Documentación de Programación de Robots

Versión traducida

Rev.2
ESM268S9135M

Índice

Conceptos básicos de Python

Estándares básicos de Python

Esta sección explica algunos conceptos básicos de programación en Python que pueden utilizarse en el editor de aplicaciones. Esto es solo un resumen de los estándares más importantes de Python (3.13); por supuesto, existen muchas más funciones integradas en Python. Tenga en cuenta que una aplicación robótica no es simplemente una aplicación Python, por lo que algunas funcionalidades de Python están restringidas o no pueden utilizarse. Para usuarios avanzados de Python o tutoriales de programación, remitimos a la documentación oficial en línea de Python 3.13.

Comentarios

Las siguientes líneas de código representan comentarios que no se ejecutan durante el tiempo de ejecución.

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

Definiciones de variables

Python admite muchos tipos de variables diferentes y no diferencia entre la definición y la declaración de variables.

```
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"
```

Conversión de variables

Las variables pueden convertirse fácilmente de un tipo a otro si los valores son 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
```

Estructuras de datos

Python admite cuatro tipos diferentes de estructuras de datos:

- Una lista contiene un conjunto ordenado de elementos.

```
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
```

- Una tupla contiene un conjunto ordenado pero inmutable (no modificable) de elementos.

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

- Un conjunto contiene una colección no ordenada de elementos sin duplicados.

```
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 diccionario contiene una combinación no ordenada de claves y valores.

```
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
```

Bibliotecas e importaciones

Python proporciona un conjunto de módulos integrados (la biblioteca estándar). Los siguientes módulos son compatibles:

- Operaciones numéricas y aleatoriedad – útil para cálculos, funciones matemáticas y generación de valores aleatorios

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

- Gestión de tiempo y fecha – trabajo con marcas de tiempo, retardos y valores de fecha y hora

```
import time, datetime
```

- Almacenamiento de datos y formatos de archivo – lectura y escritura de datos estructurados como formatos JSON o CSV

```
import json, csv
```

- Procesamiento de texto – manipulación de cadenas y búsqueda de patrones mediante expresiones regulares

```
import re, string
```

- Herramientas avanzadas de iteración y funciones – bucles eficientes, combinaciones y funciones de orden superior

```
import itertools, functools
```

Además de la biblioteca estándar, están disponibles las siguientes bibliotecas de terceros:

- Cálculo numérico – manejo eficiente de matrices y datos numéricos

```
import numpy
```

- Matemáticas de orientación 3D – representación de cuaterniones y transformaciones (relacionado con robótica)

```
import pyquaternion
```

- Comunicación HTTP – envío de solicitudes a servicios web y APIs REST

```
import requests
```

- Configuración y serialización de datos – trabajo con datos en formato YAML

```
import yaml
```

Los siguientes módulos estándar de Python no están disponibles en aplicaciones de robot:

- Acceso al sistema operativo (p. ej. `os`, `shutil`, `subprocess`, `sys`)
- Paralelización personalizada (p. ej. `multiprocessing`, `threading`)
- Inspección y depuración personalizadas (p. ej. `inspect`, `traceback`)

Además, no se admite la instalación de bibliotecas de terceros personalizadas en el entorno Python del robot.

Funciones básicas de Python

Argumentos obligatorios y opcionales de funciones

Consideremos que una función ha sido definida con los siguientes argumentos:

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

Los argumentos de función arg1 y arg2 son argumentos estándar que deben proporcionarse al llamar a esta función. Los argumentos arg3 y arg4 tienen un valor predeterminado y, por lo tanto, no es obligatorio proporcionarlos al llamar a la función. Todas las siguientes llamadas a funciones son válidas:

```
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")
```

Argumentos de función desempaquetados

Algunas funciones mencionadas en la documentación del código utilizan argumentos desempaquetados. Estos argumentos son útiles cuando la cantidad de argumentos de una función puede variar. Consideremos los siguientes argumentos de función:

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

Entonces, estas funciones pueden utilizarse de la siguiente manera:

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

Operadores básicos de Python

Operadores de cálculo

```
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
```

Operadores de comparación

Aquí x e y pueden ser de cualquier tipo.

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

Aquí x e y son números.

```
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
```

Operadores lógicos

Aquí x e y son valores booleanos.

```
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
```

Operadores bit a bit

Aquí x e y son enteros.

```
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
```

Programación condicional

Condición If–Else

Las condiciones `if`, `elif` (else if) y `else` se pueden combinar según las siguientes reglas:

- El bloque `if` se ejecuta únicamente si su condición devuelve el valor True.
- Un bloque `elif` (else if) es opcional y se ejecuta únicamente si todas las condiciones anteriores `if` o `elif` devuelven False y su propia condición devuelve True.
- Un bloque `else` es opcional y se ejecuta únicamente si todas las condiciones previas `if` o `elif` devolvieron False.

El código general if-elif-else se ve de la siguiente manera:

```
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
    ...
```

Ejemplos:

Verificar si una variable `my_integer` es igual a 5, 10 o a ninguno de esos números.

```
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.")
```

Bucle While

Un bucle while se ejecuta y se repite según las siguientes reglas:

- El bloque `while` se ejecuta repetidamente mientras su condición devuelva True.
- La palabra clave `continue` es opcional y puede usarse para saltar a la siguiente iteración del bucle ignorando el código restante dentro del while.
- La palabra clave `break` es opcional y puede usarse para salir del bucle.
- Un bloque `else` es opcional (y raramente usado) y se ejecuta si la condición del bucle se vuelve False. En otras palabras, este bloque no se ejecuta si el bucle while termina con 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
    ...
```

Ejemplos:

Imprimiendo todos los números del 1 al 10.

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

Bucle Infinito: Los bucles infinitos son útiles cuando se desea que el robot repita una tarea hasta detenerla manualmente, o salir del bucle usando una condición o función específica. Si el bucle tiende a ejecutarse demasiado rápido, se recomienda agregar comandos de espera para dar tiempo al procesador.

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

Bucle For

Un bucle for se ejecuta y se repite según las siguientes reglas:

- El bloque `for` se ejecuta repetidamente, una vez por cada elemento en una secuencia iterable (p. ej., lista, set, diccionario o cadena).
- La palabra clave `continue` es opcional y puede usarse para saltar a la siguiente iteración del bucle ignorando el código restante dentro del for.
- La palabra clave `break` es opcional y puede usarse para salir del bucle.
- Un bloque `else` es opcional (y raramente usado) y se ejecuta después de que se ha procesado el último elemento de la secuencia del bucle for. En otras palabras, este bloque no se ejecuta si el bucle for termina con 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'.
    ...
```

Ejemplos:

Imprimiendo todos los números del 1 al 5.

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

Imprimiendo mis frutas favoritas.

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

Contando las letras en `python`.

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

Funciones principales

class Core

Nota

Esta es una colección de todas las funciones principales. Estas funciones están disponibles directamente y pueden ser llamadas en el código de las aplicaciones.

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

Mover una o varias articulaciones del robot a un conjunto específico de ángulos de articulación. Este movimiento sincroniza todas las articulaciones para que alcancen su posición objetivo al mismo tiempo.

Parámetros:

- **joints** (*int, str, list*) –
  Los IDs de los actuadores a mover. Especifique el valor como sigue:
  - el ID de una sola articulación (p. ej., *6*)
  - una lista de IDs de articulaciones para varias articulaciones (p. ej., *[1, 4, 6]*)
  - la constante *ALL_KIN_JOINTS* para mover las seis articulaciones
- **joint_position** (*float, list of float*) –
  Las posiciones angulares de las articulaciones a mover en [°]. Especifique el valor como sigue:
  - un solo número para mover todas las articulaciones especificadas a la misma posición (p. ej., *-30*)
  - una lista de números para mover cada articulación especificada a una posición diferente (p. ej., *[90, -45, 30]*)
- **joint_velocity** (*float or None*) – La velocidad máxima en porcentaje del límite de velocidad de las articulaciones. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *45* %). Si no se especifica ningún valor, se usa la velocidad global de las articulaciones.
- **joint_acceleration** (*float or None*) – La aceleración máxima en porcentaje del límite de aceleración de las articulaciones en movimiento. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *60* %). Si no se especifica ningún valor, se usa la aceleración global de las articulaciones.
- **relative** (*bool*) – Indica si las posiciones de las articulaciones se consideran un desplazamiento relativo respecto a las posiciones actuales. Por defecto, se usan valores de posición absolutos.
- **block** (*bool*) – Indica si se debe esperar a que termine este movimiento antes de continuar con la ejecución del programa. Tenga en cuenta que solo los movimientos sincronizados que usan las mismas articulaciones pueden ejecutarse sucesivamente de forma no bloqueante.

Ejemplos:

Moviendo todas las articulaciones del robot sincronizadas a 0°, que es la posición vertical de pie del robot:

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

Moviendo la articulación 6 -30° desde su posición actual:

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

Moviendo las articulaciones 2, 3 y 5 a 20°, -40° y 20°, respectivamente. El movimiento se realizará de modo que cada articulación termine al mismo tiempo. Este movimiento usará la aceleración máxima permitida para las articulaciones, pero solo el 50% de la velocidad máxima permitida.

```
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*)

Moviendo el TCP (punto central de la herramienta) del robot a un conjunto específico de coordenadas cartesianas. El robot se mueve siguiendo una trayectoria óptima en tiempo, resultando en perfiles de velocidad trapezoidales en cada articulación.

Parámetros:

- **coordinates** (*Coordinates, list or dict*) –
  Las coordenadas objetivo del TCP. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **configuration** (*str or None*) – La configuración de la pose objetivo. Es una cadena de “c1” a “c8”.
- **joint_velocity** (*float or None*) – La velocidad máxima en porcentaje del límite de velocidad de las articulaciones. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *45* %). Si no se especifica ningún valor, se usa la velocidad global de las articulaciones.
- **joint_acceleration** (*float or None*) – La aceleración máxima en porcentaje del límite de aceleración de las articulaciones en movimiento. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *60* %). Si no se especifica ningún valor, se usa la aceleración global de las articulaciones.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

- **relative** (*bool*) – Indica si las coordenadas se consideran un desplazamiento relativo a las coordenadas actuales. Por defecto, se usan valores de coordenadas absolutas.
- **block** (*bool*) – Indica si se debe esperar a que la ejecución del programa continúe hasta que este movimiento haya finalizado. Por defecto, esta función es bloqueante.

Ejemplos:

Las coordenadas objetivo se pueden proporcionar de varias formas. Todos los comandos especificados mueven el TCP a la posición x = 700 mm, y = -125 mm, z = 500 mm y ángulos de orientación roll = 180°, pitch = 0° y yaw = -90° en el marco de trabajo 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 no se especifican todas las coordenadas, se utilizan las coordenadas actuales del TCP para las restantes. Por ejemplo, solo se puede cambiar la componente x y el roll de la siguiente manera:

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

En los ejemplos anteriores, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también se pueden definir explícitamente en la llamada a la función:

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

lo que mueve -100 mm adicionales en la dirección y y -60° en la dirección yaw con el 50% de la velocidad máxima y con la aceleración máxima. Además, el TCP se desplaza 20mm en la dirección x debido al desplazamiento de la herramienta.

Hay múltiples opciones para definir el movimiento en el espacio de la herramienta, una de ellas es:

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

que se mueve a lo largo de la dirección z de la herramienta.

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

Moviendo el TCP (punto central de la herramienta) del robot a una pose específica. El robot se mueve a lo largo de una trayectoria óptima en el tiempo, lo que produce perfiles de velocidad trapezoidales en cada articulación.

Parámetros:

- **pose** (*Pose, str, or int*) –
  La pose objetivo del TCP. Especifique el valor de la siguiente manera:
  - un objeto Pose
  - el nombre de una pose predefinida (guardada en la base de datos)
  - el ID de una pose predefinida (guardada en la base de datos)
- **joint_velocity** (*float or None*) – La velocidad máxima en porcentaje del límite de velocidad de las articulaciones. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *45* %). Si no se especifica ningún valor, se usa la velocidad global de las articulaciones.
- **joint_acceleration** (*float or None*) – La aceleración máxima en porcentaje del límite de aceleración de las articulaciones en movimiento. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *60* %). Si no se especifica ningún valor, se usa la aceleración global de las articulaciones.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates

- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.
- **block** (*bool*) – Indica si se debe esperar a que la ejecución del programa continúe hasta que este movimiento haya finalizado. Por defecto, esta función es bloqueante.

Ejemplos:

Moviéndose a una pose de la base de datos:

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

También se puede especificar una pose personalizada en el script utilizando la función `create_pose()`:

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

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

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

lo que mueve el robot a la «final_pose» de la base de datos con el 45% de la velocidad máxima de las articulaciones y el 60% de la aceleración máxima de las articulaciones. Además, se aplica un desplazamiento al marco de trabajo, desplazando la pose objetivo 50 mm en la dirección z.

Hay múltiples opciones para definir el movimiento en el espacio de la herramienta, una de ellas es:

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

que se mueve a lo largo de la dirección z de la herramienta.

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*)

Moviendo el TCP (punto central de la herramienta) del robot desde su pose actual hasta una pose objetivo en línea recta. Tenga en cuenta que tanto la posición como la orientación se interpolan linealmente a lo largo de la trayectoria. Este movimiento solo puede ejecutarse si cada pose a lo largo de la trayectoria es alcanzable.

Parámetros:

- **coordinates** (*Coordinates, list, or dict*) –
  Las coordenadas objetivo del TCP. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **linear_velocity** (*float or None*) – La velocidad lineal máxima de la herramienta [mm/s] para todas las partes móviles del robot.
- **linear_acceleration** (*float or None*) – La aceleración lineal máxima de la herramienta [mm/s^2] para todas las partes móviles del robot.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates

- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.
- **relative** (*bool*) – Indica si las coordenadas se consideran un desplazamiento relativo a las coordenadas actuales. Por defecto, se usan valores de coordenadas absolutas.
- **block** (*bool*) – Indica si se debe esperar a que la ejecución del programa continúe hasta que este movimiento haya finalizado. Por defecto, esta función es bloqueante.

Ejemplos:

Las coordenadas objetivo pueden proporcionarse en varias formas. Todos los comandos especificados mueven el TCP linealmente a la posición x = 600 mm, y = -125 mm, z = 500 mm y ángulos de orientación roll = 180°, pitch = 0° y yaw = -90° en el marco de trabajo 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 no se especifican todas las coordenadas, se utilizan las coordenadas TCP actuales para las coordenadas restantes. Por ejemplo, se pueden cambiar los componentes x y roll de la siguiente manera:

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

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

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

lo que mueve 100 mm adicionales en la dirección x con una velocidad de 500 mm/s y una aceleración de 1000 mm/s². Además, el marco de trabajo se sobrescribe con las coordenadas dadas.

Hay múltiples opciones para definir el movimiento en el espacio de la herramienta, una de ellas es:

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

lo que se mueve linealmente a lo largo de la dirección z de la herramienta.

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*)

Moviendo el TCP (punto central de la herramienta) del robot desde su pose actual hasta una pose objetivo a lo largo de una trayectoria circular, pasando por una posición intermedia. Tenga en cuenta que la orientación se interpola linealmente a lo largo de la trayectoria. Este movimiento solo puede ejecutarse si cada pose a lo largo de la trayectoria es alcanzable.

Parámetros:

- **intermediate_position** (*list*) – Una posición intermedia en el camino hacia las coordenadas objetivo. La posición intermedia determina la curvatura del movimiento circular.
- **coordinates** (*Coordinates, list, or dict*) –
  Las coordenadas objetivo del TCP. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **linear_velocity** (*float or None*) – La velocidad lineal máxima de la herramienta [mm/s] para todas las partes móviles del robot.
- **linear_acceleration** (*float*) – La aceleración lineal máxima de la herramienta [mm/s^2] para todas las partes móviles del robot.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –

La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:

- un objeto Coordinates
- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.
- **relative** (*bool*) – Indica si las coordenadas se consideran un desplazamiento relativo a las coordenadas actuales. Por defecto, se usan valores de coordenadas absolutas.
- **block** (*bool*) – Indica si se debe esperar a que la ejecución del programa continúe hasta que este movimiento haya finalizado. Por defecto, esta función es bloqueante.

Ejemplos:

El robot puede moverse circularmente a la posición objetivo x = 0 mm, y = -500 mm, z = 500 mm y ángulos de orientación roll = 180°, pitch = 0° y yaw = -90° en el marco de trabajo global. Durante su movimiento pasará por la posición intermedia 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 no se especifican todas las coordenadas, se utilizan las coordenadas TCP actuales para las coordenadas 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})
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
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)
```

