机器人编程文档

翻译版

© Seiko Epson Corporation 2026

Rev.2
SCM268S9129M

目录

Python 基础

Python 基本规范

本节介绍了一些可在应用程序编辑器中使用的 Python 基础编程知识。这只是对 Python（3.13）最重要规范的简要概述；当然，Python 还提供了更多内置函数。请注意，机器人应用并不仅仅是普通的 Python 应用，因此某些 Python 功能受到限制或无法使用。对于高级 Python 用户或编程教程，请参考官方的 Python 3.13 文档。

注释

以下代码行表示代码注释，在运行时不会被执行。

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

变量定义

Python 支持多种不同的变量类型，并且不区分变量定义和变量声明。

```
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"
```

变量转换

只要数值兼容，变量就可以很方便地从一种类型转换为另一种类型。

```
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
```

数据结构

Python 支持四种不同的数据结构类型：

- 列表（List）包含一个有序的元素序列。

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

- 元组（Tuple）包含一个有序但不可变（不可修改）的元素序列。

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

- 集合（Set）包含一个无序且不含重复元素的集合。

```
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
```

- 字典（Dictionary）包含键和值的无序组合。

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

库与导入

Python 提供了一组内置模块（标准库）。以下模块均受支持：

- 数值运算与随机数 – 用于计算、数学函数以及生成随机值

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

- 时间与日期处理 – 处理时间戳、延迟以及日期/时间值

```
import time, datetime
```

- 数据存储与文件格式 – 读写结构化数据，例如 JSON 或 CSV 格式

```
import json, csv
```

- 文本处理 – 字符串操作以及使用正则表达式进行模式匹配

```
import re, string
```

- 高级迭代与函数式工具 – 高效循环、组合以及高阶函数

```
import itertools, functools
```

除标准库外，还提供以下第三方库：

- 数值计算 – 高效处理数组与数值数据

```
import numpy
```

- 三维姿态数学 – 四元数表示与变换（机器人相关）

```
import pyquaternion
```

- HTTP 通信 – 向 Web 服务和 REST API 发送请求

```
import requests
```

- 配置与数据序列化 – 处理 YAML 格式数据

```
import yaml
```

以下标准 Python 模块在机器人应用中不可用：

- 操作系统访问（例如 `os` 、 `shutil` 、 `subprocess` 、 `sys` ）
- 自定义并行处理（例如 `multiprocessing` 、 `threading` ）
- 自定义检查与调试（例如 `inspect` 、 `traceback` ）

此外，在机器人 Python 环境中不支持安装自定义第三方库。

Python 基本函数

必需与可选的函数参数

假设定义了一个具有以下参数的函数：

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

函数参数 arg1 和 arg2 是标准参数，在调用函数时必须提供。参数 arg3 和 arg4 具有默认值，因此在调用函数时不一定需要提供。以下所有函数调用都是有效的：

```
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")
```

解包函数参数

代码文档中提到的一些函数使用了解包参数。当函数参数数量可能变化时，这种方式非常有用。让我们来看以下函数参数：

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

这些函数随后可以按如下方式使用：

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

Python 基本运算符

算术运算符

```
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
```

比较运算符

此处 x 和 y 可以是任意类型。

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

此处 x 和 y 为数值类型。

```
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
```

逻辑运算符

此处 x 和 y 为布尔值。

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

位运算符

此处 x 和 y 为整数。

```
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
```

条件编程

If–Else 条件

`if`、`elif`（else if）和 `else` 条件可以按照以下规则组合：

- 仅当 `if` 条件返回 True 时，`if` 块才会被执行。
- `elif`块是可选的，且仅在所有前面的 `if` 或 `elif` 条件返回 False 且自身条件返回 True 时执行。
- `else` 块是可选的，且仅在所有前面的 `if` 或 `elif` 条件都返回 False 时执行。

一般的 if-elif-else 结构如下所示：

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

示例：

检查变量 `my_integer` 是否等于 5、10，或两者都不是。

```
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 循环

While 循环按以下规则执行和重复：

- 只要条件返回 True，`while` 块就会被重复执行。
- `continue` 关键字是可选的，可用于跳到下一次循环迭代，忽略 while 循环中剩余的代码。
- `break` 关键字是可选的，可用于跳出当前循环。
- `else` 块是可选的（很少使用），仅在循环条件变为 False 时执行。换句话说，如果 while 循环通过 break 退出，该块不会被执行。

