une position intermédiaire [x, y, z] sur le chemin vers les coordonnées cibles. La position intermédiaire détermine la courbure du mouvement circulaire.
- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet CircularSegment en définissant tous les paramètres:

```
circular_seg = CircularSegment([0, 300, 300], [100, 200, 300, 15, -15, 30],
                               linear_velocity=500, linear_acceleration=1000,
                               blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
circular_seg = CircularSegment([0, 300, 300], [100, 200, 300, 15, -15, 30])
```

***property* intermediate_position**

La position intermédiaire de ce segment. Elle inclut les coordonnées x, y, z: [x, y, z]

**Renvoie:**

La position intermédiaire de ce segment.

**Type renvoyé:**

list

***property* coordinates**

Les coordonnées cibles de ce segment, qui est un objet Coordinates contenant les valeurs de position et d'orientation.

**Renvoie:**

Les coordonnées cibles de ce segment.

**Type renvoyé:**

Coordinates

***property* blend_radius**

Le rayon de mixage au début de ce segment.

**Renvoie:**

Le rayon de mixage initial.

**Type renvoyé:**

float

***property* linear_acceleration**

L'accélération linéaire souhaitée pour ce segment. Si elle est None, la valeur globale d'accélération linéaire est utilisée.

**Renvoie:**

L'accélération linéaire souhaitée.

**Type renvoyé:**

float or None

***property* linear_velocity**

La vitesse linéaire souhaitée pour ce segment. Si elle est None, la valeur globale de vitesse linéaire est utilisée.

**Renvoie:**

La vitesse linéaire souhaitée.

**Type renvoyé:**

float or None

## *class* OrientationSegment

L'objet OrientationSegment décrit un segment de trajectoire d'orientation pure où le TCP (point central de l'outil) tourne selon l'orientation souhaitée tout en gardant sa position constante. De plus, il inclut des informations sur la vitesse angulaire souhaitée, l'accélération angulaire et le rayon de courbure au début du segment. Si le rayon de courbure est supérieur à 0, la rotation du TCP commence avant d'atteindre la position cible.

### **__init__(*orientation_segment*)**

Créer un objet OrientationSegment.

**Paramètres:**

- **orientation** (*list*) – Orientation cible. Notez que la position reste constante.
- **angular_velocity** (*float or None*) – Vitesse angulaire souhaitée [°/s]. Si aucune vitesse angulaire n'est définie, la valeur globale sera utilisée.
- **angular_acceleration** (*float or None*) – Accélération angulaire souhaitée [°/s$^2$]. Si aucune accélération angulaire n'est définie, la valeur globale sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Un rayon supérieur à 0 permet au point central de l'outil (TCP) de tourner avant que la position de ce segment ne soit atteinte. Si aucun rayon n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet OrientationSegment en définissant tous les paramètres:

```
orientation_seg = OrientationSegment([15, -15, 30],
                                     angular_velocity=250,
angular_acceleration=500,
                                     blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
orientation_seg = OrientationSegment([15, -15, 30])
```

***property* orientation**

L'orientation cible de ce segment. Donnée en angles nautiques [roll, pitch, yaw] en degrés.

**Renvoie:**

L'orientation cible de ce segment.

**Type renvoyé:**

list

***property* angular_velocity**

La vitesse angulaire souhaitée pour ce segment. Si elle est None, la valeur globale est utilisée.

**Renvoie:**

La vitesse angulaire souhaitée.

**Type renvoyé:**

float or None

***property* angular_acceleration**

L'accélération angulaire souhaitée pour ce segment. Si elle est None, la valeur globale est utilisée.

**Renvoie:**

L'accélération angulaire souhaitée.

**Type renvoyé:**

float or None

***property* blend_radius**

Le rayon de mixage au début de ce segment.

**Renvoie:**

Le rayon de mixage initial.

**Type renvoyé:**

float

## *class* CoordinatesSegment

L'objet CoordinatesSegment décrit un mouvement linéaire dans l'espace des articulations vers les coordonnées cibles. Le TCP (point central de l'outil), en revanche, ne se déplace généralement pas de manière linéaire. Il inclut également des informations sur la vitesse linéaire souhaitée, l'accélération linéaire et le rayon de courbure au début du segment. Ce rayon de courbure permet une déviation de la trajectoire pour assurer une transition en douceur entre les segments.

