

機器人程式設計說明書

翻譯版

© Seiko Epson Corporation 2026

Rev.2
TCM268S9130M

目录

• Python 基礎	3
• 核心功能	12
• 附加功能函式	156
◦ Cognex 附加功能函式	157
◦ Modbus 附加功能函式	163
◦ REST 附加功能函式	167
◦ ROS 附加功能函式	173

Python 基礎

基本 Python 標準

本節說明可在應用程式編輯器中使用的一些基本 Python 程式設計知識。這僅是最重要的 Python (3.14) 標準摘要，當然還有更多可用的 Python 內建函式。請注意，機器人應用程式不只是一個 Python 應用程式，因此部分 Python 功能受到限制或無法使用。進階 Python 使用者或需要程式設計教學者，請參閱線上 [Python 3.14 Documentation](#)。

註解

這幾行程式碼代表在執行時不會被執行的程式碼註解。

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

變數定義

Python 支援許多不同的變數型別，且不區分變數定義與變數宣告。

```
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"
```

變數轉換

若數值相容，變數可輕易從一種型別轉換為另一種型別。

```
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
```

資料結構

Python 支援四種不同的資料結構型別：

- List 包含一個有序的元素陣列。

```
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
```

- Tuple 包含一個有序但不可變（無法變更）的元素陣列。

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

- Set 包含一個無序且不含重複項的元素陣列。

```
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
```

- Dictionary 包含鍵與值的無序組合。

```
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
```

函式庫與匯入

Python 提供一組內建模組 (標準函式庫)。支援以下模組：

- 數值運算與亂數 – 適用於計算、數學函式以及產生隨機值

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

- 時間與日期處理 – 處理時間戳記、延遲以及日期 / 時間值

```
import time, datetime
```

- 資料儲存與檔案格式 – 讀取與寫入 JSON 或 CSV 等結構化資料

```
import json, csv
```

- 文字處理 – 字串操作以及使用正規表示式進行模式比對

```
import re, string
```

- 進階迭代與函數式工具 – 高效的迴圈、組合以及高階函式

```
import itertools, functools
```

除標準函式庫外，亦提供以下第三方函式庫：

- 數值計算 – 高效處理陣列與數值資料

```
import numpy
```

- 3D 方向數學 – 四元數表示法與變換（與機器人相關）

```
import pyquaternion
```

- HTTP 通訊 – 向網頁服務與 REST API 發送請求

```
import requests
```

- 設定與資料序列化 – 處理 YAML 格式的資料

```
import yaml
```

以下標準 Python 模組在機器人應用程式中無法使用：

- 作業系統存取（例如 `os`、`shutil`、`subprocess`、`sys`）
- 自訂平行處理（例如 `multiprocessing`、`threading`）
- 自訂檢視與疑難排解（例如 `inspect`、`traceback`）

此外，機器人 Python 環境不支援安裝自訂的第三方函式庫。

基本 Python 函式

必要與選用的函式引數

假設有一個函式以下列引數定義：

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

函式引數 `arg1` 與 `arg2` 是標準引數，呼叫此函式時必須提供。引數 `arg3` 與 `arg4` 具有預設值，因此呼叫此函式時不一定要提供。以下所有函式呼叫皆為有效：

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

解包的函式引數

程式碼文檔中提到的少數函式使用解包引數。當函式的引數數量可能變動時，這些引數就很有用。假設有以下函式引數：

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

那麼這些函式可以下列方式使用：

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

基本 Python 運算子

計算運算子

```
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
```

比較運算子

此處 x 與 y 可為任意型別。

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

此處 x 與 y 為數字。

```
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
```

邏輯運算子

此處 x 與 y 為布林值。

```
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
```

位元運算子

此處 x 與 y 為整數。

```
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
```

條件式程式設計

If-Else 條件

`if`、`elif` (else if)` 與 ``else` 條件可依下列規則組合：

- 當且僅當 `if` 的條件回傳值 `True` 時，才執行其範圍內的程式碼。
- `elif` (else if)` 範圍為選用，當且僅當所有先前的 ``if` 或 `elif` 條件回傳值 `False` 且其自身條件回傳值 `True` 時才會執行。
- `else` 範圍為選用，當且僅當所有先前的 `if` 或 `elif` 條件回傳值 `False` 時才會執行。

一般的 `if-elif-else` 程式碼如下所示：

```
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
    ...
```

範例：

檢查變數 `my_integer` 是否等於 5、10，或兩者皆非。

```

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.")

```

While 迴圈

while 迴圈依下列規則執行與重複：

- 只要 `while` 的條件回傳值 `True`，其迴圈範圍內的程式碼就會重複執行。
- `continue` 關鍵字為選用，可用於跳至下一次迴圈迭代，忽略 `while` 迴圈內剩餘的程式碼。
- `break` 關鍵字為選用，可用於跳出此迴圈。
- `else` 範圍為選用（且很少使用），當迴圈條件轉為 `False` 時執行。換言之，若 `while` 迴圈以 `break` 敘述結束，則不會執行此範圍。

```

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
    ...

```

範例：

列印從 1 到 10 的所有數字。

```

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

```

無限迴圈：當您希望機器人持續重複某項任務，直到您手動停止，或透過特定條件或函式跳出迴圈時，無限迴圈很有用。若迴圈可能以任意高的速度執行，建議加入等待指令，讓處理器有喘息的時間。

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

For 迴圈

for 迴圈依下列規則執行與重複：

- `for` 迴圈範圍內的程式碼會重複執行，對可迭代序列（例如清單、集合、字典或字串）中的每一項各執行一次。
- `continue` 關鍵字為選用，可用於跳至下一次迴圈迭代，忽略 `for` 迴圈內剩餘的程式碼。
- `break` 關鍵字為選用，可用於跳出此迴圈。
- `else` 範圍為選用（且很少使用），在 `for` 迴圈序列中的最後一項處理完畢後執行。換言之，若 `for` 迴圈以 `break` 敘述結束，則不會執行此範圍。

```
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'.
    ...
```

範例：

列印從 1 到 5 的所有數字。

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

列印我最喜歡的水果。

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

數出 `python` 中的字母。

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

核心功能

class Core

備註

這是所有核心功能的集合。這些功能可直接使用，並可在應用程式的程式碼中呼叫。

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

將機器人的一個或多個關節移動到特定的關節角度。此動作會同步所有關節，使其同時到達目標位置。

參數：

- **`joints`** (*int, str, list*) -- 要移動的致動器 ID。請按以下方式指定值：
 - 單一關節的 ID (例如 6)
 - 多個關節的關節 ID 清單 (例如 [1, 4, 6])
 - 常數 `ALL_KIN_JOINTS`，用於移動全部六個關節
- **`joint_position`** (*float, list of float*) -- 移動關節的角度位置，單位為 [deg]。請按以下方式指定值：
 - 單一數值，將所有指定關節移動到相同位置 (例如 -30)
 - 數值清單，將每個指定關節移動到不同位置 (例如 [90, -45, 30])
- **`joint_velocity`** (*float or None*) -- 最大速度，以關節速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 45 百分比)。若未指定值，則使用全域關節速度。
- **`joint_acceleration`** (*float or None*) -- 移動關節的最大加速度，以加速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 60 百分比)。若未指定值，則使用全域關節加速度。
- **`relative`** (*bool*) -- 關節位置是否視為相對於目前關節位置的偏移量。預設使用絕對位置值。
- **`block`** (*bool*) -- 是否等待此動作結束後才繼續執行程式。請注意，只有使用相同關節的同步運動才能以非阻斷式的方式連續執行。

範例：

將機器人所有關節同步移動到 0°，即機器人的直立位置：

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

將關節 6 從目前位置移動 -30°：

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

將關節 2、3 和 5 分別移動到 20° 、 -40° 和 20° 。此運動會使每個關節同時完成移動。此運動將使用最大允許的關節加速度，但僅使用最大允許關節速度的 50%。

```
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)
```

將機器人的 TCP (工具中心點) 移動到特定的笛卡兒坐標。機器人沿時間最佳軌跡移動，在每個關節上產生梯形速度曲線。

參數:

- **coordinates** (*Coordinates, list or dict*) --
TCP 的目標坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **configuration** (*str or None*) -- 目標位姿的配置。這是從 'c1' 到 'c8' 的字串。
- **joint_velocity** (*float or None*) -- 最大速度，以關節速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 45 百分比)。若未指定值，則使用全域關節速度。
- **joint_acceleration** (*float or None*) -- 移動關節的最大加速度，以加速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 60 百分比)。若未指定值，則使用全域關節加速度。
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --

用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：

- 一個 **Coordinates** 物件
- 格式為 `[x, y, z, roll, pitch, yaw]` 的笛卡兒坐標清單
- 格式為 `{"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}` 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

- **relative** (*bool*) -- 坐標是否視為相對於目前坐標的偏移量。預設使用絕對坐標值。
- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。預設此函式為阻斷式。

範例：

目標坐標可以多種形式給出。所有指定的指令都會將 TCP 移動到全域工件坐標系中的位置 $x = 700\text{mm}$ 、 $y = -125\text{mm}$ 、 $z = 500\text{mm}$ ，方向角為 $\text{roll} = 180^\circ$ 、 $\text{pitch} = 0^\circ$ 和 $\text{yaw} = -90^\circ$ 。

```
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]))
```

若未指定所有坐標，其餘坐標將使用目前的 TCP 坐標。例如，僅變更 x 與 roll 分量可如下進行：

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

上述範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

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

此指令在 y 方向額外移動 -100mm ，在 yaw 方向額外旋轉 -60° ，使用最大速度的 50% 和最大加速度。此外，TCP 透過工具偏移在 x 方向平移 20mm 。

定義工具空間中運動的方式有多種，其中之一為：

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

此指令沿工具的 z 方向移動。

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

將機器人的 TCP (工具中心點) 移動到特定位姿。機器人沿時間最佳軌跡移動，在每個關節上產生梯形速度曲線。

參數:

- **pose** (*Pose, str, or int*) --
TCP 的目標位姿。請按以下方式指定值：
 - 一個 Pose 物件
 - 預先定義姿勢的名稱 (已儲存於資料庫中)
 - 預先定義姿勢的 ID (已儲存於資料庫中)
- **joint_velocity** (*float or None*) -- 最大速度，以關節速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 45 百分比)。若未指定值，則使用全域關節速度。
- **joint_acceleration** (*float or None*) -- 移動關節的最大加速度，以加速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 60 百分比)。若未指定值，則使用全域關節加速度。
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單

- 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。
- **block** (bool) -- 是否等待此動作結束後才繼續執行程式。預設此函式為阻斷式。

範例：

移動到資料庫中的某個姿勢：

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

也可以在腳本中使用 `create_pose()` 函式指定自訂位姿：

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

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

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

此指令將機器人移動到資料庫中的 "final_pose"，使用最大關節速度的 45% 和最大關節加速度的 60%。此外，對工件坐標系套用偏移，使目標位姿在 z 方向平移 50mm。

定義工具空間中運動的方式有多種，其中之一為：

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

此指令沿工具的 z 方向移動。

```
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)
```

將機器人的 TCP (工具中心點) 從目前位姿以直線移動到目標位姿。請注意，位置與方向都會沿軌跡線性插值。請注意，只有當路徑上的每個位姿皆可到達時，才能執行此動作。

參數:

- **coordinates** (*Coordinates, list, or dict*) --
TCP 的目標坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **linear_velocity** (*float or None*) -- 機器人所有運動部件的最大線性工具速度 [mm/s]。
- **linear_acceleration** (*float or None*) -- 機器人所有運動部件的最大線性工具加速度 [mm/s²]。
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。

- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。

- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。

- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單

- 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

- **relative** (*bool*) -- 坐標是否視為相對於目前坐標的偏移量。預設使用絕對坐標值。
- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。預設此函式為阻斷式。

