

AX Portal ソフトウェアコマンドリファレンス

翻訳版

© Seiko Epson Corporation 2026

Rev.2
JAM268S9128F

目次

• Python基礎	3
• コア関数	12
• アドオン関数	159
◦ Cognexアドオン関数	160
◦ Modbusアドオン関数	166
◦ RESTアドオン関数	170
◦ ROSアドオン関数	176

Python基礎

Python基本規約

ここでは、アプリケーションエディターで利用できる基本的な [Python](#) プログラミングについて説明します。ここでは、Python (3.13)の規約の主なものについてまとめています。ここで説明する以外に、多くのPython組み込み関数可以使用されます。ロボットアプリケーションは単純なPythonアプリケーションではないため、一部のPython機能は制限されるか使用できないことがあります。高度なPythonユーザーやプログラミングチュートリアルについては、オンラインの [Python 3.13 Documentation](#) を参照してください。

コメント

以下のコード行は、実行時に処理されないコードコメントです。

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

変数定義

Pythonは多くの変数型をサポートし、変数定義と変数宣言を区別しません。

```
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"
```

変数変換

値が互換性を持つ場合、変数は簡単に別の型に変換できます。

```
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
```

データ構造

Pythonは4種類のデータ構造をサポートします。

- リストは順序付きの要素配列を含みます。

```
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
```

- タプルは順序付きですが変更できない要素配列を含みます。

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

- セットは順序なしで重複のない要素配列を含みます。

```
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
```

- 辞書はキーと値の順序なしの組み合わせを含みます。

```
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
```

ライブラリーとインポート

Pythonには、組み込みモジュール群 (標準ライブラリー) が用意されています。次のモジュールをサポートしています。

- 数値演算および乱数: 計算、数学関数、乱数生成に使用します。

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

- 時刻および日付の処理: タイムスタンプ、待機時間、日時データを扱います。

```
import time, datetime
```

- データ保存およびファイル形式: JSONやCSVなどの構造化データを読み書きします。

```
import json, csv
```

- テキスト処理: 文字列操作や正規表現によるパターンマッチングを使用します。

```
import re, string
```

- 高度な反復処理および関数ツール: 効率的なループ、組み合わせ処理、高階関数を使用します。

```
import itertools, functools
```

標準ライブラリーに加えて、次のサードパーティーライブラリーを使用できます。

- 数値計算: 配列や数値データを効率的に処理します。

```
import numpy
```

- 3D姿勢演算: クォータニオンによる表現および変換を扱います (ロボット関連)。

```
import pyquaternion
```

- HTTP通信: ウェブサービスやREST APIにリクエストを送信します。

```
import requests
```

- 設定およびデータのシリアル化: YAML形式のデータを扱います。

```
import yaml
```

次の標準Pythonモジュールは、ロボットアプリケーションでは使用できません。

- オペレーティングシステムへのアクセス (例: `os`, `shutil`, `subprocess`, `sys`)
- カスタム並列処理 (例: `multiprocessing`, `threading`)
- カスタムの解析およびトラブルシューティング (例: `inspect`, `traceback`)

また、ロボット用Python環境では、独自のサードパーティーライブラリーをインストールできません。

Pythonの基本関数

必須引数とオプション引数

以下の引数を持つ関数が定義された例を使って説明します。

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

`arg1`と`arg2`は標準引数であり、関数を呼び出すときに指定する必要があります。`arg3`と`arg4`は初期値を持つデフォルト引数であり、関数を呼び出すときに指定は必須ではありません。以下の関数呼び出しは、すべて有効です。

```

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

```

アンパックされた関数引数

コードドキュメントで説明されている一部の関数はアンパックされた引数を使用します。この引数は、関数によって引数の数が異なる場合に便利です。以下の関数引数を例に説明します。

```

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

```

これらの関数は、以下のように使用できます。

```

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

```

Pythonの基本演算子

計算演算子

```

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

```

比較演算子

以下の例では、xとyは任意の型です。

```

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

```

以下の例では、xとyは数値です。

```

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

```

論理演算子

以下の例では、xとyはブール値です。

```
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
```

ビット演算子

以下の例では、xとyは整数です。

```
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
```

条件付きプログラミング

If-Else条件

`if`、`elif (else if)`、`else` 条件は以下のルールに従って組み合わせできます。

- `if` 文は条件がTrueを返す場合のみ処理されます。
- `elif (else if)` 文は任意で記述します。前の `if` または `elif` 条件がすべてFalseを返し、自身の条件がTrueを返す場合のみ処理されます。
- `else` 文は任意で記述します。前の `if` または `elif` 文がすべてFalseを返す場合のみ処理されます。

一般的なif-elif-elseコードは以下のようになります。

```
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
    ...
```

使用例:

変数 `my_integer` が5、10、またはそれ以外の値かを確認します。

```
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ループ

`while` ループは以下のルールにしたがって、同じ処理を繰り返し実行します。

- `while` ループは条件がTrueを返す間、処理を繰り返します。
- 任意で `continue` キーワードを使用して、ループ内の残りのコードを無視して、次のループに移ることができます。
- 任意で `break` キーワードを使用して、このループから抜けることができます。
- `else` 文は任意で使用できますが、使用することはほとんどありません。ループ条件がFalseになった場合に実行されます。`while` ループがbreak文で終了した場合、else文は実行されません。

```
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
    ...
```

使用例:

1から10までのすべての数値を出力します。

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

無限ループ: 無限ループは、手動で停止するまでロボットにタスクを繰り返させたい場合や、特定の条件や関数でループを抜きたい場合に便利です。ループが極端に速く実行される場合は、プロセッサに余裕を与えるために待機コマンドを追加することを推奨します。

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

Forループ

`for` ループは以下のルールにしたがって、同じ処理を繰り返し実行します。

- `for` ループは反復可能なシーケンス (リスト、セット、辞書、文字列など) の各項目に対して1回ずつ繰り返し処理します。
- 任意で `continue` キーワードを使用して、ループ内の残りのコードを無視して、次のループに移ることができます。
- 任意で `break` キーワードを使用して、このループから抜けることができます。
- `else` 文は任意で使用できますが、使用することはほとんどありません。ループシーケンス内の最後の項目が処理された後に実行されます。`for` ループが`break`文で終了した場合、`else`文は実行されません。

```
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'.
    ...
```

使用例:

1から5までのすべての数値を出力します。

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

お気に入りのフルーツを出力します。

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

`python` の文字を数えます。

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

コア関数

class Core

① 注釈

ここでは、すべてのコア関数を説明します。これらの関数はアプリケーションコードで直接使用および呼び出しできます。

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

ロボットの単一または複数の関節を指定した角度に移動します。この動作では、すべての関節が同時に目標位置に到達するように同期します。

パラメータ:

- **`joints`** (*int, str, list*) -- 動作させるアクチュエーターのID。値は次のとおり指定します。
 - 単一の関節ID (例: 6)
 - 複数関節のIDリスト (例: [1, 4, 6])
 - 6つの関節すべてを動作する定数 `ALL_KIN_JOINTS`
- **`joint_position`** (*float, list of float*) -- 移動する関節の角度位置 [deg]。値は次のとおり指定します。
 - 指定したすべての関節を同じ角度に移動する単一の数値 (例: -30)
 - 指定した各関節を異なる角度に移動する複数の数値リスト (例: [90, -45, 30])
- **`joint_velocity`** (*float or None*) -- 関節の速度制限に対する最大速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 45 %)。値を指定しない場合、グローバル関節速度が使用されます。
- **`joint_acceleration`** (*float or None*) -- 移動する関節の加速度制限に対する最大加速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 60 %)。値を指定しない場合、グローバル関節加速度が使用されます。
- **`relative`** (*bool*) -- 関節位置を現在の関節位置に対する相対オフセットとして扱うかどうかを指定します。デフォルトでは絶対位置の値が使用されます。
- **`block`** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。同じ関節を使用する同期動作のみ、非ブロッキングで連続実行できます。

使用例:

ロボットのすべての関節を同期して0°に移動します。これはロボットの直立姿勢です。

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

第6関節を現在の位置から-30°移動します。

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

第2, 3, 5関節をそれぞれ20°, -40°, 20°移動します。各関節が同時に動作を終了するように移動します。この動作では最大許容加速度を使用し、速度は最大許容速度の50%のみを使用します。

```
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)
```

ロボットのTCP (ツールセンターポイント)を指定したデカルト座標に移動します。ロボットは時間最適軌跡で移動し、各関節は台形速度プロファイルで制御されます。

パラメータ:

- **coordinates** (*Coordinates, list or dict*) --
TCPの目標座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
- **configuration** (*str or None*) -- 目標ポーズの姿勢設定。'c1'から'c8'の文字列で指定します。
- **joint_velocity** (*float or None*) -- 関節の速度制限に対する最大速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 45 %)。値を指定しない場合、グローバル関節速度が使用されます。
- **joint_acceleration** (*float or None*) -- 移動する関節の加速度制限に対する最大加速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 60 %)。値を指定しない場合、グローバル関節加速度が使用されます。
- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト

- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates*, *list*, *dict*, or *None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。
- **relative** (*bool*) -- 座標を現在の座標に対する相対オフセットとして扱うかどうかを指定します。デフォルトでは絶対座標の値が使用されます。
- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。デフォルトではこの関数はブロッキングです。

使用例:

目標座標はさまざまな形式で指定できます。すべての指定コマンドは、グローバルワークフレームにおいて、x = 700mm, y = -125mm, z = 500mm, ロール = 180°, ピッチ = 0°, ヨー = -90°の位置へTCPを移動します。

```
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]))
```

すべての座標が指定されていない場合、残りの座標には現在のTCP座標が使用されます。たとえば、x座標とロール値のみを変更するには、以下のように記述します。

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

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

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

この動作では、Y軸方向に-100 mm、ヨー方向に-60°の追加移動を、速度50%、加速度最大で実行します。また、ツールオフセットにより、TCPはX軸方向に20 mmシフトします。

ツール空間での動作を定義する方法はいくつかあり、以下に一例を示します。

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

この動作では、ツールのZ軸方向に沿って移動します。

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

ロボットのTCP (ツールセンターポイント)を指定したポーズに移動します。ロボットは時間最適軌跡で移動し、各関節は台形速度プロファイルで制御されます。

パラメータ:

- **pose** (*Pose, str, or int*) -- TCPの目標ポーズ。値は次のとおり指定します。
 - Poseオブジェクト
 - データベースに保存された定義済みポーズの名前
 - データベースに保存された定義済みポーズのID
- **joint_velocity** (*float or None*) -- 関節の速度制限に対する最大速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 45 %)。値を指定しない場合、グローバル関節速度が使用されます。
- **joint_acceleration** (*float or None*) -- 移動する関節の加速度制限に対する最大加速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 60 %)。値を指定しない場合、グローバル関節加速度が使用されます。
- **tool_offset** (*Coordinates, list, dict, or None*) -- ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) -- フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) -- ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。
- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。デフォルトではこの関数はブロッキングです。

使用例:

データベースに保存されたポーズへ移動します。

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

スクリプトで `create_pose()` 関数を使用してカスタムポーズを指定することもできます。

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

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

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

この動作では、データベースに保存された"final_pose"に、関節速度45%、関節加速度60%で移動します。また、ワークフレームにオフセットが適用され、目標ポーズはZ軸方向に50mmシフトします。

ツール空間での動作を定義する方法はいくつかあり、以下に一例を示します。

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

この動作では、ツールのZ軸方向に沿って移動します。

```
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)
```

ロボットのTCP (ツールセンターポイント)を現在のポーズから目標のポーズへ直線的に移動します。位置と向きは軌跡に沿って線形補間されます。この動作は、経路上のすべてのポーズが到達可能な場合のみ実行できます。

パラメータ:

- **coordinates** (*Coordinates, list, or dict*) --
TCPの目標座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
- **linear_velocity** (*float or None*) -- ロボットのすべての可動部に対するツールの最大直線速度 (単位: [mm/s])。
- **linear_acceleration** (*float or None*) -- ロボットのすべての可動部に対するツールの最大直線加速度 (単位: [mm/s²])。
- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。
- **relative** (*bool*) -- 座標を現在の座標に対する相対オフセットとして扱うかどうかを指定します。デフォルトでは絶対座標の値が使用されます。
- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。デフォルトではこの関数はブロッキングです。

使用例:

目標座標はさまざまな形式で指定できます。すべての指定コマンドは、グローバルワークフレームにおいて、x = 600mm, y = -125mm, z = 500mm, ロール = 180°, ピッチ = 0°, ヨー = -90°の位置へTCPを直線的に移動します。

```
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]))
```

すべての座標が指定されていない場合、残りの座標には現在のTCP座標が使用されます。たとえば、x座標とロール値は、以下のように変更できます。

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

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

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

この動作では、X軸方向に100 mmの追加移動を、速度500 mm/s、加速度1000 mm/s²で実行します。また、ワークフレームは指定した座標で上書きされます。

ツール空間での動作を定義する方法はいくつかあり、以下に一例を示します。

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

この動作では、ツールのZ軸方向に沿って直線的に移動します。