### **__init__(*coordinates_segment*)**

Créer un objet CoordinatesSegment.

**Paramètres:**

- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet CoordinatesSegment en définissant tous les paramètres:

```
coordinates_seg = CoordinatesSegment([100, 200, 300, 15, -15, 30],
                                     linear_velocity=250,
linear_acceleration=500,
                                     blend_radius=100)
```

ou en utilisant les valeurs par défaut:

```
coordinates_seg = CoordinatesSegment([100, 200, 300, 15, -15, 30])
```

***property* coordinates**

Les coordonnées cibles de ce segment, qui est un objet Coordinates contenant les valeurs de position et d'orientation.

**Renvoie:**

Les coordonnées cibles de ce segment.

**Type renvoyé:**

Coordinates

***property* blend_radius**

Le rayon de mixage au début de ce segment.

**Renvoie:**

Le rayon de mixage initial.

**Type renvoyé:**

float

***property* linear_acceleration**

L'accélération linéaire souhaitée pour ce segment. Si elle est None, la valeur globale d'accélération linéaire est utilisée.

**Renvoie:**

L'accélération linéaire souhaitée.

**Type renvoyé:**

float or None

***property* linear_velocity**

La vitesse linéaire souhaitée pour ce segment. Si elle est None, la valeur globale de vitesse linéaire est utilisée.

**Renvoie:**

La vitesse linéaire souhaitée.

**Type renvoyé:**

float or None

## *class* PoseSegment

L'objet PoseSegment décrit un mouvement linéaire dans l'espace des articulations vers la pose cible. Le TCP (point central de l'outil) ne se déplace généralement pas de manière linéaire. Il inclut également des informations sur la vitesse linéaire souhaitée, l'accélération linéaire et le rayon de courbure au début du segment. Ce rayon de courbure permet une déviation de la trajectoire pour assurer une transition en douceur entre les segments.

### **__init__(*pose_segment*)**

Créer un objet PoseSegment.

**Paramètres:**

- **pose** (*Pose*) – Pose cible.
- **linear_velocity** (*float or None*) – Vitesse linéaire souhaitée [mm/s]. Si aucune vitesse linéaire n'est définie, la valeur globale de vitesse linéaire sera utilisée.
- **linear_acceleration** (*float or None*) – Accélération linéaire souhaitée [mm/s$^2$]. Si aucune accélération linéaire n'est définie, la valeur globale d'accélération linéaire sera utilisée.
- **blend_radius** (*float or None*) – Rayon de courbure au début du segment [mm]. Si aucun rayon de courbure n'est défini, 0.0 sera utilisé.

**Constructeur:**

Création d'un objet PoseSegment en définissant tous les paramètres:

```
pose_seg = PoseSegment(create_pose([100, 200, 300, 15, -15, 30], "c1"),
                       linear_velocity=250, linear_acceleration=500,
                       blend_radius=100)
```

ou en utilisant les valeurs par défaut et la pose depuis la base de données:

```
pose_seg = PoseSegment("target_pose")
```

***property* pose**

La pose cible de ce segment, qui est un objet Pose contenant les valeurs de position, d'orientation et de configuration.

**Renvoie:**

La pose cible de ce segment.

**Type renvoyé:**

Pose

***property* blend_radius**

Le rayon de mixage au début de ce segment.

**Renvoie:**

Le rayon de mixage initial.

**Type renvoyé:**

float

***property* linear_acceleration**

L'accélération linéaire souhaitée pour ce segment. Si elle est None, la valeur globale d'accélération linéaire est utilisée.

**Renvoie:**

L'accélération linéaire souhaitée.

**Type renvoyé:**

float or None

***property* linear_velocity**

La vitesse linéaire souhaitée pour ce segment. Si elle est None, la valeur globale de vitesse linéaire est utilisée.

**Renvoie:**

La vitesse linéaire souhaitée.