```
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
    ...
```

示例：

打印从 1 到 10 的所有数字。

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

无限循环：当你希望机器人不断重复任务直到手动停止，或者通过特定条件或函数跳出循环时，无限循环非常有用。如果循环运行速度过快，建议添加等待命令以给处理器一些缓冲时间。

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

For 循环

For 循环按以下规则执行和重复：

- `for` 块会被重复执行，每个可迭代序列中的元素执行一次（例如列表、集合、字典或字符串）。
- `continue` 关键字是可选的，可用于跳到下一次循环迭代，忽略 for 循环中剩余的代码。
- `break` 关键字是可选的，可用于跳出当前循环。
- `else` 块是可选的（很少使用），在 for 循环序列的最后一个元素处理完后执行。换句话说，如果 for 循环通过 break 退出，该块不会被执行。

```
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'.
    ...
```

示例：

打印从 1 到 5 的所有数字。

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

打印我最喜欢的水果。

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

统计 `python` 中的字母。

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

核心功能

class Core

备注

这是所有核心功能的集合。这些功能可以直接使用，并可在应用程序代码中调用。

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

将机器人的一个或多个关节移动到特定的关节角度。此动作会同步所有关节，使它们同时到达目标位置。

参数:

- **joints** (*int, str, list*) --
  要移动的执行器ID。指定如下：
  - 单个关节的ID（例如 *6* ）
  - 多个关节的ID列表（例如 *[1, 4, 6]* ）
  - 常量 *ALL_KIN_JOINTS* 移动全部六个关节
- **joint_position** (*float, list of float*) --
  移动关节的角度位置[°]。指定如下：
  - 单个数值将所有指定关节移动到相同位置（例如 *-30* ）
  - 数值列表将每个指定关节移动到不同位置（例如 *[90, -45, 30]* ）
- **joint_velocity** (*float or None*) -- 关节最大速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *45* %）。如未指定值，将使用全局关节速度。
- **joint_acceleration** (*float or None*) -- 移动关节最大加速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *60* %）。如未指定值，将使用全局关节加速度。
- **relative** (*bool*) -- 关节位置是否作为相对于当前关节位置的相对偏移。默认使用绝对位置值。
- **block** (*bool*) -- 是否等待此动作完成后再继续执行程序。注意，只有使用相同关节的同步动作可以连续以非阻塞方式执行。

示例:

将机器人的所有关节同步移动到0°，即机器人的站立直立位置：

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

将第6关节从当前位置移动-30°：

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

将第2、3和5关节分别移动到20°、-40°和20°。运动将以每个关节同时完成移动的方式进行。此运动将使用允许的最大关节加速度，但只使用最大允许关节速度的50%。

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

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

将机器人的TCP（工具中心点）移动到特定的笛卡尔坐标集合。机器人沿时间最优轨迹运动，在每个关节上产生梯形速度曲线。

参数:

- **coordinates** (*Coordinates, list or dict*) --
  TCP的目标坐标。按如下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典
- **configuration** (*str or None*) -- 目标位姿的配置。这是一个从 'c1' 到 'c8' 的字符串。
- **joint_velocity** (*float or None*) -- 关节最大速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *45* %）。如未指定值，将使用全局关节速度。
- **joint_acceleration** (*float or None*) -- 移动关节最大加速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *60* %）。如未指定值，将使用全局关节加速度。
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --

用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：

- 一个Coordinates对象
- 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
- 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

- **relative** (*bool*) -- 坐标是否被视为相对于当前位置的偏移量。默认情况下使用绝对坐标值。
- **block** (*bool*) -- 是否在此运动完成之前暂停程序的进一步执行。默认情况下，此函数是阻塞的。

示例:

目标坐标可以以多种形式给出。所有指定的命令都会将 TCP 移动到全局工作坐标系中的位置 x = 700mm，y = -125mm，z = 500mm，且方向角为 roll = 180°、pitch = 0° 和 yaw = -90°。