```
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)
```

ロボットのTCP (ツールセンターポイント)を現在のポーズから中間位置を経由して目標のポーズへ円弧軌跡に沿って移動します。向きは軌跡に沿って線形補間されます。この動作は、経路上のすべてのポーズが到達可能な場合のみ実行できます。

パラメータ:

- **intermediate_position** (*list*) -- 目標座標に向かうパス上で経由する中間位置。中間位置に応じて、円弧動作の曲率が決まります。
- **coordinates** (*Coordinates, list, or dict*) -- TCPの目標座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

- **linear_velocity** (*float or None*) -- ロボットのすべての可動部に対するツールの最大直線速度 (単位: [mm/s])。

- **linear_acceleration** (*float*) -- ロボットのすべての可動部に対するツールの最大直線加速度 (単位: [mm/s²])。

- **tool_offset** (*Coordinates, list, dict, or None*) -- ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) -- フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) -- ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

- **relative** (*bool*) -- 座標を現在の座標に対する相対オフセットとして扱うかどうかを指定します。デフォルトでは絶対座標の値が使用されます。
- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。デフォルトではこの関数はブロッキングです。

使用例:

ロボットはグローバルワークフレームにおいて、 $x = 0\text{mm}$, $y = -500\text{mm}$, $z = 500\text{mm}$, ロール = 180° , ピッチ = 0° , ヨー = -90° の位置へ円弧軌跡で移動します。動作中に、中間位置 $x = 0\text{mm}$, $y = -500\text{mm}$, $z = 500\text{mm}$ を経由します。

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

すべての座標が指定されていない場合、残りの座標には現在のTCP座標が使用されます。

```
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})
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
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)
```

この動作では、速度 500 mm/s 、加速度 1000 mm/s^2 でTCPを指定した座標に移動します。また、ツールオフセットにより、TCPはX軸方向に 20 mm シフトします。

```
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)
```

向きのみが変化するようにTCP (ツールセンターポイント)を移動します。ロボットは位置を固定したまま、指定したツール方向に応じて線形に回転します。この動作は、経路上のすべてのポーズが到達可能な場合のみ実行できます。

パラメータ:

- **tool_orientation** (*list*) -- 航法角 (ロール、ピッチ、ヨー)で指定するツールの方向 (単位: [deg])。
- **angular_velocity** (*float or None*) -- TCPのツール最大角速度 (単位: [deg/s])。
- **angular_acceleration** (*float or None*) -- TCPのツール最大角加速度 (単位: [deg/s²])。
- **tool_offset** (*Coordinates, list, dict, or None*) -- ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) -- フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) -- ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) -- ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト

- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

- **relative** (*bool*) -- 座標を現在の座標に対する相対オフセットとして扱うかどうかを指定します。デフォルトでは絶対座標の値が使用されます。
- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。デフォルトではこの関数はブロッキングです。

使用例:

目標方向は、すべての角度を指定するか、一部のみ指定し、残りを'None'にできます。後者の場合、指定されていない角度には、現在の方向の値が使用されます。

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

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

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

この動作では、ロール軸方向に-60°の相対移動を、速度250°/s、加速度3000°/s²で実行します。また、ツールフレームは指定した座標で上書きされます。

ツール空間での動作を定義する方法はいくつかあり、以下に一例を示します。

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

この動作では、ツールのZ軸方向を中心に回転します。

```
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)
```

現在のロボットポーズから開始し、定義された経由点を經由して、TCP (ツールセンターポイント) を移動します。

パラメータ:

- **poses** (*list of str, int or Pose*) -- 経由点のリスト。
- **joint_velocity** (*float or None*) -- 関節の速度制限に対する最大速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 45 %)。値を指定しない場合、グローバル関節速度が使用されます。
- **joint_acceleration** (*float or None*) -- 移動する関節の加速度制限に対する最大加速度 (単位: パーセント)。1つの数値で、すべての関節を同じ最大値に制限します (例: 60 %)。値を指定しない場合、グローバル関節加速度が使用されます。
- **blend_radius** (*float or None*) -- 経由点間の移動時にツール空間で許容される最大偏差を定義するブレード半径 (単位: [mm])。
- **tool_offset** (*Coordinates, list, dict, or None*) -- ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) -- フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) -- ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - *Coordinates*オブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --

ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。

- Coordinatesオブジェクト
- [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。同じ関節を使用する同期動作のみ、非ブロッキングで連続実行できます。

使用例:

経由点は、複数の形式で指定できます (`Pose` オブジェクト、データベース内のポーズ名、データベース内のポーズ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])
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

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

この動作では、経由点に沿って移動し、最大50mmの偏差を許容します。速度50%、加速度最大で動作します。

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

定義したセグメントに従ってTCP (ツールセンターポイント)を移動します。現在のロボットポーズから開始します。

パラメータ:

- **segments** (*list*) -- パスを構成するセグメントのリスト。
- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。同じ関節を使用する同期動作のみ、非ブロッキングで連続実行できます。

使用例:

セグメントは、スクリプトでSegmentオブジェクトを使用して定義できます (`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])
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
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)
```

この動作では、ツールオフセットで定義されたTCPのわずかなシフトを使用してセグメントを実行します。動作は非ブロッキングに設定されます。

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

指定されたパスに従ってTCP (ツールセンターポイント)を移動します。最初に、ロボットはデフォルトの速度と加速度で、パスの初期ポーズに移動します。その後、セグメントは仕様に従って順に実行されます。

パラメータ:

- **path** (*Path, str, or int*) -- TCPがたどるパス。
- **previous_angles** (*list or None*) --
開始姿勢に対して、最も近い解となる関節角度を計算します。値は次のとおり指定します。
 - 現在のロボット関節角度を使用するNone
 - 関節角度のリスト (単位 :[deg])

- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト

- [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

- **block** (*bool*) -- 動作が終了するまでプログラムの実行を待機するかどうかを指定します。同じ関節を使用する同期動作のみ、非ブロッキングで連続実行できます。

使用例:

データベースに保存されたパスを実行します。

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

スクリプトで `create_path()` 関数を使用してカスタムパスを指定することもできます。

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

`Path` オブジェクトを使用してパスを作成し、実行することもできます。

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

run_path(path)
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

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

この動作では、ツールフレームで定義されたTCPのわずかなシフトを使用して"fancy_path"を実行します。動作は非ブロッキングに設定されます。

wait_for_motor(*actuator_ids=None*)

単一または複数の関節モーターが停止するまで待機します。

パラメータ:

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

待機対象とするアクチュエーターのID。値は次のとおり指定します。

- 単一の関節ID (例: 6)
- 複数関節のIDリスト (例: [1, 4, 6])
- 複数関節の関節名セット (例: {"1", "4", "6"})
- すべての関節を読み取る定数 *ALL_KIN_JOINTS* または *None*

使用例:

この例は、ロボットの移動中の時間をカスタム用途に利用します。

```
# 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()

接続されているすべてのアクチュエーターに停止信号を送信し、動作を停止します。

使用例:

この例は、非ブロッキング動作を'stop_motion'関数で中止します。ここでは、特定のデジタル入力が切り替わったときに動作を中止します。

```
# 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*)

単一または複数のモーターが動作中か確認します。

パラメータ:

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

確認するアクチュエーターのID。値は次のとおり指定します。

- 単一の関節ID (例: 6)
- 複数関節のIDリスト (例: [1, 4, 6])
- 複数関節の関節名セット (例: {1, 4, 6})
- すべての関節を読み取る定数 *ALL_KIN_JOINTS* または *None*

戻り値:

指定したモーターが動作中かどうか。以下のいずれかです。

- 単一のモーターを確認したときはブール値
- 複数のモーターを確認したときはブール値のリスト

戻り値の型:

bool or list of bool

使用例:

この例は、非ブロッキング動作を'stop_motion'関数で中止します。ここでは、特定のデジタル入力が切り替わったときに動作を中止します。

```
# 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)

PTP動作関数で使用する `joint_velocity` 引数の新しいグローバルデフォルト値を設定します。関節速度の値は、最大関節速度制限に対するパーセントで指定します。

パラメータ:

joint_velocity (*float*) -- 最大関節速度に対するグローバル関節速度 (単位: [%])。

使用例:

同じ速度制限を使用して、2つの連続するPTP動作を呼び出します。

```
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*)

PTP動作関数で使用する `joint_acceleration` 引数の新しいグローバルデフォルト値を設定します。関節加速度の値は、最大関節加速度制限に対するパーセントで指定します。

パラメータ:

joint_acceleration (*float*) -- 最大関節加速度に対するグローバル関節加速度 (単位: [%])。

使用例:

同じ加速度制限を使用して、2つの連続するPTP動作を呼び出します。

```
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*)

ツール動作やパスプランナー関数で使用する `linear_velocity` 引数のグローバルデフォルト値を設定します。直線速度の値は、ツールやロボットのすべての可動部の最大直線速度を制限します。

パラメータ:

linear_velocity (*float*) -- グローバル直線速度 (単位: [mm/s])。

使用例:

同じ速度制限を使用して、2つの連続する直線動作を呼び出します。

```
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*)

ツール動作やパスプランナー関数で使用する `linear_acceleration` 引数のグローバルデフォルト値を設定します。直線加速度の値は、ツールやロボットのすべての可動部の最大直線加速度を制限します。

パラメータ:

linear_acceleration (*float*) -- グローバル直線加速度 (単位: [mm/s²])。

使用例:

同じ加速度制限を使用して、2つの連続する直線動作を呼び出します。

```
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*)

回転動作関数で使用する `angular_velocity` 引数のグローバルデフォルト値を設定します。
角速度の値は、ツールの回転速度を制限します。

パラメータ:

angular_velocity (*float*) -- グローバル角速度 (単位: [deg/s])。

使用例:

同じ速度制限を使用して、2つの連続する回転動作を呼び出します。

```
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*)

回転動作関数で使用する `angular_acceleration` 引数のグローバルデフォルト値を設定します。角加速度の値は、ツールの回転加速度を制限します。

パラメータ:

angular_acceleration (*float*) -- グローバル角加速度 (単位: [deg/s²])。

使用例:

同じ加速度制限を使用して、2つの連続する回転動作を呼び出します。

```
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*)

動作関数やパス関数で使用する `blend_radius` 引数のグローバルデフォルト値を設定します。

パラメータ:

blend_radius (*float*) -- グローバルブレンド半径 (単位: [mm])。

使用例:

同じブレンド半径を使用して、2つの連続する経由点を呼び出します。

```
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)

動作関数やキネマティクス関数で使用する `work_frame` 引数のグローバルデフォルト値を設定します。ワークフレームは、ロボットの基準フレームを変更するための変換で、ベースフレームに対して指定します。

パラメータ:

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

ベースフレームに対して指定したグローバルワークフレーム。値は次のとおり指定します。

- Coordinatesオブジェクト
- [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

使用例:

同じワークフレームを使用して、2つの連続する動作を呼び出します。

```
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)

動作関数やキネマティクス関数で使用する `tool_frame` 引数のグローバルデフォルト値を設定します。ツールフレームは、ロボットのTCPを変更するための変換で、フランジフレームに対して指定します。

パラメータ:

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

フランジフレームに対して指定したグローバルツールフレーム。値は次のとおり指定します。

- Coordinatesオブジェクト
- [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

使用例:

同じツールフレームを使用して、2つの連続する動作を呼び出します。

```
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)
```

ロボットの現在のTCP (ツールセンターポイント) ポーズを読み取ります。フレームを指定しない場合、グローバルフレーム基準のポーズを読み取ります。

パラメータ:

- **tool_offset** (*Coordinates, list, dict, or None*) --
 ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
 フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
 ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
 ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

ロボットの現在のTCPポーズ。

戻り値の型:

Pose

使用例:

2通りの方法でポーズのプロパティを読み取ります。

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

現在のポーズはセンサーデータから読み取ります。センサー値は変動し、常に正確な値を返すとは限りません。そのため、次の例のように現在のポーズを基準に前後に移動すると、時間の経過とともに予期しないずれが発生します。この場合は、代わりに `get_reference_pose` 関数を使用することを推奨します。

```
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)
```

指定された座標フレームに対するロボットの現在のTCP (ツールセンターポイント)座標を読み取ります。フレームを指定しない場合、グローバルフレーム基準の座標を読み取ります。

パラメータ:

- **tool_offset** (*Coordinates, list, dict, or None*) --
 ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
 フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
 ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
 ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

ロボットの現在のTCP座標。

戻り値の型:

[Coordinates](#)

使用例:

グローバル座標フレーム基準で現在のTCP座標を読み取ります。

```
coordinates = get_current_coordinates()
```

現在の座標はセンサーデータから読み取ります。センサー値は変動し、常に正確な値を返すとは限りません。そのため、次の例のように現在のポーズを基準に前後に移動すると、時間の経過とともに予期しないずれが発生します。この場合は、代わりに *get_reference_coordinates* 関数を使用することを推奨します。