**Type renvoyé:**

float or None

## *class* Path

L'objet Path définit comment le robot se déplace entre deux poses ou plus. Un chemin continu est défini uniquement dans l'espace de l'outil par plusieurs segments, comme linéaires ou circulaires. Un chemin est dit sans couture si les solutions discrètes du chemin ne nécessitent pas que le robot se repositionne (changement de configuration). Un chemin est dit continu si l'accélération est continue le long du chemin. Un chemin commence à une pose spécifiée (plutôt qu'à des coordonnées). Les segments du chemin définissent généralement des coordonnées, car la configuration du chemin est définie par la pose initiale.

### __init__(*path*)

Créer un objet Path, soit en chargeant un chemin sauvegardé par nom ou ID.

**Paramètres:**

**path** (*str, int, or Path*) –

Référence du chemin à charger, qui peut être l'un des suivants:

- le nom (str) d'une trajectoire enregistrée dans la base de données
- l'ID (int) d'une trajectoire enregistrée dans la base de données
- un objet Path (copie d'un autre objet Path)

**Constructeur:**

Charger un chemin depuis la base de données.

```
path = Path("pick_place_path")
```

Créer un objet Path contenant la pose initiale spécifiée et les segments de chemin définis en utilisant la fonction `create_path`. Notez que des segments peuvent également être ajoutés ultérieurement.

```
path = create_path(initial_pose, segments)
```

***property* initial_pose**

La pose initiale du robot pour le chemin.

**Renvoie:**

Pose initiale du robot.

**Type renvoyé:**

Pose

***property* segments**

Les segments inclus dans le chemin. Notez que les segments n'ont pas tous besoin d'être du même type.

**Renvoie:**

Segments inclus.

**Type renvoyé:**

list of Segments (LinearSegment, CircularSegment, OrientationSegment, CoordinatesSegment, PoseSegment)

**add_segment(*segment*)**

Ajoute un segment à la fin du chemin.

**Paramètres:**

**segment** (*LinearSegment*, *CircularSegment*, *OrientationSegment*, *CoordinatesSegment*, *PoseSegment*) – Segment à ajouter.

**Exemples:**

Un segment peut être ajouté comme suit:

```
# creating and adding a segment
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_segment(lin_seg)
```

**add_linear(*coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Ajoute un segment linéaire à la fin du chemin.

**Paramètres:**

- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles du chemin.
- **linear_velocity** (*float*) – Vitesse linéaire maximale autorisée de l'outil [mm/s].
- **linear_acceleration** (*float*) – Accélération linéaire maximale autorisée de l'outil [mm/s²].
- **blend_radius** (*float*) – Rayon de fusion autorisé [mm] au début du segment.

**Exemples:**

Un segment linéaire peut être ajouté comme suit:

```
# add a linear segment with different argument types for the "coordinates"
parameter
path.add_linear([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_linear({"x": 700.0, "y": -125.0, "z": 500.0,
                 "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_linear(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
# first linear segment is added
coord = Coordinates([700.0, -250.0, 500.0, 180.0, 0.0, -90.0])
path.add_linear(coord, linear_velocity=500.0, linear_acceleration=1000.0)

# second linear segment is added
path.add_linear([700.0, 250.0, 500.0, 180.0, 0.0, -90.0],
                linear_velocity=250.0, linear_acceleration=500.0,
blend_radius=100.0)
```

Le premier segment définit un mouvement linéaire vers les coordonnées cibles spécifiées avec une vitesse maximale de 500 mm/s et une accélération maximale de 1000 mm/s². Notez que ces coordonnées ne sont pas atteintes exactement, car le segment suivant définit un rayon de fusion de 100 mm, ce qui permet une déviation de ces coordonnées pour assurer une transition plus fluide et efficace. Le deuxième segment ajoute ensuite un mouvement linéaire vers les coordonnées suivantes avec une vitesse maximale de 250 mm/s et une accélération maximale de 500 mm/s².

**add_circular(*intermediate_position*, *coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Ajoute un segment circulaire à la fin du chemin.

**Paramètres:**

- **intermediate_position** (*list*) – Une position intermédiaire ([x, y, z]) sur le chemin vers les coordonnées cibles. La position intermédiaire détermine la courbure du mouvement circulaire.
- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles du chemin.
- **linear_velocity** (*float*) – Vitesse linéaire maximale autorisée de l'outil [mm/s].
- **linear_acceleration** (*float*) – Accélération linéaire maximale autorisée de l'outil [mm/s²].
- **blend_radius** (*float*) – Rayon de fusion autorisé [mm] au début du segment.