```
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]))
```

如果未指定所有坐标，则会使用当前 TCP 坐标作为其余坐标。例如，仅更改 x 和 roll 分量可以按如下方式进行：

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

在上述示例中，未指定任何可选参数。因此，使用全局参数。也可以在函数调用中显式定义这些参数：

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

这将使 TCP 在 y 方向额外移动 -100mm，yaw 方向旋转 -60°，速度为最大速度的 50%，并使用最大加速度。此外，TCP 通过工具偏移在 x 方向移动 20mm。

在工具空间中定义运动有多种选项，其中之一是：

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

沿工具的 z 方向移动。

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

将机器人的 TCP（工具中心点）移动到特定姿态。机器人沿时间最优轨迹移动，从而在每个关节产生梯形速度曲线。

参数:

- **pose** (*Pose, str, or int*) --
  TCP 的目标姿态。指定值如下：
  - 一个 Pose 对象
  - 预定义姿态的名称（已保存在数据库中）
  - 预定义姿态的 ID（已保存在数据库中）
- **joint_velocity** (*float or None*) -- 关节最大速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *45* %）。如未指定值，将使用全局关节速度。
- **joint_acceleration** (*float or None*) -- 移动关节最大加速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *60* %）。如未指定值，将使用全局关节加速度。
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表

    - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

- **block** (*bool*) -- 是否在此运动完成之前暂停程序的进一步执行。默认情况下，此函数是阻塞的。

示例:

从数据库中移动到一个姿态：

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

也可以在脚本中使用 `create_pose()` 函数指定自定义姿态：

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

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

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

这会将机器人移动到数据库中的"final_pose"，使用最大关节速度的45%和最大关节加速度的60%。此外，还会对工作坐标系应用偏移，将目标姿态在z方向上移动50mm。

在工具空间中定义运动有多种选项，其中之一是：

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

沿工具的 z 方向移动。

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*)

将机器人的TCP（工具中心点）从当前姿态直线移动到目标姿态。注意，位置和方向沿轨迹线性插值。只有当路径上的每个位姿都可达时，才能执行此运动。

参数:

- **coordinates** (*Coordinates, list, or dict*) --
  TCP的目标坐标。按如下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典
- **linear_velocity** (*float or None*) -- 机器人所有运动部件的最大线速度 [mm/s]。
- **linear_acceleration** (*float or None*) -- 机器人所有运动部件的最大线加速度 [mm/s^2]。
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表

- 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

- **relative** (*bool*) -- 坐标是否被视为相对于当前位置的偏移量。默认情况下使用绝对坐标值。
- **block** (*bool*) -- 是否在此运动完成之前暂停程序的进一步执行。默认情况下，此函数是阻塞的。

示例:

目标坐标可以有多种形式。所有指定的命令将TCP线性移动到全局工作坐标系中的位置 x = 600mm, y = -125mm, z = 500mm，姿态角 roll = 180°，pitch = 0°，yaw = -90°。

```
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]))
```

如果未指定所有坐标，则使用当前TCP坐标作为剩余坐标。例如，x 和 roll 分量可以如下更改：

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

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

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

这将在x方向额外移动100mm，速度为500mm/s，加速度为1000mm/s²。此外，工作坐标系将被给定坐标覆盖。

在工具空间中定义运动有多种选项，其中之一是：

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

沿工具的z方向线性移动。

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*)

将机器人的TCP（工具中心点）从当前姿态沿圆形轨迹移动到目标姿态，经过中间位置。注意方向沿轨迹线性插值。只有当路径上的每个位姿都可达时，才能执行此运动。

参数:

- **intermediate_position** (*list*) -- 目标坐标路径上的中间位置。中间位置决定圆周运动的曲率。
- **coordinates** (*Coordinates, list, or dict*) --
  TCP的目标坐标。按如下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典
- **linear_velocity** (*float or None*) -- 机器人所有运动部件的最大线速度 [mm/s]。
- **linear_acceleration** (*float*) -- 机器人所有运动部件的最大线加速度 [mm/s^2]。
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表

- 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

- **relative** (*bool*) -- 坐标是否被视为相对于当前位置的偏移量。默认情况下使用绝对坐标值。
- **block** (*bool*) -- 是否在此运动完成之前暂停程序的进一步执行。默认情况下，此函数是阻塞的。

示例:

机器人可以沿圆弧移动到目标位置 x = 0mm, y = -500mm, z = 500mm，姿态角 roll = 180°，pitch = 0°，yaw = -90°（全局工作坐标系）。在运动过程中，它将经过中间位置 x = 0mm, y = -500mm, z = 500mm。

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

如果未指定所有坐标，则剩余坐标使用当前TCP坐标。