```
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()

現在のロボットポーズの姿勢設定を読み取ります。姿勢設定は現在のグローバルフレームに依存しません。

戻り値:

現在の姿勢設定 ('c1'から'c8'のいずれか)。

戻り値の型:

str

使用例:

2通りの方法で現在のロボットポーズの姿勢設定を読み取ります。

```
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)
```

ロボットの基準TCP (ツールセンターポイント) ポーズを取得します。フレームを指定しない場合、グローバルフレーム基準のポーズを返します。このポーズは、ドライブに送信された基準角度から計算されます。

パラメータ:

- **tool_offset** (*Coordinates, list, dict, or None*) --
 ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
 フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
 ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
 ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

ロボットの基準TCPポーズ。

戻り値の型:

Pose

使用例:

2通りの方法で基準ポーズのプロパティーを読み取ります。

```
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)
```

指定された座標フレームに対するロボットの基準TCP (ツールセンターポイント)座標を取得します。フレームを指定しない場合、グローバルフレーム基準の座標を読み取ります。この座標は、ドライブに送信された基準角度から計算されます。

パラメータ:

- **tool_offset** (*Coordinates, list, dict, or None*) --
 ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
 フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
 ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
 ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

ロボットの基準TCP座標。

戻り値の型:

[Coordinates](#)

使用例:

グローバル座標フレーム基準で基準TCP座標を読み取ります。

```
coordinates = get_reference_coordinates()
```

get_reference_configuration()

基準ロボットポーズの姿勢設定を取得します。姿勢設定は基準グローバルフレームに依存しません。この姿勢設定は、ドライブに送信された基準角度から計算されます。

戻り値:

基準姿勢設定 ('c1'から'c8'のいずれか)。

戻り値の型:

str

使用例:

2通りの方法で基準ロボットポーズの姿勢設定を読み取ります。

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

create_pose(coordinates, configuration)

指定した座標と姿勢設定からPoseオブジェクトを作成します。

パラメータ:

- **coordinates** (*Coordinates*, *list*, or *dict*) --
ポーズのTCP座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
- **configuration** (*str*) -- ポーズの姿勢設定。'c1'から'c8'のいずれかを指定します。

戻り値:

□ロボットのPoseオブジェクト。

戻り値の型:

Pose

使用例:

座標値のリストと指定した姿勢設定からポーズを作成します。

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

get_pose(*pose*)

データベースに保存された定義済みポーズからPoseオブジェクトを読み取ります。

パラメータ:

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

座標を読み取る対象のロボットポーズ。以下のいずれかです。

- Poseオブジェクト
- データベースに保存されたポーズの名前 (str)
- データベースに保存されたポーズのID (int)

戻り値:

Poseオブジェクト。

戻り値の型:

Pose

使用例:

名前が"grasping_pose"のポーズをデータベースから読み込みます。

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

create_path(*initial_pose*, *segments*)

指定した初期ポーズとセグメントからPathオブジェクトを作成します。

パラメータ:

- **initial_pose** (*Pose*) -- 初期ロボットポーズ。
- **segments** (*list*) -- パスを構成するセグメントのリスト。segmentsはあとから追加することもできます。

戻り値:

ロボットのPathオブジェクト。

戻り値の型:

[Path](#)

使用例:

初期ポーズと複数のセグメントからパスを作成します。

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

get_path(*path*)

データベースに保存された定義済みパスからPathオブジェクトを読み取ります。

パラメータ:

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

取得するロボットのパス。以下のいずれかです。

- Pathオブジェクト
- データベースに保存されたパスの名前 (str)
- データベースに保存されたパスのID (int)

戻り値:

Pathオブジェクト。

戻り値の型:

Path

使用例:

名前が"grasping_path"のパスをデータベースから読み込みます。

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

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

LineSegmentオブジェクトを作成します。

パラメータ:

- **coordinates** (*list, dict, or [Coordinates](#)*) -- 目標座標。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、LineSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

```
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)
```

CircularSegmentオブジェクトを作成します。

パラメータ:

- **intermediate_position** (*list*) -- 目標座標に向かうパス上で経由する中間位置 [x, y, z]。中間位置に応じて、円弧動作の曲率が決まります。
- **coordinates** (*list, dict or [Coordinates](#)*) -- 目標座標。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、CircularSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

```
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)
```

OrientationSegmentオブジェクトを作成します。

パラメータ:

- **orientation** (*list*) -- 目標方向。位置は一定に保たれます。
- **angular_velocity** (*float or None*) -- 指定する角速度 (単位: [deg/s])。角速度を指定しない場合、グローバル角速度が使用されます。
- **angular_acceleration** (*float or None*) -- 指定する角加速度 (単位: [deg/s²])。角加速度を指定しない場合、グローバル角加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を0より大きくすると、ツールセンターポイント (TCP)はセグメントの位置に到達する前に回転し始めます。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、OrientationSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

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

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

CoordinatesSegmentオブジェクトを作成します。

パラメータ:

- **coordinates** (*list, dict or Coordinates*) -- 目標座標。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、CoordinatesSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

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

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

PoseSegmentオブジェクトを作成します。

パラメータ:

- **pose** (*Pose*) -- 目標ポーズ。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、PoseSegmentオブジェクトを作成します。

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

デフォルト値とデータベース内のポーズを使用します。

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

get_coordinates(*pose*)

指定したポーズの座標を読み取ります。座標は現在のグローバルフレームに依存しません。

パラメータ:

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

座標を読み取る対象のロボットポーズ。以下のいずれかです。

- Poseオブジェクト
- データベースに保存されたポーズの名前 (*str*)
- データベースに保存されたポーズのID (*int*)

戻り値:

ポーズの座標。

戻り値の型:

Coordinates

使用例:

"grasping_pose"と名前をつけて保存したポーズの座標をデータベースから読み取ります。

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

ID 3として保存したポーズの座標をデータベースから読み取ります。

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

get_configuration(pose)

指定したポーズの姿勢設定を読み取ります。

パラメータ:

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

姿勢設定を読み取るポーズ。値は次のとおり指定します。

- Poseオブジェクト
- データベースに保存された定義済みポーズの名前
- データベースに保存された定義済みポーズのID

戻り値:

ポーズの姿勢設定 ('c1'から'c8'のいずれか)。

戻り値の型:

str

使用例:

名前が"grasping_pose"のポーズの姿勢設定をデータベースから読み取ります。

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

2通りの方法でPoseオブジェクトから姿勢設定を読み込みます。

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

parse_coordinates(coordinates)

Coordinatesオブジェクトを作成します。

パラメータ:

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

特定の座標セットは以下のように指定します。

- [x, y, z, ロール, ピッチ, ヨー]形式のリスト (位置の単位: [mm]、方向の単位: [deg])
- {'x': x, 'y': y, 'z': z, 'roll': ロール, 'pitch': ピッチ, 'yaw': ヨー}形式の辞書 (位置の単位: [mm]、方向の単位: [deg])
- Coordinatesオブジェクト (ほかのCoordinatesオブジェクトを複製)

コンストラクター:

座標値からCoordinatesオブジェクトを作成します。

```
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)
```

相対ツールフレームとワークフレームの差を使用して、異なるフレーム間でロボット座標を変換します。相対ツールフレームとワークフレームのどちらも指定しない場合、座標は変換されず、変更されていないCoordinatesオブジェクトを返します。

パラメータ:

- **coordinates** (*Coordinates, list, or dict*) --
変換するTCP座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
- **relative_tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **relative_work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

変換後のロボット座標。

戻り値の型:

[Coordinates](#)

使用例:

座標セットを別のワークフレームに変換します。ワークフレームの原点が変換対象の座標と同じ場合、変換結果はすべての値がゼロのCoordinatesオブジェクトになります。

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

get_grid_points(*grid_name*)

データベースに保存された特定のグリッドから、すべてのグリッドポイントの位置を読み取ります。

パラメータ:

grid_name (*str*) -- ポイントを抽出するグリッド名。

戻り値:

グリッドプロパティで定義された順序で並んだグリッドポイントのリスト。

戻り値の型:

list

使用例:

定義済みグリッド内のすべてのポイントに移動します。

```
# 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)
```

順キネマティクスは、指定した関節角度からTCP (ツールセンターポイント)のポーズを計算するロボットキネマティクスの概念です。この関数の逆は 逆キネマティクス 関数で、ロボットのTCPポーズから関節角度を計算します。キネマティクスは通常、ロボットの内蔵ベースフレームとフランジフレーム間で計算されます。ただし、ワークフレームとツールフレームを定義することで基準フレームを変更できます。デフォルトでは、グローバルに定義されたワークフレームとツールフレームが使用されます。

パラメータ:

- **joint_angles** (*list of float*) -- 6つの関節角度のリスト (単位 :[deg])
- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

ロボットのPoseオブジェクト。

戻り値の型:

Pose

使用例:

特異点では、複数の関節角度のセットが同じTCPポーズになります。以下の例では、両方の関節角度が同じポーズになります。

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

2通りの方法で現在のTCPポーズを読み取ります。

```
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)
```

逆キネマティクスはTCP (ツールセンターポイント)のポーズから関節角度のセットを計算するロボットキネマティクスの概念です。この関数の逆は 順キネマティクス 関数で、ロボットの関節角度からTCPポーズを計算します。逆キネマティクス には通常複数の解がありますが、そのうちのひとつだけが返されます。

パラメータ:

- **pose** (*Pose, str, or int*) --
対応する関節角度を計算するポーズ。値は次のとおり指定します。
 - Poseオブジェクト
 - データベースに保存された定義済みポーズの名前
 - データベースに保存された定義済みポーズのID
- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
 デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト

- [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

6つの関節角度のリスト (単位 :[deg])

戻り値の型:

list of float

使用例:

データベースに保存されたポーズの関節角度を計算します。

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

特定の姿勢設定で指定座標に到達する関節角度を計算します。

```
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)
```

現在のロボットのポーズまたは特定のポーズに最も近い 逆キネマティクス の解を計算します (詳細は `inverse_kinematics` 関数を参照してください)。

パラメータ:

- **coordinates** (*Coordinates, list, or dict*) --
対応する関節角度を計算するTCP座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
- **previous_angles** (*list or None*) --
最も近い解となる関節角度を計算します。値は次のとおり指定します。
 - 現在のロボット関節角度を使用するNone
 - 関節角度のリスト (単位 :[deg])

- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。

- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。

- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。

- **work_frame** (*Coordinates, list, dict, or None*) --

ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。

- Coordinatesオブジェクト
- [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

6つの関節角度のリスト (単位 :[deg])

戻り値の型:

list of float

使用例:

逆キネマティクスを使用して、指定した座標がキネマティクスの解を持つかを確認できます。

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

同じポーズに対して複数の解を取得するには、previous_poseの記述を変更します。以下の例では、直立したロボットを前方と下方にわずかに移動した場合の2つの解を計算します。

```
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)
```

現在のロボットのポーズまたは特定のポーズに最も近い 逆キネマティクス の解を計算します (詳細は `inverse_kinematics` 関数を参照してください)。

パラメータ:

- **coordinates** (*Coordinates, list, or dict*) --
対応する関節角度を計算するTCP座標。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書
- **tool_offset** (*Coordinates, list, dict, or None*) --
ツールフレームに対する相対値で指定する、ロボットのTCPを移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではツールオフセットはなく、TCPはツールフレームの原点にあります。
- **tool_frame** (*Coordinates, list, dict, or None*) --
フランジフレームに対する相対値で指定する、ロボットのTCPを変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルツールフレームが適用されます。ツールフレームの座標がすべてゼロの場合、フランジフレームと一致します。
- **work_offset** (*Coordinates, list, dict, or None*) --
ワークフレームに対する相対値で指定する、ロボットの基準を移動するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト
 - {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトではワークオフセットはなく、基準はワークフレームの原点にあります。
- **work_frame** (*Coordinates, list, dict, or None*) --
ベースフレームに対する相対値で指定する、ロボットの基準を変更するための変換。値は次のとおり指定します。
 - Coordinatesオブジェクト
 - [x, y, z, ロール, ピッチ, ヨー]形式の座標リスト

- {"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標辞書

デフォルトでは、グローバルワークフレームが適用されます。ワークフレームの座標がすべてゼロの場合、ベースフレームと一致します。

戻り値:

6軸の角度 [deg]と姿勢設定を{'c1': [...], 'c2': [...], ...}形式で表現したリスト。

戻り値の型:

dict

使用例:

以下の例では、同じTCP座標に到達するすべての関節角度セットにロボットを移動します。

```
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)
```

移動させたいポーズに到達する関節角度のセットがひとつわかれば、この方法で異なる姿勢設定で同じポーズに到達するすべての解を計算できます。

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

wait(seconds)

指定した時間、コードの実行を停止して待機します。

パラメータ:

seconds (*float*) -- 待機時間 (単位: [s])。

使用例:

以下の例では、ロボットを定義済みポーズに移動し、動作間で3秒待機します。

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

以下の例では、指定したコードを無限に繰り返します。実行速度を制限するため、各ループの最後で100ms待機します。

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

以下の例は上記の例と似ていますが、一定時間の待機ではなく、コードを毎秒一定間隔で実行できるように待機時間を動的に設定します。

```
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)

メインアプリケーション内でブロッキングアプリケーションを実行します。

パラメータ:

- **script_name** (*str*) -- 実行するアプリケーションの名前。アプリケーションがフォルダー内にある場合は、アプリケーションへのフルパスを指定します (例: *folder/subfolder/app*)。
- **arguments** (*dict*) -- 実行するアプリケーションに渡す引数。呼び出されたアプリケーションでは、組み込み変数 *script_arguments* を使用して引数を読み取ります。

使用例:

環境温度を測定する独立したアプリケーションを例に説明します。以下の例では、1分ごとにアプリケーションを繰り返し呼び出します。

```
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)

メインアプリケーションと並行して、非ブロッキングアプリケーションを実行します。メインアプリケーションはすぐに処理を続行し、メインアプリケーションが終了するとすべての並列アプリケーションは中止します。

パラメータ:

script_name (*str*) -- 開始する並列アプリケーションの名前。アプリケーションがフォルダー内にある場合は、アプリケーションへのフルパスを指定します (例: *folder/subfolder/app*)。

使用例:

以下の例では、ロボットが特定の温度範囲でのみ動作するように、温度センサーを監視する並列アプリケーションを開始します。メインアプリケーションは継続的に動作し、動作温度の基準を満たさなくなったときに中止します。

```
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()

現在実行中のアプリケーションを停止します。メインアプリケーションとメインアプリケーションから開始されたすべてのアプリケーションに適用されます。バックグラウンドアプリケーションは実行を継続します。

例外:

ScriptInterrupt -- *run_script* 関数を使用して開始したアプリケーションからこの関数を実行した場合、メインアプリケーションで例外が発生します。

使用例:

以下のコードサンプルは バックグラウンドアプリケーションで使用でき、実行中のメインアプリケーションを一時停止し、手動で再開できます。

```
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()

現在実行中のアプリケーションを一時停止します。メインアプリケーションとメインアプリケーションから開始されたすべてのアプリケーションに適用されます。バックグラウンドアプリケーションは実行を継続します。

使用例:

この関数を使用する *stop_application* の例を参照してください。

resume_application()

一時停止したアプリケーションを再開します。この関数は 一時停止 状態でのみ実行できるため、バックグラウンドアプリケーションでのみ使用できます。

使用例:

この関数を使用する *stop_application* の例を参照してください。

stop_and_release()

現在実行中のアプリケーションを停止し、すべての関節のブレーキを解除します (重力補正モードを有効にします)。以降のアプリケーションを実行するには、手動ですべての関節ブレーキを保持する (重力補正モードを無効にする) 必要があります。

is_status(status)

プログラムの状態が指定された状態に一致するかを確認します。

パラメータ:

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

確認するプログラムの状態は、以下のいずれか、または複数です。

- 'ready' (ロボットが待機中で、コマンドの受信を待っています)
- 'processing' (ロボットがデータを処理中で、時間がかかる場合があります)
- 'hand_guidance_control' (ロボットがハンドガイド制御モードです)
- 'running' (ロボットがアプリケーションや動作を実行中です)
- 'paused' (ロボットがアプリケーションや動作を一時停止しています)
- 'emergency_stop' (非常停止がトリガーされました)
- 'normal_stop' (通常停止がトリガーされました)
- 'protective_stop' (保護停止がトリガーされました)
- 'error' (エラーまたは安全違反が発生しました)
- 'position_limit' (安全位置監視違反が発生しました)
- 'safeguard' (安全停止割り込みがトリガーされました)
- 'collision' (衝突割り込みがトリガーされました)
- 'proceed' (ロボットが割り込みから復帰しています)
- 'stopping' (ロボットがプログラムを停止しています)

戻り値:

プログラムの状態が指定された状態 (または複数の状態のいずれか) に一致するかどうか

戻り値の型:

bool

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

ボタン付きのダイアログを作成します。提示されたいずれかの選択肢でユーザーがダイアログに
応答するまで、プログラムの処理を停止します。

パラメータ:

- **text** (*str*) -- ユーザーに提示するメッセージまたは質問。
- **options** (*list of str*) -- 各ボタンのラベルとして表示される選択肢。
- **title** (*str*) -- ダイアログポップアップのタイトル。
- **dialog_type** (*int*) --
ユーザーに提示するダイアログの種類。以下のいずれかです。
 - 0: 情報 / 青色
 - 1: 成功 / 緑色
 - 2: 警告 / オレンジ色
 - 3: エラー / 赤色
- **dialog_value** (*str*) -- ダイアログに応答がない場合に返すデフォルト値。
- **mandatory** (*bool*) -- プログラムの処理を続行するためにダイアログへの応答
が必須かどうか。 *True* に設定すると、ポップアップウィンドウを閉じるなどして
ダイアログを無視できず、応答があるまで処理を停止します。

戻り値:

ユーザーが選択した値 (ユーザーが押したボタンのラベル)。

戻り値の型:

`str`

使用例:

以下の例では、"Continue"ボタンがついたかんたんな情報ポップアップウィンドウを作成しま
す。ボタンを押すと、プログラムはコードの実行を継続します。

```
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)
```

選択肢をドロップダウンリストとして表示するダイアログを作成します。ユーザーがいずれかの選択肢を選択、確定してダイアログに応答するまで、プログラムの処理を停止します。

パラメータ:

- **text** (*str*) -- ユーザーに提示するメッセージまたは質問。
- **options** (*list of str*) -- ドロップダウンリスト選択肢のラベルとして表示される選択肢。
- **title** (*str*) -- ダイアログポップアップのタイトル。
- **dialog_type** (*int*) -- ユーザーに提示するダイアログの種類。以下のいずれかです。
 - 0: 情報 / 青色
 - 1: 成功 / 緑色
 - 2: 警告 / オレンジ色
 - 3: エラー / 赤色
- **dialog_value** (*str*) -- ダイアログに応答がない場合に返すデフォルト値。
- **mandatory** (*bool*) -- プログラムの処理を続行するためにダイアログへの応答が必須かどうか。 *True* に設定すると、ポップアップウィンドウを閉じるなどしてダイアログを無視できず、応答があるまで処理を停止します。

戻り値:

ユーザーが選択した値 (ドロップダウンリストで選択した項目のラベル)。

戻り値の型:

`str`

使用例:

以下の例では、警告ポップアップウィンドウを作成し、ユーザーが選択した内容に応じてプログラムを分岐します。

```
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)
```

テキストフィールド付きのダイアログを作成します。ユーザーがテキストを入力しダイアログに回答するまで、プログラムの処理を停止します。

パラメータ:

- **text** (*str*) -- ユーザーに提示するメッセージまたは質問。
- **title** (*str*) -- ダイアログポップアップのタイトル。
- **dialog_type** (*int*) -- ユーザーに提示するダイアログの種類。以下のいずれかです。
 - 0: 情報 / 青色
 - 1: 成功 / 緑色
 - 2: 警告 / オレンジ色
 - 3: エラー / 赤色
- **dialog_value** (*str*) -- ダイアログに回答がない場合に返すデフォルト値。
- **mandatory** (*bool*) -- プログラムの処理を続行するためにダイアログへの回答が必須かどうか。 *True* に設定すると、ポップアップウィンドウを閉じるなどしてダイアログを無視できず、応答があるまで処理を停止します。

戻り値:

テキストフィールドに入力されたテキスト。

戻り値の型:

`str`

使用例:

以下の例では、名前を入力するテキストフィールドがついたかんたんなポップアップウィンドウを作成します。ユーザーが入力した名前を、ログ記録などプログラム内で使用できます。

```
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)

Yes、No ボタン付きのダイアログを作成します。ユーザーがいずれかのボタンを押すまで、プログラムの処理を停止します。

パラメータ:

- **text** (*str*) -- ユーザーに提示するメッセージまたは質問。
- **title** (*str*) -- ダイアログポップアップのタイトル。
- **dialog_type** (*int*) -- ユーザーに提示するダイアログの種類。以下のいずれかです。
 - 0: 情報 / 青色
 - 1: 成功 / 緑色
 - 2: 警告 / オレンジ色
 - 3: エラー / 赤色
- **dialog_value** (*str*) -- ダイアログに応答がない場合に返すデフォルト値。
- **mandatory** (*bool*) -- プログラムの処理を続行するためにダイアログへの応答が必須かどうか。True に設定すると、ポップアップウィンドウを閉じるなどしてダイアログを無視できず、応答があるまで処理を停止します。

戻り値:

ユーザーが"yes"を選択した場合True、"no"を選択した場合False。

戻り値の型:

bool

使用例:

以下の例では、ユーザーがアプリケーションを継続するか、中止するかを選択できるかんたんな確認ダイアログを作成します。

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

print(*args)

アプリケーションの出力にメッセージを出力します。この関数は、Pythonの組み込み関数 *print* と使用方法が似ています。複数の引数やさまざまな型の引数をサポートしています。引数は自動的に文字列に変換され、空白スペースで連結されます。

パラメータ:

args (*tuple of various*) -- 出力する引数。

使用例:

変数の値を整形して出力します。1回目の呼び出しでは自動変換と連結を使用し、2回目の呼び出しではf文字列を使用しますが、同じ結果が出力されます。

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

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

インターフェースにカスタム通知メッセージを表示します。これはプログラムのフローに影響を与えません。通知をフェードアウトさせるか、インターフェースに残すかは、設定メニューでユーザーが指定できます。

パラメータ:

- **message_text** (*str*) -- インターフェースに表示する通知テキスト。
- **message_title** (*str*) -- 通知ポップアップのタイトル。

raise_error(message)

実行中のアプリケーションにカスタムエラーメッセージを発生します。エラーの型は `ApplicationError` で、アプリケーション内で捕捉できます。捕捉されない場合、実行中のアプリケーションは中止します。エラーはログシステムの一部である `error.log` ファイルにも記録されます。ログは、インターフェースで `CTRL + SHIFT + L` を押すとダウンロードできます。

パラメータ:

message (str) -- アプリケーションで発生させるエラーメッセージ。

例外:

ApplicationError -- カスタムメッセージを含みます。

使用例:

バックグラウンドアプリケーションはデジタル入力を監視し、特定のエラー入力トリガーされているか確認します。エラー入力トリガーされた場合、アプリケーションにエラーが発生します。

```
# 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)
```

このアプリケーションエラーは、ロボットを制御するメインアプリケーションで捕捉され、処理されます。

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

raise_warning(message)

インターフェースにカスタム警告メッセージを表示します。プログラムの処理には影響しません。警告をインターフェースでフェード表示するか固定表示するかは、設定メニューで指定できます。

パラメータ:

message (*str*) -- インターフェースに表示する警告メッセージ。

使用例:

アナログ入力に温度センサーが取り付けられている場合を例に説明します。この例では、値がしきい値を超えるとインターフェースに警告を表示し、再びしきい値を下回るとリセットします。

```
# 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)
```

カスタムログファイルにメッセージを記録します。

パラメータ:

- **message** (*str*) -- 記録するカスタムメッセージ。
- **logger** (*str*) -- 記録するログファイル名。ログファイルはログフォルダー内の `custom/<logger>.log` にあります。デフォルトのファイル名は `custom/custom.log` です。ログは、インターフェースで `CTRL + SHIFT + L` を押すとダウンロードできます。
- **include_timestamp** (*bool*) -- 記録されたメッセージにタイムスタンプを追加するかどうか。

使用例:

センサー測定値を `custom/measurements.log` ファイルに記録します。

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

clear_log()

すべてのカスタムログファイルをクリアし、削除します。

set_shared_variable(name, value)

共有変数の値を設定します。

共有変数はメインアプリケーションが実行されている間のみ存在します。メインアプリケーションが終了すると、すべての共有変数は自動的に削除されます。アプリケーション間で変数の値を保持したり、ロボットの再起動後も値を保持する必要がある場合は、グローバル変数を使用してください。

パラメータ:

- **name** (*str*) -- 設定する変数名。
- **value** (*various*) -- 変数の新しい値。

使用例:

以下の例では、共有変数を使用して、メインアプリケーションから並列アプリケーションを終了させます。並列アプリケーションは終了フラグを監視し、フラグが *True* に設定されるとすぐに中止します。共有変数は効果がなくなった時点で削除されます。共有変数はメインアプリケーション終了時に消去されるため、削除の処理は必須ではありませんが、記述しておくことを推奨します。

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

メインアプリケーションは終了フラグがあることを確認し、並列アプリケーションを実行します。その後、終了フラグを *True* に設定し、並列アプリケーションを終了します。

```
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)