**Exemples:**

Un segment circulaire peut être ajouté comme suit:

```
# add a circular segment with different argument types for the "coordinates"
parameter
path.add_circular([0.0, -500.0, 500.0], [-500.0, -125.0, 500.0, 180.0, 0.0,
-90.0])
path.add_circular([0.0, -500.0, 500.0], {"x": -500.0, "y": -125.0, "z": 500.0,
                  "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_circular([0.0, -500.0, 500.0],
                  Coordinates([-500.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
# first circular segment is added
coord = Coordinates([-500.0, 250.0, 500.0, 180.0, 0.0, -90.0])
path.add_circular([-700.0, 0.0, 500.0], coord,
                  linear_velocity=500.0, linear_acceleration=1000.0)

# second circular segment is added
path.add_circular([-700.0, 0.0, 500.0], [-500.0, -250.0, 500.0, 180.0, 0.0,
-90.0],
                  linear_velocity=250.0, linear_acceleration=500.0,
blend_radius=0.0)
```

Le premier segment définit un mouvement circulaire passant par la position intermédiaire spécifiée vers les coordonnées cibles avec une vitesse maximale de 500 mm/s et une accélération maximale de 1000 mm/s². Notez que ces coordonnées ne sont pas atteintes exactement, car le segment suivant définit un rayon de mixage de 100 mm, ce qui permet une déviation de ces coordonnées pour assurer une transition plus fluide et efficace. Le

deuxième segment ajoute ensuite un mouvement circulaire passant par la position intermédiaire suivante vers les coordonnées cibles suivantes avec une vitesse maximale de 250 mm/s et une accélération maximale de 500 mm/s².

**add_orientation(*coordinates*, *angular_velocity=None*, *angular_acceleration=None*, *blend_radius=None*)**

Ajoute un segment d'orientation à la fin du chemin.

**Paramètres:**

- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles du chemin.
- **angular_velocity** (*float*) – Vitesse angulaire maximale autorisée de l'outil [°/s].
- **angular_acceleration** (*float*) – Accélération angulaire maximale autorisée de l'outil [°/s²].
- **blend_radius** (*float*) – Rayon de mixage autorisé [mm] au début du segment. Une valeur supérieure à 0 signifie qu'il peut déjà tourner avant d'atteindre la position souhaitée de ce segment.

**Exemples:**

Un segment d'orientation peut être ajouté comme suit:

```
# add an orientation segment
path.add_orientation([150.0, 0.0, -90.0])
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
# first orientation segment is added
path.add_orientation([180.0, 0.0, -90.0],
                     angular_velocity=250.0, angular_acceleration=500.0)

# second orientation segment is added
path.add_orientation([150.0, 0.0, -90.0], angular_velocity=500.0,
                     angular_acceleration=1000.0, blend_radius=100.0)
```

Le premier segment définit une rotation vers l'orientation cible spécifiée avec une vitesse maximale de 250 °/s et une accélération maximale de 500 °/s². Notez que lors de l'exécution de ce chemin, cette première rotation sera ignorée car le rayon de mixage du segment suivant est supérieur à 0 (voir `OrientationSegment.__init__`), et le second segment sera exécuté directement. Ce segment définit une rotation vers l'orientation cible de [150.0, 0.0, -90.0] avec une vitesse maximale de 500 °/s et une accélération maximale de 1000 °/s². En revanche, si le rayon de mixage du second segment était défini à 0, les deux rotations seraient exécutées en séquence.

**add_coordinates(*coordinates*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Ajoute un segment de coordonnées à la fin du chemin.

**Paramètres:**

- **coordinates** (*list, dict or Coordinates*) – Coordonnées cibles du chemin.
- **linear_velocity** (*float*) – Vitesse linéaire maximale autorisée de l'outil [mm/s].
- **linear_acceleration** (*float*) – Accélération linéaire maximale autorisée de l'outil [mm/s$^2$].
- **blend_radius** (*float*) – Rayon de fusion autorisé [mm] au début du segment.