lo que mueve el TCP a las coordenadas especificadas con una velocidad de 500 mm/s y una aceleración de 1000 mm/s^2. Además, el TCP se desplaza 20 mm en la dirección x mediante el desplazamiento de la herramienta.

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*)

Moviendo el TCP (punto central de la herramienta) de manera que solo cambie la orientación. Luego, el robot rota linealmente según la orientación deseada de la herramienta mientras mantiene su posición constante. Este movimiento solo puede ejecutarse si cada pose a lo largo de la trayectoria es alcanzable.

Parámetros:

- **tool_orientation** (*list*) – Orientación deseada de la herramienta como ángulos náuticos (roll, pitch, yaw) [°]
- **angular_velocity** (*float or None*) – Velocidad angular máxima de la herramienta [°/s] del TCP.
- **angular_acceleration** (*float or None*) – Aceleración angular máxima de la herramienta [°/s²] del TCP.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
    - un objeto Coordinates
    - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
    - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.
- **relative** (*bool*) – Indica si las coordenadas se consideran un desplazamiento relativo a las coordenadas actuales. Por defecto, se usan valores de coordenadas absolutas.
- **block** (*bool*) – Indica si se debe esperar a que la ejecución del programa continúe hasta que este movimiento haya finalizado. Por defecto, esta función es bloqueante.

Ejemplos:

La orientación objetivo puede especificar todos los ángulos o solo algunos, dejando “None” para los demás. En este caso, se utiliza la orientación actual para los ángulos faltantes.

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

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

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

Lo que corresponde a un movimiento relativo de -60° en la dirección de roll con una velocidad de 250°/s y una aceleración de 3000°/s². Además, el marco de la herramienta se sobrescribe con las coordenadas dadas.

Hay múltiples opciones para definir el movimiento en el espacio de la herramienta, una de ellas es:

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

Que rota alrededor de la dirección z de la herramienta.

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*)

Mueve el TCP (punto central de la herramienta) a través de los puntos intermedios definidos, comenzando desde la pose actual del robot.

Parámetros:

- **poses** (*list of str, int or Pose*) – Una lista de puntos intermedios.
- **joint_velocity** (*float or None*) – La velocidad máxima en porcentaje del límite de velocidad de las articulaciones. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *45* %). Si no se especifica ningún valor, se usa la velocidad global de las articulaciones.
- **joint_acceleration** (*float or None*) – La aceleración máxima en porcentaje del límite de aceleración de las articulaciones en movimiento. Este es un único número para limitar todas las articulaciones al mismo valor máximo (p. ej. *60* %). Si no se especifica ningún valor, se usa la aceleración global de las articulaciones.
- **blend_radius** (*float or None*) – Radio de mezcla permitido [mm] que define la desviación máxima de los puntos intermedios en el espacio de la herramienta durante la transición entre puntos.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]

- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.
- **block** (*bool*) – Indica si se debe esperar a que termine este movimiento antes de continuar con la ejecución del programa. Tenga en cuenta que solo los movimientos sincronizados que usan las mismas articulaciones pueden ejecutarse sucesivamente de forma no bloqueante.

Ejemplos:

Los puntos intermedios se pueden dar en diferentes formas (`Pose` objeto, nombre de la pose en la base de datos, ID de la pose en la base de datos):

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

En los ejemplos anteriores, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también se pueden definir explícitamente en la llamada a la función:

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

Que se mueve a lo largo de los puntos intermedios permitiendo una desviación máxima de 50 mm. Utiliza el 50% de la velocidad máxima y la aceleración máxima.

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

Mueve el TCP (punto central de la herramienta) según los segmentos definidos. Comienza en la pose actual del robot.

Parámetros:

- **segments** (*list*) – Una lista de segmentos que componen la trayectoria.
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.
- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

- **block** (*bool*) – Indica si se debe esperar a que termine este movimiento antes de continuar con la ejecución del programa. Tenga en cuenta que solo los movimientos sincronizados que usan las mismas articulaciones pueden ejecutarse sucesivamente de forma no bloqueante.

Ejemplos:

Los segmentos se pueden definir en el script utilizando objetos 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])
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
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)
```

Que ejecuta los segmentos usando un ligero desplazamiento del TCP definido por el offset de la herramienta. Además, el movimiento se establece como no bloqueante.

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

Mueve el TCP (punto central de la herramienta) según la trayectoria especificada. Primero, el robot se mueve a la pose inicial de la trayectoria utilizando las velocidades y aceleraciones por defecto. Desde allí, los segmentos se ejecutan uno tras otro según su especificación.

Parámetros:

- **path** (*Path, str, or int*) – La trayectoria que debe seguir el TCP.
- **previous_angles** (*list or None*) –
  Los ángulos para los que se calculará la solución más cercana para la pose inicial. Especifique el valor de la siguiente manera:
  - None, para usar los ángulos actuales del robot
  - Una lista de ángulos articulares [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.
- **work_frame** (*Coordinates, list, dict, or None*) –

La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:

- un objeto Coordinates
- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

- **block** (*bool*) – Indica si se debe esperar a que termine este movimiento antes de continuar con la ejecución del programa. Tenga en cuenta que solo los movimientos sincronizados que usan las mismas articulaciones pueden ejecutarse sucesivamente de forma no bloqueante.

Ejemplos:

Ejecutar una trayectoria desde la base de datos:

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

También se puede especificar una trayectoria personalizada en el script usando la función `create_path()`:

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

Otra opción es usar el objeto `Path` para crear una trayectoria y luego ejecutarla:

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

run_path(path)
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

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

Que ejecuta la «fancy_path» usando un ligero desplazamiento del TCP definido por el marco de la herramienta. Además, el movimiento se establece como no bloqueante.

wait_for_motor(*actuator_ids=None*)

Esperando a que uno o varios motores de articulación dejen de moverse.

Parámetros:

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

Los IDs de los actuadores a esperar. Especifique el valor de la siguiente manera:

- el ID de una sola articulación (p. ej., *6*)
- una lista de IDs de articulaciones para varias articulaciones (p. ej., *[1, 4, 6]*)
- Un conjunto de nombres de articulaciones para múltiples articulaciones (p. ej., *{«1», «4», «6»}*)
- la constante *ALL_KIN_JOINTS* o *None* para leer de todas las articulaciones

Ejemplos:

Este ejemplo muestra que el tiempo mientras un robot se mueve se puede utilizar para fines personalizados.

```
# 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()

Deteniendo el movimiento enviando una señal de parada a todos los actuadores conectados.

Ejemplos:

Este ejemplo muestra que los movimientos no bloqueantes se pueden abortar utilizando la función “stop_motion”. Aquí queremos abortar el movimiento si un input digital específico cambia.

```
# 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*)

Comprobar si uno o varios motores están en movimiento

Parámetros:

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

Los IDs de los actuadores a comprobar. Especifique el valor de la siguiente manera:

- el ID de una sola articulación (p. ej., *6*)
- una lista de IDs de articulaciones para varias articulaciones (p. ej., *[1, 4, 6]*)
- un conjunto de nombres de articulaciones para múltiples articulaciones (p. ej., *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* o *None* para leer de todas las articulaciones

Devuelve:

Indica si los motores especificados se están moviendo. Puede ser uno de los siguientes:

- un booleano, si se comprobó un solo motor
- una lista de booleanos, si se comprobaron varios motores

Tipo del valor devuelto:

bool or list of bool

Ejemplos:

Este ejemplo muestra que los movimientos no bloqueantes se pueden abortar utilizando la función “stop_motion”. Aquí queremos abortar el movimiento si un input digital específico cambia.

```
# 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*)

Establece un nuevo valor predeterminado global para el argumento `joint_velocity` utilizado en las funciones de movimiento punto a punto. El valor de la velocidad de la articulación se da en porcentaje del límite máximo de velocidad de la articulación.

Parámetros:

joint_velocity (*float*) – La velocidad global de la articulación en [%] del límite máximo de velocidad de la articulación.

Ejemplos:

Llamando a dos movimientos consecutivos punto a punto usando los mismos límites de velocidad.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `joint_acceleration` utilizado en las funciones de movimiento punto a punto. El valor de aceleración de las articulaciones se da como porcentaje del límite máximo de aceleración de las articulaciones.

Parámetros:

joint_acceleration (*float*) – La aceleración global de las articulaciones en [%] del límite máximo de aceleración de las articulaciones.

Ejemplos:

Llamando a dos movimientos consecutivos punto a punto usando los mismos límites de aceleración.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `linear_velocity` utilizado en las funciones de movimiento de herramientas y planificación de trayectoria. El valor de velocidad lineal limita la velocidad lineal máxima de la herramienta y de todas las partes móviles del robot.

Parámetros:

linear_velocity (*float*) – La velocidad lineal global en [mm/s].

Ejemplos:

Llamando a dos movimientos lineales consecutivos usando los mismos límites de velocidad.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `linear_acceleration` utilizado en las funciones de movimiento de herramientas y planificación de trayectoria. El valor de aceleración lineal limita la aceleración lineal máxima de la herramienta y de todas las partes móviles del robot.

Parámetros:

linear_acceleration (*float*) – La aceleración lineal global en [mm/s^2].

Ejemplos:

Llamando a dos movimientos lineales consecutivos usando los mismos límites de aceleración.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `angular_velocity` utilizado en las funciones de movimiento rotacional. El valor de velocidad angular limita la velocidad de rotación de la herramienta.

Parámetros:

angular_velocity (*float*) – La velocidad angular global en [°/s].

Ejemplos:

Llamando a dos movimientos rotacionales consecutivos usando los mismos límites de velocidad.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `angular_acceleration` utilizado en las funciones de movimiento rotacional. El valor de aceleración angular limita la aceleración rotacional de la herramienta.

Parámetros:

angular_acceleration (*float*) – La aceleración angular global en [°/s^2].

Ejemplos:

Llamando a dos movimientos rotacionales consecutivos usando los mismos límites de aceleración.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `blend_radius` utilizado en funciones de movimiento y planificación de trayectorias.

Parámetros:

blend_radius (*float*) – El radio de fusión global en [mm].

Ejemplos:

Llamando a dos movimientos consecutivos de puntos de trayectoria usando el mismo radio de fusión.

```
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*)

Estableciendo un nuevo valor predeterminado global para el argumento `work_frame` utilizado en funciones de movimiento y cinemática. El marco de trabajo es la transformación que altera el marco de referencia del robot, dado con respecto al marco base.

Parámetros:

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

El marco de trabajo global, dado en relación con el marco base. Especifique el valor de la siguiente manera:

- un objeto Coordinates
- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Ejemplos:

Ejecutando dos movimientos consecutivos usando el mismo marco de trabajo.

```
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*)

Estableciendo un nuevo valor predeterminado global para el `tool_frame` que se utiliza en funciones de movimiento y cinemática. El marco de la herramienta es la transformación para alterar el TCP del robot, dado en relación con el marco del brida.

Parámetros:

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

El marco de herramienta global, dado en relación con el marco de la brida. Especifique el valor de la siguiente manera:

- un objeto Coordinates
- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Ejemplos:

Ejecutando dos movimientos consecutivos usando el mismo marco de herramienta.

```
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*)

Lectura de la pose actual del TCP (punto central de la herramienta) del robot. Si no se proporcionan marcos, la pose se lee con respecto a los marcos globales.

Parámetros:

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

La pose actual del TCP del robot.

Tipo del valor devuelto:

Pose

Ejemplos:

Lectura de las propiedades de una pose de dos maneras diferentes.

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

Tenga en cuenta que la pose actual se lee a partir de los datos del sensor. Los sensores pueden fluctuar y no siempre devolver valores precisos, por lo que el siguiente ejemplo de moverse hacia adelante y hacia atrás con respecto a la pose actual puede provocar un desplazamiento inesperado con el tiempo. En este caso, es mejor usar la función *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*)

Lectura de las coordenadas actuales del TCP (punto central de la herramienta) del robot con respecto a los marcos de coordenadas dados. Si no se especifican marcos, devuelve las coordenadas con respecto a los marcos globales.

Parámetros:

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

Las coordenadas actuales del TCP del robot.

Tipo del valor devuelto:

Coordinates

Ejemplos:

Lectura de las coordenadas actuales del TCP en los marcos de coordenadas globales.