特定の共有変数の値を読み取ります。デフォルトでは、すべての共有変数の値を読み取ります。

共有変数はメインアプリケーションが実行されている間のみ存在します。メインアプリケーションが終了すると、すべての共有変数は自動的に削除されます。アプリケーション間で変数の値を保持したり、ロボットの再起動後も値を保持する必要がある場合は、グローバル変数を使用してください。

パラメータ:

name (*str (or None)*) -- 値を返す変数名 (すべての変数を返す場合はNone)。

戻り値:

値を返す変数の値 (または、すべての変数を含む辞書)

戻り値の型:

various (or dict)

使用例:

`set_shared_variable` の例を参照してください。すべての共有変数関数の使用方法を組み合わせています。

remove_shared_variable(name)

共有変数を削除します。削除後、その値にはアクセスできません。

共有変数はメインアプリケーションが実行されている間のみ存在します。メインアプリケーションが終了すると、すべての共有変数は自動的に削除されます。アプリケーション間で変数の値を保持したり、ロボットの再起動後も値を保持する必要がある場合は、グローバル変数を使用してください。

パラメータ:

name (str) -- 削除する変数名。

使用例:

`set_shared_variable` の例を参照してください。すべての共有変数関数の使用方法を組み合わせています。

set_global_variable(name, value)

グローバル変数の値を設定します。

グローバル変数の変更は永続的であり、ロボットをシャットダウンや再起動しても保持されます。そのため、不要になった グローバル変数 は削除することを推奨します。単一のアプリケーションを実行する間のみ変数が必要な場合は、共有変数 を使用してください。

パラメータ:

- **name** (*str*) -- 設定する変数名。
- **value** (*various*) -- 変数の新しい値。

使用例:

以下の例では、処理を繰り返すアプリケーションで処理が成功した回数を数えます。初回実行時にグローバルカウンターを初期化し、各ループの最後にカウンターを増やすことで実現します。

```
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)

特定のグローバル変数の値を読み取ります。デフォルトでは、すべてのグローバル変数の値を読み取ります。

グローバル変数の変更は永続的であり、ロボットをシャットダウンや再起動しても保持されます。そのため、不要になったグローバル変数は削除することを推奨します。単一のアプリケーションを実行する間のみ変数が必要な場合は、共有変数を使用してください。

パラメータ:

name (str (or None)) -- 値を返す変数名 (すべての変数を返す場合はNone)。

戻り値:

値を返す変数の値 (または、すべての変数を含む辞書)

戻り値の型:

various (or dict)

使用例:

`set_global_variable` の例を参照してください。すべての `グローバル変数` 関数の使用方法を組み合わせています。

remove_global_variable(name)

グローバル変数を削除します。削除後、その値にはアクセスできません。

グローバル変数の変更は永続的であり、ロボットをシャットダウンや再起動しても保持されます。そのため、不要になった グローバル変数 は削除することを推奨します。単一のアプリケーションを実行する間のみ変数が必要な場合は、共有変数 を使用してください。

パラメータ:

name (str) -- 削除する変数名。

使用例:

以下の例では、すべてのグローバル変数を削除します。

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

set_state_variable(name, value)

状態変数の値を設定します。

状態変数はアプリケーションごとに個別に定義され、メインアプリケーションでのみ使用されます。状態変数は、アプリケーションエディターの 変数 タブで設定、カスタマイズができます。状態変数にはそれぞれ特定のデータ型があり、同一型の値のみ受けつけます。実行中のアプリケーションを監視、制御するため、パネルインターフェースに変数の値をリアルタイムで表示することに、状態変数は主に使用されます。

パラメータ:

- **name** (*str*) -- 設定する変数名。
- **value** (*various*) -- 変数の新しい値。

使用例:

percent 型で *progress* と名前をつけた読み取り専用の状態変数と、*bool* 型で *logging* と名前をつけた動的に編集できる状態変数を定義したアプリケーションを例に説明します。*progress* 変数は、プログラムの現在の進捗をパーセントで保持するために使用します。*logging* フラグは、進捗の更新をログに記録するかどうかを示すために使用します。ユーザーは *logging* の値をいつでも変更でき、動的にプログラムに適用されます。

```
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)

特定の状態変数の値を読み取ります。デフォルトでは、すべての状態変数の値を読み取ります。

状態変数はアプリケーションごとに個別に定義され、メインアプリケーションでのみ使用されます。状態変数は、アプリケーションエディターの 変数 タブで設定、カスタマイズができます。状態変数にはそれぞれ特定のデータ型があり、同一型の値のみ受けつけます。実行中のアプリケーションを監視、制御するため、パネルインターフェースに変数の値をリアルタイムで表示することに、状態変数は主に使用されます。

パラメータ:

name (*str (or None)*) -- 値を返す変数名 (すべての変数を返す場合はNone)。

戻り値:

値を返す変数の値 (または、すべての変数を含む辞書)

戻り値の型:

various (or dict)

使用例:

`set_state_variable` の例を参照してください。すべての 状態変数 関数の使用方法を組み合わせています。

read_actuator_position(*actuator_ids=None*)

単一、複数、すべてのアクチュエーターの角度位置を読み取ります。

パラメータ:

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

読み取るアクチュエーターのID。値は次のとおり指定します。

- 単一の関節ID (例: 6)
- 複数関節のIDリスト (例: [1, 4, 6])
- 複数関節の関節名セット (例: {1, 4, 6})
- すべての関節を読み取る定数 *ALL_KIN_JOINTS* または *None*

戻り値:

読み取ったアクチュエーターの角度位置 (単位: [deg])。以下の通りです。

- 単一の関節位置の値 -> float
- 関節位置のリスト -> floatのリスト
- 関節IDと位置の値の組み合わせ -> floatの辞書

戻り値の型:

float, or list of float, or dict of float

使用例:

手首関節アクチュエーターの現在位置を出力します。

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

read_actuator_velocity(*actuator_ids=None*)

単一、複数、すべてのアクチュエーターの角速度を読み取ります。

パラメータ:

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

読み取るアクチュエーターのID。値は次のとおり指定します。

- 単一の関節ID (例: 6)
- 複数関節のIDリスト (例: [1, 4, 6])
- 複数関節の関節名セット (例: {1, 4, 6})
- すべての関節を読み取る定数 *ALL_KIN_JOINTS* または *None*

戻り値:

読み取ったアクチュエーターの角速度 (単位: [deg/s])。以下の通りです。

- 単一の関節速度の値 -> float
- 関節速度のリスト -> floatのリスト
- 関節IDと速度の値の組み合わせ -> floatの辞書

戻り値の型:

float, or list of float, or dict of float

使用例:

動作中の速度を監視し、最も速く動作しているアクチュエーターを特定します。

```
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()

ロボットのTCP (ツールセンターポイント)の直線速度を取得します。

戻り値:

ツールの直線速度 (単位: [mm/s])。

戻り値の型:

list

使用例:

現在の直線速度を取得し、出力します。

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

get_angular_velocity()

ロボットのTCP (ツールセンターポイント)の角速度を取得します。

戻り値:

ツールの角速度 (単位: [deg/s])。

戻り値の型:

list

使用例:

現在の角速度を取得し、出力します。

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

read_actuator_current(*actuator_ids=None*)

単一、複数、すべてのアクチュエーターに流れる電流値を読み取ります。

パラメータ:

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

読み取るアクチュエーターのID。値は次のとおり指定します。

- 単一の関節ID (例: 6)
- 複数関節のIDリスト (例: [1, 4, 6])
- 複数関節の関節名セット (例: {1, 4, 6})
- すべての関節を読み取る定数 *ALL_KIN_JOINTS* または *None*

戻り値:

読み取ったアクチュエーターの電流値 (単位: [A])。以下の通りです。

- 単一の関節電流値 -> float
- 関節電流値のリスト -> floatのリスト
- 関節IDと電流値の組み合わせ -> floatの辞書

戻り値の型:

float, or list of float, or dict of float

使用例:

特定のポーズで、ロボットアクチュエーターの電流を2回出力します。1回目はオブジェクトをピックする前、2回目はピックした後です。

```
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()

ツールのピック機能を有効にしてオブジェクトを把持します。機械式のツールではクランプし、空気圧式のツールでは吸着します。

使用例:

ツールのピック機能を有効にします。

```
tool_pick()
```

tool_place()

ツールのプレース機能を有効にしてオブジェクトを解放します。機械式のツールではクランプを開き、空気圧式のツールでは吸着を停止します。

使用例:

ツールのプレース機能を有効にします。

```
tool_place()
```

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

ロボットの動作中にツールのペイロードを動的に設定します。

パラメータ:

- **payload** (*float*) -- ツールのペイロード (重量 + 把持物、単位: [kg])
- **center_of_mass** (*list of float*) -- フランジフレームに対する重心[x, y, z] (単位: [m])
- **inertia** (*list of float*) -- イナーシャマトリックス (長さ9のリスト) (単位: [kg·m²])

使用例:

物体をピックおよびプレースするときに、ペイロードを変更します。

```
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*)

'デジタルメイン入力'カテゴリーの単一、複数、すべてのI/Oの値を読み取ります。

デジタルI/Oの2つの状態は以下のとおりです。

- *HIGH*: ブール値の *True* で表されます
- *LOW*: ブール値の *False* で表されます

パラメータ:

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

読み取るI/Oを指定します。identifiersは、I/O番号、またはI/Oに割り当て可能な一意のカスタム名です。以下のいずれかです。

- 単一の番号 (*int*)、またはカスタム名 (*str*) -> ブール値を返します
- 番号 (*int*)、カスタム名 (*str*)、または混在したリスト -> ブール値のリストを返します
- 番号 (*int*)、カスタム名 (*str*)、または混在したセット -> identifiersがキーとなるブール値の辞書を返します

戻り値:

読み取ったデジタルI/Oのブール値 (*HIGH = True, LOW = False*)

戻り値の型:

bool, list, dict

使用例:

デジタル入力の1つに'trigger'という名前が割り当てられている場合を例に説明します。このコードブロックは、triggerがHIGHに切り替わるまで待機し、その後実行を継続します。

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

最初の8個のデジタルI/Oを読み取り、ブール値のバイナリリストを0~255の整数に変換します。

```
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*)

'デジタルツール入力'カテゴリーの単一、複数、すべてのI/Oの値を読み取ります。

詳細は、使用方法が類似する'read_digital_main_input'を参照してください。

read_analog_main_input(identifiers)

'アナログメイン入力'カテゴリーの単一、複数、すべてのI/Oの値を読み取ります。

アナログI/Oは、電圧モード、電流モードのいずれかに設定できます。値は0.1刻みで設定および読み取ります。I/O値55は実際の電圧 5.5Vまたは電流5.5mAを示します。アナログI/Oの値の範囲は以下の通りです。

- 0.0～10.0 V (電圧モード)
- 4.0～20.0 mA (電流モード)

パラメータ:

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

読み取るI/Oを指定します。identifiersは、I/O番号、またはI/Oに割り当て可能な一意のカスタム名です。以下のいずれかです。

- 単一の番号 (int)、またはカスタム名 (str) -> ブール値を返します
- 番号 (int)、カスタム名 (str)、または混在したリスト -> ブール値のリストを返します
- 番号 (int)、カスタム名 (str)、または混在したセット -> identifiersがキーとなるブール値の辞書を返します

戻り値:

指定したアナログI/Oの数値

戻り値の型:

float, list, dict

使用例:

電流モードで、アナログ入力の1つに'temperature'と名前をつけた場合を例に説明します。この入力に温度センサーを接続し、戻り値10.6 mAが超えてはならない臨界温度のしきい値であると判断しました。メインアプリケーションとは別に次のプログラムを実行します。このプログラムは、臨界しきい値に到達した場合に処理を中止します。

```
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*)

'アナログツールI/O'カテゴリーの単一、複数、すべてのI/Oの値を読み取ります。

詳細は、使用方法が類似する'read_analog_main_input'を参照してください。

write_digital_main_output(*identifiers, values*)

'デジタルメイン出力'カテゴリーの単一、複数、すべてのI/Oに値を書き込みます。

デジタルI/Oの2つの状態は以下のとおりです。

- *HIGH*: ブール値の *True* で表されます
- *LOW*: ブール値の *False* で表されます

パラメータ:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) --
書き込むI/Oを指定します。identifiersは、I/O番号、またはI/Oに割り当て可能な一意のカスタム名です。以下のいずれかです。
 - 単一の番号 (*int*)、またはカスタム名 (*str*) -> *values* はブール値
 - 番号 (*int*)、カスタム名 (*str*)、または混在したリスト -> *values* はブール値のリスト
 - 番号 (*int*)、カスタム名 (*str*)、または混在したセット -> *values* は identifiersがキーとなるブール値の辞書
- **values** (*bool, list (of bool), dict (of bool)*) --
書き込むデジタルI/Oのブール値 (*HIGH = True, LOW = False*) identifiers に応じて、以下のいずれかです。
 - identifiers が単一のI/Oの場合、単一のブール値
 - identifiers がI/Oリストの場合、ブール値のリスト
 - identifiers がI/Oセットの場合、identifiersがキーとなるブール値の辞書

使用例:

デジタル出力の1つに'trigger'という名前が割り当てられている場合を例に説明します。以下のコードでは、デジタル出力を100 msの間HIGHに設定し、その後LOWに戻してインパルスを送信します。

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

最初の8個のデジタルI/Oに8ビット整数 (0~255)を書き込みます。まず整数をブール値リストに変換し、その後8つの値を同時に書き込みます。