```
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})
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
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)
```

这将TCP移动到指定坐标，速度为500mm/s，加速度为1000mm/s^2。此外，TCP沿x方向被工具偏移20mm。

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

移动TCP（工具中心点），仅改变方向。机器人按所需工具方向线性旋转，同时保持位置不变。只有当路径上的每个位姿都可达时，才能执行此运动。

参数:

- **tool_orientation** (*list*) -- 工具的期望方向，以航海角（roll, pitch, yaw）表示 [°]
- **angular_velocity** (*float or None*) -- TCP 的最大工具角速度 [°/秒]。
- **angular_acceleration** (*float or None*) -- TCP 的最大工具角加速度 [°/秒^2]。
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。
- **relative** (*bool*) -- 坐标是否被视为相对于当前位置的偏移量。默认情况下使用绝对坐标值。

- **block** (*bool*) -- 是否在此运动完成之前暂停程序的进一步执行。默认情况下，此函数是阻塞的。

示例:

目标方向可以指定所有角度或仅指定部分角度，其他角度为 'None'。在后一种情况下，缺失的角度使用当前方向。

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

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

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

对应于绕滚转方向相对移动 -60°，速度为 250°/秒，加速度为 3000°/秒^2。此外，工具坐标系被提供的坐标覆盖。

在工具空间中定义运动有多种选项，其中之一是：

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

绕工具的 z 方向旋转。

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*)

将 TCP（工具中心点）沿定义的路径点移动，从当前机器人位姿开始。

参数:

- **poses** (*list of str, int or Pose*) -- 路径点列表。
- **joint_velocity** (*float or None*) -- 关节最大速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *45* %）。如未指定值，将使用全局关节速度。
- **joint_acceleration** (*float or None*) -- 移动关节最大加速度的百分比。使用单个数值限制所有关节到相同的最大值（例如 *60* %）。如未指定值，将使用全局关节加速度。
- **blend_radius** (*float or None*) -- 允许的混合半径 [毫米]，定义在工具空间中路径点之间过渡时的最大偏差。
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

- **block** (*bool*) -- 是否等待此动作完成后再继续执行程序。注意，只有使用相同关节的同步动作可以连续以非阻塞方式执行。

示例:

路径点可以用不同形式提供（ `Pose` 对象，数据库中姿态的名称，数据库中姿态的 ID）：

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

在上述示例中，未指定任何可选参数。因此，使用全局参数。也可以在函数调用中显式定义这些参数：

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

沿路径点移动，允许最大偏差为 50 毫米。使用最大速度的 50% 和最大加速度。

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

根据定义的段移动 TCP（工具中心点）。从当前机器人位姿开始。

参数:

- **segments** (*list*) -- 路径由多个段组成的列表。
- **tool_offset** (*Coordinates*, *list*, *dict*, *or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates*, *list*, *dict*, *or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates*, *list*, *dict*, *or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates*, *list*, *dict*, *or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。
- **block** (*bool*) -- 是否等待此动作完成后再继续执行程序。注意，只有使用相同关节的同步动作可以连续以非阻塞方式执行。

示例:

段可以在脚本中使用 Segment 对象定义（ `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])
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
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)
```

使用工具偏移定义的 TCP 轻微移动来执行这些段。此外，运动设置为非阻塞。

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

根据指定路径移动 TCP（工具中心点）。首先，机器人使用默认速度和加速度移动到路径的初始位姿。然后根据规范依次执行各段。

参数:

- **path** (*Path, str, or int*) -- TCP 应该遵循的路径。
- **previous_angles** (*list or None*) --
  用于计算起始位姿最接近解的关节角度。请按如下方式指定该值：
  - None，用于使用当前机器人的关节角度
  - 关节角度列表 [度]
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

- **block** (*bool*) -- 是否等待此动作完成后再继续执行程序。注意，只有使用相同关节的同步动作可以连续以非阻塞方式执行。

示例:

从数据库运行路径：

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

也可以在脚本中使用 `create_path()` 函数指定自定义路径：

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

另一种选择是使用 `Path` 对象创建路径并运行它：

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

run_path(path)
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

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

执行 "fancy_path"，使用工具坐标系定义的 TCP 轻微偏移。此外，运动设置为非阻塞。

wait_for_motor(*actuator_ids=None*)

等待一个或多个关节电机停止运动。

参数:

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

要等待的执行器ID。按如下方式指定值：

- 单个关节的ID（例如 *6* ）
- 多个关节的ID列表（例如 *[1, 4, 6]* ）
- 多个关节的关节名称集合（例如：*{"1", "4", "6"}* ）
- 使用常量 *ALL_KIN_JOINTS* 或 *None* 从所有关节读取