範例：

目標坐標可以多種形式給出。所有指定的指令都會將 TCP 線性移動到全域工件坐標系中的位置 $x = 600\text{mm}$ 、 $y = -125\text{mm}$ 、 $z = 500\text{mm}$ ，方向角為 $\text{roll} = 180^\circ$ 、 $\text{pitch} = 0^\circ$ 和 $\text{yaw} = -90^\circ$ 。

```
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]))
```

若未指定所有坐標，其餘坐標將使用目前的 TCP 坐標。例如， x 與 roll 分量可如下變更：

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

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

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

此指令在 x 方向額外移動 100mm ，速度為 500mm/s ，加速度為 1000mm/s^2 。此外，工件坐標系會被給定的坐標覆寫。

定義工具空間中運動的方式有多種，其中之一為：

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

此指令沿工具的 z 方向線性移動。

```
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)
```

將機器人的 TCP (工具中心點) 從目前位姿沿圓形軌跡移動到目標位姿，並經過一個中間位置。請注意，方向會沿軌跡線性插值。請注意，只有當路徑上的每個位姿皆可到達時，才能執行此動作。

參數:

- **intermediate_position** (*list*) -- 通往目標坐標的路徑上的中間位置。中間位置決定圓弧運動的曲率。
- **coordinates** (*Coordinates, list, or dict*) -- TCP 的目標坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **linear_velocity** (*float or None*) -- 機器人所有運動部件的最大線性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 機器人所有運動部件的最大線性工具加速度 [mm/s²]。
- **tool_offset** (*Coordinates, list, dict, or None*) -- 用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) -- 用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) -- 用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --

用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：

- 一個 **Coordinates** 物件
- 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
- 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

- **relative** (*bool*) -- 坐標是否視為相對於目前坐標的偏移量。預設使用絕對坐標值。
- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。預設此函式為阻斷式。

範例：

機器人可以圓弧方式移動到全域工件坐標系中的目標位置 $x = 0\text{mm}$ 、 $y = -500\text{mm}$ 、 $z = 500\text{mm}$ ，方向角為 $\text{roll} = 180^\circ$ 、 $\text{pitch} = 0^\circ$ 和 $\text{yaw} = -90^\circ$ 。運動過程中會經過中間位置 $x = 0\text{mm}$ 、 $y = -500\text{mm}$ 、 $z = 500\text{mm}$ 。

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

若未指定所有坐標，其餘坐標將使用目前的 TCP 坐標。

```
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})
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
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)
```

此指令將 TCP 移動到指定坐標，速度為 500mm/s ，加速度為 1000mm/s^2 。此外，TCP 透過工具偏移在 x 方向平移 20mm 。

```
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)
```

移動 TCP (工具中心點)，使其僅改變方向。機器人隨後依照所需的工具方向線性旋轉，同時保持位置不變。請注意，只有當路徑上的每個位姿皆可到達時，才能執行此動作。

參數：

- **tool_orientation** (*list*) -- 工具的目標方向，以航海角 (roll、pitch、yaw) 表示 [deg]
- **angular_velocity** (*float or None*) -- TCP 的最大角速度 [deg/s]。
- **angular_acceleration** (*float or None*) -- TCP 的最大角加速度 [deg/s²]。
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。

- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。

- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。

- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

- **relative** (*bool*) -- 坐標是否視為相對於目前坐標的偏移量。預設使用絕對坐標值。

- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。預設此函式為阻斷式。

範例：

目標方向可以指定所有角度，或僅指定部分角度而其餘設為 'None'。在後者的情況下，缺少的角度將使用目前的方向。

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

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

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

此指令相當於在 roll 方向進行 -60° 的相對運動，速度為 $250^\circ/\text{s}$ ，加速度為 $3000^\circ/\text{s}^2$ 。此外，工具坐標系會被給定的坐標覆寫。

定義工具空間中運動的方式有多種，其中之一為：

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

此指令繞工具的 z 方向旋轉。

```
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)
```

從目前的機器人位姿開始，將 TCP (工具中心點) 依序移動經過所定義的路徑點。

參數:

- **poses** (*list of str, int or Pose*) -- 路徑點清單。
- **joint_velocity** (*float or None*) -- 最大速度，以關節速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 45 百分比)。若未指定值，則使用全域關節速度。
- **joint_acceleration** (*float or None*) -- 移動關節的最大加速度，以加速度上限的百分比表示。這是單一數值，將所有關節限制在相同的最大值 (例如 60 百分比)。若未指定值，則使用全域關節加速度。
- **blend_radius** (*float or None*) -- 允許的混合半徑 [mm]，定義在路徑點之間過渡時，工具空間中相對於路徑點的最大偏差。
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。

- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。

- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。

- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。請注意，只有使用相同關節的同步運動才能以非阻斷式的方式連續執行。

範例：

路徑點可以不同形式給出 (Pose 物件、資料庫中姿勢的名稱、資料庫中姿勢的 ID)：

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

上述範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

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

此指令沿路徑點移動，允許相對於路徑點的最大偏差為 50mm。它使用最大速度的 50% 和最大加速度。

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

依照所定義的區段移動 TCP (工具中心點)。運動從目前的機器人位姿開始。

參數：

- **segments** (*list*) -- 構成路徑的區段清單。
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 *Coordinates* 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 *Coordinates* 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 *Coordinates* 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 *Coordinates* 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。
- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。請注意，只有使用相同關節的同步運動才能以非阻斷式的方式連續執行。

範例：

區段可以在腳本中使用 *Segment* 物件定義 (`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])
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
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)
```

此指令執行這些區段，並透過工具偏移使 TCP 產生輕微平移。此外，此運動設定為非阻斷式。

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

依照指定的路徑移動 TCP (工具中心點)。首先，機器人會使用預設速度與加速度移動到路徑的初始位姿。接著依照各區段的規格依序執行。

參數：

- **path** (*Path, str, or int*) -- TCP 應遵循的路徑。
- **previous_angles** (*list or None*) --
用於計算起始位姿最接近解的角度。請按以下方式指定值：
 - None，表示使用目前的機器人角度
 - 關節角度清單 [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。
- **block** (*bool*) -- 是否等待此動作結束後才繼續執行程式。請注意，只有使用相同關節的同步運動才能以非阻斷式的方式連續執行。

範例：

執行資料庫中的某條路徑：

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

也可以在腳本中使用 `create_path()` 函式指定自訂路徑：

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

另一種方式是使用 `Path` 物件建立路徑並執行該路徑：

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

run_path(path)
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

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

此指令執行 "fancy_path"，並透過工具坐標系使 TCP 產生輕微平移。此外，此運動設定為非阻斷式。

wait_for_motor(*actuator_ids*=None)

等待一個或多個關節馬達停止運動。

參數:

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

要等待的致動器 ID。請按以下方式指定值：

- 單一關節的 ID (例如 6)
- 多個關節的關節 ID 清單 (例如 [1, 4, 6])
- 多個關節的關節名稱集合 (例如 {"1", "4", "6"})
- 常數 *ALL_KIN_JOINTS* 或 *None* · 用於讀取所有關節

範例：

此範例說明機器人運動期間的時間可用於自訂用途。

```
# 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()

透過向所有已連接的致動器發送停止訊號來停止運動。

範例：

此範例說明可使用 'stop_motion' 函式中止非阻斷式運動。此處我們希望在特定數位輸入切換時中止運動。

```
# 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*)

檢查一個或多個馬達是否正在運動

參數:

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

要檢查的致動器 ID。請按以下方式指定值：

- 單一關節的 ID (例如 6)
- 多個關節的關節 ID 清單 (例如 [1, 4, 6])
- 多個關節的關節名稱集合 (例如 {1, 4, 6})
- 常數 *ALL_KIN_JOINTS* 或 *None*，用於讀取所有關節

回傳:

指定的馬達是否正在運動。可為以下其中一種：

- 一個 bool，若檢查的是單一馬達
- 一個 bool 清單，若檢查的是多個馬達

回傳型別:

bool or list of bool

範例：

此範例說明可使用 'stop_motion' 函式中止非阻斷式運動。此處我們希望在特定數位輸入切換時中止運動。

```
# 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*)

為點對點運動函式所使用的 `joint_velocity` 參數設定新的全域預設值。關節速度值以最大關節速度上限的百分比表示。

參數：

joint_velocity (*float*) -- 全域關節速度，以最大關節速度的 [%] 表示。

範例：

使用相同的速度上限呼叫兩個連續的點對點運動。

```
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*)

為點對點運動函式所使用的 `joint_acceleration` 參數設定新的全域預設值。關節加速度值以最大關節加速度上限的百分比表示。

參數:

joint_acceleration (*float*) -- 全域關節加速度，以最大關節加速度的 [%] 表示。

範例：

使用相同的加速度上限呼叫兩個連續的點對點運動。

```
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*)

為工具運動與路徑規劃函式所使用的 `linear_velocity` 參數設定新的全域預設值。線性速度值限制工具及機器人所有運動部件的最大線性速度。

參數:

linear_velocity (*float*) -- 全域線性速度，單位為 [mm/s]。

範例：

使用相同的速度上限呼叫兩個連續的線性運動。

```
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*)

為工具運動與路徑規劃函式所使用的 `linear_acceleration` 參數設定新的全域預設值。線性加速度值限制工具及機器人所有運動部件的最大線性加速度。

參數:

linear_acceleration (*float*) -- 全域線性加速度，單位為 mm/s^2 。

範例：

使用相同的加速度上限呼叫兩個連續的線性運動。

```
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*)

為旋轉運動函式所使用的 `angular_velocity` 參數設定新的全域預設值。角速度值限制工具的旋轉速度。

參數:

angular_velocity (*float*) -- 全域角速度，單位為 [deg/s]。

範例：

使用相同的速度上限呼叫兩個連續的旋轉運動。

```
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*)

為旋轉運動函式所使用的 `angular_acceleration` 參數設定新的全域預設值。角加速度值限制工具的旋轉加速度。

參數:

angular_acceleration (*float*) -- 全域角加速度，單位為 [deg/s²]。

範例：

使用相同的加速度上限呼叫兩個連續的旋轉運動。

```
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*)

為運動與路徑函式所使用的 `blend_radius` 參數設定新的全域預設值。

參數:

blend_radius (*float*) -- 全域混合半徑，單位為 [mm]。

範例：

使用相同的混合半徑呼叫兩個連續的路徑點運動。

```
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*)

為運動與運動學函式所使用的 `work_frame` 參數設定新的全域預設值。工件坐標系是用於變更機器人參考坐標系的變換，相對於基坐標系給出。

參數：

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

全域工件坐標系，相對於基坐標系給出。請按以下方式指定值：

- 一個 `Coordinates` 物件
- 格式為 `[x, y, z, roll, pitch, yaw]` 的笛卡兒坐標清單
- 格式為 `{"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}` 的笛卡兒坐標字典

範例：

使用相同的工件坐標系呼叫兩個連續的運動。

```
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)

為運動與運動學函式所使用的 `tool_frame` 設定新的全域預設值。工具坐標系是用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。

參數:

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

全域工具坐標系，相對於法蘭坐標系給出。請按以下方式指定值：

- 一個 Coordinates 物件
- 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
- 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

範例：

使用相同的工具坐標系呼叫兩個連續的運動。

```
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)
```

讀取機器人目前的 TCP (工具中心點) 位姿。若未給定坐標系，則相對於全域坐標系讀取位姿。

參數:

- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

機器人目前的 TCP 位姿。

回傳型別:

Pose

範例：

以兩種不同方式讀取位姿的屬性。

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

請注意，目前位姿是從感測器資料讀取的。感測器可能會波動，不一定回傳精確值，因此以下相對於目前位姿來回移動的範例，會隨時間產生非預期的漂移。在這種情況下，建議改用 `get_reference_pose` 函式。

```
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)
```

相對於給定的坐標系讀取機器人 TCP (工具中心點) 目前的坐標。若未指定坐標系，則回傳相對於全域坐標系的坐標。

參數:

- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

機器人目前的 TCP 坐標。

回傳型別:

Coordinates

範例：

在全域坐標系中讀取目前的 TCP 坐標。

```
coordinates = get_current_coordinates()
```

請注意，目前坐標是從感測器資料讀取的。感測器可能會波動，不一定回傳精確值，因此以下相對於目前位姿來回移動的範例，會隨時間產生非預期的漂移。在這種情況下，建議改用 *get_reference_coordinates* 函式。

```
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()

讀取機器人目前位姿配置的名稱。請注意，此配置與目前的全域坐標系無關。

回傳:

目前的配置 (為 'c1' 到 'c8' 其中之一)。

回傳型別:

str

範例 :

以兩種不同方式讀取機器人目前位姿的配置。

```
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)
```

取得機器人的參考 TCP (工具中心點) 位姿。若未給定坐標系，則回傳相對於全域坐標系的位姿。請注意，此位姿是根據發送給驅動器的參考角度計算而得。

參數:

- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

機器人的參考 TCP 位姿。

回傳型別:

Pose

範例：

以兩種不同方式讀取參考位姿的屬性。

```
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)
```

相對於給定的坐標系取得機器人 TCP (工具中心點) 的參考坐標。若未指定坐標系，則回傳相對於全域坐標系的坐標。請注意，這些坐標是根據發送給驅動器的參考角度計算而得。

參數:

- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

機器人的參考 TCP 坐標。

回傳型別:

Coordinates

範例：

在全域坐標系中讀取參考 TCP 坐標。

```
coordinates = get_reference_coordinates()
```

get_reference_configuration()

取得機器人參考位姿配置的名稱。請注意，此配置與參考全域坐標系無關。請注意，此配置是根據發送給驅動器的參考角度計算而得。

回傳:

參考配置 (為 'c1' 到 'c8' 其中之一)。

回傳型別:

str

範例 :

以兩種不同方式讀取機器人參考位姿的配置。

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

create_pose(coordinates, configuration)

根據給定的坐標與配置建立一個 Pose 物件。

參數:

- **coordinates** (*Coordinates*, list, or dict) -- 位姿的 TCP 坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **configuration** (str) -- 位姿的配置。為 'c1' 到 'c8' 其中之一。

回傳:

機器人位姿物件。

回傳型別:

Pose

範例：

以特定配置，從坐標值清單建立一個位姿。

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

get_pose(*pose*)

從資料庫中已儲存的預先定義姿勢讀取位姿物件。

參數:

pose (*Pose, str, or int*) --

要讀取坐標的機器人位姿。可為以下其中一種：

- 一個 Pose 物件
- 資料庫中已儲存姿勢的名稱 (str)
- 資料庫中已儲存姿勢的 ID (int)

回傳:

位姿物件。

回傳型別:

Pose

範例：

從資料庫載入名為 "grasping_pose" 的姿勢。

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

create_path(*initial_pose*, *segments*)

根據給定的初始位姿與區段建立一個 Path 物件。

參數:

- **initial_pose** (*Pose*) -- 機器人的初始位姿。
- **segments** (*list*) -- 構成路徑的區段清單。請注意，區段也可以之後再加入。

回傳:

機器人路徑物件。

回傳型別:

[Path](#)

範例：

從初始位姿與多個區段建立一條路徑。

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

get_path(*path*)

從資料庫中已儲存的預先定義路徑讀取路徑物件。

參數:

path (*Path, str, or int*) --

要取得的機器人路徑。可為以下其中一種：

- 一個 Path 物件
- 資料庫中已儲存路徑的名稱 (str)
- 資料庫中已儲存路徑的 ID (int)

回傳:

路徑物件。

回傳型別:

Path

範例：

從資料庫載入名為 "grasping_path" 的路徑。

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

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

建立一個 LinearSegment 物件。

參數:

- **coordinates** (*list, dict, or Coordinates*) -- 目標坐標。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 LinearSegment 物件：

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

或使用預設值：

```
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)
```

建立一個 CircularSegment 物件。

參數:

- **intermediate_position** (*list*) -- 通往目標坐標的路徑上的中間位置 [x, y, z]。中間位置決定圓弧運動的曲率。
- **coordinates** (*list, dict or [Coordinates](#)*) -- 目標坐標。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 CircularSegment 物件：

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

或使用預設值：

```
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)
```

建立一個 OrientationSegment 物件。

參數：

- **orientation** (*list*) -- 目標方向。請注意，位置保持不變。
- **angular_velocity** (*float or None*) -- 所需的角速度 [deg/s]。若未定義角速度，則使用全域角速度值。
- **angular_acceleration** (*float or None*) -- 所需的角加速度 [deg/s²]。若未定義角加速度，則使用全域角加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。混合半徑大於 0 時，工具中心點 (TCP) 可在到達此區段的位置之前開始旋轉。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 OrientationSegment 物件：

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

或使用預設值：

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

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

建立一個 CoordinatesSegment 物件。

參數：

- **coordinates** (*list, dict or Coordinates*) -- 目標坐標。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 CoordinatesSegment 物件：

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

或使用預設值：

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

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

建立一個 PoseSegment 物件。

參數:

- **pose** (*Pose*) -- 目標位姿。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 PoseSegment 物件：

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

或使用預設值與資料庫中的姿勢：

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

get_coordinates(*pose*)

讀取指定位姿的坐標。請注意，這些坐標與目前的全域坐標系無關。

參數:

pose (*Pose, str, or int*) --

要讀取坐標的機器人位姿。可為以下其中一種：

- 一個 Pose 物件
- 資料庫中已儲存姿勢的名稱 (str)
- 資料庫中已儲存姿勢的 ID (int)

回傳:

此位姿的坐標。

回傳型別:

Coordinates

範例：

讀取先前以名稱 "grasping_pose" 儲存於資料庫中的姿勢坐標。

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

讀取先前以 ID 3 儲存於資料庫中的姿勢坐標。

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

get_configuration(*pose*)

讀取特定位姿的配置。

參數:

pose (*Pose, str, or int*) --

要讀取配置的位姿。請按以下方式指定值：

- 一個 Pose 物件
- 預先定義姿勢的名稱 (已儲存於資料庫中)
- 預先定義姿勢的 ID (已儲存於資料庫中)

回傳:

此位姿的配置 (為 'c1' 到 'c8' 其中之一)。

回傳型別:

str

範例：

讀取以名稱 "grasping_pose" 儲存於資料庫中的姿勢配置。

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

以兩種不同方式從 Pose 物件讀取配置。

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

parse_coordinates(coordinates)

建立一個 Coordinates 物件。

參數:

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

一組特定的坐標，可為以下其中一種：

- 形式為 [x, y, z, roll, pitch, yaw] 的清單，位置值單位為 [mm]，方向值單位為 [deg]
- 形式為 {'x': x, 'y': y, 'z': z, 'roll': roll, 'pitch': pitch, 'yaw': yaw} 的字典，位置值單位為 [mm]，方向值單位為 [deg]
- 一個 Coordinates 物件 (複製另一個 Coordinates 物件)

建構函式：

從坐標值建立一個 Coordinates 物件。

```
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)
```

利用相對工具坐標系與相對工件坐標系之間的差異，在不同坐標系之間變換機器人坐標。若相對工具坐標系與相對工件坐標系皆未指定，則不進行變換，回傳未變更的坐標物件。

參數：

- **coordinates** (*Coordinates, list, or dict*) --
要變換的 TCP 坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **relative_tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **relative_work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳：

變換後的機器人坐標。

回傳型別：

Coordinates

範例：

將一組坐標變換到另一個工件坐標系。若工件坐標系的原點恰好位於要變換的坐標處，則變換會回傳一個所有值皆為零的坐標物件。

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

get_grid_points(*grid_name*)

讀取先前儲存於資料庫中某個特定網格的所有網格點位置。

參數:

grid_name (*str*) -- 要擷取點位的網格名稱。

回傳:

網格點清單，依網格屬性中定義的迭代順序排列。

回傳型別:

list

範例：

移動到預先定義網格中的所有點位。

```
# 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)
```

正向運動學是機器人運動學中的概念，用於從給定的一組關節角度計算 TCP（工具中心點）的位姿。此函式的反向操作是逆運動學函式，它根據機器人的 TCP 位姿計算關節角度。運動學計算通常在機器人內建的基坐標系與法蘭坐標系之間進行。不過使用者可以透過定義工件坐標系與工具坐標系來變更參考坐標系。預設使用全域定義的工件坐標系與工具坐標系。

參數:

- **joint_angles** (*list of float*) -- 六個關節角度的清單，單位為 [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

機器人位姿物件。

回傳型別:

Pose

範例：

在奇異點上，多組關節角度會產生相同的 TCP 位姿。在此範例中，兩組關節角度得到完全相同的位姿。

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

以兩種不同方法讀取目前的 TCP 位姿。

```
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)
```

逆運動學是機器人運動學中的概念，用於從 TCP (工具中心點) 的位姿計算一組關節角度。此函式的反向操作是 正向運動學 函式，它根據機器人的關節角度計算 TCP 位姿。請注意，逆運動學通常有多組解，但只會回傳其中一組。

參數：

- **pose** (*Pose, str, or int*) --
要計算對應關節角度的位姿。請按以下方式指定值：
 - 一個 Pose 物件
 - 預先定義姿勢的名稱 (已儲存於資料庫中)
 - 預先定義姿勢的 ID (已儲存於資料庫中)
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
 預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

六個關節角度的清單，單位為 [deg]

回傳型別:

list of float

範例：

計算先前儲存於資料庫中某個姿勢的關節角度。

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

計算以特定配置到達特定坐標的關節角度。

```
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)
```

此函式計算最接近目前機器人位姿或最接近特定位姿的 **逆運動學解** (詳見 [inverse_kinematics](#) 函式)。

參數:

- **coordinates** (*Coordinates, list, or dict*) --
要計算對應關節角度的 TCP 坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **previous_angles** (*list or None*) --
用於計算最接近解的角度。請按以下方式指定值：
 - None · 表示使用目前的機器人角度
 - 關節角度清單 [deg]
- **tool_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** (*Coordinates, list, dict, or None*) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單

- 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

六個關節角度的清單，單位為 [deg]

回傳型別:

list of float

範例：

逆運動學可用於簡單檢查特定的一組坐標是否有運動學解。

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

若要取得同一位姿的多組解，可相應變更前述的位姿敘述。在此範例中，計算了直立機器人略微向前並向下平移後的兩組解。

```
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)
```

此函式計算最接近目前機器人位姿或最接近特定位姿的 逆運動學解 (詳見 [inverse_kinematics](#) 函式)。

參數:

- **coordinates** ([Coordinates](#), list, or dict) --
要計算對應關節角度的 TCP 坐標。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典
- **tool_offset** ([Coordinates](#), list, dict, or None) --
用於平移機器人 TCP 的變換，相對於工具坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工具偏移，因此 TCP 位於工具坐標系的原點。
- **tool_frame** ([Coordinates](#), list, dict, or None) --
用於變更機器人 TCP 的變換，相對於法蘭坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工具坐標系。若工具坐標系僅包含零坐標，則與法蘭坐標系重合。
- **work_offset** ([Coordinates](#), list, dict, or None) --
用於平移機器人參考點的變換，相對於工件坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設沒有工件偏移，因此參考點位於工件坐標系的原點。
- **work_frame** ([Coordinates](#), list, dict, or None) --
用於變更機器人參考點的變換，相對於基坐標系給出。請按以下方式指定值：
 - 一個 Coordinates 物件
 - 格式為 [x, y, z, roll, pitch, yaw] 的笛卡兒坐標清單
 - 格式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的笛卡兒坐標字典

預設套用全域工件坐標系。若工件坐標系僅包含零坐標，則與基坐標系重合。

回傳:

六個關節角度的清單及其配置，格式為 {'c1': [...], 'c2': [...], ...}，單位為 [deg]

回傳型別:

dict

範例：

在此範例中，機器人移動到所有可到達相同 TCP 坐標的關節角度組合。