**Exemples:**

Un segment de coordonnées peut être ajouté comme suit:

```
# different argument types for the "coordinates" parameter
path.add_coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_coordinates({"x": 700.0, "y": -125.0, "z": 500.0,
                      "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_coordinates(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
# first coordinates segment is added
coord = Coordinates([700.0, -250.0, 500.0, 180.0, 0.0, -90.0])
path.add_coordinates(coord, linear_velocity=500.0, linear_acceleration=1000.0)

# second coordinates segment is added
path.add_coordinates([700.0, 250.0, 500.0, 180.0, 0.0, -90.0],
linear_velocity=250.0,
                     linear_acceleration=500.0, blend_radius=100.0)
```

Le premier segment définit un mouvement vers les coordonnées cibles spécifiées avec une vitesse maximale de 500 mm/s et une accélération maximale de 1000 mm/s$^2$. Notez que ces coordonnées ne sont pas atteintes exactement, car le segment suivant définit un rayon de mixage de 100 mm, ce qui permet une déviation de ces coordonnées pour assurer une transition plus fluide et efficace. Le second segment ajoute ensuite un mouvement vers les coordonnées suivantes avec une vitesse maximale de 250 mm/s et une accélération maximale de 500 mm/s$^2$.

**add_pose(*pose*, *linear_velocity=None*, *linear_acceleration=None*, *blend_radius=None*)**

Ajoute un segment de pose à la fin du chemin.

**Paramètres:**

- **pose** (*Pose*) – Coordonnées cibles du chemin.
- **linear_velocity** (*float*) – Vitesse linéaire maximale autorisée de l'outil [mm/s].
- **linear_acceleration** (*float*) – Accélération linéaire maximale autorisée de l'outil [mm/s²].
- **blend_radius** (*float*) – Rayon de fusion autorisé [mm] au début du segment.

**Exemples:**

Un segment de pose peut être ajouté comme suit:

```
# using a pose from the database
pose_db = get_pose("pick_pose")
path.add_pose(pose_db)

# using a newly created pose
new_pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
path.add_pose(new_pose)
```

Dans ces exemples, aucun paramètre optionnel n'a été spécifié. Par conséquent, les paramètres globaux sont utilisés. Les paramètres peuvent également être définis explicitement dans l'appel de fonction:

```
# first pose segment is added
new_pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
path.add_pose(new_pose, linear_velocity=500.0, linear_acceleration=1000.0)

# second pose segment is added
path.add_pose(get_pose("pick_pose"), linear_velocity=250.0,
              linear_acceleration=500.0, blend_radius=100.0)
```

Le premier segment définit un mouvement vers la pose cible spécifiée avec une vitesse maximale de 500 mm/s et une accélération maximale de 1000 mm/s². Notez que cette pose n'est pas atteinte exactement, car le segment suivant définit un rayon de mixage de 100 mm, ce qui permet une déviation des coordonnées correspondantes pour assurer une transition plus fluide et efficace. Le second segment ajoute ensuite un mouvement vers la pose suivante avec une vitesse maximale de 250 mm/s et une accélération maximale de 500 mm/s².

## *class* Socket

Le Socket est un objet renvoyé par les fonctions `create_tcp_client`, `create_tcp_server` et `create_udp_socket`. Utilisez cet objet pour envoyer et recevoir des données via le socket.

### receive(*timeout*=*None*)

Réception d'un message via ce socket.

**Paramètres:**

**timeout** (*None or float*) – Le temps maximum d'attente pour un message. Si non spécifié, attente indéfinie.

**Renvoie:**

Le message reçu, ou une chaîne vide si un délai d'attente est survenu.

**Type renvoyé:**

str

**Exemples:**

Attente d'une réponse pendant 3 secondes et affichage lorsqu'elle est reçue.

```
print(socket.receive(timeout=3.0))
```

**send(*message*)**

Envoi d'un message via ce socket.

**Paramètres:**

**message** (*str*) – Le message à envoyer.

**Exemples:**

Envoyer un message à tous les pairs connectés.

```
socket.send("Hello world!")
```

### close()

Fermeture de la connexion du socket.