```
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*)

'デジタルツール出力'カテゴリーの単一、複数、すべてのI/Oに値を書き込みます。

詳細は、使用方法が類似する'write_digital_main_output'を参照してください。

write_analog_main_output(*identifiers, values*)

'アナログメイン出力'カテゴリーの単一、複数、すべてのI/Oに値を書き込みます。

アナログI/Oは、電圧モード、電流モードのいずれかに設定できます。値は0.1刻みで設定および読み取ります。I/O値55は実際の電圧 5.5Vまたは電流5.5mAを示します。アナログI/Oの値の範囲は以下の通りです。

- 0.0～10.0 V (電圧モード)
- 4.0～20.0 mA (電流モード)

パラメータ:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) --
書き込むI/Oを指定します。identifiersは、I/O番号、またはI/Oに割り当て可能な一意のカスタム名です。以下のいずれかです。
 - 単一の番号 (int)、またはカスタム名 (str) -> *values* はfloat
 - 番号 (int)、カスタム名 (str)、または混在したリスト -> *values* はfloatのリスト
 - 番号 (int)、カスタム名 (str)、または混在したセット -> *values* は identifiersがキーとなるfloatの辞書
- **values** (*float, list (of float), dict (of float)*) --
書き込むアナログI/Oの数値 *identifiers* に応じて、以下のいずれかです。
 - *identifiers* が単一のI/Oの場合、単一のfloat
 - *identifiers* がI/Oリストの場合、floatのリスト
 - *identifiers* がI/Oセットの場合、identifiersがキーとなるブール値の辞書

使用例:

RGBの値が制御可能なLEDがそれぞれ"red"、"green"、"blue"というカスタム名を持つ3つのアナログI/Oに直接接続されている場合を例に説明します。この場合、値の最大範囲[0, 255]は電圧の最大範囲[0.0 V, 10.0 V]に対応します。以下の例では、これらの値を同時に書き込みます。

```
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)`

'アナログツールI/O'カテゴリーの単一、複数、すべてのI/Oに値を書き込みます。

詳細は、使用方法が類似する'`write_analog_main_output`'を参照してください。

wait_for_digital_main_input(*identifier*, *value*)

'デジタルメイン入力'カテゴリーのI/Oが特定の値を読み取るまで待機します。

デジタルI/Oの2つの状態は以下のとおりです。

- *HIGH*: ブール値の *True* で表されます
- *LOW*: ブール値の *False* で表されます

パラメータ:

- **identifier** (*int*, *str*) -- 待機するI/Oを指定します。identifiersは、I/O番号 (*int*)、またはI/Oに割り当てられる一意のカスタム名 (*str*)です。
- **value** (*bool*) -- 待機する値。

使用例:

デジタル入力の1つに'trigger'という名前が割り当てられている場合を例に説明します。このコードブロックは、triggerがHIGHに切り替わるまで待機し、その後実行を継続します。

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

まれに、プログラムを一時停止し、I/Oがトリガーされた場合に待機関数を終了させたい場合があるかもしれません。この待機関数では実現できませんが、以下の方法で実現できます。

以下のコードをバックグラウンドアプリケーションに追加します。特定のI/Oの値を監視し、指定した値に到達した場合はフラグを設定します。

```
# 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)
```

以下のコードをメインアプリケーションに追加し、バックグラウンド監視を開始し、フラグが設定されるまで待機します。

```
# 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*)

'デジタルツール入力'カテゴリのI/Oが特定の値を読み取るまで待機します。

詳細は、使用方法が類似する'wait_for_digital_main_input'を参照してください。

wait_for_analog_main_input(*identifier*, *value_range*)

'アナログメイン入力'カテゴリーのI/Oがしきい値を超えるまで待機します。

アナログI/Oは、電圧モード、電流モードのいずれかに設定できます。値は0.1刻みで設定および読み取ります。I/O値55は実際の電圧 5.5Vまたは電流5.5mAを示します。アナログI/Oの値の範囲は以下の通りです。

- 0.0～10.0 V (電圧モード)
- 4.0～20.0 mA (電流モード)

パラメータ:

- **identifier** (*int*, *str*) -- 待機するI/Oを指定します。identifiersは、I/O番号 (*int*)、またはI/Oに割り当てられる一意のカスタム名 (*str*)です。
- **value_range** (*list of float*) --
I/Oが範囲内の値に到達するまで待機します。以下のいずれかを指定します。
 - 下限値と上限値の両方 (例: [5.5, 7.0])
 - 下限値のみ (例: [5.5, None])
 - 上限値のみ (例: [None, 7.0])

使用例:

アナログI/Oが電圧モードに設定されている場合を例に説明します。この例では、電圧値が8.0 Vを超えるまで待機します。8.0 Vは0.1 V刻みで80として表されます。

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

wait_for_analog_tool_io(*identifier, value_range*)

'アナログツール入力'カテゴリーのI/Oがしきい値を超えるまで待機します。

詳細は、使用方法が類似する'wait_for_analog_main_input'を参照してください。

get_runtime_position_offsets()

この関数はロボットの位置オフセットを返します。

戻り値:

ロボットの位置オフセット

戻り値の型:

list of float

read_button_state(channel=None)

ロボットのインターフェースでカスタマイズ可能な単一またはすべてのボタンの状態 (押されているかどうか)を読み取ります。特定の機能が指定されたボタンは、ほかの用途で使用するため、実行時に状態を読み取れません。

ボタンは、ロボットアームの最上部の関節に配置されています。

パラメータ:

channel (*int or None*) -- ボタンを識別子、1からボタンの数までの数字。デフォルトでは、カスタマイズ可能なすべてのボタンの状態を読み取ります。

戻り値:

指定したボタンの現在の状態を示すブール値、または{1: bool, 2: bool, ...}の形式で各ボタンの状態を記述した辞書。

戻り値の型:

bool or dict of bool

使用例:

以下の例では、カスタマイズ可能なボタンの状態が変わるたびに出力します。

```
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)
```

以下の例をバックグラウンドアプリケーションとして実行します。ボタン1と2を使用して、実行中のメインアプリケーションを一時停止および再開できます。

```
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)

メッセージを送信するためのTCPソケットクライアントを作成します。クライアントは指定したアドレスのサーバーに継続的に接続を試みます。メッセージは *UTF-8* でエンコードされます。メッセージはEnd of Text文字 *0x03* (Pythonでは `\x03`) で区切られます。

パラメータ:

- **ip** (*str*) -- サーバーのIPアドレス (例: `"192.168.80.2"`)。
- **port** (*int*) -- サーバーにアクセス可能なポート番号 (例: `60001`)。ポート番号は60000～61000の範囲に制限されています。

戻り値:

メッセージの送受信に使用するTCPソケットクライアントオブジェクト。

戻り値の型:

[Socket](#)

使用例:

以下の例では、アプリケーション内のTCPソケットクライアントと同じネットワーク内のTCPソケットサーバーの間でハンドシェイクを実行します。はじめに接続を確立します。次に、クライアントはサーバーから初期メッセージが送信されるのを待機します。初期メッセージを受信すると、クライアントは自身のメッセージで応答し、接続を終了します。

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

create_tcp_server(port)

20クライアントまで処理できるTCPソケットサーバーを作成します。メッセージは *UTF-8* でエンコードされます。メッセージはEnd of Text文字 *0x03* (Pythonでは `\x03`) で区切られます。

パラメータ:

port (*int*) -- クライアントがこのサーバーにアクセスするためのポート番号 (例: `60002`)。ポート番号は60000～61000の範囲に制限されています。

戻り値:

メッセージの送受信に使用するTCPソケットサーバーオブジェクト。

戻り値の型:

`Socket`

使用例:

以下の例では、アプリケーション内のTCPソケットサーバーと最初に接続するクライアントの間でハンドシェイクを実行します。はじめに、サーバーは特定のポートで接続を開き、クライアントの接続を待機します。次に、サーバーはクライアントから初期メッセージが送信されるのを待機します。初期メッセージを受信すると、サーバーは自身のメッセージで応答し、すべてのクライアントとの接続を終了します。

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

create_udp_socket(port)

メッセージを送信するためのUDPソケットを作成します。メッセージは *UTF-8* でエンコードされます。メッセージはEnd of Text文字 *0x03* (Pythonでは `\x03`) で区切られます。

パラメータ:

port (*int*) -- このソケットにアクセスするためのポート番号 (例: `60003`)。ポート番号は60000～61000の範囲に制限されています。

戻り値:

メッセージの送受信に使用するUDPソケットオブジェクト。

戻り値の型:

[Socket](#)

使用例:

以下の例では、アプリケーション内のUDPソケットと最初の接続相手との間でハンドシェイクを実行します。はじめに、ソケットは特定のポートで接続を開き、相手の接続を待機します。次に、ソケットは相手から初期メッセージが送信されるのを待機します。初期メッセージを受信すると、ソケットは自身のメッセージで応答し、接続を終了します。

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

send_message(message)

開いているTCP接続に'script_send'キーを使用してメッセージを送信します。

パラメータ:

message (*str*) -- 送信するメッセージ。

使用例:

TCP経由で、受信可能な任意のデバイスにメッセージを送信します。

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

接続したデバイス側では、データは以下の形式で受信します。

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

receive_message(timeout=None)

開いているTCP接続から、'script_receive'アクションを呼び出してメッセージを受信します。

パラメータ:

timeout (*None or float*) -- メッセージを待機する最大時間。指定しない場合は、無期限に待機します。

使用例:

外部機器がTCP経由でロボットに接続し、以下のデータを送信します。

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

ロボット側では、この関数を使ってメッセージを受信、出力します。

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

clear_messages()

未読の可能性がある以前の受信メッセージをTCPメッセージキューからクリアします。

使用例:

新しいメッセージを待機する前に、古いメッセージを破棄することが有効な場合があります。

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

***class* Coordinates**

Coordinatesオブジェクトは、動作関数の目標座標や座標フレームの定義に使用できる数学的変換を定義します。

`__init__(coordinates)`

特定の座標値を指定するか、現在のロボット座標を含むCoordinatesオブジェクトを作成します。

パラメータ:

coordinates (*list, dict, or [Coordinates](#)*) --

特定の座標セットは以下のように指定します。

- [x, y, z, ロール, ピッチ, ヨー]形式のリスト (位置の単位: [mm]、方向の単位: [deg])
- {'x': x, 'y': y, 'z': z, 'roll': ロール, 'pitch': ピッチ, 'yaw': ヨー}形式の辞書 (位置の単位: [mm]、方向の単位: [deg])
- Coordinatesオブジェクト (ほかのCoordinatesオブジェクトを複製)

コンストラクター:

座標値からCoordinatesオブジェクトを作成します。

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

`get_current_coordinates` 関数を使用して、現在のロボット座標を含むCoordinatesオブジェクトを作成します。

```
current_coordinates = get_current_coordinates()
```

property position

Coordinatesオブジェクトの位置部分。

戻り値:

[x, y, z]形式で表現される位置情報 (単位: [mm])。

戻り値の型:

list of float

property orientation

Coordinatesオブジェクトの方向を示す部分。

戻り値:

[ロール, ピッチ, ヨー]形式で表現される方向情報 (単位: [deg])。

戻り値の型:

list of float

to_list()

Coordinatesオブジェクトを[x, y, z, ロール, ピッチ, ヨー]形式の座標値リストに変換します (位置の単位: [mm]、方向の単位: [deg])。

戻り値:

リスト形式の座標値。

戻り値の型:

list of float

使用例:

座標をリスト形式で使用する場合に、この関数を使用できます。

```
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()

Coordinatesオブジェクトを{"x": x, "y": y, "z": z, "roll": ロール, "pitch": ピッチ, "yaw": ヨー}形式の座標値辞書に変換します (位置の単位: [mm]、方向の単位: [deg])。

戻り値:

辞書形式の座標値。

戻り値の型:

dict of float

使用例:

座標を辞書形式で使用する場合に、この関数を使用できます。

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

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

inverted()

Coordinatesオブジェクトの逆変換を計算します。

戻り値:

変換オブジェクトの逆変換

戻り値の型:

Transform

使用例:

```
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オブジェクトは、座標 (位置と方向)と姿勢設定 (同じ座標に到達する複数の姿勢設定のどれを使用するか)で表現されるロボットのポーズを定義します。

`__init__(pose)`

名前またはIDを指定して保存したポーズを読み込むか、ロボットの現在のポーズからPoseオブジェクトを作成します。

パラメータ:

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

読み込むポーズは、以下のいずれかの方法で参照します。

- データベースに保存されたポーズの名前 (*str*)
- データベースに保存されたポーズのID (*int*)
- Poseオブジェクト (ほかのPoseオブジェクトを複製)

コンストラクター:

データベースからポーズを読み込みます。

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

`get_current_pose` 関数を使用して、現在のロボットポーズを含むPoseオブジェクトを作成します。