```
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)
```

若已知一組可到達目標位姿的關節角度，即可用此方式計算所有其他到達相同位姿但關節配置不同的解。

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

wait(seconds)

等待並因此主動阻斷後續程式碼的執行一段指定的時間。

參數：

seconds (*float*) -- 等待的時間長度，單位為 [s]。

範例：

此範例將機器人移動到預先定義的位姿，並在各次運動之間等待 3 [s]。

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

此範例會不斷重複您的某段程式碼。為確保執行速度受到限制，程式碼會在每次迭代結束時等待 100 [ms]。

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

此範例與上述範例類似，但不是等待固定的時間，而是動態選擇等待時間，使您的程式碼能以每秒固定的間隔執行。

```
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)

在主應用程式內執行一個（阻斷式）應用程式。

參數:

- **script_name** (*str*) -- 要執行的應用程式名稱。若應用程式位於某個資料夾中，請使用應用程式的完整路徑（例如 *folder/subfolder/app*）
- **arguments** (*dict*) -- 傳遞給所執行應用程式的引數。您可以在被呼叫的應用程式中使用內建的 *script_arguments* 變數讀取這些引數。

範例：

假設我們有一個用於量測環境溫度的獨立應用程式。此範例每分鐘重複呼叫該應用程式一次。

```
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)

與主應用程式並行執行一個（非阻斷式）應用程式。請注意，主應用程式會立即繼續執行，且所有並行應用程式會在主應用程式終止時中止。

參數：

script_name (*str*) -- 要啟動的並行應用程式名稱。若應用程式位於某個資料夾中，請使用應用程式的完整路徑（例如 *folder/subfolder/app*）

範例：

此範例觸發一個並行應用程式，用於監控某個溫度感測器，以確保機器人僅在特定溫度範圍內運作。主應用程式持續執行，並在不再滿足運作溫度條件時中止。

```
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()

停止目前正在執行的應用程式。這包括主應用程式，以及所有由主應用程式觸發的應用程式。請注意，正在執行的 *background* 應用程式會繼續執行。

引發:

ScriptInterrupt -- 若從使用 *run_script* 函式觸發的應用程式中執行此函式，則會引發至主應用程式。

範例：

此程式碼範例可用於 *background* 應用程式中，以暫時暫停並手動恢復正在執行的主應用程式。

```
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()

暫停目前正在執行的應用程式。這包括主應用程式，以及所有由主應用程式觸發的應用程式。請注意，正在執行的 *background* 應用程式會繼續執行。

範例：

請參閱 *stop_application* 的綜合範例，其中使用了此函式。

resume_application()

恢復目前暫停中的應用程式。由於此函式僅能在 *paused* 狀態下執行，因此只能從 *background* 應用程式中使用。

範例：

請參閱 *stop_application* 的綜合範例，其中使用了此函式。

stop_and_release()

停止目前正在執行的應用程式，並釋放所有關節（啟用重力補償模式）。請注意，使用者必須手動鎖定所有關節（停用重力補償模式），才能繼續執行後續的應用程式。

is_status(status)

檢查程式狀態是否符合給定的狀態。

參數:

status (*str or list of str*) --

要檢查的程式狀態，可為以下其中一個或多個：

- 'ready' (機器人閒置中，等待接收指令)
- 'processing' (機器人正在處理資料，可能需要一段時間)
- 'hand_guidance_control' (機器人處於使能控制模式)
- 'running' (機器人正在執行應用程式或運動)
- 'paused' (機器人正在暫停應用程式或運動)
- 'emergency_stop' (已觸發急停)
- 'normal_stop' (已觸發一般停止)
- 'protective_stop' (已觸發防護停止)
- 'error' (發生錯誤或安全違規)
- 'position_limit' (發生安全限制位置違規)
- 'safeguard' (已觸發安全防護中斷)
- 'collision' (已觸發碰撞中斷)
- 'proceed' (機器人正在中斷後繼續運行)
- 'stopping' (機器人正在停止其程式)

回傳:

程式狀態是否符合給定的狀態 (或為給定狀態其中之一)

回傳型別:

bool

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

建立一個帶按鈕的對話框。此對話框會阻斷程式流程，直到使用者以所提供的選項之一回應對話框為止。

參數：

- **text** (*str*) -- 向使用者顯示的訊息（或問題）。
- **options** (*list of str*) -- 可供選擇的選項，即各按鈕上的標籤。
- **title** (*str*) -- 對話框彈出視窗的標題。
- **dialog_type** (*int*) -- 要向使用者顯示的對話框類型。可為以下其中一種：
 - 0：資訊 / 藍色
 - 1：成功 / 綠色
 - 2：警告 / 橘色
 - 3：錯誤 / 紅色
- **dialog_value** (*str*) -- 若對話框未獲回應時要回傳的預設值。
- **mandatory** (*bool*) -- 是否必須回應對話框，程式流程才能繼續。若設為 *True*，則無法透過關閉彈出視窗等方式忽略對話框，且會持續阻斷直到收到回應為止。

回傳：

所選的選項，即使用者按下的按鈕上的標籤。

回傳型別：

str

範例：

此範例說明如何建立一個含有 "Continue" 按鈕的簡單資訊彈出視窗。按下按鈕後，程式將繼續執行其程式碼。

```
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)
```

建立一個以下拉式選單顯示選項的對話框。此對話框會阻斷程式流程，直到使用者選擇並確認其中一個選項為止。

參數：

- **text** (*str*) -- 向使用者顯示的訊息（或問題）。
- **options** (*list of str*) -- 可供選擇的選項，即下拉式選單各選項的標籤。
- **title** (*str*) -- 對話框彈出視窗的標題。
- **dialog_type** (*int*) -- 要向使用者顯示的對話框類型。可為以下其中一種：
 - 0：資訊 / 藍色
 - 1：成功 / 綠色
 - 2：警告 / 橘色
 - 3：錯誤 / 紅色
- **dialog_value** (*str*) -- 若對話框未獲回應時要回傳的預設值。
- **mandatory** (*bool*) -- 是否必須回應對話框，程式流程才能繼續。若設為 *True*，則無法透過關閉彈出視窗等方式忽略對話框，且會持續阻斷直到收到回應為止。

回傳：

所選的選項，即所選下拉式選單選項的標籤。

回傳型別：

str

範例：

此範例說明如何建立一個警告彈出視窗，並讓使用者在多個選項之間選擇以影響程式流程。

```
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)
```

建立一個帶文字欄位的對話框。此對話框會阻斷程式流程，直到使用者輸入文字並確認對話框為止。

參數:

- **text** (*str*) -- 向使用者顯示的訊息 (或問題)。
- **title** (*str*) -- 對話框彈出視窗的標題。
- **dialog_type** (*int*) -- 要向使用者顯示的對話框類型。可為以下其中一種：
 - 0：資訊 / 藍色
 - 1：成功 / 綠色
 - 2：警告 / 橘色
 - 3：錯誤 / 紅色
- **dialog_value** (*str*) -- 若對話框未獲回應時要回傳的預設值。
- **mandatory** (*bool*) -- 是否必須回應對話框，程式流程才能繼續。若設為 *True*，則無法透過關閉彈出視窗等方式忽略對話框，且會持續阻斷直到收到回應為止。

回傳:

輸入到文字欄位中的文字。

回傳型別:

str

範例：

此範例說明如何建立一個帶文字欄位的簡單彈出視窗，用於輸入您的姓名。使用者輸入姓名後，該姓名即可在程式中進一步使用，例如用於記錄。

```
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)

建立一個帶 Yes 與 No 按鈕的對話框。此對話框會阻斷程式流程，直到使用者按下其中一個按鈕為止。

參數:

- **text** (*str*) -- 向使用者顯示的訊息 (或問題)。
- **title** (*str*) -- 對話框彈出視窗的標題。
- **dialog_type** (*int*) -- 要向使用者顯示的對話框類型。可為以下其中一種：
 - 0：資訊 / 藍色
 - 1：成功 / 綠色
 - 2：警告 / 橘色
 - 3：錯誤 / 紅色
- **dialog_value** (*str*) -- 若對話框未獲回應時要回傳的預設值。
- **mandatory** (*bool*) -- 是否必須回應對話框，程式流程才能繼續。若設為 *True*，則無法透過關閉彈出視窗等方式忽略對話框，且會持續阻斷直到收到回應為止。

回傳:

若使用者選擇 "yes" 則為 *True*，若使用者選擇 "no" 則為 *False*。

回傳型別:

bool

範例：

此範例使用一個簡單の確認對話框，讓使用者選擇要繼續還是中止應用程式。

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

print(*args)

將訊息列印到應用程式輸出中。請注意，此函式的用法類似於 Python 內建的 *print* 函式。它支援多個引數與各種引數型別。引數會自動轉換為字串，並以空白字元分隔串接。

參數：

args (*tuple of various*) -- 要列印的引數。

範例：

以美觀的格式列印變數的值。請注意，第一次呼叫使用自動轉換與串接，第二次呼叫則使用 f-string，兩者結果相同。

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

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

在介面中顯示自訂通知訊息。此操作不影響程式流程。通知在介面中是淡出還是持續顯示，使用者可在設定選單中指定。

參數：

- **message_text** (*str*) -- 要在介面中顯示的通知文字。
- **message_title** (*str*) -- 通知彈出視窗的標題。

raise_error(message)

將自訂錯誤訊息引發至正在執行的應用程式中。此錯誤的型別為 `ApplicationError`，可在應用程式中攔截。若未攔截，正在執行的應用程式將中止。此錯誤同時記錄於 `error.log` 檔案中，該檔案屬於記錄系統的一部分。記錄可透過介面按下 `CTRL + SHIFT + L` 下載。

參數:

message (str) -- 要引發至應用程式中的錯誤訊息。

引發:

ApplicationError -- 包含自訂訊息。

範例：

背景應用程式監控數位輸入，並檢查是否觸發了特定的錯誤輸入。若觸發了錯誤輸入，則將此錯誤引發至應用程式中。

```
# 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)
```

此應用程式錯誤隨後可由控制機器人的主應用程式攔截並處理。

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

raise_warning(message)

將自訂警告訊息框引發至介面中。此操作不影響程式流程。警告在介面中是淡出還是持續顯示，使用者可在設定選單中指定。

參數：

message (*str*) -- 要引發至介面中的警告訊息。

範例：

假設我們在類比輸入上連接了一個溫度感測器。此範例會在數值超過某個閾值時將警告引發至介面中，並在數值再次低於某個閾值時重置。

```
# 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)

將訊息記錄到自訂記錄檔中。

參數：

- **message** (*str*) -- 要記錄的自訂訊息。
- **logger** (*str*) -- 要記錄到的記錄檔名稱。記錄檔位於 `custom/<logger>.log`，即記錄資料夾中。預設檔案名稱為 `custom/custom.log`。記錄可透過介面按下 `CTRL + SHIFT + L` 下載。
- **include_timestamp** (*bool*) -- 是否在記錄的訊息中加入時間戳記。

範例：

將感測器量測值記錄到 `custom/measurements.log` 檔案中。

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

clear_log()

清除並移除所有自訂記錄檔。

set_shared_variable(name, value)

設定共用變數的值。

請注意，*共用變數* 僅在主應用程式執行期間存在。主應用程式終止後，所有 *共用變數* 都會自動移除。若您需要變數在應用程式執行期間之外、甚至在機器人重新啟動後仍保留其值，請改用 *全域變數*。

參數：

- **name** (*str*) -- 要設定的變數名稱。
- **value** (*various*) -- 此變數的新值。

範例：

此範例說明主應用程式如何使用共用變數終止並行應用程式。並行應用程式只需監聽一個終止旗標，並在旗標設為 *True* 時立即中止。此外，共用變數在不再有作用後即被移除。由於共用變數在主應用程式終止時就會被清除，此步驟並非必要，但屬於良好做法。

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

主應用程式會確保終止旗標存在、執行並行應用程式，最後將終止旗標設為 *True* 以結束並行應用程式。

```
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)