**Exemples:**

Fermer le socket.

```
socket.close()
```

# Fonctions du module complémentaire

# Fonctions de l’add-on Cognex

# *class* Cognex

Le module Cognex prend en charge l'utilisation des caméras Cognex. Notez que la configuration de la caméra Cognex nécessite l'installation de Cognex In-Sight Explorer.

Pour plus d'informations sur la manière de configurer correctement et de connecter votre caméra Cognex au robot, veuillez consulter le manuel utilisateur.

Toutes les fonctions de cet add-on sont appelées en utilisant le préfixe `cognex.`

## connect_to_camera(*ip*, *user*='*admin*', *password*='')

Configuration d'un client TCP se connectant à la caméra Cognex connectée via le port ISE pour les commandes natives.

**Paramètres:**

- **ip** (*str*) – L'adresse IP de la caméra, par ex. `'10.0.0.210'`.
- **user** (*str*) – Le nom d'utilisateur pour se connecter à la caméra (tel que précédemment défini dans Cognex In-Sight Explorer). Le nom d'utilisateur par défaut est `'admin'`.
- **password** (*str*) – Le mot de passe pour se connecter à la caméra (tel que précédemment défini dans Cognex In-Sight Explorer). Le mot de passe par défaut est une chaîne vide.

**Lève:**

**MinorError** – si les informations d'identification sont incorrectes, la connexion est interrompue pendant la connexion ou la connexion a pris trop de temps.

**Exemples:**

Se connecter à une caméra fraîchement livrée avec l'IP et les identifiants par défaut.

```
cognex.connect_to_camera("10.0.0.210")
```

Se connecter à une caméra avec une adresse IP et des identifiants modifiés précédemment.

```
cognex.connect_to_camera("192.168.80.235", user="Developer",
password="my_password_1234")
```

**trigger_camera(*ip*)**

Configuration de l'événement de déclenchement de la caméra (« SW8 ») qui prend une nouvelle image et renvoie les données traitées par le travail Cognex actif.

**Paramètres:**

**ip** (*str*) – L'adresse IP de la caméra, par ex. `'10.0.0.210'`.

**Renvoie:**

Le message de réponse de la caméra, contenant les entités détectées.

**Type renvoyé:**

str

### change_job(*ip*, *job_name*)

Changement du travail Cognex via la commande native de la caméra « LF », qui charge un fichier de travail déjà stocké sur la caméra. Notez que cette commande peut prendre plusieurs secondes à s'exécuter.

**Paramètres:**

- **ip** (*str*) – L'adresse IP de la caméra, par ex. `'10.0.0.210'`.
- **job_name** (*str*) – Le nom du travail vers lequel passer (tel que défini précédemment dans Cognex In-Sight Explorer).

**Lève:**

**MinorError** – si la caméra n'existe pas ou si le changement de travail a échoué.

**Exemples:**

Passer à un travail précédemment stocké sous "CircleDetection.job", puis prendre une photo pour détecter les cercles dans la vue actuelle de la caméra.

```
ip_address = "10.0.0.210"
cognex.change_job(ip, "CircleDetection")
detected_circle_message = cognex.trigger_camera(ip)
```

### get_camera_message(*ip*, *event_id*, *keyword*, *timeout=None*)

Déclencher un événement sur la caméra et demander un message socket spécifique côté Cognex.

**Paramètres:**

- **ip** (*str*) – L'adresse IP de la caméra, par ex. `'10.0.0.210'`.
- **event_id** (*int*) – L'ID de l'événement Cognex à déclencher.
- **keyword** (*str*) – Le mot-clé à attendre.
- **timeout** (*float or None*) – Le délai d'attente de la requête (en [s]). Utilisez `None` pour attendre indéfiniment.

**Lève:**

**TimeOutError** – si la caméra n'a pas répondu une chaîne valide avant l'expiration du délai.

**Exemples:**

Demande le message contenant la chaîne "my_message", lié à l'événement 1 (SW1) du travail Cognex In-Sight.

```
message = cognex.get_camera_message("10.0.0.210", 1, "my_message")
```

**disconnect_camera(*ip*)**

Déconnexion d'une caméra spécifique.