```
coordinates = get_current_coordinates()
```

Tenga en cuenta que las coordenadas actuales se leen a partir de datos de sensores. Los sensores pueden fluctuar y no siempre devolver valores precisos, por lo que el siguiente ejemplo de mover hacia adelante y atrás en relación con la pose actual provocará una deriva inesperada con el tiempo. En este caso, es mejor usar la función *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()

Lectura del nombre de la configuración de la pose actual del robot. Tenga en cuenta que esta configuración es independiente de los marcos globales actuales.

Devuelve:

La configuración actual (que es una de “c1” a “c8”).

Tipo del valor devuelto:

str

Ejemplos:

Lectura de la configuración de la pose actual del robot de dos maneras diferentes.

```
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*)

Obtención de la pose TCP (punto central de la herramienta) de referencia del robot. Si no se proporcionan marcos, la pose se devuelve con respecto a los marcos globales. Tenga en cuenta que esta pose se calcula a partir de los ángulos de referencia enviados a los actuadores.

Parámetros:

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]

- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

La pose TCP de referencia del robot.

Tipo del valor devuelto:

Pose

Ejemplos:

Lectura de las propiedades de una pose de referencia de dos maneras diferentes.

```
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*)

Obtención de las coordenadas de referencia del TCP (punto central de la herramienta) del robot con respecto a los marcos de coordenadas dados. Si no se especifican marcos, se devuelven las coordenadas con respecto a los marcos globales. Tenga en cuenta que estas coordenadas se calculan a partir de los ángulos de referencia enviados a los actuadores.

Parámetros:

- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.

- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.

- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]

- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

Las coordenadas TCP de referencia del robot.

Tipo del valor devuelto:

Coordinates

Ejemplos:

Lectura de las coordenadas TCP de referencia en los marcos de coordenadas globales.

```
coordinates = get_reference_coordinates()
```

get_reference_configuration()

Obtención del nombre de la configuración de la pose de referencia del robot. Tenga en cuenta que esta configuración es independiente de los marcos globales de referencia y que se calcula a partir de los ángulos de referencia enviados a los actuadores.

Devuelve:

La configuración de referencia (que es una de “c1” a “c8”).

Tipo del valor devuelto:

str

Ejemplos:

Lectura de la configuración de la pose de referencia del robot de dos maneras diferentes.

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

create_pose(*coordinates*, *configuration*)

Crea un objeto Pose basado en las coordenadas y la configuración proporcionadas.

Parámetros:

- **coordinates** (*Coordinates, list, or dict*) –
  Las coordenadas TCP de la pose. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **configuration** (*str*) – La configuración de la pose. Es una de “c1” a “c8”.

Devuelve:

El objeto de pose del robot.

Tipo del valor devuelto:

Pose

Ejemplos:

Creación de una pose a partir de una lista de valores de coordenadas, en una configuración específica.

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

get_pose(*pose*)

Lectura del objeto de pose desde una pose predefinida guardada en la base de datos.

Parámetros:

pose (*Pose, str, or int*) –

La pose del robot de la que se leen las coordenadas. Puede ser una de las siguientes:

- un objeto Pose
- el nombre (str) de una pose guardada en la base de datos.
- el ID (int) de una pose guardada en la base de datos

Devuelve:

El objeto de pose.

Tipo del valor devuelto:

Pose

Ejemplos:

Cargando una pose desde la base de datos con el nombre «grasping_pose».

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

create_path(*initial_pose*, *segments*)

Crea un objeto Path basado en la pose inicial y los segmentos proporcionados.

Parámetros:

- **initial_pose** (*Pose*) – La pose inicial del robot.
- **segments** (*list*) – Una lista de segmentos de los que consta la trayectoria. Tenga en cuenta que los segmentos también pueden añadirse posteriormente.

Devuelve:

El objeto de trayectoria del robot.

Tipo del valor devuelto:

Path

Ejemplos:

Creando una trayectoria a partir de una pose inicial y múltiples segmentos.

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

get_path(*path*)

Leyendo el objeto de trayectoria desde una trayectoria predefinida guardada en la base de datos.

Parámetros:

path (*Path, str, or int*) –

La trayectoria del robot a obtener. Puede ser una de las siguientes:

- un objeto Path
- el nombre (str) de una trayectoria guardada en la base de datos
- el ID (int) de una trayectoria guardada en la base de datos

Devuelve:

El objeto de trayectoria.

Tipo del valor devuelto:

Path

Ejemplos:

Cargando una trayectoria desde la base de datos con el nombre «grasping_path».

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

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

Crear un objeto LinearSegment.

Parámetros:

- **coordinates** (*list, dict, or Coordinates*) – Coordenadas objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Crear un objeto LinearSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

```
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*)

Crear un objeto CircularSegment.

Parámetros:

- **intermediate_position** (*list*) – Una posición intermedia [x, y, z] en el camino hacia las coordenadas objetivo. La posición intermedia determina la curvatura del movimiento circular.
- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto CircularSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

```
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*)

Crear un objeto OrientationSegment.

Parámetros:

- **orientation** (*list*) – Orientación objetivo. Tenga en cuenta que la posición permanece constante.
- **angular_velocity** (*float or None*) – Velocidad angular deseada [°/s]. Si no se define velocidad angular, se utilizará el valor de velocidad angular global.
- **angular_acceleration** (*float or None*) – Aceleración angular deseada [°/s^2]. Si no se define aceleración angular, se utilizará el valor de aceleración angular global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Un radio de mezcla mayor a 0 permite que el punto central de la herramienta (TCP) rote antes de alcanzar la posición de este segmento. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto OrientationSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

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

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

Crear un objeto CoordinatesSegment.

Parámetros:

- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto CoordinatesSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

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

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

Crear un objeto PoseSegment.

Parámetros:

- **pose** (*Pose*) – Pose objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto PoseSegment definiendo todos los parámetros:

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

o usando los valores predeterminados y la pose de la base de datos:

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

get_coordinates(*pose*)

Leyendo las coordenadas de la pose especificada. Tenga en cuenta que estas coordenadas son independientes de los marcos globales actuales.

Parámetros:

pose (*Pose, str, or int*) –

La pose del robot de la que se leen las coordenadas. Puede ser una de las siguientes:

- un objeto Pose
- el nombre (str) de una pose guardada en la base de datos.
- el ID (int) de una pose guardada en la base de datos

Devuelve:

Las coordenadas de esta pose.

Tipo del valor devuelto:

Coordinates

Ejemplos:

Leyendo las coordenadas de la pose previamente guardada en la base de datos con el nombre «grasping_pose».

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

Leyendo las coordenadas de la pose previamente guardada en la base de datos con el ID 3.

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

get_configuration(*pose*)

Leyendo la configuración de una pose específica.

Parámetros:

pose (*Pose, str, or int*) –

La pose de la que se desea leer la configuración. Especifique el valor de la siguiente manera:

- un objeto Pose
- el nombre de una pose predefinida (guardada en la base de datos)
- el ID de una pose predefinida (guardada en la base de datos)

Devuelve:

La configuración de esta pose (que es una de “c1” a “c8”).

Tipo del valor devuelto:

str

Ejemplos:

Leyendo la configuración de una pose guardada en la base de datos con el nombre «grasping_pose».

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

Leyendo la configuración desde un objeto Pose de dos maneras diferentes.

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

parse_coordinates(*coordinates*)

Crear un objeto Coordinates.

Parámetros:

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

Un conjunto específico de coordenadas, que puede ser uno de los siguientes:

- una lista en la forma [x, y, z, roll, pitch, yaw] con valores de posición en [mm] y valores de orientación en [grados]
- un diccionario en la forma {“x”: x, “y”: y, “z”: z, “roll”: roll, “pitch”: pitch, “yaw”: yaw} con valores de posición en [mm] y valores de orientación en [grados]
- un objeto Coordinates (copiando otro objeto Coordinates)

Constructor:

Crear un objeto Coordinates a partir de valores de coordenadas.

```
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*)

Transformando las coordenadas del robot entre diferentes marcos utilizando la diferencia entre los marcos relativos de herramienta y de trabajo. Si no se especifica ni el marco de herramienta relativo ni el marco de trabajo relativo, las coordenadas no se transforman y se devuelve el objeto de coordenadas sin cambios.

Parámetros:

- **coordinates** (*Coordinates, list, or dict*) –
  Las coordenadas TCP a transformar. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **relative_tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **relative_work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

Las coordenadas del robot transformadas.

Tipo del valor devuelto:

Coordinates

Ejemplos:

Transformación de un conjunto de coordenadas a otro marco de trabajo. Si el origen del marco de trabajo coincide exactamente con las coordenadas a transformar, la transformación devuelve un objeto de coordenadas con todos los valores en cero.

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

get_grid_points(*grid_name*)

Lectura de las posiciones de todos los puntos de una cuadrícula específica previamente guardada en la base de datos.

Parámetros:

grid_name (*str*) – El nombre de la cuadrícula de la que se extraerán los puntos.

Devuelve:

La lista de puntos de la cuadrícula, en el orden de iteración definido en las propiedades de la cuadrícula.

Tipo del valor devuelto:

list

Ejemplos:

Movimiento a todos los puntos de una cuadrícula predefinida.

```
# 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 *cinemática directa* es un concepto de la cinemática robótica que calcula la pose del TCP (punto central de la herramienta) a partir de un conjunto dado de ángulos articulares. La función inversa es la *cinemática inversa*, que calcula los ángulos articulares a partir de la pose TCP del robot. Los cálculos cinemáticos normalmente se realizan entre los marcos base y de brida integrados del robot. No obstante, el usuario puede modificar los marcos de referencia definiendo los marcos de trabajo y de herramienta. De forma predeterminada, se utilizan los marcos de trabajo y de herramienta definidos globalmente.

Parámetros:

- **joint_angles** (*list of float*) – La lista de los seis ángulos articulares en [deg].
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.
- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates

- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

El objeto de pose del robot.

Tipo del valor devuelto:

Pose

Ejemplos:

En una singularidad, varios conjuntos de ángulos articulares conducen a la misma pose TCP. En este ejemplo, ambos conjuntos de ángulos dan como resultado exactamente la misma pose.

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

Lectura de la pose TCP actual utilizando dos enfoques diferentes.

```
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 *cinemática inversa* es un concepto de la cinemática robótica que calcula un conjunto de ángulos articulares a partir de la pose del TCP (punto central de la herramienta). La función inversa es la *cinemática directa*, que calcula la pose TCP a partir de los ángulos articulares del robot. Tenga en cuenta que la *cinemática inversa* suele tener múltiples soluciones, pero solo devuelve una de ellas.

Parámetros:

- **pose** (*Pose, str, or int*) –
  La pose para la cual se deben calcular los ángulos articulares correspondientes. Especifique el valor de la siguiente manera:
  - un objeto Pose
  - el nombre de una pose predefinida (guardada en la base de datos)
  - el ID de una pose predefinida (guardada en la base de datos)
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

La lista de los seis ángulos articulares en [deg].

Tipo del valor devuelto:

list of float

Ejemplos:

Cálculo de los ángulos articulares de una pose previamente guardada en la base de datos.

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

Calcular los ángulos articulares que alcanzan un conjunto específico de coordenadas en una configuración específica.

```
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*)

Esta función calcula la solución de la *cinemática inversa* (consulte la función `inverse_kinematics` para más detalles) que está más próxima a la pose actual del robot o más próxima a una pose específica.

Parámetros:

- **coordinates** (*Coordinates, list, or dict*) –
  Las coordenadas TCP para las cuales se deben calcular los ángulos articulares correspondientes. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **previous_angles** (*list or None*) –
  Los ángulos para los que se calculará la solución más cercana. Especifique el valor de la siguiente manera:
  - None, para usar los ángulos actuales del robot
  - Una lista de ángulos articulares [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates

- una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
- un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

La lista de los seis ángulos articulares en [deg].

Tipo del valor devuelto:

list of float

Ejemplos:

La cinemática inversa se puede usar para verificar si un conjunto específico de coordenadas devuelve una solución cinemática.

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

Para obtener múltiples soluciones para la misma pose, la declaración de la pose anterior puede modificarse en consecuencia. En este ejemplo, se calculan dos soluciones para un robot de pie, ligeramente desplazado hacia adelante y hacia abajo.

```
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*)

Esta función calcula la solución de la *cinemática inversa* (consulte la función `inverse_kinematics` para más detalles) que está más próxima a la pose actual del robot o más próxima a una pose específica.

Parámetros:

- **coordinates** (*Coordinates, list, or dict*) –
  Las coordenadas TCP para las cuales se deben calcular los ángulos articulares correspondientes. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}
- **tool_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar el TCP del robot, dada en relación con el marco de la herramienta. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento de la herramienta, de modo que el TCP se encuentra en el origen del marco de la herramienta.
- **tool_frame** (*Coordinates, list, dict, or None*) –
  La transformación para alterar el TCP del robot, dada con respecto al marco de brida. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de herramienta global. Si el marco de herramienta contiene solo coordenadas cero, coincide con el marco de brida.
- **work_offset** (*Coordinates, list, dict, or None*) –
  La transformación para desplazar la referencia del robot, dada con respecto al marco de trabajo. Especifica el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, no hay desplazamiento del trabajo, de modo que la referencia se encuentra en el origen del marco de trabajo.

- **work_frame** (*Coordinates, list, dict, or None*) –
  La transformación para modificar la referencia del robot, dada relativa al marco base. Especifique el valor de la siguiente manera:
  - un objeto Coordinates
  - una lista de coordenadas cartesianas en el formato [x, y, z, roll, pitch, yaw]
  - un diccionario de coordenadas cartesianas en el formato {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}

  Por defecto, se aplica el marco de trabajo global. Si el marco de trabajo contiene solo coordenadas cero, coincide con el marco base.

Devuelve:

Las listas de seis ángulos de articulación en [grados] y sus configuraciones en el formato {"c1": [...], "c2": [...], ...}

Tipo del valor devuelto:

dict

Ejemplos:

En este ejemplo, el robot se mueve a todos los conjuntos de ángulos de articulación que alcanzan las mismas coordenadas 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 conocemos un conjunto de ángulos de articulación que alcanza una pose deseada, de esta manera podemos calcular todas las demás soluciones que alcanzan la misma pose pero con diferentes configuraciones de articulación.