讀取特定共用變數的值。預設情況下，此函式會讀取所有共用變數的值。

請注意，*共用變數* 僅在主應用程式執行期間存在。主應用程式終止後，所有 *共用變數* 都會自動移除。若您需要變數在應用程式執行期間之外、甚至在機器人重新啟動後仍保留其值，請改用 *全域變數*。

參數:

name (*str (or None)*) -- 要回傳的變數名稱 (若要回傳所有變數則為 None)

回傳:

要回傳的變數值 (或包含所有變數的字典)

回傳型別:

various (or dict)

範例 :

請參閱 *set_shared_variable* 的範例，其中結合了所有 *共用變數* 函式的用法。

remove_shared_variable(name)

移除共用變數。移除後，即無法再存取其值。

請注意，*共用變數* 僅在主應用程式執行期間存在。主應用程式終止後，所有 *共用變數* 都會自動移除。若您需要變數在應用程式執行期間之外、甚至在機器人重新啟動後仍保留其值，請改用 *全域變數*。

參數:

name (*str*) -- 要移除的變數名稱

範例：

請參閱 *set_shared_variable* 的範例，其中結合了所有 *共用變數* 函式的用法。

set_global_variable(*name*, *value*)

設定全域變數的值。

請注意，對 全域變數 的變更是永久性的，即使關閉並重新啟動機器人也會保留。因此強烈建議在 全域變數 不再有任何用途時將其移除。若您只在單一應用程式執行期間需要變數，請改用 共用變數。

參數：

- **name** (*str*) -- 要設定的變數名稱。
- **value** (*various*) -- 此變數的新值。

範例：

此範例說明重複性應用程式如何計算其成功迭代的次數。做法是在首次執行時初始化一個全域計數器，並在每次迴圈迭代結束時遞增該計數器。

```
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)

讀取特定全域變數的值。預設情況下，此函式會讀取所有全域變數的值。

請注意，對 全域變數 的變更是永久性的，即使關閉並重新啟動機器人也會保留。因此強烈建議在 全域變數 不再有任何用途時將其移除。若您只在單一應用程式執行期間需要變數，請改用 共用變數。

參數:

name (*str (or None)*) -- 要回傳的變數名稱 (若要回傳所有變數則為 None)

回傳:

要回傳的變數值 (或包含所有變數的字典)

回傳型別:

various (or dict)

範例 :

請參閱 `set_global_variable` 的範例，其中結合了 全域變數 函式的用法。

remove_global_variable(*name*)

移除全域變數。移除後，即無法再存取其值。

請注意，對 全域變數 的變更是永久性的，即使關閉並重新啟動機器人也會保留。因此強烈建議在 全域變數 不再有任何用途時將其移除。若您只在單一應用程式執行期間需要變數，請改用 共用變數。

參數：

name (*str*) -- 要移除的變數名稱

範例：

此範例會移除所有全域變數。

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

set_state_variable(name, value)

設定狀態變數的值。

請注意，狀態變數是為每個應用程式個別定義的，且僅用於主應用程式中。它們可在應用程式編輯器的 *variables* 分頁中設定與自訂。每個狀態變數都有特定的型別，且只接受該特定型別的值。狀態變數的主要用途是其值會即時顯示在 *Panel Interface* 中，以便觀察與控制正在執行的應用程式。

參數:

- **name** (*str*) -- 要設定的變數名稱。
- **value** (*various*) -- 此變數的新值。

範例：

假設此應用程式定義了一個型別為 *percent*、名稱為 *progress* 的唯讀狀態變數，以及一個型別為 *boolean*、名稱為 *logging* 的可動態編輯狀態變數。*progress* 變數的用途是以百分比儲存程式目前的進度。*logging* 旗標的用途是決定是否記錄進度更新。請注意，使用者可隨時切換 *logging* 的值，藉此動態調整程式。

```
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)

讀取特定狀態變數的值。預設情況下，此函式會讀取所有狀態變數的值。

請注意，*狀態變數* 是為每個應用程式個別定義的，且僅用於主應用程式中。它們可在應用程式編輯器的 *variables* 分頁中設定與自訂。每個 *狀態變數* 都有特定的型別，且只接受該特定型別的值。*狀態變數* 的主要用途是其值會即時顯示在 *Panel Interface* 中，以便觀察與控制正在執行的應用程式。

參數:

name (*str* (or *None*)) -- 要回傳的變數名稱 (若要回傳所有變數則為 *None*)

回傳:

要回傳的變數值 (或包含所有變數的字典)

回傳型別:

various (or dict)

範例 :

請參閱 *set_state_variable* 的範例，其中結合了 *狀態變數* 函式的用法。

read_actuator_position(*actuator_ids=None*)

讀取一個、多個或所有致動器的角度位置。

參數:

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

要讀取的致動器 ID。請按以下方式指定值：

- 單一關節的 ID (例如 6)
- 多個關節的關節 ID 清單 (例如 [1, 4, 6])
- 多個關節的關節名稱集合 (例如 {1, 4, 6})
- 常數 *ALL_KIN_JOINTS* 或 *None*，用於讀取所有關節

回傳:

所請求致動器的角度位置 (單位為 [deg])，可為：

- 單一關節位置值 -> float
- 關節位置值的清單 -> list of float
- 關節 ID 與其位置值之間的對應 -> dict of float

回傳型別:

float, or list of float, or dict of float

範例：

列印腕部關節致動器目前的位置。

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

read_actuator_velocity(*actuator_ids=None*)

讀取一個、多個或所有致動器的角速度。

參數:

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

要讀取的致動器 ID。請按以下方式指定值：

- 單一關節的 ID (例如 6)
- 多個關節的關節 ID 清單 (例如 [1, 4, 6])
- 多個關節的關節名稱集合 (例如 {1, 4, 6})
- 常數 *ALL_KIN_JOINTS* 或 *None*，用於讀取所有關節

回傳:

所請求致動器的角速度 (單位為 °/s)，可為：

- 單一關節速度值 -> float
- 關節速度值的清單 -> list of float
- 關節 ID 與其速度值之間的對應 -> dict of float

回傳型別:

float, or list of float, or dict of float

範例：

在運動期間監控速度，以判斷運動最快的致動器。

```
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()

取得機器人 TCP (工具中心點) 的線性速度。

回傳:

線性工具速度，單位為 [mm/s]。

回傳型別:

list

範例：

取得並列印目前的線性速度。

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

get_angular_velocity()

取得機器人 TCP (工具中心點) 的角速度。

回傳:

角度工具速度，單位為 [deg/s]。

回傳型別:

list

範例：

取得並列印目前的角速度。

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

read_actuator_current(*actuator_ids=None*)

讀取一個、多個或所有致動器的施加電流。

參數:

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

要讀取的致動器 ID。請按以下方式指定值：

- 單一關節的 ID (例如 6)
- 多個關節的關節 ID 清單 (例如 [1, 4, 6])
- 多個關節的關節名稱集合 (例如 {1, 4, 6})
- 常數 *ALL_KIN_JOINTS* 或 *None*，用於讀取所有關節

回傳:

所請求致動器的電流值 (單位為 A)，可為：

- 單一關節電流值 -> float
- 關節電流值的清單 -> list of float
- 關節 ID 與其電流值之間的對應 -> dict of float

回傳型別:

float, or list of float, or dict of float

範例：

在特定位姿列印機器人致動器的電流兩次，一次在拾取物體之前，一次在拾取之後。

```
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()

啟動工具的拾取功能以抓取物體。對機械式工具而言可能是夾持，對氣動式工具而言可能是施加吸力。

範例：

啟動工具拾取功能。

```
tool_pick()
```

tool_place()

啟動工具的放置功能以釋放物體。對機械式工具而言可能是鬆開夾爪，對氣動式工具而言可能是停止吸力。

範例：

啟動工具放置功能。

```
tool_place()
```

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

在機器人運作期間動態設定工具負載。

參數:

- **payload** (*float*) -- 工具負載 (重量 + 抓取物件) · 單位為 [kg]
- **center_of_mass** (*list of float*) -- 相對於法蘭坐標系的質心 [x, y, z] · 單位為 [m]
- **inertia** (*list of float*) -- 切分為長度 9 之清單的慣性矩陣 [kg m²]

範例：

在將物體從一個位姿拾取並放置到另一個位姿的過程中變更負載。

```
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*)

讀取 'Digital Main Inputs' 類別中一個、多個或所有 I/O 的值。

數位 I/O 的兩種狀態為：

- *HIGH*，以布林值 *True* 表示
- *LOW*，以布林值 *False* 表示

參數：

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

指定要讀取哪些 I/O。識別碼可以是 I/O 編號，或是指派給這些 I/O 的唯一自訂名稱。可為以下其中一種：

- 單一編號 (*int*) 或自訂名稱 (*str*) -> 回傳一個 *bool*
- 編號 (*int*) 或自訂名稱 (*str*) 的清單，或兩者混合 -> 回傳一個 *bool* 清單
- 編號 (*int*) 或自訂名稱 (*str*) 的集合，或兩者混合 -> 回傳一個以識別碼為鍵、*bool* 為值的 *dict*

回傳：

所請求數位 I/O 的布林值 (*HIGH* = *True*，*LOW* = *False*)

回傳型別：

bool, list, dict

範例：

假設其中一個數位輸入被指派了名稱 'trigger'。此程式碼區塊會等待該 trigger 切換為 HIGH 後才繼續執行。

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

讀取前 8 個數位 I/O，並將二進位的 *bool* 清單轉換為 0 到 255 之間的整數。

```
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*)

讀取 'Digital Tool Inputs' 類別中一個、多個或所有 I/O 的值。

詳情請參閱用法類似的 'read_digital_main_input'。

read_analog_main_input(identifiers)

讀取 'Analog Main Inputs' 類別中一個、多個或所有 I/O 的值。

類比 I/O 可為電壓模式或電流模式。數值以 0.1 為級距施加與讀取，亦即 I/O 值 55 代表實際電壓值 5.5 V 或電流值 5.5 mA。類比 I/O 的數值範圍為：

- 介於 0.0 與 10.0 V 之間 (電壓模式)
- 介於 4.0 與 20.0 mA 之間 (電流模式)

參數:

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

指定要讀取哪些 I/O。識別碼可以是 I/O 編號，或是指派給這些 I/O 的唯一自訂名稱。可為以下其中一種：

- 單一編號 (int) 或自訂名稱 (str) -> 回傳一個 bool
- 編號 (int) 或自訂名稱 (str) 的清單，或兩者混合 -> 回傳一個 bool 清單
- 編號 (int) 或自訂名稱 (str) 的集合，或兩者混合 -> 回傳一個以識別碼為鍵、bool 為值的 dict

回傳:

所請求類比 I/O 的數值

回傳型別:

float, list, dict

範例：

假設其中一個電流模式的類比輸入被指派了名稱 'temperature'。有一個溫度感測器連接到此輸入，並確定回傳值 10.6 mA 代表不應超過的臨界溫度閾值。接著與您的主應用程式並行執行以下程式，它會在達到臨界閾值時中止程式。

```
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*)

讀取 'Analog Tool I/O' 類別中一個、多個或所有 I/O 的值。

詳情請參閱用法類似的 'read_analog_main_input'。

write_digital_main_output(*identifiers, values*)

寫入 'Digital Main Output' 類別中一個、多個或所有 I/O 的值。

數位 I/O 的兩種狀態為：

- *HIGH* · 以布林值 *True* 表示
- *LOW* · 以布林值 *False* 表示

參數：

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) --
指定要寫入哪些 I/O。識別碼可以是 I/O 編號，或是指派給這些 I/O 的唯一自訂名稱。可為以下其中一種：
 - 單一編號 (*int*) 或自訂名稱 (*str*) -> *values* 為一個 *bool*
 - 編號 (*int*) 或自訂名稱 (*str*) 的清單，或兩者混合 -> *values* 為一個 *bool* 清單
 - 編號 (*int*) 或自訂名稱 (*str*) 的集合，或兩者混合 -> *values* 為一個以識別碼為鍵、*bool* 為值的 *dict*
- **values** (*bool, list (of bool), dict (of bool)*) --
要寫入之數位 I/O 的布林值 (*HIGH* = *True* · *LOW* = *False*)。視 *identifiers* 而定，可為以下其中一種：
 - 單一 *bool* -> 若 *identifiers* 為單一 I/O
 - *bool* 清單 -> 若 *identifiers* 為 I/O 清單
 - 以識別碼為鍵、*bool* 為值的 *dict* -> 若 *identifiers* 為 I/O 集合

範例：

假設其中一個數位輸出被指派了名稱 'trigger'。此程式碼會將該數位輸出設為 *HIGH* 持續 100 ms，然後再設回 *LOW*，藉此發送一個脈衝。

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

將一個 8 位元整數 (介於 0 與 255 之間) 寫入前 8 個數位 I/O，做法是先將該數字轉換為 *bool* 清單，再同時寫入全部 8 個值。