**Paramètres:**

**ip** (*str*) – L'adresse IP de la caméra, par ex. `'10.0.0.210'`.

# Fonctions de l’extension Modbus

# *class* Modbus

Le module Modbus contient des fonctions pour envoyer des requêtes à un serveur REST externe.

Toutes les fonctions de cette extension sont appelées en utilisant le préfixe `modbus.`

### write_user_register(*address*, *values*)

Écriture dans les registres de sortie Modbus.

**Paramètres:**

- **address** (*int*) – L'adresse du registre à partir de laquelle commencer l'écriture. Les valeurs possibles vont de 0x10 (16) à 0xFF (255).
- **values** (*int or list of int*) – La ou les valeurs à écrire. Chaque registre peut stocker jusqu'à 2 octets de données. La valeur maximale pouvant être stockée est donc 65535 (=$2^{16}$-1). Si plusieurs valeurs sont fournies, elles sont écrites dans des registres consécutifs.

**Exemples:**

Écrire une seule valeur à une adresse spécifique des registres de sortie Modbus.

```
# write 11111 to the address 0x20
modbus.write_user_register(0x20, 11111)
```

Écrire plusieurs valeurs dans des registres de sortie consécutifs à l'aide d'une seule commande.

```
# write 555 to the address 0x30, and 777 to the address 0x31
modbus.write_user_register(0x30, [555, 777])
```

**read_user_register(*address*, *count=1*)**

Lecture des registres d'entrée Modbus.

**Paramètres:**

- **address** (*int*) – L'adresse du registre à partir de laquelle commencer la lecture. Les valeurs possibles vont de 0x10 (16) à 0xFF (255).
- **count** (*int*) – Le nombre de registres consécutifs à lire. Par défaut, seul un registre est lu.

**Renvoie:**

Les valeurs lues à partir des registres. La longueur de cette liste est égale à l'argument *count*.

**Type renvoyé:**

list of int

**Exemples:**

Lire une seule valeur à une adresse spécifique des registres d'entrée Modbus.

```
# read value from the address 0x20
value = modbus.read_user_register(0x20)[0]
```

Lire plusieurs valeurs de registres d'entrée consécutifs à l'aide d'une seule commande.

```
# read values from the addresses 0x30 and 0x31
values = modbus.read_user_register(0x30, 2)
```

### wait_for_event(*index*)

Attente jusqu'à ce que l'événement avec l'index spécifié soit déclenché. Les événements sont définis depuis l'extérieur par le client Modbus.

**Paramètres:**

**index** (*int*) – L'index de l'événement à attendre. Les indices possibles vont de 0 à 15.

**Exemples:**

Attendre qu'un événement soit déclenché.

```
modbus.wait_for_event(2)
```

# Fonctions du module complémentaire REST

## *class* Rest

Le module REST contient des fonctions pour envoyer des requêtes à un serveur REST externe.

Toutes les fonctions de ce module complémentaire sont appelées en utilisant le préfixe `rest.`

### setup_client_connection(*target_url*)

Configuration d'une connexion client pour envoyer des requêtes à un serveur REST.

**Paramètres:**

**target_url** (*str*) – L'URL (point de terminaison général de l'API) du serveur REST auquel se connecter.

**Renvoie:**

L'objet de connexion client REST.

**Type renvoyé:**

RestClientConnection

**Exemples:**

Établir une connexion à un serveur REST.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# ... this is similar to ...
connection = rest.setup_client_connection("https://some_url:5000/endpoint/")
```

## *class* RestClientConnection

RestClientConnection est un objet renvoyé par `rest.setup_client_connection`. Il est utilisé pour envoyer des requêtes au serveur REST connecté.

### get_default_headers(*method*)

Lecture des en-têtes par défaut envoyés avec chaque requête.

**Paramètres:**

**method** (*str*) – La méthode (« get » ou « post ») pour laquelle récupérer les en-têtes.

**Renvoie:**

Les en-têtes par défaut actuellement définis pour cette méthode de requête.

**Type renvoyé:**

dict

**Exemples:**

Lecture des en-têtes par défaut des méthodes GET et POST.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# reading the default headers of the GET method, which by default are set to
#     {'Accept': 'application/json'}
headers = connection.get_default_headers("get")
# reading the default headers of the POST method, which by default are set to
#     {'Accept': 'application/json', 'Content-Type': 'application/json'}
headers = connection.get_default_headers("post")
```

### set_default_headers(*method*, *new_headers*)

Modifier les en-têtes par défaut envoyés avec chaque requête.