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

wait(*seconds*)

Esperando y bloqueando activamente la ejecución del código durante un tiempo definido.

Parámetros:

seconds (*float*) – La duración de la espera en [s].

Ejemplos:

Este ejemplo mueve el robot a poses predefinidas y espera 3 [s] entre los movimientos.

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

Este ejemplo repite parte de su código indefinidamente. Para asegurar que la velocidad de ejecución esté limitada, el código espera 100 [ms] al final de cada iteración.

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

Este ejemplo es similar al anterior, pero en lugar de esperar un tiempo constante, el tiempo de espera se elige dinámicamente para poder ejecutar su código en un intervalo constante cada segundo.

```
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*)

Ejecutando una aplicación (bloqueante) desde dentro de la aplicación principal.

Parámetros:

- **script_name** (*str*) – El nombre de la aplicación a ejecutar. Si la aplicación está en una carpeta, use la ruta completa a la aplicación (p. ej., *folder/subfolder/app*)
- **arguments** (*dict*) – Los argumentos pasados a la aplicación para ejecutar. Puede leerlos en la aplicación llamada usando la variable incorporada *script_arguments*.

Ejemplos:

Supongamos que tenemos una aplicación separada que mide la temperatura del entorno. Este ejemplo llama a esta aplicación repetidamente, una vez cada minuto.

```
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*)

Ejecutando una aplicación (no bloqueante) en paralelo con la aplicación principal. Tenga en cuenta que la aplicación principal continúa de inmediato y que todas las aplicaciones en paralelo se abortan una vez que la aplicación principal termina.

Parámetros:

script_name (*str*) – El nombre de la aplicación paralela a iniciar. Si la aplicación está en una carpeta, use la ruta completa a la aplicación (p. ej., *folder/subfolder/app*)

Ejemplos:

Este ejemplo activa una aplicación paralela con el propósito de monitorear un sensor de temperatura para asegurar que el robot solo opere en un rango de temperatura específico. La aplicación principal se ejecuta continuamente y se aborta si el criterio de temperatura operativa ya no se cumple.

```
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()

Deteniendo las aplicaciones que se están ejecutando actualmente. Esto incluye la aplicación principal y cualquier aplicación que haya sido activada por la aplicación principal. Tenga en cuenta que una aplicación en *background* en ejecución continúa funcionando.

Muestra:

ScriptInterrupt – Se lanza en la aplicación principal si se ejecuta esta función desde una aplicación activada mediante la función *run_script*.

Ejemplos:

Este ejemplo de código se puede usar en una aplicación *background* para pausar temporalmente y reanudar manualmente las aplicaciones principales en ejecución.

```
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()

Pausando las aplicaciones actualmente en ejecución. Esto incluye la aplicación principal y cualquier aplicación activada por la aplicación principal. Tenga en cuenta que una aplicación *background* en ejecución sigue funcionando.

Ejemplos:

Consulte el ejemplo combinado de *stop_application*, que utiliza esta función.

resume_application()

Reanudando las aplicaciones que están en pausa. Dado que esta función solo se puede ejecutar en estado *paused*, solo puede usarse desde una aplicación *background*.

Ejemplos:

Consulte el ejemplo combinado de *stop_application*, que utiliza esta función.

stop_and_release()

Deteniendo las aplicaciones actualmente en ejecución y liberando todas las articulaciones (activando el modo de compensación de gravedad). Tenga en cuenta que el usuario debe sostener manualmente todas las articulaciones (desactivando el modo de compensación de gravedad) para continuar ejecutando aplicaciones adicionales posteriormente.

is_status(*status*)

Verificando si el estado del programa corresponde al estado dado.

Parámetros:

status (*str or list of str*) –

El estado del programa a verificar puede ser uno o varios de los siguientes:

- "ready" (el robot está inactivo y esperando recibir comandos)
- "processing" (el robot está procesando datos, lo cual puede tardar)
- "hand_guidance_control" (el robot está en modo de control manual)
- "running" (el robot está ejecutando una aplicación o un movimiento)
- "paused" (el robot está pausando una aplicación o un movimiento)
- "emergency_stop" (se activó una parada de emergencia)
- "normal_stop" (se activó una parada normal)
- "protective_stop" (se activó una parada protectora)
- "error" (se produjo un error o una infracción de seguridad)
- "position_limit" (se produjo una violación del límite de posición segura)
- "safeguard" (se activó una interrupción de seguridad)
- "collision" (se activó una interrupción por colisión)
- "proceed" (el robot continúa desde una interrupción)
- "stopping" (el robot está deteniendo su programa)

Devuelve:

Si el estado del programa coincide con el estado dado (o es uno de los estados dados)

Tipo del valor devuelto:

bool

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

Creando un diálogo con botones. Este diálogo bloquea el flujo del programa hasta que el usuario responda con una de las opciones presentadas.

Parámetros:

- **text** (*str*) – El mensaje (o pregunta) presentado al usuario.
- **options** (*list of str*) – Las opciones que se pueden elegir, que son las etiquetas de cada botón.
- **title** (*str*) – El título del cuadro de diálogo emergente.
- **dialog_type** (*int*) –
  El tipo de diálogo que se mostrará al usuario. Puede ser uno de los siguientes:
  - 0: información / color azul
  - 1: éxito / color verde
  - 2: advertencia / color naranja
  - 3: error / color rojo
- **dialog_value** (*str*) – El valor predeterminado a devolver si el diálogo no recibe respuesta.
- **mandatory** (*bool*) – Indica si el diálogo debe ser respondido para que el flujo del programa continúe. Si se establece en *True*, el diálogo no puede ser ignorado, por ejemplo, cerrando la ventana emergente, y bloquea hasta que se reciba una respuesta.

Devuelve:

La opción seleccionada, que es la etiqueta del botón presionado por el usuario.

Tipo del valor devuelto:

str

Ejemplos:

Este ejemplo muestra cómo crear una ventana emergente informativa simple que contiene un botón «Continuar». Al presionar el botón, el programa continuará ejecutando su código.

```
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*)

Crear un diálogo con opciones presentadas en una lista desplegable. El diálogo bloquea el flujo del programa hasta que el usuario seleccione y confirme una de las opciones.

Parámetros:

- **text** (*str*) – El mensaje (o pregunta) presentado al usuario.
- **options** (*list of str*) – Las opciones que se pueden elegir, que son las etiquetas de las opciones desplegables.
- **title** (*str*) – El título del cuadro de diálogo emergente.
- **dialog_type** (*int*) –
  El tipo de diálogo que se mostrará al usuario. Puede ser uno de los siguientes:
  - 0: información / color azul
  - 1: éxito / color verde
  - 2: advertencia / color naranja
  - 3: error / color rojo
- **dialog_value** (*str*) – El valor predeterminado a devolver si el diálogo no recibe respuesta.
- **mandatory** (*bool*) – Indica si el diálogo debe ser respondido para que el flujo del programa continúe. Si se establece en *True*, el diálogo no puede ser ignorado, por ejemplo, cerrando la ventana emergente, y bloquea hasta que se reciba una respuesta.

Devuelve:

La opción seleccionada, que es la etiqueta de la opción desplegable seleccionada.

Tipo del valor devuelto:

str

Ejemplos:

Este ejemplo muestra cómo crear una ventana emergente de advertencia y permite al usuario elegir entre varias opciones para influir en el flujo del programa.

```
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*)

Crear un diálogo con un campo de texto. El diálogo bloquea el flujo del programa hasta que el usuario ingrese un texto y confirme el diálogo.

Parámetros:

- **text** (*str*) – El mensaje (o pregunta) presentado al usuario.
- **title** (*str*) – El título del cuadro de diálogo emergente.
- **dialog_type** (*int*) –
  El tipo de diálogo que se mostrará al usuario. Puede ser uno de los siguientes:
  - 0: información / color azul
  - 1: éxito / color verde
  - 2: advertencia / color naranja
  - 3: error / color rojo
- **dialog_value** (*str*) – El valor predeterminado a devolver si el diálogo no recibe respuesta.
- **mandatory** (*bool*) – Indica si el diálogo debe ser respondido para que el flujo del programa continúe. Si se establece en *True*, el diálogo no puede ser ignorado, por ejemplo, cerrando la ventana emergente, y bloquea hasta que se reciba una respuesta.

Devuelve:

El texto ingresado en el campo de texto.

Tipo del valor devuelto:

str

Ejemplos:

Este ejemplo muestra cómo crear una ventana emergente simple con un campo de texto para ingresar su nombre. Después de que el usuario ingrese su nombre, se puede usar más adelante en el programa, por ejemplo, para registro.

```
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*)

Crear un diálogo con botones *Sí* y *No*. El diálogo bloquea el flujo del programa hasta que el usuario presione uno de estos botones.

Parámetros:

- **text** (*str*) – El mensaje (o pregunta) presentado al usuario.
- **title** (*str*) – El título del cuadro de diálogo emergente.
- **dialog_type** (*int*) –
  El tipo de diálogo que se mostrará al usuario. Puede ser uno de los siguientes:
  - 0: información / color azul
  - 1: éxito / color verde
  - 2: advertencia / color naranja
  - 3: error / color rojo
- **dialog_value** (*str*) – El valor predeterminado a devolver si el diálogo no recibe respuesta.
- **mandatory** (*bool*) – Indica si el diálogo debe ser respondido para que el flujo del programa continúe. Si se establece en *True*, el diálogo no puede ser ignorado, por ejemplo, cerrando la ventana emergente, y bloquea hasta que se reciba una respuesta.

Devuelve:

Verdadero si el usuario selecciona «sí», Falso si el usuario selecciona «no».

Tipo del valor devuelto:

bool

Ejemplos:

Este ejemplo utiliza un diálogo de confirmación simple para permitir al usuario elegir si continuar o abortar la aplicación.

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

print(**args*)

Imprimir un mensaje en la salida de la aplicación. Tenga en cuenta que esta función es similar en uso a la función incorporada *print* de Python. Soporta múltiples argumentos y varios tipos de argumentos. Los argumentos se convierten automáticamente a cadenas y se concatenan con un separador de espacio en blanco.

Parámetros:

args (*tuple of various*) – Los argumentos a imprimir.

Ejemplos:

Imprimir el valor de una variable en un estilo bien formateado. Tenga en cuenta que la primera llamada usa conversión y concatenación automáticas, mientras que la segunda llamada usa f-strings con el mismo resultado.

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

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

Mostrar un mensaje de notificación personalizado en la interfaz. Esto no tiene efecto sobre el flujo del programa. El usuario puede especificar en el menú de configuración si las notificaciones desaparecen o permanecen.

Parámetros:

- **message_text** (*str*) – El texto de notificación que se mostrará en la interfaz.
- **message_title** (*str*) – El título de la notificación emergente.

raise_error(*message*)

Generar un mensaje de error personalizado en la aplicación en ejecución. El error es de tipo `ApplicationError` y puede ser capturado en la aplicación. Si no se captura, la aplicación en ejecución se aborta. El error también se registra en el archivo `error.log`, que forma parte del sistema de registro. Los registros pueden descargarse desde la interfaz pulsando `CTRL + SHIFT + L`.

Parámetros:

message (*str*) – El mensaje de error que se debe generar en la aplicación.

Muestra:

ApplicationError – Que contiene el mensaje personalizado.

Ejemplos:

Una aplicación en segundo plano supervisa entradas digitales y comprueba si se activa una entrada de error específica. Si se activa una entrada de error, este error se genera en la aplicación.

```
# 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)
```

Este error de la aplicación puede ser capturado y gestionado por la aplicación principal que controla el robot.

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

raise_warning(*message*)

Mostrar un cuadro de mensaje de advertencia personalizado en la interfaz. Esto no tiene efecto sobre el flujo del programa. El usuario puede especificar en el menú de configuración si las advertencias desaparecen o permanecen en la interfaz.

Parámetros:

message (*str*) – El mensaje de advertencia que se mostrará en la interfaz.

Ejemplos:

Supongamos que tenemos un sensor de temperatura conectado a una entrada analógica. Este ejemplo muestra una advertencia en la interfaz si el valor supera un determinado umbral, y la restablece si el valor vuelve a caer por debajo de dicho umbral.

```
# 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*)

Registrar un mensaje en un archivo de registro personalizado.

Parámetros:

- **message** (*str*) – El mensaje personalizado que se va a registrar.
- **logger** (*str*) – El nombre del archivo de registro en el que se escribirá. El archivo de registro se encuentra en `custom/<logger>.log` dentro de la carpeta de registros. El nombre de archivo predeterminado es `custom/custom.log`. Los registros pueden descargarse desde la interfaz pulsando `CTRL + SHIFT + L`.
- **include_timestamp** (*bool*) – Si se añade una marca de tiempo al mensaje registrado.

Ejemplos:

Registrar el valor de una medición de un sensor en el archivo *custom/measurements.log*.

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

clear_log()

Borrar y eliminar todos los archivos de registro personalizados.

set_shared_variable(*name*, *value*)

Establecer el valor de una variable compartida.

Tenga en cuenta que las *variables compartidas* solo existen mientras se ejecuta la aplicación principal. Tras finalizar la aplicación principal, todas las *variables compartidas* se eliminan automáticamente. Si necesita que las variables conserven sus valores durante la ejecución de la aplicación o incluso tras reiniciar el robot, utilice *variables globales*.

Parámetros:

- **name** (*str*) – El nombre de la variable que se va a establecer.
- **value** (*various*) – El nuevo valor de esta variable.

Ejemplos:

Este ejemplo muestra cómo una aplicación paralela puede ser finalizada por la aplicación principal usando variables compartidas. La aplicación paralela simplemente escucha una bandera de terminación y se aborta en cuanto la bandera se establece en *True*. Además, la variable compartida se elimina una vez que deja de tener efecto. Dado que las variables compartidas se borran cuando finaliza la aplicación principal, esto no es necesario, pero es una buena práctica.