```
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*)

寫入 'Digital Tool Output' 類別中一個、多個或所有 I/O 的值。

詳情請參閱用法類似的 'write_digital_main_output'。

write_analog_main_output(*identifiers, values*)

寫入 'Analog Main Outputs' 類別中一個、多個或所有 I/O 的值。

類比 I/O 可為電壓模式或電流模式。數值以 0.1 為級距施加與讀取，亦即 I/O 值 55 代表實際電壓值 5.5 V 或電流值 5.5 mA。類比 I/O 的數值範圍為：

- 介於 0.0 與 10.0 V 之間 (電壓模式)
- 介於 4.0 與 20.0 mA 之間 (電流模式)

參數:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) --
指定要寫入哪些 I/O。識別碼可以是 I/O 編號，或是指派給這些 I/O 的唯一自訂名稱。可為以下其中一種：
 - 單一編號 (int) 或自訂名稱 (str) -> *values* 為一個 float
 - 編號 (int) 或自訂名稱 (str) 的清單，或兩者混合 -> *values* 為一個 float 清單
 - 編號 (int) 或自訂名稱 (str) 的集合，或兩者混合 -> *values* 為一個以識別碼為鍵、float 為值的 dict
- **values** (*float, list (of float), dict (of float)*) --
要寫入之類比 I/O 的數值。視 *identifiers* 而定，可為以下其中一種：
 - 單一 float -> 若 *identifiers* 為單一 I/O
 - float 清單 -> 若 *identifiers* 為 I/O 清單
 - 以識別碼為鍵、float 為值的 dict -> 若 *identifiers* 為 I/O 集合

範例：

假設某個可控制 LED 的 R (紅)、G (綠)、B (藍) 值直接連接到三個類比 I/O，其自訂名稱為 "red"、"green" 和 "blue"，使得最大數值範圍 [0, 255] 直接對應到最大電壓範圍 [0.0 V, 10.0 V]。此範例會同時寫入這些值。

```
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)`

寫入 'Analog Tool I/O' 類別中一個、多個或所有 I/O 的值。

詳情請參閱用法類似的 'write_analog_main_output'。

wait_for_digital_main_input(*identifier*, *value*)

等待直到 'Digital Main Inputs' 類別中的某個 I/O 讀取到特定值。

數位 I/O 的兩種狀態為：

- *HIGH* · 以布林值 *True* 表示
- *LOW* · 以布林值 *False* 表示

參數：

- **identifier** (*int*, *str*) -- 指定要等待哪一個 I/O。識別碼可以是 I/O 編號 (*int*)，或是指派給此 I/O 的唯一自訂名稱 (*str*)。
- **value** (*bool*) -- 要等待的值。

範例：

假設其中一個數位輸入被指派了名稱 'trigger'。此程式碼區塊會等待該 trigger 切換為 HIGH 後才繼續執行。

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

在少數情況下，您可能希望能夠暫停程式，同時仍希望等待函式在 I/O 被觸發時結束。此等待函式無法用於這種做法，但以下範例正是如此實作：

將此程式碼加入背景應用程式中。它會監控特定 I/O 的值，並在監控期間達到指定值時設定一個旗標。

```
# 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)
```

並將此程式碼加入對應的主應用程式中，由它觸發背景監控並等待監控程式設定該旗標。

```
# 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*)

等待直到 'Digital Tool Inputs' 類別中的某個 I/O 讀取到特定值。

詳情請參閱用法類似的 'wait_for_digital_main_input'。

wait_for_analog_main_input(*identifier*, *value_range*)

等待直到 'Analog Main Inputs' 類別中的某個 I/O 超過閾值。

類比 I/O 可為電壓模式或電流模式。數值以 0.1 為級距施加與讀取，亦即 I/O 值 55 代表實際電壓值 5.5 V 或電流值 5.5 mA。類比 I/O 的數值範圍為：

- 介於 0.0 與 10.0 V 之間 (電壓模式)
- 介於 4.0 與 20.0 mA 之間 (電流模式)

參數：

- **identifier** (*int*, *str*) -- 指定要等待哪一個 I/O。識別碼可以是 I/O 編號 (*int*)，或是指派給此 I/O 的唯一自訂名稱 (*str*)。
- **value_range** (*list of float*) --
等待此 I/O 達到此範圍內的值。為以下其中一種：
 - 同時指定下限與上限 (例如 [5.5, 7.0])
 - 僅指定下限 (例如 [5.5, None])
 - 僅指定上限 (例如 [None, 7.0])

範例：

以電壓模式的類比 I/O 為例。此範例等待其電壓值增長到超過 8.0 V，該值以 80 表示 (級距為 0.1 V)。

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

`wait_for_analog_tool_io(identifier, value_range)`

等待直到 'Analog Tool I/O' 類別中的某個 I/O 超過閾值。

詳情請參閱用法類似的 'wait_for_analog_main_input'。

get_runtime_position_offsets()

此函式回傳機器人的位置偏移量。

回傳:

機器人的位置偏移量

回傳型別:

list of float

read_button_state(channel=None)

讀取機器人介面上一個或所有可自訂按鈕的目前狀態 (按下或未按下)。請注意，已指派特定功能的按鈕無法在執行期間讀取，因為它們另有用途。

這些按鈕位於機械手臂最上方的關節上。

參數:

channel (*int or None*) -- 按鈕識別碼，為介於 1 與按鈕數量之間的數字。預設會讀取所有可自訂按鈕的狀態。

回傳:

所請求按鈕的目前狀態，以 **bool** 表示；或各按鈕的目前狀態，封裝成 {1: bool, 2: bool, ...} 形式的字典。

回傳型別:

bool or dict of bool

範例：

此範例會在可自訂按鈕的狀態變更時列印訊息。

```
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)
```

將此範例作為背景應用程式執行。它讓您可以使用按鈕 1 和 2 暫停與恢復正在執行的主應用程式。

```
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)

建立一個用於傳送訊息的 TCP socket 用戶端。用戶端會持續嘗試連線到指定位址的伺服器。訊息以 *UTF-8* 編碼標準編碼。此外，訊息以文字結束字元 *0x03`* (*Python* 中為 *`x03`*) 分隔。

參數:

- **ip** (*str*) -- 伺服器的 IP 位址 (例如 `"192.168.80.2"`)
- **port** (*int*) -- 可存取伺服器的連接埠號 (例如 `60001`)。連接埠限制在 60000 到 61000 的範圍內。

回傳:

TCP socket 用戶端物件，用於傳送與接收訊息。

回傳型別:

Socket

範例 :

此範例在應用程式中的 TCP socket 用戶端與同一網路中的 TCP socket 伺服器之間執行一次交握。首先建立連線。接著用戶端等待伺服器發送初始訊息。收到訊息後，用戶端回覆自己的訊息，最後關閉連線。

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

create_tcp_server(port)

建立一個最多可處理 20 個用戶端的 TCP socket 伺服器。訊息以 *UTF-8* 編碼標準編碼。此外，訊息以文字結束字元 *0x03`* (*Python* 中為 *`x03`*) 分隔。

參數:

port (*int*) -- 用戶端可存取此伺服器的連接埠號 (例如 `60002`)。連接埠限制在 60000 到 61000 的範圍內。

回傳:

TCP socket 伺服器物件，用於傳送與接收訊息。

回傳型別:

[Socket](#)

範例：

此範例在應用程式中的 TCP socket 伺服器與第一個連線到它的用戶端之間執行一次交握。首先，伺服器在特定連接埠上開啟連線並等待用戶端連線。接著伺服器等待用戶端發送初始訊息。收到訊息後，伺服器回覆自己的訊息，最後關閉與所有已連線用戶端的連線。

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

create_udp_socket(port)

建立一個用於傳送訊息的 UDP socket。訊息以 *UTF-8* 編碼標準編碼。此外，訊息以文字結束字元 *0x03* (*Python* 中為 *\x03*) 分隔。

參數:

port (*int*) -- 可存取此 socket 的連接埠號 (例如 `60003`)。連接埠限制在 60000 到 61000 的範圍內。

回傳:

UDP socket 物件，用於傳送與接收訊息。

回傳型別:

[Socket](#)

範例：

此範例在應用程式中的 UDP socket 與第一個對它開啟的連線之間執行一次交握。首先，socket 在特定連接埠上開啟連線並等待對方連線。接著 socket 等待對方發送初始訊息。收到訊息後，socket 回覆自己的訊息，最後關閉連線。

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

send_message(message)

以鍵值 'script_send' 向已開啟的 TCP 連線傳送訊息。

參數:

message (*str*) -- 要傳送的訊息。

範例：

透過 TCP 向任何可能監聽的裝置傳送訊息：

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

在已連線的裝置端，此資料會以下列格式接收：

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

receive_message(*timeout=None*)

呼叫動作 'script_receive'，從已開啟的 TCP 連線接收訊息。

參數:

timeout (*None or float*) -- 等待訊息的最長時間。若未指定，則無限期等待。

範例：

外部裝置透過 TCP 連線到機器人並發送以下資料：

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

在機器人端，接著可使用此函式接收並列印該訊息：

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

clear_messages()

清除 TCP 訊息佇列中先前傳入但可能尚未讀取的訊息。

範例：

在等待新訊息之前，有時確保舊訊息被排除是合理的做法。

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

class Coordinates

Coordinates 物件定義一個數學變換，可用於例如運動函式中的目標坐標，或用於定義坐標系。

`__init__(coordinates)`

建立一個 Coordinates 物件，可帶有特定的坐標值，或包含機器人目前的坐標。

參數：

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

一組特定的坐標，可為以下其中一種：

- 形式為 [x, y, z, roll, pitch, yaw] 的清單，位置值單位為 [mm]，方向值單位為 [deg]
- 形式為 {'x': x, 'y': y, 'z': z, 'roll': roll, 'pitch': pitch, 'yaw': yaw} 的字典，位置值單位為 [mm]，方向值單位為 [deg]
- 一個 Coordinates 物件（複製另一個 Coordinates 物件）

建構函式：

從坐標值建立一個 Coordinates 物件。

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

使用 `get_current_coordinates` 函式建立一個包含機器人目前坐標的 Coordinates 物件。

```
current_coordinates = get_current_coordinates()
```

property position

Coordinates 物件的位置部分。

回傳:

位置，形式為 [x, y, z]，單位為 [mm]。

回傳型別:

list of float

property orientation

Coordinates 物件的方向部分。

回傳:

方向，形式為 [roll, pitch, yaw]，單位為 [deg]。

回傳型別:

list of float

to_list()

將坐標物件轉換為形式為 [x, y, z, roll, pitch, yaw] 的坐標值清單，其中位置值單位為 [mm]，旋轉值單位為 [deg]。

回傳:

清單格式的坐標值。

回傳型別:

list of float

範例：

當需要以清單形式使用 Coordinates 時，可使用此函式：

```
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()

將坐標物件轉換為形式為 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 的坐標值字典，其中位置值單位為 [mm]，旋轉值單位為 [deg]。

回傳:

字典格式的坐標值。

回傳型別:

dict of float

範例：

當需要以字典形式使用 `Coordinates` 時，可使用此函式：

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

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

inverted()