```
current_pose = get_current_pose()
```

`create_pose` 関数を使用して、指定した座標と姿勢設定を含むPoseオブジェクトを作成します。

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

property coordinates

位置と方向の値を含むCoordinatesオブジェクトとして表現されるポーズの座標。

戻り値:

ポーズの座標。

戻り値の型:

Coordinates

property configuration

特定のポーズ座標には最大8つのロボット関節の組み合わせが考えられます。一意のポーズを定義するため、数値 (0から7)または文字列 ('c1'から'c8')で姿勢設定を指定します。

戻り値:

ポーズ構成のインデックス (0～7)

戻り値の型:

int

class LinearSegment

LinearSegmentオブジェクトは、TCP (ツールセンターポイント)が指定座標まで移動する直線パスセグメントを記述します。指定する直線速度、直線加速度、セグメント開始時のブレンド半径の情報もあわせて記述します。ブレンド半径により、直線パスからのわずかな逸脱が可能になり、セグメント間のスムーズな遷移を実現します。

__init__ (*linear_segment*)

LinearSegmentオブジェクトを作成します。

パラメータ:

- **coordinates** (*list, dict, or Coordinates*) -- 目標座標。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、LinearSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

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

property coordinates

位置と方向の値を含むCoordinatesオブジェクトとして表現されるセグメントの目標座標。

戻り値:

セグメントの目標座標。

戻り値の型:

Coordinates

property blend_radius

セグメント開始時のブレンド半径。

戻り値:

開始時のブレンド半径。

戻り値の型:

float

property linear_acceleration

セグメントに指定する直線加速度。指定しない場合、グローバル直線加速度が使用されます。

戻り値:

指定する直線加速度。

戻り値の型:

float or None

property linear_velocity

セグメントに指定する直線速度。指定しない場合、グローバル直線速度が使用されます。

戻り値:

指定する直線速度。

戻り値の型:

float or None

class CircularSegment

CircularSegmentオブジェクトは、TCP (ツールセンターポイント)が中間位置を経由して、指定座標まで移動する円弧動作を記述します。中間ポーズの方向は、アルゴリズムによって定義されます。指定する直線速度、直線加速度、セグメント開始時のブレンド半径の情報もあわせて記述します。ブレンド半径により、円弧経路からのわずかな逸脱が可能になり、セグメント間のスムーズな遷移を実現します。

`__init__(circular_segment)`

CircularSegmentオブジェクトを作成します。

パラメータ:

- **intermediate_position** (*list*) -- 目標座標に向かうパス上で経由する中間位置 [x, y, z]。中間位置に応じて、円弧動作の曲率が決まります。
- **coordinates** (*list, dict or [Coordinates](#)*) -- 目標座標。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、CircularSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

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

property intermediate_position

セグメントの中間位置。[x, y, z]形式で、x, y, zの座標で表現します。

戻り値:

セグメントの中間位置。

戻り値の型:

list

property coordinates

位置と方向の値を含むCoordinatesオブジェクトとして表現されるセグメントの目標座標。

戻り値:

セグメントの目標座標。

戻り値の型:

[Coordinates](#)

property blend_radius

セグメント開始時のブレンド半径。

戻り値:

開始時のブレンド半径。

戻り値の型:

float

property linear_acceleration

セグメントに指定する直線加速度。指定しない場合、グローバル直線加速度が使用されます。

戻り値:

指定する直線加速度。

戻り値の型:

float or None

property linear_velocity

セグメントに指定する直線速度。指定しない場合、グローバル直線速度が使用されます。

戻り値:

指定する直線速度。

戻り値の型:

float or None

***class* OrientationSegment**

OrientationSegmentオブジェクトは、TCP (ツールセンターポイント)が位置を保持したまま、指定したツールの方向に回転する純粋な方向パスセグメントを記述します。指定する角速度、角加速度、セグメント開始時のブレンド半径の情報もあわせて記述します。ブレンド半径を0より大きくすると、TCPは目標位置に到達する前に回転し始めます。

***__init__*(orientation_segment)**

OrientationSegmentオブジェクトを作成します。

パラメータ:

- **orientation** (*list*) -- 目標方向。位置は一定に保たれます。
- **angular_velocity** (*float or None*) -- 指定する角速度 (単位: [deg/s])。角速度を指定しない場合、グローバル角速度が使用されます。
- **angular_acceleration** (*float or None*) -- 指定する角加速度 (単位: [deg/s²])。角加速度を指定しない場合、グローバル角加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を0より大きくすると、ツールセンターポイント (TCP)はセグメントの位置に到達する前に回転し始めます。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、OrientationSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

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

property orientation

セグメントの目標方向。航法角 [ロール, ピッチ, ヨー]を指定します (単位: [deg])。

戻り値:

セグメントの目標方向。

戻り値の型:

list

property angular_velocity

セグメントに指定する角速度。指定しない場合、グローバル角速度が使用されます。

戻り値:

指定する角速度。

戻り値の型:

float or None

property angular_acceleration

セグメントに指定する角加速度。指定しない場合、グローバル角加速度が使用されます。

戻り値:

指定する角加速度。

戻り値の型:

float or None

property blend_radius

セグメント開始時のブレンド半径。

戻り値:

開始時のブレンド半径。

戻り値の型:

float

***class* CoordinatesSegment**

CoordinatesSegmentオブジェクトは、関節空間において目標座標へ向かう直線動作を記述します。TCP (ツールセンタポイント)は通常、直線的に移動しません。指定する直線速度、直線加速度、セグメント開始時のブレンド半径の情報もあわせて記述します。ブレンド半径により、パスからのわずかな逸脱が可能になり、セグメント間のスムーズな遷移を実現します。

***__init__*(coordinates_segment)**

CoordinatesSegmentオブジェクトを作成します。

パラメータ:

- **coordinates** (*list, dict or Coordinates*) -- 目標座標。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、CoordinatesSegmentオブジェクトを作成します。

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

デフォルト値を使用します。

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

property coordinates

位置と方向の値を含むCoordinatesオブジェクトとして表現されるセグメントの目標座標。

戻り値:

セグメントの目標座標。

戻り値の型:

Coordinates

property blend_radius

セグメント開始時のブレンド半径。

戻り値:

開始時のブレンド半径。

戻り値の型:

float

property linear_acceleration

セグメントに指定する直線加速度。指定しない場合、グローバル直線加速度が使用されます。

戻り値:

指定する直線加速度。

戻り値の型:

float or None

property linear_velocity

セグメントに指定する直線速度。指定しない場合、グローバル直線速度が使用されます。

戻り値:

指定する直線速度。

戻り値の型:

float or None

class PoseSegment

Poseオブジェクトは、関節空間において目標ポーズへ向かう直線動作を記述します。TCP (ツールセンターポイント)は通常、直線的に移動しません。指定する直線速度、直線加速度、セグメント開始時のブレンド半径の情報もあわせて記述します。ブレンド半径により、パスからのわずかな逸脱が可能になり、セグメント間のスムーズな遷移を実現します。

`__init__(pose_segment)`

PoseSegmentオブジェクトを作成します。

パラメータ:

- **pose** (*Pose*) -- 目標ポーズ。
- **linear_velocity** (*float or None*) -- 指定する直線速度 (単位: [mm/s])。直線速度を指定しない場合、グローバル直線速度が使用されます。
- **linear_acceleration** (*float or None*) -- 指定する直線加速度 (単位: [mm/s²])。直線加速度を指定しない場合、グローバル直線加速度が使用されます。
- **blend_radius** (*float or None*) -- セグメント開始時のブレンド半径 (単位: [mm])。ブレンド半径を指定しない場合、0.0が使用されます。

コンストラクター:

すべてのパラメーターを指定して、PoseSegmentオブジェクトを作成します。

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

デフォルト値とデータベース内のポーズを使用します。

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

property pose

位置、方向、姿勢設定の値を含むPoseオブジェクトとして表現されるセグメントの目標ポーズ。

戻り値:

セグメントの目標ポーズ。

戻り値の型:

Pose

property blend_radius

セグメント開始時のブレンド半径。

戻り値:

開始時のブレンド半径。

戻り値の型:

float

property linear_acceleration

セグメントに指定する直線加速度。指定しない場合、グローバル直線加速度が使用されます。

戻り値:

指定する直線加速度。

戻り値の型:

float or None

property linear_velocity

セグメントに指定する直線速度。指定しない場合、グローバル直線速度が使用されます。

戻り値:

指定する直線速度。

戻り値の型:

float or None

class Path

Pathオブジェクトはロボットが2つ以上のポーズの間をどのように動くかを定義します。連続パスは、ツール空間でのみ定義され、直線や円弧など複数のセグメントで定義されます。離散化したパスで、ロボットが姿勢設定変更なしで移動できる場合、パスはシームレスと呼ばれます。パスに沿って加速度が連続している場合、パスは連続的と呼ばれます。パスは座標ではなく、指定したポーズから開始します。パスの姿勢設定は初期ポーズで決まるため、パスのセグメントは通常座標で定義されます。

`__init__(path)`

名前またはIDで保存されたパスを読み込んで、Pathオブジェクトを作成します。

パラメータ:

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

読み込むパスは、以下のいずれかの方法で参照します。

- データベースに保存されたパスの名前 (*str*)
- データベースに保存されたパスのID (*int*)
- Pathオブジェクト (ほかのPathオブジェクトを複製)

コンストラクター:

データベースからパスを読み込みます。

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

`create_path` 関数を使用して、指定した初期ポーズと定義済みパスセグメントを含むPathオブジェクトを作成します。segmentsはあとから追加することもできます。

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

property initial_pose

パスの初期ロボットポーズ。

戻り値:

初期ロボットポーズ。

戻り値の型:

[Pose](#)

property segments

パスに含まれるセグメント。すべてのセグメントが同一の型である必要はありません。

戻り値:

含まれるセグメント。

戻り値の型:

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

add_segment(segment)

パスの末尾にセグメントを追加します。

パラメータ:

segment ([LinearSegment](#), [CircularSegment](#), [OrientationSegment](#), [CoordinatesSegment](#),
[PoseSegment](#)) -- 追加するセグメント。

使用例:

セグメントは以下のように追加します。

```
# 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)
```

パスの末尾に直線セグメントを追加します。

パラメータ:

- **coordinates** (*list, dict or [Coordinates](#)*) -- パスの目標座標。
- **linear_velocity** (*float*) -- ツール直線速度の最大値 (単位: [mm/s])。
- **linear_acceleration** (*float*) -- ツール直線加速度の最大値 (単位: [mm/s²])。
- **blend_radius** (*float*) -- セグメント開始時の許容ブレンド半径 (単位: [mm])。

使用例:

直線セグメントは以下のように追加します。

```
# 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]))
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
# 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)
```

最初のセグメントは、指定した目標座標へ最大速度500 mm/s、最大加速度1000 mm/s²で移動する直線動作が定義されています。後続のセグメントでブレンド半径を100 mmに設定して、座標からの逸脱を許容してスムーズで効率的に遷移するよう指定しているため、指定した目標座標には正確には到達しません。2つ目のセグメントは、次の座標へ最大速度250 mm/s、最大加速度500 mm/s²で移動する直線動作を追加しています。

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

パスの末尾に円弧セグメントを追加します。

パラメータ:

- **intermediate_position** (*list*) -- 目標座標に向かうパス上で経由する中間位置 [x, y, z]。中間位置に応じて、円弧動作の曲率が決まります。
- **coordinates** (*list, dict or Coordinates*) -- パスの目標座標。
- **linear_velocity** (*float*) -- ツール直線速度の最大値 (単位: [mm/s])。
- **linear_acceleration** (*float*) -- ツール直線加速度の最大値 (単位: [mm/s²])。
- **blend_radius** (*float*) -- セグメント開始時の許容ブレンド半径 (単位: [mm])。

使用例:

円弧補間セグメントは以下のように追加します。

```
# 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]))
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
# 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)
```

最初のセグメントは、中間位置を経由して指定した目標座標へ最大速度500 mm/s、最大加速度1000 mm/s²で移動する円弧動作が定義されています。後続のセグメントでブレンド半径を100 mmに設定して、座標からの逸脱を許容してスムーズで効率的に遷移するよう指定しているため、指定した目標座標には正確には到達しません。2つ目のセグメントは、次の中間位置を経由して次の座標へ最大速度250 mm/s、最大加速度500 mm/s²で移動する円弧動作を追加しています。

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

パスの末尾に方向セグメントを追加します。

パラメータ:

- **coordinates** (*list, dict or [Coordinates](#)*) -- パスの目標座標。
- **angular_velocity** (*float*) -- ツール角速度の最大値 (単位: [deg/s])。
- **angular_acceleration** (*float*) -- ツール角加速度の最大値 (単位: [deg/s²])。
- **blend_radius** (*float*) -- セグメント開始時の許容ブレンド半径 (単位: [mm])。ブレンド半径を0より大きくすると、セグメント目標位置に到達する前に回転し始めます。

使用例:

方向セグメントは以下のように追加します。

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

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
# 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)
```