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

La aplicación principal se asegura de que la bandera de terminación exista, ejecuta la aplicación paralela y finalmente establece la bandera de terminación en *True* para finalizar la aplicación paralela.

```
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*)

Leer el valor de una variable compartida específica. De forma predeterminada, esta función lee los valores de todas las variables compartidas.

Tenga en cuenta que las *variables compartidas* solo existen mientras se ejecuta la aplicación principal. Tras finalizar la aplicación principal, todas las *variables compartidas* se eliminan automáticamente. Si necesita que las variables conserven sus valores durante la ejecución de la aplicación o incluso tras reiniciar el robot, utilice *variables globales*.

Parámetros:

name (*str (or None)*) – Nombre de la variable a devolver (o None si se deben devolver todas las variables).

Devuelve:

Valor de la variable a devolver (o un diccionario que contiene todas las variables).

Tipo del valor devuelto:

various (or dict)

Ejemplos:

Consulte los ejemplos de *set_shared_variable*, que combinan el uso de todas las funciones de *variables compartidas*.

remove_shared_variable(*name*)

Eliminar una variable compartida. Tras eliminarla, su valor ya no puede ser accedido.

Tenga en cuenta que las *variables compartidas* solo existen mientras se ejecuta la aplicación principal. Tras finalizar la aplicación principal, todas las *variables compartidas* se eliminan automáticamente. Si necesita que las variables conserven sus valores durante la ejecución de la aplicación o incluso tras reiniciar el robot, utilice *variables globales*.

Parámetros:

name (*str*) – nombre de la variable a eliminar

Ejemplos:

Consulte los ejemplos de *set_shared_variable*, que combinan el uso de todas las funciones de *variables compartidas*.

set_global_variable(*name*, *value*)

Establecer el valor de una variable global.

Tenga en cuenta que los cambios en las *variables globales* son permanentes y persisten incluso al apagar y reiniciar el robot. Por ello, se recomienda encarecidamente eliminar las *variables globales* una vez que ya no cumplan ninguna función. Si solo necesita variables durante la ejecución de una sola aplicación, utilice *variables compartidas*.

Parámetros:

- **name** (*str*) – El nombre de la variable que se va a establecer.
- **value** (*various*) – El nuevo valor de esta variable.

Ejemplos:

Este ejemplo muestra cómo una aplicación repetitiva cuenta el número de iteraciones exitosas. Esto se realiza inicializando un contador global en la primera ejecución y aumentando este contador al final de cada iteración del bucle.

```
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*)

Leer el valor de una variable global específica. De forma predeterminada, esta función lee los valores de todas las variables globales.

Tenga en cuenta que los cambios en las *variables globales* son permanentes y persisten incluso al apagar y reiniciar el robot. Por ello, se recomienda encarecidamente eliminar las *variables globales* una vez que ya no cumplan ninguna función. Si solo necesita variables durante la ejecución de una sola aplicación, utilice *variables compartidas*.

Parámetros:

name (*str (or None)*) – Nombre de la variable a devolver (o None si se deben devolver todas las variables).

Devuelve:

Valor de la variable a devolver (o un diccionario que contiene todas las variables).

Tipo del valor devuelto:

various (or dict)

Ejemplos:

Consulte los ejemplos de *set_global_variable*, que combinan el uso de las funciones de *variables globales*.

remove_global_variable(*name*)

Eliminar una variable global. Después de eliminarla, ya no se podrá acceder a su valor.

Tenga en cuenta que los cambios en las *variables globales* son permanentes y persisten incluso al apagar y reiniciar el robot. Por ello, se recomienda encarecidamente eliminar las *variables globales* una vez que ya no cumplan ninguna función. Si solo necesita variables durante la ejecución de una sola aplicación, utilice *variables compartidas*.

Parámetros:

name (*str*) – nombre de la variable a eliminar

Ejemplos:

Este ejemplo elimina todas las variables globales.

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

set_state_variable(*name*, *value*)

Establecer el valor de una variable de estado.

Tenga en cuenta que las *variables de estado* se definen individualmente para cada aplicación y solo se utilizan en aplicaciones principales. Pueden configurarse y personalizarse en la pestaña *variables* del editor de aplicaciones. Cada *variable de estado* tiene un tipo específico y solo acepta valores de ese tipo. El propósito principal de las *variables de estado* es que su valor se muestre en tiempo real en la *Interfaz del Panel* para observar y controlar la aplicación en ejecución.

Parámetros:

- **name** (*str*) – El nombre de la variable que se va a establecer.
- **value** (*various*) – El nuevo valor de esta variable.

Ejemplos:

Supongamos que para esta aplicación se ha definido una variable de estado de solo lectura de tipo *percent* llamada *progress* y una variable de estado editable dinámicamente de tipo *boolean* llamada *logging*. El propósito de la variable *progress* es almacenar el progreso actual del programa en porcentaje. El propósito del indicador *logging* es determinar si la actualización del progreso debe registrarse. Tenga en cuenta que el usuario puede cambiar el valor de *logging* en cualquier momento, adaptando dinámicamente el programa.

```
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*)

Leer el valor de una variable de estado específica. De forma predeterminada, esta función lee los valores de todas las variables de estado.

Tenga en cuenta que las *variables de estado* se definen individualmente para cada aplicación y solo se utilizan en aplicaciones principales. Pueden configurarse y personalizarse en la pestaña *variables* del editor de aplicaciones. Cada *variable de estado* tiene un tipo específico y solo acepta valores de ese tipo. El propósito principal de las *variables de estado* es que su valor se muestre en tiempo real en la *Interfaz del Panel* para observar y controlar la aplicación en ejecución.

Parámetros:

name (*str (or None)*) – Nombre de la variable a devolver (o None si se deben devolver todas las variables).

Devuelve:

Valor de la variable a devolver (o un diccionario que contiene todas las variables).

Tipo del valor devuelto:

various (or dict)

Ejemplos:

Consulte los ejemplos de *set_state_variable*, que combinan el uso de las funciones de *variables de estado*.

read_actuator_position(*actuator_ids*=*None*)

Leer la posición angular de uno, varios o todos los actuadores.

Parámetros:

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

Los ID de los actuadores desde los que se va a leer. Especifique el valor de la siguiente manera:

- el ID de una sola articulación (p. ej., *6*)
- una lista de IDs de articulaciones para varias articulaciones (p. ej., *[1, 4, 6]*)
- un conjunto de nombres de articulaciones para múltiples articulaciones (p. ej., *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* o *None* para leer de todas las articulaciones

Devuelve:

La posición angular de los actuadores solicitados (en [deg]), que puede ser:

- un valor de posición de una sola articulación → float
- una lista de valores de posición de articulaciones → lista de float
- una asignación entre los ID de articulaciones y sus valores de posición → diccionario de float

Tipo del valor devuelto:

float, or list of float, or dict of float

Ejemplos:

Imprimir las posiciones actuales de los actuadores de las articulaciones de la muñeca.

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

read_actuator_velocity(*actuator_ids=None*)

Lectura de la velocidad angular de uno, varios o todos los actuadores.

Parámetros:

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

Los ID de los actuadores desde los que se va a leer. Especifique el valor de la siguiente manera:

- el ID de una sola articulación (p. ej., *6*)
- una lista de IDs de articulaciones para varias articulaciones (p. ej., *[1, 4, 6]*)
- un conjunto de nombres de articulaciones para múltiples articulaciones (p. ej., *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* o *None* para leer de todas las articulaciones

Devuelve:

La velocidad angular de los actuadores solicitados (en °/s), que puede ser:

- un único valor de velocidad de una articulación -> float
- una lista de valores de velocidad de articulaciones -> lista de float
- un mapeo entre IDs de articulaciones y sus valores de velocidad -> diccionario de float

Tipo del valor devuelto:

float, or list of float, or dict of float

Ejemplos:

Monitorización de la velocidad durante un movimiento para determinar el actuador que se desplaza más rápido.

```
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()

Obtención de la velocidad lineal del TCP (punto central de la herramienta) del robot.

Devuelve:

La velocidad lineal de la herramienta en [mm/s].

Tipo del valor devuelto:

list

Ejemplos:

Obtener e imprimir la velocidad lineal actual.

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

get_angular_velocity()

Obtención de la velocidad angular del TCP (punto central de la herramienta) del robot.

Devuelve:

La velocidad angular de la herramienta en [deg/s].

Tipo del valor devuelto:

list

Ejemplos:

Obtener e imprimir la velocidad angular actual.

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

read_actuator_current(*actuator_ids=None*)

Lectura de la corriente aplicada en uno, varios o todos los actuadores.

Parámetros:

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

Los ID de los actuadores desde los que se va a leer. Especifique el valor de la siguiente manera:

- el ID de una sola articulación (p. ej., *6*)
- una lista de IDs de articulaciones para varias articulaciones (p. ej., *[1, 4, 6]*)
- un conjunto de nombres de articulaciones para múltiples articulaciones (p. ej., *{1, 4, 6}*)
- la constante *ALL_KIN_JOINTS* o *None* para leer de todas las articulaciones

Devuelve:

Los valores de corriente de los actuadores solicitados (en A), que pueden ser:

- un único valor de corriente de una articulación -> float
- una lista de valores de corriente de articulaciones -> lista de float
- un mapeo entre IDs de articulaciones y sus valores de corriente -> diccionario de float

Tipo del valor devuelto:

float, or list of float, or dict of float

Ejemplos:

Impresión de las corrientes de los actuadores del robot en una posición específica dos veces, una antes y otra después de recoger un objeto.

```
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()

Activación de la función de recogida de la herramienta para sujetar un objeto. Esto puede implicar el cierre de pinzas en herramientas mecánicas o la aplicación de succión en herramientas neumáticas.

Ejemplos:

Activar la función de recogida de la herramienta.

```
tool_pick()
```

tool_place()

Activación de la función de colocación de la herramienta para soltar un objeto. Esto puede implicar la apertura de una pinza en herramientas mecánicas o la detención de la succión en herramientas neumáticas.

Ejemplos:

Activar la función de colocación de la herramienta.

```
tool_place()
```

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

Configurar dinámicamente la carga útil de la herramienta durante la operación del robot.

Parámetros:

- **payload** (*float*) – la carga útil de la herramienta (peso + elementos agarrados) en [kg]
- **center_of_mass** (*list of float*) – centro de masa [x, y, z] relativo al marco del brida en [m]
- **inertia** (*list of float*) – matriz de inercia convertida en lista de longitud 9 [kg m^2]

Ejemplos:

Cambiar la carga útil al recoger y colocar objetos de una posición a otra.

```
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*)

Leer el valor de uno, varios o todos los E/S de la categoría “Digital Main Inputs”.

Los dos estados de las E/S digitales son:

- *HIGH*, que se representa con el valor booleano *True*
- *LOW*, que se representa con el valor booleano *False*

Parámetros:

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

Especifica qué E/S se van a leer. Los identificadores son los números de E/S o los nombres personalizados únicos que se pueden asignar a estas E/S. Esto puede ser uno de los siguientes:

- un solo número (int) o nombre personalizado (str) -> devuelve un bool
- una lista de números (int) o nombres personalizados (str), o mixtos -> devuelve una lista de bool
- un conjunto de números (int) o nombres personalizados (str), o mixtos -> devuelve un diccionario con valores booleanos y los identificadores como claves

Devuelve:

Los valores booleanos de las E/S digitales solicitadas (HIGH = True, LOW = False)

Tipo del valor devuelto:

bool, list, dict

Ejemplos:

Supongamos que se asignó el nombre “trigger” a una de las entradas digitales. Este bloque de código espera hasta que este disparador cambie a HIGH antes de continuar con su ejecución.

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

Leer las primeras 8 E/S digitales y convertir la lista binaria de valores booleanos a un número entero entre 0 y 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*)

Leer el valor de uno, varios o todos los E/S de la categoría “Digital Tool Inputs”.

Para más detalles, consulte “read_digital_main_input”, que tiene un uso similar.

read_analog_main_input(*identifiers*)

Leer el valor de uno, varios o todos los E/S de la categoría “Analog Main Inputs”.

Las E/S analógicas pueden estar en modo de voltaje o de corriente. Los valores se aplican y se leen en pasos de 0.1, lo que significa que el valor de E/S 55 representa el valor real de voltaje de 5.5 V o el valor de corriente de 5.5 mA. El rango de valores de las E/S analógicas es:

- Entre 0.0 y 10.0 V (en modo de voltaje)
- Entre 4.0 y 20.0 mA (en modo de corriente)

Parámetros:

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

Especifica qué E/S se van a leer. Los identificadores son los números de E/S o los nombres personalizados únicos que se pueden asignar a estas E/S. Esto puede ser uno de los siguientes:

- un solo número (int) o nombre personalizado (str) -> devuelve un bool
- una lista de números (int) o nombres personalizados (str), o mixtos -> devuelve una lista de bool
- un conjunto de números (int) o nombres personalizados (str), o mixtos -> devuelve un diccionario con valores booleanos y los identificadores como claves

Devuelve:

Los valores numéricos de las E/S analógicas solicitadas

Tipo del valor devuelto:

float, list, dict

Ejemplos:

Supongamos que a una de las entradas analógicas en modo de corriente se le asignó el nombre “temperature”. A esta entrada se conectó un sensor de temperatura y se determinó que un valor de retorno de 10.6 mA representa un umbral crítico de temperatura que no debe superarse. A continuación, ejecute el siguiente programa en paralelo con su aplicación principal, el cual aborta el programa si se alcanza el umbral crítico.

```
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*)

Leer el valor de uno, varios o todos los E/S de la categoría “Analog Tool I/O”.

Para más detalles, consulte “read_analog_main_input”, que tiene un uso similar.

write_digital_main_output(*identifiers*, *values*)

Escribir el valor de uno, varios o todos los E/S de la categoría “Digital Main Output”.

Los dos estados de las E/S digitales son:

- *HIGH*, que se representa con el valor booleano *True*
- *LOW*, que se representa con el valor booleano *False*

Parámetros:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) –
  Especifica qué E/S se van a escribir. Los identificadores son los números de E/S o los nombres personalizados únicos que se pueden asignar a estas E/S. Esto puede ser uno de los siguientes:
  - un solo número (int) o nombre personalizado (str) -> *values* es un bool
  - una lista de números (int) o nombres personalizados (str), o mixtos -> *values* es una lista de bool
  - un conjunto de números (int) o nombres personalizados (str), o mixtos -> *values* es un diccionario con valores booleanos y los identificadores como claves
- **values** (*bool, list (of bool), dict (of bool)*) –
  Los valores booleanos de las E/S digitales a escribir (HIGH = True, LOW = False). Dependiendo de los *identifiers*, esto puede ser uno de los siguientes:
  - un solo bool -> si *identifiers* es una sola E/S
  - una lista de booleanos -> si *identifiers* es una lista de E/S
  - un diccionario con valores booleanos y identificadores como claves -> si *identifiers* es un conjunto de E/S

Ejemplos:

Supongamos que se asignó el nombre “trigger” a una de las salidas digitales. Este código envía un pulso a esta salida digital estableciéndola en HIGH durante 100 ms y luego volviéndola a LOW.

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

Escribir un número entero de 8 bits (entre 0 y 255) en las primeras 8 E/S digitales, convirtiendo primero el número a una lista de booleanos y luego escribiendo los 8 valores simultáneamente.