計算此坐標物件的反變換。

回傳:

變換物件的反變換

回傳型別:

Transform

範例 :

```
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

Pose 物件定義機器人的一個位姿，由其坐標（位置與方向）及其配置構成（因為同一組坐標可透過多種配置到達）。

`__init__(pose)`

建立一個 Pose 物件，可透過名稱或 ID 載入已儲存的姿勢，或從機器人目前的位姿建立。

參數：

pose (*str, int, or Pose*) --

要載入之姿勢的參照，可為以下其中一種：

- 資料庫中已儲存姿勢的名稱 (*str*)
- 資料庫中已儲存姿勢的 ID (*int*)
- 一個 Pose 物件（複製另一個 Pose 物件）

建構函式：

從資料庫載入一個姿勢。

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

使用 `get_current_pose` 函式建立一個包含機器人目前位姿的 Pose 物件。

```
current_pose = get_current_pose()
```

使用 `create_pose` 函式建立一個包含指定坐標與配置的 Pose 物件。

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

property coordinates

此位姿的坐標，為一個包含位置與方向值的 `Coordinates` 物件。

回傳:

此位姿的坐標。

回傳型別:

`Coordinates`

property configuration

由於一組位姿坐標最多可透過 8 種機器人關節配置到達，因此配置以數字 (0 到 7) 或字串 ('c1' 到 'c8') 定義，以唯一確定該位姿。

回傳:

位姿配置的索引 (0 到 7)。

回傳型別:

`int`

class LinearSegment

LinearSegment 物件描述 TCP (工具中心點) 通往目標坐標的線性路徑區段。此外，它包含所需線性速度、線性加速度以及區段起點混合半徑的資訊。此混合半徑允許偏離線性路徑，以確保區段之間平順過渡。

`__init__(linear_segment)`

建立一個 LinearSegment 物件。

參數：

- **coordinates** (*list, dict, or Coordinates*) -- 目標坐標。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 LinearSegment 物件：

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

或使用預設值：

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

property coordinates

此區段的目標坐標，為一個包含位置與方向值的 `Coordinates` 物件。

回傳:

此區段的目標坐標。

回傳型別:

`Coordinates`

property blend_radius

此區段起點的混合半徑。

回傳:

起點混合半徑。

回傳型別:

`float`

property linear_acceleration

此區段所需的線性加速度。若為 `None`，則使用全域線性加速度值。

回傳:

所需的線性加速度。

回傳型別:

`float` or `None`

property linear_velocity

此區段所需的線性速度。若為 `None`，則使用全域線性速度值。

回傳:

所需的線性速度。

回傳型別:

`float` or `None`

class CircularSegment

CircularSegment 物件描述 TCP (工具中心點) 經過中間位置通往目標坐標的圓弧運動。請注意，該中間位姿的方向由演算法決定。此外，它包含所需線性速度、線性加速度以及區段起點混合半徑的資訊。此混合半徑允許偏離圓弧路徑，以確保區段之間平順過渡。

`__init__(circular_segment)`

建立一個 CircularSegment 物件。

參數：

- **intermediate_position** (*list*) -- 通往目標坐標的路徑上的中間位置 [x, y, z]。中間位置決定圓弧運動的曲率。
- **coordinates** (*list, dict or Coordinates*) -- 目標坐標。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 CircularSegment 物件：

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

或使用預設值：

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

property intermediate_position

此區段的中間位置。它包含 x、y、z 坐標：[x, y, z]

回傳：

此區段的中間位置。

回傳型別：

list

property coordinates

此區段的目標坐標，為一個包含位置與方向值的 Coordinates 物件。

回傳：

此區段的目標坐標。

回傳型別：

Coordinates

property blend_radius

此區段起點的混合半徑。

回傳：

起點混合半徑。

回傳型別：

float

property linear_acceleration

此區段所需的線性加速度。若為 None，則使用全域線性加速度值。

回傳：

所需的線性加速度。

回傳型別：

float or None

property linear_velocity

此區段所需的線性速度。若為 None，則使用全域線性速度值。

回傳:

所需的線性速度。

回傳型別:

float or None

class OrientationSegment

OrientationSegment 物件描述純方向路徑區段，其中 TCP（工具中心點）依照所需的工具方向旋轉，同時保持位置不變。此外，它包含所需角速度、角加速度以及區段起點混合半徑的資訊。若混合半徑大於 0，TCP 會在到達目標位置之前就開始旋轉。

`__init__(orientation_segment)`

建立一個 OrientationSegment 物件。

參數：

- **orientation** (*list*) -- 目標方向。請注意，位置保持不變。
- **angular_velocity** (*float or None*) -- 所需的角速度 [deg/s]。若未定義角速度，則使用全域角速度值。
- **angular_acceleration** (*float or None*) -- 所需的角加速度 [deg/s²]。若未定義角加速度，則使用全域角加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。混合半徑大於 0 時，工具中心點 (TCP) 可在到達此區段的位置之前開始旋轉。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 OrientationSegment 物件：

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

或使用預設值：

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

property orientation

此區段的目標方向。以航海角 [roll, pitch, yaw] 表示，單位為度

回傳:

此區段的目標方向。

回傳型別:

list

property angular_velocity

此區段所需的角速度。若為 None，則使用全域角速度值。

回傳:

所需的角速度。

回傳型別:

float or None

property angular_acceleration

此區段所需的角加速度。若為 None，則使用全域角加速度值。

回傳:

所需的角加速度。

回傳型別:

float or None

property blend_radius

此區段起點的混合半徑。

回傳:

起點混合半徑。

回傳型別:

float

class CoordinatesSegment

CoordinatesSegment 物件描述在關節空間中通往目標坐標的線性運動。相對地，TCP (工具中心點) 一般不會線性移動。此外，它包含所需線性速度、線性加速度以及區段起點混合半徑的資訊。此混合半徑允許偏離路徑，以確保區段之間平順過渡。

`__init__(coordinates_segment)`

建立一個 CoordinatesSegment 物件。

參數：

- **coordinates** (*list, dict or [Coordinates](#)*) -- 目標坐標。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 CoordinatesSegment 物件：

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

或使用預設值：

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

property coordinates

此區段的目標坐標，為一個包含位置與方向值的 `Coordinates` 物件。

回傳:

此區段的目標坐標。

回傳型別:

`Coordinates`

property blend_radius

此區段起點的混合半徑。

回傳:

起點混合半徑。

回傳型別:

`float`

property linear_acceleration

此區段所需的線性加速度。若為 `None`，則使用全域線性加速度值。

回傳:

所需的線性加速度。

回傳型別:

`float` or `None`

property linear_velocity

此區段所需的線性速度。若為 `None`，則使用全域線性速度值。

回傳:

所需的線性速度。

回傳型別:

`float` or `None`

class PoseSegment

Pose 物件描述在關節空間中通往目標位姿的線性運動。相對地，TCP（工具中心點）一般不會線性移動。此外，它包含所需線性速度、線性加速度以及區段起點混合半徑的資訊。此混合半徑允許偏離路徑，以確保區段之間平順過渡。

`__init__(pose_segment)`

建立一個 PoseSegment 物件。

參數：

- **pose** (*Pose*) -- 目標位姿。
- **linear_velocity** (*float or None*) -- 所需的線性速度 [mm/s]。若未定義線性速度，則使用全域線性速度值。
- **linear_acceleration** (*float or None*) -- 所需的線性加速度 [mm/s²]。若未定義線性加速度，則使用全域線性加速度值。
- **blend_radius** (*float or None*) -- 此區段起點的混合半徑 [mm]。若未定義混合半徑，則使用 0.0。

建構函式：

透過定義所有參數建立一個 PoseSegment 物件：

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

或使用預設值與資料庫中的姿勢：

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

property pose

此區段的目標位姿，為一個包含位置、方向與配置值的 Pose 物件。

回傳:

此區段的目標位姿。

回傳型別:

Pose

property blend_radius

此區段起點的混合半徑。

回傳:

起點混合半徑。

回傳型別:

float

property linear_acceleration

此區段所需的線性加速度。若為 None，則使用全域線性加速度值。

回傳:

所需的線性加速度。

回傳型別:

float or None

property linear_velocity

此區段所需的線性速度。若為 None，則使用全域線性速度值。

回傳:

所需的線性速度。

回傳型別:

float or None

class Path

Path 物件定義機器人在兩個或更多位姿之間的移動方式。連續路徑僅在工具空間中由多個區段定義，例如線性或圓弧區段。若路徑的離散解不需要機器人重新調整自身位姿（配置變更），則該路徑稱為無縫路徑。若加速度沿路徑連續，則該路徑稱為連續路徑。路徑從一個指定的位姿開始（而非從坐標開始）。路徑的區段通常定義坐標，因為路徑的配置由初始位姿決定。

`__init__(path)`

建立一個 Path 物件，透過名稱或 ID 載入已儲存的路徑。

參數：

path (*str, int, or Path*) --

要載入之路徑的參照，可為以下其中一種：

- 資料庫中已儲存路徑的名稱 (str)
- 資料庫中已儲存路徑的 ID (int)
- 一個 Path 物件（複製另一個 Path 物件）

建構函式：

從資料庫載入一條路徑。

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

使用 `create_path` 函式建立一個包含指定初始位姿與所定義路徑區段的 Path 物件。請注意，區段也可以之後再加入。

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

property initial_pose

路徑的初始機器人位姿。

回傳:

初始機器人位姿。

回傳型別:

Pose

property segments

路徑中包含的區段。請注意，這些區段不必全部屬於同一類型。

回傳:

包含的區段。

回傳型別:

list of Segments ([LinearSegment](#), [CircularSegment](#), [OrientationSegment](#),
[CoordinatesSegment](#), [PoseSegment](#))

add_segment(segment)

在路徑末端加入一個區段。

參數:

segment ([LinearSegment](#), [CircularSegment](#), [OrientationSegment](#), [CoordinatesSegment](#),
[PoseSegment](#)) -- 要加入的區段。

範例：

可依下列方式加入區段：

```
# 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)
```

在路徑末端加入一個線性區段。

參數：

- **coordinates** (*list, dict or [Coordinates](#)*) -- 路徑的目標坐標。
- **linear_velocity** (*float*) -- 允許的最大線性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允許的最大線性工具加速度 [mm/s²]。
- **blend_radius** (*float*) -- 此區段起點允許的混合半徑 [mm]。

範例：

可依下列方式加入線性區段：

```
# 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]))
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
# 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)
```

第一個區段定義通往指定目標坐標的線性運動，最大速度為 500 mm/s，最大加速度為 1000 mm/s²。請注意，由於後續區段定義了 100 mm 的混合半徑，這些坐標不會被精確到達，該混合半徑允許偏離這些坐標，以確保更平順且更高效的過渡。第二個區段接著加入通往下一組坐標的線性運動，最大速度為 250 mm/s，最大加速度為 500 mm/s²。

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

在路徑末端加入一個圓弧區段。

參數：

- **intermediate_position** (*list*) -- 通往目標坐標的路徑上的中間位置 ([x, y, z])。中間位置決定圓弧運動的曲率。
- **coordinates** (*list, dict or Coordinates*) -- 路徑的目標坐標。
- **linear_velocity** (*float*) -- 允許的最大線性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允許的最大線性工具加速度 [mm/s²]。
- **blend_radius** (*float*) -- 此區段起點允許的混合半徑 [mm]。

範例：

可依下列方式加入圓弧區段：