最初のセグメントは、指定した目標方向へ最大速度250 deg/s、最大加速度500 deg/s²で回転する動作が定義されています。このパスを実行すると、後続セグメントのブレンド半径が0より大きいいため、最初の回転はスキップされ、2番目のセグメントが直接実行されます。詳細は `OrientationSegment.__init__` を参照してください。このセグメントは、指定した目標方向 [150.0, 0.0, -90.0] へ最大速度500 deg/s、最大加速度1000 deg/s²で回転する動作が定義されています。2つ目のセグメントのブレンド半径を0.0に設定した場合、両方の回転動作が順に実行されます。

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

パスの末尾に座標セグメントを追加します。

パラメータ:

- **coordinates** (*list, dict or [Coordinates](#)*) -- パスの目標座標。
- **linear_velocity** (*float*) -- ツール直線速度の最大値 (単位: [mm/s])。
- **linear_acceleration** (*float*) -- ツール直線加速度の最大値 (単位: [mm/s²])。
- **blend_radius** (*float*) -- セグメント開始時の許容ブレンド半径 (単位: [mm])。

使用例:

座標セグメントは以下のように追加します。

```
# 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]))
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
# 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)
```

最初のセグメントは、指定した目標座標へ最大速度500 mm/s、最大加速度1000 mm/s²で移動する動作が定義されています。後続のセグメントでブレンド半径を100 mmに設定して、座標からの逸脱を許容してスムーズで効率的に遷移するよう指定しているため、指定した目標座標には正確には到達しません。2つ目のセグメントは、次の座標へ最大速度250 mm/s、最大加速度500 mm/s²で移動する動作を追加しています。

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

パスの末尾にポーズセグメントを追加します。

パラメータ:

- **pose** (*Pose*) -- パスの目標座標。
- **linear_velocity** (*float*) -- ツール直線速度の最大値 (単位: [mm/s])。
- **linear_acceleration** (*float*) -- ツール直線加速度の最大値 (単位: [mm/s²])。
- **blend_radius** (*float*) -- セグメント開始時の許容ブレンド半径 (単位: [mm])。

使用例:

ポーズセグメントは以下のように追加します。

```
# 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)
```

上記の例では、オプションパラメーターは指定されていません。そのため、グローバルパラメーターが使用されます。パラメーターは関数呼び出しで明示的に定義することもできます。

```
# 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)
```

最初のセグメントは、指定した目標ポーズへ最大速度500 mm/s、最大加速度1000 mm/s²で移動する動作が定義されています。後続のセグメントでブレンド半径を100 mmに設定して、対応する座標からの逸脱を許容してスムーズで効率的に遷移するよう指定しているため、指定した目標ポーズには正確には到達しません。2つ目のセグメントは、次のポーズへ最大速度250 mm/s、最大加速度500 mm/s²で移動する動作を追加しています。

***class* Socket**

Socketは、`create_tcp_client`、`create_tcp_server`、`create_udp_socket` 関数で返されるオブジェクトです。このオブジェクトを使用して、ソケット経由でデータを送受信します。

`receive(timeout=None)`

ソケット経由でメッセージを受信します。

パラメータ:

timeout (*None or float*) -- メッセージを待機する最大時間。指定しない場合は、無期限に待機します。

戻り値:

受信したメッセージ、またはタイムアウトが発生したときは空文字列。

戻り値の型:

str

使用例:

3秒間応答を待機し、受信時に出力します。

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

send(*message*)

ソケット経由でメッセージを送信します。

パラメータ:

message (*str*) -- 送信するメッセージ。

使用例:

接続しているすべてのピアにメッセージを送信します。

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

close()

ソケット接続を閉じます。

使用例:

ソケットを閉じます。

```
socket.close()
```

アドオン関数

Cognexアドオン関数

class Cognex

CognexモジュールはCognexカメラの使用をサポートします。Cognexカメラの設定には [Cognex In-Sight Explorer](#) のインストールが必要です。

Cognexカメラをロボットに正しく設定して接続する方法については、ユーザーマニュアルを参照してください。

このアドオンのすべての関数は、プレフィックス `cognex.` を使用して呼び出します。

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

ネイティブコマンド用にISEポート経由で接続されたCognexカメラにTCPクライアントを設定します。

パラメータ:

- **ip** (*str*) -- カメラのIPアドレス (例: `'10.0.0.210'`)。
- **user** (*str*) -- カメラにログインするためのユーザー名 (事前に [Cognex In-Sight Explorer](#) で設定します)。デフォルトのユーザー名は `'admin'` です。
- **password** (*str*) -- カメラにログインするためのパスワード (事前に [Cognex In-Sight Explorer](#) で設定します)。デフォルトのパスワードは空文字列 (設定なし) です。

例外:

MinorError -- 認証情報が間違っている場合、接続中に中断される場合、接続に時間かかる場合。

使用例:

デフォルトのIPと認証情報で新品のカメラに接続します。

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

IPアドレスと認証情報を変更済みのカメラに接続します。

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

trigger_camera(ip)

カメラのトリガーイベント ("SW8")を設定し、新しい画像を撮影して、アクティブなCognexジョブで処理されたデータを返します。

パラメータ:

ip (*str*) -- カメラのIPアドレス (例: `'10.0.0.210'`)。

戻り値:

検出されたエンティティーを含むカメラからのレスポンスメッセージ。

戻り値の型:

str

change_job(ip, job_name)

ネイティブカメラコマンド"LF"を使用してCognexジョブを変更します。このコマンドはカメラに保存されているジョブファイルを読み込みます。このコマンドの実行には数秒かかる場合があります。

パラメータ:

- **ip** (str) -- カメラのIPアドレス (例: '10.0.0.210')。
- **job_name** (str) -- 切り替えるジョブ名 (事前に [Cognex In-Sight Explorer](#) で設定します)。

例外:

MinorError -- カメラが存在しない場合、またはジョブの変更に失敗した場合。

使用例:

'CircleDetection.job' として保存されたジョブに切り替え、現在のカメラビューで円を検出するために画像を撮影します。

```
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)

カメラでイベントをトリガーし、Cognex側で特定のキーワードソケットメッセージをリクエストします。

パラメータ:

- **ip** (*str*) -- カメラのIPアドレス (例: `'10.0.0.210'`)。
- **event_id** (*int*) -- トリガーするCognexイベントのID。
- **keyword** (*str*) -- 待機するキーワード。
- **timeout** (*float or None*) -- 待機リクエストのタイムアウト (単位: 秒)。無期限に待機する場合は `None` を使用します。

例外:

TimeoutError -- タイムアウトまでにカメラが有効な文字列を返さなかった場合。

使用例:

Cognex In-Sightジョブのイベント1 (SW1)に関連している、文字列 'my_message' を含むメッセージを要求します。

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

disconnect_camera(*ip*)

特定のカメラを切断します。

パラメータ:

ip (*str*) -- カメラのIPアドレス (例: `'10.0.0.210'`)。

Modbusアドオン関数

class Modbus

Modbusモジュールには、外部RESTサーバーにリクエストを送信する関数があります。

このアドオンのすべての関数は、プレフィックス `modbus.` を使用して呼び出します。

write_user_register(address, values)

Modbus出力レジスターに書き込みます。

パラメータ:

- **address** (*int*) -- 書き込みを開始するレジスターのアドレス。設定可能な値は 0x10 (16) から 0xFF (255) です。
- **values** (*int or list of int*) -- 書き込む値。各レジスタには最大2バイトのデータを格納できます。格納できる最大値は65535 ($=2^{16}-1$) です。複数の値を指定した場合、値は連続するレジスターに書き込まれます。

使用例:

Modbus出力レジスターの特定アドレスに単一の値を書き込みます。

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

1つのコマンドで複数の値を連続する出力レジスターに書き込みます。

```
# 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)

Modbus入力レジスターから読み取ります。

パラメータ:

- **address** (*int*) -- 読み取りを開始するレジスターのアドレス。設定可能な値は 0x10 (16)から0xFF (255)です。
- **count** (*int*) -- 読み取る連続レジスターの数。デフォルトでは1つのレジスターのみを読み取ります。

戻り値:

レジスターから読み取った値。このリストの長さは *count* 引数と同じです。

戻り値の型:

list of int

使用例:

Modbus入力レジスターの特定アドレスから単一の値を読み取ります。

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

1つのコマンドで複数の値を連続する入力レジスターから読み取ります。

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

wait_for_event(*index*)

指定したインデックスのイベントが設定されるまで待機します。イベントはModbusクライアントによって外部から設定されます。

パラメータ:

index (*int*) -- 待機するイベントのインデックス。設定可能なインデックスは0から15です。

使用例:

イベントが設定されるまで待機します。

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

RESTアドオン関数

class Rest

RESTモジュールには、外部RESTサーバーにリクエストを送信する関数があります。

このアドオンのすべての関数は、プレフィックス `rest.` を使用して呼び出します。

`setup_client_connection(target_url)`

RESTサーバーにリクエストを送信するためのクライアント接続を設定します。

パラメータ:

target_url (*str*) -- 接続するRESTサーバーのURL (一般的なAPIエンドポイント)

戻り値:

RESTクライアント接続オブジェクト。

戻り値の型:

[RestClientConnection](#)

使用例:

RESTサーバーへの接続を確立します。

```
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は、`rest.setup_client_connection` で返されるオブジェクトです。接続したRESTサーバーにリクエストを送信するために使用します。

get_default_headers(*method*)

すべてのリクエストに付加されるデフォルトのヘッダーを読み取ります。

パラメータ:

method (*str*) -- ヘッダーを取得するメソッド ("get"または"post")。

戻り値:

リクエストメソッドに現在設定されているデフォルトのヘッダー。

戻り値の型:

dict

使用例:

GETメソッドとPOSTメソッドのデフォルトヘッダーを読み取ります。

```
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*)

すべてのリクエストに付加されるデフォルトヘッダーを変更します。

パラメータ:

- **method** (*str*) -- ヘッダーを設定するメソッド ("get"または"post")。
- **new_headers** (*dict*) -- 上書きする新しいデフォルトヘッダー。

使用例:

RESTサーバーへの接続を確立します。

```
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)
```

RESTサーバーにGETリクエストを送信します。

パラメータ:

- **resource** -- アクセスするリソース名。リソース名はURLに追加されます (<url>/<resource>)。先頭に '/' を含めることもできます。
- **headers_override** (*dict*) -- 個別リクエストのヘッダーを設定します。指定しない場合は、デフォルトのヘッダーを使用します。
- **query_params** (*dict*) -- クエリの指定 (ある場合)。
- **secure_connection** (*bool*) -- セキュア接続 (証明書の検証)を使用するかどうかを指定します。

戻り値:

レスポンスコード (HTTPコード、例: 正常時は 200)とレスポンスペイロード (レスポンスデータ、ある場合)。

戻り値の型:

(int, dict)

使用例:

RESTサーバーにGETリクエストを送信し、レスポンスコードを確認します。

```
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)
```

RESTサーバーにPOSTリクエストを送信します。

パラメータ:

- **resource** (*str*) -- アクセスするリソース名。リソース名はURLに追加されます (<url>/<resource>)。先頭に '/' を含めることもできます。
- **data** (*str or dict*) -- POSTリクエストの本文/データ。
- **headers_override** (*dict*) -- 個別リクエストのヘッダーを設定します。指定しない場合は、デフォルトのヘッダーを使用します。
- **query_params** (*dict*) -- クエリの指定 (ある場合)。
- **secure_connection** (*bool*) -- セキュア接続 (証明書の検証)を使用するかどうかを指定します。

戻り値:

レスポンスコード (HTTPコード、例: 正常時は 200) とレスポンスペイロード (レスポンスデータ、ある場合)。

戻り値の型:

(int, dict)

使用例:

RESTサーバーにPOSTリクエストを送信し、レスポンスコードを確認します。

```
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アドオン関数

class ROSHandler

ROSモジュールは、他のアドオンやユーザーが利用できるようにROS機能を追加します。

このアドオンのすべての関数は、プレフィックス `ros_handler.` を使用して呼び出します。

`ros_node()`

ROSモジュールは独自のROSノードを作成し、この関数を使ってアクセスできます。スクリプト内でpublisherやsubscriberなどのオブジェクトを作成するときに使用できます。また、`rclpy` をアプリケーションに直接インポートすることもできます。

ⓘ 警告

このノードでは `rclpy.spin()` を実行しないでください。すでにバックグラウンドで実行されています。

使用例:

ロボットがパブリッシュしているすべてのトピック名を表示します。

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

戻り値:

ROSモジュールのROSノード。

戻り値の型:

`rclpy.node.Node`