```
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*)

Escribir el valor de uno, varios o todos los E/S de la categoría “Digital Tool Output”.

Para más detalles, consulte “write_digital_main_output”, que tiene un uso similar.

write_analog_main_output(*identifiers*, *values*)

Escribir el valor de uno, varios o todos los E/S de la categoría “Analog Main Outputs”.

Las E/S analógicas pueden estar en modo de voltaje o de corriente. Los valores se aplican y se leen en pasos de 0.1, lo que significa que el valor de E/S 55 representa el valor real de voltaje de 5.5 V o el valor de corriente de 5.5 mA. El rango de valores de las E/S analógicas es:

- Entre 0.0 y 10.0 V (en modo de voltaje)
- Entre 4.0 y 20.0 mA (en modo de corriente)

Parámetros:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) –
  Especifica qué E/S se van a escribir. Los identificadores son los números de E/S o los nombres personalizados únicos que se pueden asignar a estas E/S. Esto puede ser uno de los siguientes:
  - un solo número (int) o nombre personalizado (str) -> *values* es un float
  - una lista de números (int) o nombres personalizados (str), o mixtos -> *values* es una lista de float
  - un conjunto de números (int) o nombres personalizados (str), o mixtos -> *values* es un diccionario con valores float y los identificadores como claves
- **values** (*float, list (of float), dict (of float)*) –
  Los valores numéricos de las E/S analógicas a escribir. Dependiendo de los *identifiers*, esto puede ser uno de los siguientes:
  - un float único -> si *identifiers* es un solo I/O
  - una lista de float -> si *identifiers* es una lista de I/O
  - un dict con valores float y los identificadores como claves -> si *identifiers* es un conjunto de I/O

Ejemplos:

Supongamos que los valores RGB de un LED controlable están conectados directamente a tres E/S analógicas con los nombres personalizados “red”, “green” y “blue”, de modo que el rango máximo de valores [0, 255] se asigna directamente al rango máximo de voltaje [0.0 V, 10.0 V]. Este ejemplo escribe estos valores simultáneamente.

```
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*)

Escribir el valor de uno, varios o todos los E/S de la categoría “Analog Tool I/O”.

Para más detalles, consulte “write_analog_main_output”, que tiene un uso similar.

wait_for_digital_main_input(*identifier*, *value*)

Esperando hasta que una E/S de la categoría “Digital Main Inputs” lea un valor específico.

Los dos estados de las E/S digitales son:

- *HIGH*, que se representa con el valor booleano *True*
- *LOW*, que se representa con el valor booleano *False*

Parámetros:

- **identifier** (*int, str*) – Especifica qué E/S esperar. El identificador puede ser el número de E/S (int) o el nombre personalizado único (str) que se puede asignar a esta E/S.
- **value** (*bool*) – El valor a esperar.

Ejemplos:

Supongamos que se asignó el nombre “trigger” a una de las entradas digitales. Este bloque de código espera hasta que este disparador cambie a HIGH antes de continuar con su ejecución.

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

En casos raros, es posible que desees poder pausar un programa y aún así quieras que la función de espera termine si se activa la E/S. Esta función de espera no se puede usar para este enfoque, pero el siguiente ejemplo hace exactamente eso:

Agrega este código a una aplicación en segundo plano. Supervisa el valor de una E/S específica y establece una bandera si se alcanza el valor especificado durante la supervisión.

```
# 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)
```

Y agrega este código a la contraparte de la aplicación principal que activa el monitor en segundo plano y espera hasta que el monitor establezca la bandera.

```
# 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*)

Esperando hasta que una E/S de la categoría “Digital Tool Inputs” lea un valor específico.

Para más detalles, consulte “wait_for_digital_main_input”, que tiene un uso similar.

wait_for_analog_main_input(*identifier*, *value_range*)

Esperando hasta que una E/S de la categoría “Analog Main Inputs” lea un valor específico.

Las E/S analógicas pueden estar en modo de voltaje o de corriente. Los valores se aplican y se leen en pasos de 0.1, lo que significa que el valor de E/S 55 representa el valor real de voltaje de 5.5 V o el valor de corriente de 5.5 mA. El rango de valores de las E/S analógicas es:

- Entre 0.0 y 10.0 V (en modo de voltaje)
- Entre 4.0 y 20.0 mA (en modo de corriente)

Parámetros:

- **identifier** (*int, str*) – Especifica qué E/S esperar. El identificador puede ser el número de E/S (int) o el nombre personalizado único (str) que se puede asignar a esta E/S.
- **value_range** (*list of float*) –
  Se espera que esta E/S alcance un valor dentro de este rango. Esto es uno de los siguientes:
  - tanto el límite inferior como el superior (p. ej., [5.5, 7.0])
  - solo un límite inferior (p. ej., [5.5, None])
  - solo un límite superior (p. ej., [None, 7.0])

Ejemplos:

Consideremos una E/S analógica en modo de voltaje. Este ejemplo espera a que el valor de voltaje supere los 8.0 V, lo cual se representa mediante el valor 80 (en pasos de 0.1 V).

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

wait_for_analog_tool_io(*identifier*, *value_range*)

Esperando hasta que una E/S de la categoría “Analog Tool I/O” lea un valor específico.

Para más detalles, consulte “wait_for_analog_main_input”, que tiene un uso similar.

get_runtime_position_offsets()

Esta función devuelve los desplazamientos de posición del robot.

Devuelve:

desplazamientos de posición del robot

Tipo del valor devuelto:

list of float

read_button_state(*channel=None*)

Leer el estado actual (presionado o no) de uno o todos los botones personalizables de la interfaz del robot. Ten en cuenta que los botones con funciones asignadas específicas no se pueden leer en tiempo de ejecución, ya que se usan de otra manera.

Estos botones están ubicados en la articulación superior del brazo del robot.

Parámetros:

channel (*int or None*) – El identificador del botón, que es un número entre 1 y el número de botones. Por defecto, se leen los estados de todos los botones personalizables.

Devuelve:

El estado actual del botón solicitado como booleano, o el estado actual de cada botón, empaquetado en un diccionario de la forma {1: bool, 2: bool, ...}.

Tipo del valor devuelto:

bool or dict of bool

Ejemplos:

Este ejemplo imprime cada vez que cambia el estado de un botón personalizable.

```
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)
```

Ejecuta este ejemplo como una aplicación en segundo plano. Permite pausar y reanudar la aplicación principal en ejecución usando los botones 1 y 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*)

Creando un cliente de socket TCP para transmitir mensajes. El cliente intenta continuamente conectarse al servidor en la dirección especificada. Los mensajes se codifican con el estándar *UTF-8*. Además, los mensajes se separan por el carácter de fin de texto *0x03* (`\x03` en Python).

Parámetros:

- **ip** (*str*) – La dirección IP del servidor (por ejemplo, `"192.168.80.2"`)
- **port** (*int*) – El número de puerto a través del cual el servidor es accesible (p.ej. `60001`). Los puertos están limitados al rango entre 60000 y 61000.

Devuelve:

El objeto cliente de socket TCP, que se utiliza para enviar y recibir mensajes.

Tipo del valor devuelto:

Socket

Ejemplos:

Este ejemplo realiza un saludo (handshake) entre un cliente de socket TCP en la aplicación y un servidor de socket TCP en la misma red. Primero se establece la conexión. Luego, el cliente espera que el servidor envíe un mensaje inicial. Al recibirlo, el cliente responde con su propio mensaje y finalmente cierra la conexión.

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

create_tcp_server(*port*)

Creando un servidor de socket TCP que puede manejar hasta 20 clientes. Los mensajes se codifican con el estándar *UTF-8*. Además, los mensajes están separados por el carácter de fin de texto *0x03* (`\x03` en Python).

Parámetros:

port (*int*) – El número de puerto a través del cual los clientes pueden acceder a este servidor (p.ej. `60002`). Los puertos están limitados al rango entre 60000 y 61000.

Devuelve:

El objeto servidor de socket TCP, que se utiliza para enviar y recibir mensajes.

Tipo del valor devuelto:

Socket

Ejemplos:

Este ejemplo realiza un saludo (handshake) entre un servidor de socket TCP en la aplicación y el primer cliente que se conecta. Primero, el servidor abre la conexión en un puerto específico y espera a que un cliente se conecte. Luego, el servidor espera que el cliente envíe un mensaje inicial. Al recibirlo, el servidor responde con su propio mensaje y finalmente cierra la conexión con todos los clientes conectados.

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

create_udp_socket(*port*)

Creando un socket UDP para transmitir mensajes. Los mensajes se codifican con el estándar *UTF-8*. Además, los mensajes están separados por el carácter de fin de texto *0x03* (`\x03` en Python).

Parámetros:

port (*int*) – El número de puerto a través del cual se puede acceder a este socket (p.ej. `60003`). Los puertos están limitados al rango entre 60000 y 61000.

Devuelve:

El objeto socket UDP, que se utiliza para enviar y recibir mensajes.

Tipo del valor devuelto:

Socket

Ejemplos:

Este ejemplo realiza un saludo (handshake) entre un socket UDP en la aplicación y la primera conexión que se establece con él. Primero, el socket abre la conexión en un puerto específico y espera que alguna contraparte se conecte. Luego, el socket espera a que la contraparte envíe un mensaje inicial. Al recibirlo, el socket responde con su propio mensaje y finalmente cierra la conexión.

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

send_message(*message*)

Enviando un mensaje con la clave “script_send” a una conexión TCP abierta.

Parámetros:

message (*str*) – El mensaje a enviar.

Ejemplos:

Enviando un mensaje vía TCP a cualquier dispositivo que pueda escucharlo:

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

En el lado del dispositivo conectado, estos datos se reciben en el siguiente formato:

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

receive_message(*timeout*=*None*)

Recibiendo un mensaje desde una conexión TCP abierta llamando a la acción "script_receive".

Parámetros:

timeout (*None or float*) – El tiempo máximo de espera para un mensaje. Si no se especifica, espera indefinidamente.

Ejemplos:

Un dispositivo externo se conecta al robot vía TCP y envía los siguientes datos:

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

En el lado del robot, esta función se puede usar para recibir e imprimir el mensaje:

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

clear_messages()

Limpiando la cola de mensajes TCP de mensajes entrantes anteriores que podrían no haber sido leídos.

Ejemplos:

Antes de esperar un nuevo mensaje, a veces tiene sentido asegurarse de que los mensajes antiguos sean rechazados.

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

class Coordinates

El objeto Coordinates define una transformación matemática que se puede usar, por ejemplo, como coordenadas objetivo en funciones de movimiento, o para definir marcos de coordenadas.

__init__(*coordinates*)

Crear un objeto Coordinates, ya sea con valores de coordenadas específicos o conteniendo las coordenadas actuales del robot.

Parámetros:

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

Un conjunto específico de coordenadas, que puede ser uno de los siguientes:

- una lista en la forma [x, y, z, roll, pitch, yaw] con valores de posición en [mm] y valores de orientación en [grados]
- un diccionario en la forma {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} con valores de posición en [mm] y valores de orientación en [grados]
- un objeto Coordinates (copiando otro objeto Coordinates)

Constructor:

Crear un objeto Coordinates a partir de valores de coordenadas.

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

Crear un objeto Coordinates que contenga las coordenadas actuales del robot usando la función `get_current_coordinates`.

```
current_coordinates = get_current_coordinates()
```

***property* position**

La parte de posición del objeto Coordinates.

Devuelve:

La posición en la forma [x, y, z] en [mm].

Tipo del valor devuelto:

list of float

***property* orientation**

La parte de orientación del objeto Coordinates.

Devuelve:

La orientación en la forma [roll, pitch, yaw] en [grados].

Tipo del valor devuelto:

list of float

to_list()

Convertir el objeto coordinates a una lista de valores de coordenadas en la forma [x, y, z, roll, pitch, yaw], donde los valores de posición están en [mm] y los valores de rotación en [grados].

Devuelve:

Valores de coordenadas en formato de lista.

Tipo del valor devuelto:

list of float

Ejemplos:

Esta función se puede usar cuando se necesitan las coordenadas como lista:

```
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 el objeto coordinates a un diccionario de valores de coordenadas en la forma {«x»: x, «y»: y, «z»: z, «roll»: roll, «pitch»: pitch, «yaw»: yaw}, donde los valores de posición están en [mm] y los valores de rotación en [grados].

Devuelve:

Valores de coordenadas en formato de diccionario.

Tipo del valor devuelto:

dict of float

Ejemplos:

Esta función se puede usar cuando se necesitan las coordenadas como diccionario:

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

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

inverted()

Calcula la transformación inversa de este objeto coordinates.

Devuelve:

inverso del objeto de transformación

Tipo del valor devuelto:

Transform

Ejemplos:

```
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

El objeto Pose define una pose del robot, que consiste en sus coordenadas (posición y orientación) y su configuración (ya que el mismo conjunto de coordenadas se puede alcanzar con múltiples configuraciones).

__init__(*pose*)

Crear un objeto Pose, ya sea cargando una pose guardada por nombre o ID, o desde la pose actual del robot.

Parámetros:

pose (*str, int, or Pose*) –

Referencia a la pose a cargar, que puede ser uno de los siguientes:

- el nombre (str) de una pose guardada en la base de datos.
- el ID (int) de una pose guardada en la base de datos
- un objeto Pose (copiando otro objeto Pose)

Constructor:

Cargar una pose desde la base de datos.

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

Crear un objeto Pose que contenga la pose actual del robot usando la función `get_current_pose`.

```
current_pose = get_current_pose()
```

Crear un objeto Pose que contenga las coordenadas y configuración especificadas usando la función `create_pose`.