```
# 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]))
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
# 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)
```

第一個區段定義經過指定中間位置通往目標坐標的圓弧運動，最大速度為 500 mm/s，最大加速度為 1000 mm/s²。請注意，由於後續區段定義了 100 mm 的混合半徑，這些坐標不會被精確到達，該混合半徑允許偏離這些坐標，以確保更平順且更高效的過渡。第二個區段接著加入經過下一個中間位置通往下一組目標坐標的圓弧運動，最大速度為 250 mm/s，最大加速度為 500 mm/s²。

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

在路徑末端加入一個方向區段。

參數：

- **coordinates** (*list, dict or [Coordinates](#)*) -- 路徑的目標坐標。
- **angular_velocity** (*float*) -- 允許的最大角度工具速度 [deg/s]。
- **angular_acceleration** (*float*) -- 允許的最大角度工具加速度 [deg/s²]。
- **blend_radius** (*float*) -- 此區段起點允許的混合半徑 [mm]。大於 0 的值表示可在到達此區段的目標位置之前就開始旋轉。

範例：

可依下列方式加入方向區段：

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

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
# 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)
```

第一個區段定義旋轉到指定目標方向的運動，最大速度為 250 deg/s，最大加速度為 500 deg/s²。請注意，執行此路徑時，由於後續區段的混合半徑大於 0 (參見 `OrientationSegment.__init__`)，第一次旋轉會被略過，並直接執行第二個區段。此區段定義旋轉到目標方向 [150.0, 0.0, -90.0] 的運動，最大速度為 500 deg/s，最大加速度為 1000 deg/s²。相對地，若將第二個區段的混合半徑設為 0.0，則兩次旋轉會依序執行。

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

在路徑末端加入一個坐標區段。

參數：

- **coordinates** (*list, dict or [Coordinates](#)*) -- 路徑的目標坐標。
- **linear_velocity** (*float*) -- 允許的最大線性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允許的最大線性工具加速度 [mm/s²]。
- **blend_radius** (*float*) -- 此區段起點允許的混合半徑 [mm]。

範例：

可依下列方式加入坐標區段：

```
# 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]))
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
# 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)
```

第一個區段定義通往指定目標坐標的運動，最大速度為 500 mm/s，最大加速度為 1000 mm/s²。請注意，由於後續區段定義了 100 mm 的混合半徑，這些坐標不會被精確到達，該混合半徑允許偏離這些坐標，以確保更平順且更高效的過渡。第二個區段接著加入通往下一組坐標的運動，最大速度為 250 mm/s，最大加速度為 500 mm/s²。

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

在路徑末端加入一個位姿區段。

參數：

- **pose** (*Pose*) -- 路徑的目標坐標。
- **linear_velocity** (*float*) -- 允許的最大線性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允許的最大線性工具加速度 [mm/s²]。
- **blend_radius** (*float*) -- 此區段起點允許的混合半徑 [mm]。

範例：

可依下列方式加入位姿區段：

```
# 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)
```

這些範例中未指定選用參數，因此使用全域參數。這些參數也可在函式呼叫中明確定義：

```
# 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)
```

第一個區段定義通往指定目標位姿的運動，最大速度為 500 mm/s，最大加速度為 1000 mm/s²。請注意，由於後續區段定義了 100 mm 的混合半徑，此位姿不會被精確到達，該混合半徑允許偏離對應的坐標，以確保更平順且更高效的過渡。第二個區段接著加入通往下一個位姿的運動，最大速度為 250 mm/s，最大加速度為 500 mm/s²。

class Socket

Socket 是由 `create_tcp_client`、`create_tcp_server` 和 `create_udp_socket` 函式回傳的物件。使用此物件透過該 socket 傳送與接收資料。

receive(*timeout=None*)

透過此 socket 接收訊息。

參數:

timeout (*None or float*) -- 等待訊息的最長時間。若未指定，則無限期等待。

回傳:

接收到的訊息；若發生逾時，則為空字串。

回傳型別:

str

範例：

等待答覆 3 秒，並在收到時列印該答覆。

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

send(*message*)

透過此 socket 傳送訊息。

參數：

message (*str*) -- 要傳送的訊息。

範例：

向所有已連線的對等端傳送訊息。

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

close()

關閉 socket 連線。

範例：

關閉 socket。

```
socket.close()
```

附加功能函式

Cognex 附加功能函式

class Cognex

Cognex 模組支援 Cognex 攝影機的使用。請注意，設定 Cognex 攝影機需要安裝 [Cognex In-Sight Explorer](#)。

關於如何正確設定攝影機並將其與機器人連線的詳細資訊，請參閱用戶手冊。

此附加功能的所有函式均使用 `cognex.` 前綴呼叫。

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

設定透過 ISE 連接埠連線到已連接 Cognex 攝影機的 TCP 用戶端，用於原生指令。

參數:

- **ip** (*str*) -- 攝影機的 IP 位址，例如 `'10.0.0.210'`。
- **user** (*str*) -- 登入攝影機的使用者名稱 (如先前在 [Cognex In-Sight Explorer](#) 中設定)。預設使用者名稱為 `'admin'`。
- **password** (*str*) -- 登入攝影機的密碼 (如先前在 [Cognex In-Sight Explorer](#) 中設定)。預設密碼為空字串。

引發:

MinorError -- 若憑證錯誤、連線過程中連線被中止，或連線耗時過長。

範例：

使用預設 IP 與憑證連線到全新出廠的攝影機。

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

連線到先前已修改 IP 位址與憑證的攝影機。

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

trigger_camera(ip)

設定攝影機觸發事件 ("SW8")，該事件會拍攝新影像並回傳由當前 Cognex 作業處理後的資料。

參數:

ip (*str*) -- 攝影機的 IP 位址，例如 `'10.0.0.210'`。

回傳:

來自攝影機的回應訊息，包含偵測到的實體。

回傳型別:

str

change_job(ip, job_name)

透過原生攝影機指令 "LF" 變更 Cognex 作業，該指令會載入已儲存於攝影機上的作業檔案。
請注意，此指令可能需要數秒才能執行完成。

參數:

- **ip** (str) -- 攝影機的 IP 位址，例如 `'10.0.0.210'`。
- **job_name** (str) -- 要切換到的作業名稱 (如先前在 [Cognex In-Sight Explorer](#) 中設定)。

引發:

MinorError -- 若攝影機不存在，或變更作業失敗。

範例：

切換到先前儲存為 'CircleDetection.job' 的作業，然後拍攝影像以偵測當前攝影機視野中的圓形。

```
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)

觸發攝影機上的事件，並在 Cognex 端請求特定關鍵字 socket 訊息。

參數:

- **ip** (*str*) -- 攝影機的 IP 位址，例如 `'10.0.0.210'`。
- **event_id** (*int*) -- 要觸發的 Cognex 事件 ID。
- **keyword** (*str*) -- 要等待的關鍵字。
- **timeout** (*float or None*) -- 等待請求的逾時時間（單位為 [s]）。使用 `None` 表示無限期待。

引發:

TimeoutError -- 若攝影機在逾時到期前未回應有效的字串。

範例：

請求包含字串 'my_message' 的訊息，該訊息與 Cognex In-Sight 作業中的事件 1 (SW1) 相關聯。

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

disconnect_camera(*ip*)

中斷特定攝影機的連線。

參數:

ip (*str*) -- 攝影機的 IP 位址，例如 `'10.0.0.210'`。

Modbus 附加功能函数

class Modbus

Modbus 模組包含用於向外部 REST 伺服器發送請求的函式。

此附加功能的所有函式均使用 `modbus.` 前綴呼叫。

`write_user_register(address, values)`

寫入輸出 Modbus 暫存器。

參數：

- **address** (*int*) -- 要開始寫入的暫存器位址。可能的值範圍為 0x10 (16) 到 0xFF (255)。
- **values** (*int or list of int*) -- 要寫入的一個或多個值。每個暫存器最多可儲存 2 位元組的資料，因此可儲存的最大值為 65535 ($=2^{16}-1$)。若提供多個值，這些值會寫入連續的暫存器中。

範例：

將單一值寫入 Modbus 輸出暫存器的特定位址。

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

使用單一指令將多個值寫入連續的輸出暫存器。

```
# 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)

從輸入 Modbus 暫存器讀取。

參數:

- **address** (*int*) -- 要開始讀取的暫存器位址。可能的值範圍為 0x10 (16) 到 0xFF (255)。
- **count** (*int*) -- 要讀取的連續暫存器數量。預設僅讀取一個暫存器。

回傳:

從暫存器讀取到的值。此清單的長度等於 *count* 引數的值。

回傳型別:

list of int

範例：

從 Modbus 輸入暫存器的特定位址讀取單一值。

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

使用單一指令從連續的輸入暫存器讀取多個值。

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

wait_for_event(*index*)

等待直到指定索引的事件被設定。事件由 Modbus 用戶端從外部設定。

參數:

index (*int*) -- 要等待的事件索引。可能的索引為 0 到 15。

範例：

等待事件被設定。

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

REST 附加功能函数

class Rest

REST 模組包含用於向外部 REST 伺服器發送請求的函式。

此附加功能的所有函式均使用 `rest.` 前綴呼叫。

setup_client_connection(target_url)

設定用於向 REST 伺服器發送請求的用戶端連線。

參數:

target_url (*str*) -- 要連線之 REST 伺服器的 URL (一般 API 端點)。

回傳:

REST 用戶端連線物件。

回傳型別:

[RestClientConnection](#)

範例：

建立與 REST 伺服器的連線。

```
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 是由 `rest.setup_client_connection` 回傳的物件。它用於向已連線的 REST 伺服器發送請求。

get_default_headers(*method*)

讀取隨每個請求一同發送的預設標頭。

參數:

method (*str*) -- 要取得標頭的方法 ("get" 或 "post")。

回傳:

此請求方法目前設定的預設標頭。

回傳型別:

dict

範例 :

讀取 GET 與 POST 方法的預設標頭。

```
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*)

變更隨每個請求一同發送的預設標頭。

參數:

- **method** (*str*) -- 要設定標頭的方法 ("get" 或 "post")。
- **new_headers** (*dict*) -- 要覆寫的新預設標頭。

範例：

建立與 REST 伺服器的連線。

```
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)
```

向 REST 伺服器發送 GET 請求。

參數:

- **resource** -- 要存取的資源名稱。資源名稱會附加到 URL 之後 (<url>/<resource>)，並可選擇性地包含開頭的 '/' 字元。
- **headers_override** (*dict*) -- 為此個別請求設定標頭。若未指定，則使用預設標頭。
- **query_params** (*dict*) -- 指定查詢 (若有)。
- **secure_connection** (*bool*) -- 是否使用安全連線 (驗證憑證)。

回傳:

回應碼 (HTTP 代碼，例如請求成功時為 `200`) 與回應內容 (回應資料) (若有)

回傳型別:

(int, dict)

範例：

向 REST 伺服器發送 GET 請求，並檢查其回應碼。

```
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)
```

向 REST 伺服器發送 POST 請求。

參數:

- **resource** (*str*) -- 要存取的資源名稱。資源名稱會附加到 URL 之後 (`<url>/<resource>`)，並可選擇性地包含開頭的 '/' 字元。
- **data** (*str or dict*) -- POST 請求的主體 / 資料。
- **headers_override** (*dict*) -- 為此個別請求設定標頭。若未指定，則使用預設標頭。
- **query_params** (*dict*) -- 指定查詢 (若有)。
- **secure_connection** (*bool*) -- 是否使用安全連線 (驗證憑證)。

回傳:

回應碼 (HTTP 代碼，例如請求成功時為 `200`) 與回應內容 (回應資料) (若有)

回傳型別:

(int, dict)

範例：

向 REST 伺服器發送 POST 請求，並檢查其回應碼。

```
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)
```

ROS 附加功能函数

class ROSHandler

ROS 模組為其他附加功能以及使用者的直接存取提供 ROS 功能。

此附加功能的所有函式均使用 `ros_handler.` 前綴呼叫。

ros_node()

ROS 模組會建立自己的 ROS 節點，可透過此函式直接存取。這可用於在腳本中建立發布者、訂閱者等物件。此外，也可為此目的在應用程式中直接匯入 `rclpy`。

⚠ 警告

請勿在此節點上執行 `rclpy.spin()`，因為它已在背景執行中！

範例：

列印機器人所發布之所有主題的名稱。

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

回傳：

ROS 模組的 ROS 節點。

回傳型別：

`rclpy.node.Node`