**Paramètres:**

- **method** (*str*) – La méthode (« get » ou « post ») pour laquelle définir les en-têtes.
- **new_headers** (*dict*) – Les nouveaux en-têtes par défaut à remplacer.

**Exemples:**

Établir une connexion à un serveur REST.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# resetting the headers of the GET method
connection.set_default_headers("get", {"Accept": "application/json"})
# resetting the headers of the POST method
connection.set_default_headers("post", {"Accept": "application/json",
"Content-Type": "application/json"})
```

**get(*resource*, *headers_override=None*, *query_params=None*, *secure_connection=True*)**

Envoi d'une requête GET au serveur REST.

**Paramètres:**

- **resource** – Le nom de la ressource à accéder. Le nom de la ressource est ajouté à l'URL (<url>/<resource>) et peut éventuellement contenir un caractère "/" au début.
- **headers_override** (*dict*) – Définir les en-têtes pour cette requête individuelle. Si non spécifié, les en-têtes par défaut sont utilisés.
- **query_params** (*dict*) – Spécification d'une requête (le cas échéant).
- **secure_connection** (*bool*) – Indique si une connexion sécurisée doit être utilisée (vérifier les certificats) ou non.

**Renvoie:**

Le code de réponse (code HTTP, p.ex. `200` pour une requête réussie) et la charge utile de la réponse (données de réponse) (le cas échéant).

**Type renvoyé:**

(int, dict)

**Exemples:**

Envoi d'une requête GET à un serveur REST et vérification de son code de réponse.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
code, payload = connection.get("my_resource")  # or
connection.get("/my_resource")
# print received data if request was successful:
if code == 200:
    print(payload)
```

**post(*resource*, *data*, *headers_override=None*, *query_params=None*, *secure_connection=True*)**

Envoi d'une requête POST au serveur REST.

**Paramètres:**

- **resource** (*str*) – Le nom de la ressource à accéder. Le nom de la ressource est ajouté à l'URL (<url>/<resource>) et peut éventuellement contenir un caractère "/" au début.
- **data** (*str or dict*) – Le corps/les données de la requête POST.
- **headers_override** (*dict*) – Définir les en-têtes pour cette requête individuelle. Si non spécifié, les en-têtes par défaut sont utilisés.
- **query_params** (*dict*) – Spécification d'une requête (le cas échéant).
- **secure_connection** (*bool*) – Indique si une connexion sécurisée doit être utilisée (vérifier les certificats) ou non.

**Renvoie:**

Le code de réponse (code HTTP, p.ex. `200` pour une requête réussie) et la charge utile de la réponse (données de réponse) (le cas échéant).

**Type renvoyé:**

(int, dict)

**Exemples:**

Envoi d'une requête POST à un serveur REST et vérification de son code de réponse.

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
code, payload = connection.post("my_resource")  # or
connection.post("/my_resource")
# print received data if request was successful:
if code == 200:
    print(payload)
```

# Fonctions du module complémentaire ROS

# *class* ROSHandler

Le module ROS ajoute des fonctionnalités ROS pour d'autres modules complémentaires et pour l'accès direct par les utilisateurs.

Toutes les fonctions de ce module complémentaire sont appelées en utilisant le préfixe `ros_handler.`

## ros_node()

Le module ROS crée son propre nœud ROS, auquel on peut accéder directement via cette fonction. Cela peut être utilisé pour créer des objets comme des publishers, des subscribers, etc., dans des scripts. De plus, `rclpy` peut être importé directement dans les applications à cet effet.

**Avertissement**

Ne lancez pas `rclpy.spin()` sur ce nœud, car il fonctionne déjà en arrière-plan !

**Exemples:**

Affiche les noms de tous les topics sur lesquels le robot publie.

```
for publisher in ros_handler.ros_node().publishers:
    print(publisher.topic_name)
```

**Renvoie:**

Le nœud ROS du module ROS.

**Type renvoyé:**

rclpy.node.Node