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

***property* coordinates**

Las coordenadas de esta pose, que es un objeto Coordinates que contiene los valores de posición y orientación.

Devuelve:

Las coordenadas de esta pose.

Tipo del valor devuelto:

Coordinates

***property* configuration**

Dado que un conjunto de coordenadas de pose puede alcanzarse con hasta 8 configuraciones posibles de las articulaciones del robot, la configuración se define como un número (de 0 a 7) o una cadena (de “c1” a “c8”) para definir de manera única la pose.

Devuelve:

El índice de la configuración de la pose (de 0 a 7).

Tipo del valor devuelto:

int

class LinearSegment

El objeto LinearSegment describe un segmento de trayectoria lineal del TCP (punto central de la herramienta) hacia las coordenadas deseadas. Además, incluye información sobre la velocidad lineal deseada, la aceleración lineal y el radio de mezcla al inicio del segmento. Este radio de mezcla permite una desviación de la trayectoria lineal para asegurar una transición suave entre segmentos.

__init__(*linear_segment*)

Crear un objeto LinearSegment.

Parámetros:

- **coordinates** (*list, dict, or Coordinates*) – Coordenadas objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define una velocidad lineal, se utilizará el valor global de velocidad lineal.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define una aceleración lineal, se utilizará el valor global de aceleración lineal.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define un radio de mezcla, se utilizará 0.0.

Constructor:

Crear un objeto LinearSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

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

***property* coordinates**

Las coordenadas objetivo de este segmento, que es un objeto Coordinates que contiene los valores de posición y orientación.

Devuelve:

Las coordenadas objetivo de este segmento.

Tipo del valor devuelto:

Coordinates

***property* blend_radius**

El radio de mezcla al inicio de este segmento.

Devuelve:

El radio de mezcla de inicio.

Tipo del valor devuelto:

float

***property* linear_acceleration**

La aceleración lineal deseada de este segmento. Si es None, se usa el valor global de aceleración lineal.

Devuelve:

La aceleración lineal deseada.

Tipo del valor devuelto:

float or None

***property* linear_velocity**

La velocidad lineal deseada de este segmento. Si es None, se usa el valor global de velocidad lineal.

Devuelve:

La velocidad lineal deseada.

Tipo del valor devuelto:

float or None

class CircularSegment

El objeto CircularSegment describe un movimiento circular del TCP (punto central de la herramienta) a través de la posición intermedia hacia las coordenadas deseadas. La orientación de la pose intermedia está definida por el algoritmo. Además, incluye información sobre la velocidad lineal deseada, la aceleración lineal y el radio de mezcla al inicio del segmento. Este radio de mezcla permite una desviación de la trayectoria circular para asegurar una transición suave entre segmentos.

__init__(*circular_segment*)

Crear un objeto CircularSegment.

Parámetros:

- **intermediate_position** (*list*) – Una posición intermedia [x, y, z] en el camino hacia las coordenadas objetivo. La posición intermedia determina la curvatura del movimiento circular.
- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto CircularSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

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

***property* intermediate_position**

La posición intermedia de este segmento. Incluye las coordenadas x, y, z: [x, y, z]

Devuelve:

La posición intermedia de este segmento.

Tipo del valor devuelto:

list

***property* coordinates**

Las coordenadas objetivo de este segmento, que es un objeto Coordinates que contiene los valores de posición y orientación.

Devuelve:

Las coordenadas objetivo de este segmento.

Tipo del valor devuelto:

Coordinates

***property* blend_radius**

El radio de mezcla al inicio de este segmento.

Devuelve:

El radio de mezcla de inicio.

Tipo del valor devuelto:

float

***property* linear_acceleration**

La aceleración lineal deseada de este segmento. Si es None, se usa el valor global de aceleración lineal.

Devuelve:

La aceleración lineal deseada.

Tipo del valor devuelto:

float or None

***property* linear_velocity**

La velocidad lineal deseada de este segmento. Si es None, se usa el valor global de velocidad lineal.

Devuelve:

La velocidad lineal deseada.

Tipo del valor devuelto:

float or None

class OrientationSegment

El objeto OrientationSegment describe un segmento de trayectoria de orientación pura donde el TCP (punto central de la herramienta) rota según la orientación deseada mientras mantiene su posición constante. Además, incluye información sobre la velocidad angular deseada, aceleración angular y el radio de mezcla al inicio del segmento. Si el radio de mezcla es mayor que 0, la rotación del TCP comienza antes de alcanzar la posición objetivo.

__init__(*orientation_segment*)

Crear un objeto OrientationSegment.

Parámetros:

- **orientation** (*list*) – Orientación objetivo. Tenga en cuenta que la posición permanece constante.
- **angular_velocity** (*float or None*) – Velocidad angular deseada [°/s]. Si no se define velocidad angular, se utilizará el valor de velocidad angular global.
- **angular_acceleration** (*float or None*) – Aceleración angular deseada [°/s^2]. Si no se define aceleración angular, se utilizará el valor de aceleración angular global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Un radio de mezcla mayor a 0 permite que el punto central de la herramienta (TCP) rote antes de alcanzar la posición de este segmento. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto OrientationSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

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

***property* orientation**

La orientación objetivo de este segmento. Se da como ángulos náuticos [roll, pitch, yaw] en grados

Devuelve:

La orientación objetivo de este segmento.

Tipo del valor devuelto:

list

***property* angular_velocity**

La velocidad angular deseada de este segmento. Si es None, se utiliza el valor global de velocidad angular.

Devuelve:

La velocidad angular deseada.

Tipo del valor devuelto:

float or None

***property* angular_acceleration**

La aceleración angular deseada de este segmento. Si es None, se utiliza el valor global de aceleración angular.

Devuelve:

La aceleración angular deseada.

Tipo del valor devuelto:

float or None

***property* blend_radius**

El radio de mezcla al inicio de este segmento.

Devuelve:

El radio de mezcla de inicio.

Tipo del valor devuelto:

float

class CoordinatesSegment

El objeto CoordinatesSegment describe un movimiento lineal en el espacio de articulaciones hacia las coordenadas objetivo. El TCP (punto central de la herramienta) generalmente no se mueve linealmente. Además, incluye información sobre la velocidad lineal deseada, aceleración lineal y radio de mezcla al inicio del segmento. Este radio de mezcla permite desviarse del camino para garantizar una transición suave entre segmentos.

__init__(*coordinates_segment*)

Crear un objeto CoordinatesSegment.

Parámetros:

- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto CoordinatesSegment definiendo todos los parámetros:

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

o usando los valores predeterminados:

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

***property* coordinates**

Las coordenadas objetivo de este segmento, que es un objeto Coordinates que contiene los valores de posición y orientación.

Devuelve:

Las coordenadas objetivo de este segmento.

Tipo del valor devuelto:

Coordinates

***property* blend_radius**

El radio de mezcla al inicio de este segmento.

Devuelve:

El radio de mezcla de inicio.

Tipo del valor devuelto:

float

***property* linear_acceleration**

La aceleración lineal deseada de este segmento. Si es None, se usa el valor global de aceleración lineal.

Devuelve:

La aceleración lineal deseada.

Tipo del valor devuelto:

float or None

***property* linear_velocity**

La velocidad lineal deseada de este segmento. Si es None, se usa el valor global de velocidad lineal.

Devuelve:

La velocidad lineal deseada.

Tipo del valor devuelto:

float or None

class PoseSegment

El objeto PoseSegment describe un movimiento lineal en el espacio de articulaciones hacia la pose objetivo. El TCP (punto central de la herramienta) generalmente no se mueve linealmente. Además, incluye información sobre la velocidad lineal deseada, aceleración lineal y radio de mezcla al inicio del segmento. Este radio de mezcla permite desviarse del camino para garantizar una transición suave entre segmentos.

__init__(*pose_segment*)

Crear un objeto PoseSegment.

Parámetros:

- **pose** (*Pose*) – Pose objetivo.
- **linear_velocity** (*float or None*) – Velocidad lineal deseada [mm/s]. Si no se define velocidad lineal, se utilizará el valor de velocidad lineal global.
- **linear_acceleration** (*float or None*) – Aceleración lineal deseada [mm/s^2]. Si no se define aceleración lineal, se utilizará el valor de aceleración lineal global.
- **blend_radius** (*float or None*) – Radio de mezcla al inicio del segmento [mm]. Si no se define radio de mezcla, se utilizará 0.0.

Constructor:

Creando un objeto PoseSegment definiendo todos los parámetros:

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

o usando los valores predeterminados y la pose de la base de datos:

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

***property* pose**

La pose objetivo de este segmento, que es un objeto Pose que contiene los valores de posición, orientación y configuración.

Devuelve:

La pose objetivo de este segmento.

Tipo del valor devuelto:

Pose

***property* blend_radius**

El radio de mezcla al inicio de este segmento.

Devuelve:

El radio de mezcla de inicio.

Tipo del valor devuelto:

float

***property* linear_acceleration**

La aceleración lineal deseada de este segmento. Si es None, se usa el valor global de aceleración lineal.

Devuelve:

La aceleración lineal deseada.

Tipo del valor devuelto:

float or None

***property* linear_velocity**

La velocidad lineal deseada de este segmento. Si es None, se usa el valor global de velocidad lineal.

Devuelve:

La velocidad lineal deseada.

Tipo del valor devuelto:

float or None

class Path

El objeto Path define cómo se mueve el robot entre dos o más poses. Un camino continuo se define únicamente en el espacio de la herramienta mediante múltiples segmentos, como lineales o circulares. Un camino se llama continuo sin interrupciones si las soluciones discretas del camino no requieren que el robot se reposicione (cambio de configuración). Un camino se llama continuo si la aceleración es continua a lo largo del camino. Un camino comienza en una pose específica (en lugar de coordenadas). Los segmentos del camino generalmente definen coordenadas, ya que la configuración del camino se define por la pose inicial.

__init__(*path*)

Crear un objeto Path, ya sea cargando un camino guardado por nombre o ID.

Parámetros:

path (*str, int, or Path*) –

Referencia al camino a cargar, que puede ser uno de los siguientes:

- el nombre (str) de una trayectoria guardada en la base de datos
- el ID (int) de una trayectoria guardada en la base de datos
- un objeto Path (copiando otro objeto Path)

Constructor:

Cargar un camino desde la base de datos.

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

Crear un objeto Path que contenga la pose inicial especificada y los segmentos de camino definidos usando la función `create_path`. Tenga en cuenta que los segmentos también se pueden agregar posteriormente.

```
path = create_path(initial_pose, segments)
```

***property* initial_pose**

La pose inicial del robot del camino.

Devuelve:

Pose inicial del robot.

Tipo del valor devuelto:

Pose

***property* segments**

Los segmentos incluidos en el camino. Tenga en cuenta que los segmentos no tienen que ser todos del mismo tipo.

Devuelve:

Segmentos incluidos.

Tipo del valor devuelto:

list of Segments (LinearSegment, CircularSegment, OrientationSegment, CoordinatesSegment, PoseSegment)

add_segment(*segment*)

Agrega un segmento al final del camino.

Parámetros:

segment (*LinearSegment*, *CircularSegment*, *OrientationSegment*, *CoordinatesSegment*, *PoseSegment*) – Segmento a agregar.

Ejemplos:

Un segmento se puede agregar de la siguiente manera:

```
# 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*)

Agrega un segmento lineal al final del camino.

Parámetros:

- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo del camino.
- **linear_velocity** (*float*) – Velocidad lineal máxima permitida de la herramienta [mm/s].
- **linear_acceleration** (*float*) – Aceleración lineal máxima permitida de la herramienta [mm/s^2].
- **blend_radius** (*float*) – Radio de mezcla permitido [mm] al inicio del segmento.

Ejemplos:

Un segmento lineal se puede agregar de la siguiente manera:

```
# 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]))
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
# 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)
```

El primer segmento define un movimiento lineal hacia las coordenadas objetivo especificadas con una velocidad máxima de 500 mm/s y una aceleración máxima de 1000 mm/s^2. Tenga en cuenta que estas coordenadas no se alcanzan exactamente, ya que el segmento siguiente define un radio de mezcla de 100 mm, lo que permite una desviación de esas coordenadas para asegurar una transición más suave y eficiente. El segundo segmento luego agrega un movimiento lineal hacia las siguientes coordenadas con una velocidad máxima de 250 mm/s y una aceleración máxima de 500 mm/s^2.

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

Agrega un segmento circular al final del camino.

Parámetros:

- **intermediate_position** (*list*) – Una posición intermedia ([x, y, z]) en el camino hacia las coordenadas objetivo. La posición intermedia determina la curvatura del movimiento circular.
- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo del camino.
- **linear_velocity** (*float*) – Velocidad lineal máxima permitida de la herramienta [mm/s].
- **linear_acceleration** (*float*) – Aceleración lineal máxima permitida de la herramienta [mm/s^2].
- **blend_radius** (*float*) – Radio de mezcla permitido [mm] al inicio del segmento.

Ejemplos:

Un segmento circular se puede agregar de la siguiente manera:

```
# 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]))
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
# 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)
```

El primer segmento define un movimiento circular a través de la posición intermedia especificada hacia las coordenadas objetivo con una velocidad máxima de 500 mm/s y una aceleración máxima de 1000 mm/s^2. Tenga en cuenta que estas coordenadas no se alcanzan

exactamente, ya que el segmento consecutivo define un radio de mezcla de 100 mm, lo que permite una desviación de dichas coordenadas para asegurar una transición más suave y eficiente. El segundo segmento luego agrega un movimiento circular a través de la siguiente posición intermedia hacia las siguientes coordenadas objetivo con una velocidad máxima de 250 mm/s y una aceleración máxima de 500 mm/s^2.

add_orientation(*coordinates*, *angular_velocity=None*, *angular_acceleration=None*, *blend_radius=None*)

Agrega un segmento de orientación al final de la ruta.

Parámetros:

- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo del camino.
- **angular_velocity** (*float*) – Velocidad angular máxima permitida de la herramienta [°/s].
- **angular_acceleration** (*float*) – Aceleración angular máxima permitida de la herramienta [°/s²].
- **blend_radius** (*float*) – Radio de mezcla permitido [mm] al inicio del segmento. Un valor mayor a 0 significa que puede comenzar a rotar antes de alcanzar la posición deseada de este segmento.

Ejemplos:

Se puede agregar un segmento de orientación de la siguiente manera:

```
# add an orientation segment
path.add_orientation([150.0, 0.0, -90.0])
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
# 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)
```

El primer segmento define una rotación hacia la orientación objetivo especificada con una velocidad máxima de 250 °/s y una aceleración máxima de 500 °/s². Tenga en cuenta que, al ejecutar esta ruta, esta primera rotación se omitirá ya que el radio de mezcla del segmento consecutivo es mayor que 0 (ver `OrientationSegment.__init__`) y se ejecutará directamente el segundo segmento. Este segmento define una rotación hacia la orientación objetivo de [150.0, 0.0, -90.0] con una velocidad máxima de 500 °/s y una aceleración máxima de 1000 °/s². En cambio, si el radio de mezcla del segundo segmento se estableciera en 0.0, ambas rotaciones se ejecutarían en secuencia.

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

Agrega un segmento de coordenadas al final de la ruta.

Parámetros:

- **coordinates** (*list, dict or Coordinates*) – Coordenadas objetivo del camino.
- **linear_velocity** (*float*) – Velocidad lineal máxima permitida de la herramienta [mm/s].
- **linear_acceleration** (*float*) – Aceleración lineal máxima permitida de la herramienta [mm/s²].
- **blend_radius** (*float*) – Radio de mezcla permitido [mm] al inicio del segmento.

Ejemplos:

Se puede agregar un segmento de coordenadas de la siguiente manera:

```
# 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]))
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
# 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)
```

El primer segmento define un movimiento hacia las coordenadas objetivo especificadas con una velocidad máxima de 500 mm/s y una aceleración máxima de 1000 mm/s². Tenga en cuenta que estas coordenadas no se alcanzan exactamente, ya que el segmento consecutivo define un radio de mezcla de 100 mm, lo que permite una desviación de dichas coordenadas para garantizar una transición más suave y eficiente. El segundo segmento luego agrega un movimiento hacia las siguientes coordenadas con una velocidad máxima de 250 mm/s y una aceleración máxima de 500 mm/s².

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

Agrega un segmento de pose al final de la ruta.

Parámetros:

- **pose** (*Pose*) – Coordenadas objetivo del camino.
- **linear_velocity** (*float*) – Velocidad lineal máxima permitida de la herramienta [mm/s].
- **linear_acceleration** (*float*) – Aceleración lineal máxima permitida de la herramienta [mm/s^2].
- **blend_radius** (*float*) – Radio de mezcla permitido [mm] al inicio del segmento.

Ejemplos:

Se puede agregar un segmento de pose de la siguiente manera:

```
# 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)
```

En estos ejemplos, no se especificaron parámetros opcionales. Por lo tanto, se utilizan los parámetros globales. Los parámetros también pueden definirse explícitamente en la llamada a la función:

```
# 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)
```

El primer segmento define un movimiento hacia la pose objetivo especificada con una velocidad máxima de 500 mm/s y una aceleración máxima de 1000 mm/s^2. Tenga en cuenta que esta pose no se alcanza exactamente, ya que el segmento consecutivo define un radio de mezcla de 100 mm, lo que permite una desviación de las coordenadas correspondientes para garantizar una transición más suave y eficiente. El segundo segmento luego agrega un movimiento hacia la siguiente pose con una velocidad máxima de 250 mm/s y una aceleración máxima de 500 mm/s^2.

class Socket

El Socket es un objeto que se devuelve mediante las funciones `create_tcp_client`, `create_tcp_server` y `create_udp_socket`. Use este objeto para enviar y recibir datos a través del socket.

receive(*timeout*=*None*)

Recibiendo un mensaje a través de este socket.

Parámetros:

timeout (*None or float*) – El tiempo máximo de espera para un mensaje. Si no se especifica, espera indefinidamente.

Devuelve:

El mensaje recibido, o una cadena vacía si ocurrió un tiempo de espera.

Tipo del valor devuelto:

str

Ejemplos:

Esperando una respuesta durante 3 segundos y mostrándola cuando se reciba.

```
print(socket.receive(timeout=3.0))
```

send(*message*)

Enviando un mensaje a través de este socket.

Parámetros:

message (*str*) – El mensaje a enviar.

Ejemplos:

Enviar un mensaje a todos los pares conectados.

```
socket.send("Hello world!")
```

close()

Cerrando la conexión del socket.

Ejemplos:

Cerrar el socket.

```
socket.close()
```

Funciones del complemento

Funciones del Add-on Cognex

class Cognex

El módulo Cognex soporta el uso de cámaras Cognex. Tenga en cuenta que configurar la cámara Cognex requiere la instalación de Cognex In-Sight Explorer.

Para obtener más información sobre cómo configurar y conectar correctamente su cámara Cognex con el robot, consulte el manual del usuario.

Todas las funciones de este add-on se llaman usando el prefijo `cognex.`

connect_to_camera(*ip*, *user*='*admin*', *password*='')

Configurando un cliente TCP que se conecta a la cámara Cognex conectada a través del puerto ISE para comandos nativos.

Parámetros:

- **ip** (*str*) – La dirección IP de la cámara, p.ej. `'10.0.0.210'`.
- **user** (*str*) – El nombre de usuario para iniciar sesión en la cámara (como se estableció previamente en Cognex In-Sight Explorer). El nombre de usuario predeterminado es `'admin'`.
- **password** (*str*) – La contraseña para iniciar sesión en la cámara (como se estableció previamente en Cognex In-Sight Explorer). La contraseña predeterminada es una cadena vacía.

Muestra:

MinorError – si las credenciales son incorrectas, la conexión se aborta durante la conexión o la conexión tardó demasiado.

Ejemplos:

Conéctese a una cámara recién entregada con la IP y credenciales predeterminadas.

```
cognex.connect_to_camera("10.0.0.210")
```

Conéctese a una cámara con dirección IP y credenciales modificadas previamente.

```
cognex.connect_to_camera("192.168.80.235", user="Developer",
password="my_password_1234")
```

trigger_camera(*ip*)

Configurando el evento de activación de la cámara («SW8») que toma una nueva imagen y devuelve los datos procesados por el trabajo activo de Cognex.

Parámetros:

ip (*str*) – La dirección IP de la cámara, p.ej. `'10.0.0.210'`.

Devuelve:

El mensaje de respuesta de la cámara, que contiene las entidades detectadas.

Tipo del valor devuelto:

str

change_job(*ip*, *job_name*)

Cambiando el trabajo de Cognex mediante el comando nativo de la cámara «LF», que carga un archivo de trabajo ya almacenado en la cámara. Tenga en cuenta que este comando puede tardar varios segundos en ejecutarse.

Parámetros:

- **ip** (*str*) – La dirección IP de la cámara, p.ej. `'10.0.0.210'`.
- **job_name** (*str*) – El nombre del trabajo al que cambiar (como se estableció previamente en Cognex In-Sight Explorer).

Muestra:

MinorError – si la cámara no existe o el cambio de trabajo falló.

Ejemplos:

Cambie a un trabajo almacenado previamente como “CircleDetection.job” y luego tome una foto para detectar círculos en la vista actual de la cámara.

```
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*)

Dispare un evento en la cámara y solicite un mensaje de socket de palabra clave específica en el lado de Cognex.

Parámetros:

- **ip** (*str*) – La dirección IP de la cámara, p.ej. `'10.0.0.210'`.
- **event_id** (*int*) – El ID del evento de Cognex a activar.
- **keyword** (*str*) – La palabra clave a esperar.
- **timeout** (*float or None*) – El tiempo de espera de la solicitud (en [s]). Use `None` para esperar indefinidamente.

Muestra:

TimeOutError – si la cámara no respondió una cadena válida antes de que expirara el tiempo de espera.

Ejemplos:

Solicita el mensaje que contiene la cadena “my_message”, vinculada al evento 1 (SW1) del trabajo Cognex In-Sight.

```
message = cognex.get_camera_message("10.0.0.210", 1, "my_message")
```

disconnect_camera(*ip*)

Desconectando una cámara específica.

Parámetros:

ip (*str*) – La dirección IP de la cámara, p.ej. `'10.0.0.210'`.

Funciones del complemento Modbus

class Modbus

El módulo Modbus contiene funciones para enviar solicitudes a un servidor REST externo.

Todas las funciones de este complemento se llaman usando el prefijo `modbus.`

write_user_register(*address*, *values*)

Escribiendo en los registros de salida Modbus.

Parámetros:

- **address** (*int*) – La dirección del registro donde comenzar a escribir. Los valores posibles van de 0x10 (16) a 0xFF (255).
- **values** (*int or list of int*) – El valor o valores a escribir. Cada registro puede almacenar hasta 2 bytes de datos. El valor máximo que se puede almacenar es 65535 (=2^{16}-1). Si se proporcionan varios valores, se escriben en registros consecutivos.

Ejemplos:

Escribir un único valor en una dirección específica de los registros de salida Modbus.

```
# write 11111 to the address 0x20
modbus.write_user_register(0x20, 11111)
```

Escribir varios valores en registros de salida consecutivos usando un solo comando.

```
# 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*)

Leyendo de los registros de entrada Modbus.

Parámetros:

- **address** (*int*) – La dirección del registro donde comenzar a leer. Los valores posibles van de 0x10 (16) a 0xFF (255).
- **count** (*int*) – El número de registros consecutivos a leer. Por defecto, se lee solo un registro.

Devuelve:

Los valores leídos de los registros. La longitud de esta lista es igual al argumento *count*.

Tipo del valor devuelto:

list of int

Ejemplos:

Leer un único valor de una dirección específica de los registros de entrada Modbus.

```
# read value from the address 0x20
value = modbus.read_user_register(0x20)[0]
```

Leer varios valores de registros de entrada consecutivos usando un solo comando.

```
# read values from the addresses 0x30 and 0x31
values = modbus.read_user_register(0x30, 2)
```

wait_for_event(*index*)

Esperando hasta que se establezca el evento con el índice especificado. Los eventos son establecidos desde el exterior por el cliente Modbus.

Parámetros:

index (*int*) – El índice del evento por el que esperar. Los índices posibles son de 0 a 15.

Ejemplos:

Esperar a que se establezca un evento.

```
modbus.wait_for_event(2)
```

Funciones del complemento REST

class Rest

El módulo REST contiene funciones para enviar solicitudes a un servidor REST externo.

Todas las funciones de este complemento se llaman usando el prefijo `rest.`

setup_client_connection(*target_url*)

Configurando una conexión de cliente para enviar solicitudes a un servidor REST.

Parámetros:

target_url (*str*) – La URL (punto final general de la API) del servidor REST al que conectarse.

Devuelve:

El objeto de conexión de cliente REST.

Tipo del valor devuelto:

RestClientConnection

Ejemplos:

Establecer una conexión con un servidor 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 es un objeto devuelto por `rest.setup_client_connection`. Se utiliza para enviar solicitudes al servidor REST conectado.

get_default_headers(*method*)

Leyendo los encabezados predeterminados que se envían con cada solicitud.

Parámetros:

method (*str*) – El método («get» o «post») para el cual obtener los encabezados.

Devuelve:

Los encabezados predeterminados actualmente establecidos para este método de solicitud.

Tipo del valor devuelto:

dict

Ejemplos:

Leyendo los encabezados predeterminados de los métodos GET y 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*)

Cambiar los encabezados predeterminados que se envían con cada solicitud.

Parámetros:

- **method** (*str*) – El método («get» o «post») para el cual establecer los encabezados.
- **new_headers** (*dict*) – Los nuevos encabezados predeterminados a sobrescribir.

Ejemplos:

Establecer una conexión con un servidor 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*)

Enviando una solicitud GET al servidor REST.

Parámetros:

- **resource** – El nombre del recurso al que acceder. El nombre del recurso se agrega a la URL (<url>/<resource>) y puede opcionalmente contener un carácter "/" al inicio.
- **headers_override** (*dict*) – Establecer los encabezados para esta solicitud individual. Si no se especifica, se usan los encabezados predeterminados.
- **query_params** (*dict*) – Especificando una consulta (si hay alguna).
- **secure_connection** (*bool*) – Indica si se debe usar una conexión segura (verificar certificados) o no.

Devuelve:

El código de respuesta (código HTTP, p.ej. `200` para solicitud exitosa) y la carga de respuesta (datos de respuesta) (si los hay).

Tipo del valor devuelto:

(int, dict)

Ejemplos:

Enviando una solicitud GET a un servidor REST y verificando su código de respuesta.

```
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*)

Enviando una solicitud POST al servidor REST.

Parámetros:

- **resource** (*str*) – El nombre del recurso al que acceder. El nombre del recurso se agrega a la URL (<url>/<resource>) y puede opcionalmente contener un carácter "/" al inicio.
- **data** (*str or dict*) – El cuerpo/datos de la solicitud POST.
- **headers_override** (*dict*) – Establecer los encabezados para esta solicitud individual. Si no se especifica, se usan los encabezados predeterminados.
- **query_params** (*dict*) – Especificando una consulta (si hay alguna).
- **secure_connection** (*bool*) – Indica si se debe usar una conexión segura (verificar certificados) o no.

Devuelve:

El código de respuesta (código HTTP, p.ej. `200` para solicitud exitosa) y la carga de respuesta (datos de respuesta) (si los hay).

Tipo del valor devuelto:

(int, dict)

Ejemplos:

Enviando una solicitud POST a un servidor REST y verificando su código de respuesta.

```
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)
```

Funciones del Add-on ROS

class ROSHandler

El módulo ROS agrega funcionalidades de ROS para otros add-ons y para el acceso directo de los usuarios.

Todas las funciones de este add-on se llaman usando el prefijo `ros_handler.`

ros_node()

El módulo ROS crea su propio nodo ROS, al que se puede acceder directamente mediante esta función. Esto se puede usar para crear objetos como publishers, subscribers, etc., en scripts. Además, `rclpy` se puede importar directamente en aplicaciones para este propósito.

Advertencia

No ejecute `rclpy.spin()` en este nodo, ¡ya que se está ejecutando en segundo plano!

Ejemplos:

Imprime los nombres de todos los topics a los que el robot está publicando.

```
for publisher in ros_handler.ros_node().publishers:
    print(publisher.topic_name)
```

Devuelve:

El nodo ROS del módulo ROS.

Tipo del valor devuelto:

rclpy.node.Node