示例:

此示例显示，机器人移动时的时间可以用于自定义用途。

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

stop_motion()

通过向所有连接的执行器发送停止信号来停止运动。

示例:

此示例显示可以使用 'stop_motion' 函数中止非阻塞运动。如果特定数字输入切换，我们希望中止该运动。

```
# 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*)

检查一个或多个电机是否在移动

参数:

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

要检查的执行器ID。按如下方式指定值：

- 单个关节的ID（例如 *6* ）
- 多个关节的ID列表（例如 *[1, 4, 6]* ）
- 多个关节的关节名称集合（例如：*{1, 4, 6}* ）
- 使用常量 *ALL_KIN_JOINTS* 或 *None* 从所有关节读取

返回:

指定的电机是否正在运动。可以是以下之一：

- 如果检查了单个电机，则为布尔值
- 如果检查了多个电机，则为布尔值列表

返回类型:

bool or list of bool

示例:

此示例显示可以使用 'stop_motion' 函数中止非阻塞运动。如果特定数字输入切换，我们希望中止该运动。

```
# 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*)

为点到点运动函数使用的 `joint_velocity` 参数设置新的全局默认值。关节速度值以最大关节速度限制的百分比给出。

参数:

joint_velocity (*float*) -- 全局关节速度，占最大关节速度的百分比。

示例:

使用相同速度限制连续执行两个点到点运动。

```
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*)

为用于点到点运动函数的 `joint_acceleration` 参数设置新的全局默认值。关节加速度值以最大关节加速度限制的百分比表示。

参数:

joint_acceleration (*float*) -- 全局关节加速度，占最大关节加速度的百分比 [%]。

示例:

使用相同加速度限制连续执行两个点到点运动。

```
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*)

为用于工具运动和路径规划函数的 `linear_velocity` 参数设置新的全局默认值。线速度值限制工具和机器人所有可动部件的最大线速度。

参数:

linear_velocity (*float*) -- 全局线速度 [mm/s]。

示例:

使用相同速度限制连续执行两个线性运动。

```
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*)

为用于工具运动和路径规划函数的 `linear_acceleration` 参数设置新的全局默认值。线加速度值限制工具和机器人所有可动部件的最大线加速度。

参数:

linear_acceleration (*float*) -- 全局线加速度 [mm/s^2]。

示例:

使用相同加速度限制连续执行两个线性运动。

```
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*)

为用于旋转运动函数的 `angular_velocity` 参数设置新的全局默认值。角速度值限制工具的旋转速度。

参数:

angular_velocity (*float*) -- 全局角速度 [°/s]。

示例:

使用相同速度限制连续执行两个旋转运动。

```
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*)

为用于旋转运动函数的 `angular_acceleration` 参数设置新的全局默认值。角加速度值限制工具的旋转加速度。

参数:

angular_acceleration (*float*) -- 全局角加速度 [°/s^2]。

示例:

使用相同加速度限制连续执行两个旋转运动。

```
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*)

为在运动和路径函数中使用的 `blend_radius` 参数设置新的全局默认值。

参数:

blend_radius (*float*) -- 全局混合半径 [mm]。

示例:

使用相同的混合半径连续执行两个路径点运动。

```
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*)

为在运动和运动学函数中使用的 `work_frame` 参数设置新的全局默认值。工作坐标系是用于改变机器人参考坐标系的变换，相对于基座坐标系给出。

参数:

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

全局工作坐标系，相对于基坐标系给出。指定值如下：

- 一个Coordinates对象
- 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
- 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

示例:

使用相同工作坐标系执行两次连续运动。

```
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*)

设置 `tool_frame` 的新的全局默认值，用于运动和运动学函数。工具坐标系是相对于法兰坐标系给出的 TCP 变换。

参数:

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

全局工具坐标系，相对于法兰坐标系给出。指定值如下：

- 一个Coordinates对象
- 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
- 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

示例:

使用相同工具坐标系执行两次连续运动。

```
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*)

读取机器人当前 TCP（工具中心点）姿态。如果未提供坐标系，则相对于全局坐标系读取姿态。

参数:

- **tool_offset** (*Coordinates*, *list*, *dict*, *or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。

- **tool_frame** (*Coordinates*, *list*, *dict*, *or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。

- **work_offset** (*Coordinates*, *list*, *dict*, *or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。

- **work_frame** (*Coordinates*, *list*, *dict*, *or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

机器人当前 TCP 姿态。

返回类型:

Pose

示例:

以两种不同方式读取姿态的属性。

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

注意，当前姿态是从传感器数据读取的。传感器可能波动且不总是返回精确值，因此以下示例中相对于当前姿态来回移动会导致随时间产生意外偏移。在这种情况下，最好使用 *get_reference_pose* 函数。

```
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*)

读取机器人当前 TCP（工具中心点）坐标，相对于给定坐标系。如果未指定坐标系，则返回相对于全局坐标系的坐标。

参数:

- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。

- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。

- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。

- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

机器人当前 TCP 坐标。

返回类型:

Coordinates

示例:

读取当前 TCP 在全局坐标系下的坐标。

```
coordinates = get_current_coordinates()
```

请注意，当前坐标是从传感器数据中读取的。传感器可能会波动，且不一定始终返回精确的数值，因此下面相对于当前位姿来回移动的示例会随着时间产生意外的漂移。在这种情况下，最好改用 *get_reference_coordinates* 函数。

```
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()

读取当前机器人位姿配置的名称。请注意，该配置与当前全局坐标系无关。

返回:

当前配置（为 'c1' 到 'c8' 之一）。

返回类型:

str

示例:

以两种不同方式读取当前机器人位姿的配置。

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

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

获取机器人的参考 TCP（工具中心点）位姿。如果未指定坐标系，则返回相对于全局坐标系的位姿。请注意，该位姿是根据发送到驱动器的参考角度计算得到的。

参数:

- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。

- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。

- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。

- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

机器人的参考 TCP 位姿。

返回类型:

Pose

示例:

以两种不同方式读取参考位姿的属性。

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

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

获取机器人 TCP（工具中心点）相对于给定坐标系的参考坐标。如果未指定坐标系，则返回相对于全局坐标系的坐标。请注意，这些坐标是根据发送到驱动器的参考角度计算得到的。

参数:

- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。

- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。

- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。

- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

机器人的参考 TCP 坐标。

返回类型:

Coordinates

示例:

在全局坐标系中读取参考 TCP 坐标。

```
coordinates = get_reference_coordinates()
```

get_reference_configuration()

获取参考机器人位姿配置的名称。请注意，该配置与参考全局坐标系无关，并且是根据发送到驱动器的参考角度计算得到的。

返回:

参考配置（为 'c1' 到 'c8' 之一）。

返回类型:

str

示例:

以两种不同方式读取参考机器人位姿的配置。

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

create_pose(*coordinates*, *configuration*)

根据给定的坐标和配置创建一个 Pose 对象。

参数:

- **coordinates** (*Coordinates, list, or dict*) --
  位姿的 TCP 坐标。请按以下方式指定该值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典
- **configuration** (*str*) -- 位姿的配置。为 'c1' 到 'c8' 之一。

返回:

机器人位姿对象。

返回类型:

Pose

示例:

在指定配置下，根据一组坐标值创建位姿。

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

get_pose(*pose*)

从数据库中保存的预定义位姿读取位姿对象。

参数:

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

用于读取坐标的机器人位姿。可以是以下之一：

- 一个 Pose 对象
- 数据库中保存的位姿名称（str）。
- 数据库中已保存姿态的 ID（int）

返回:

姿态对象。

返回类型:

Pose

示例:

从数据库中加载名为"grasping_pose"的姿态。

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

create_path(*initial_pose*, *segments*)

基于给定的初始姿态和分段创建一个 Path 对象。

参数:

- **initial_pose** (*Pose*) -- 机器人的初始姿态。
- **segments** (*list*) -- 路径所包含的分段列表。请注意，也可以在之后添加分段。

返回:

机器人路径对象。

返回类型:

Path

示例:

从一个初始姿态和多个分段创建路径。

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

get_path(*path*)

从数据库中读取已保存的预定义路径对象。

参数:

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

要获取的机器人路径。可以是以下之一：

- 一个 Path 对象
- 数据库中已保存路径的名称（str）
- 数据库中已保存路径的 ID（int）

返回:

路径对象。

返回类型:

Path

示例:

从数据库中加载名为"grasping_path"的路径。

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

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

创建一个 LinearSegment 对象。

参数:

- **coordinates** (*list, dict, or Coordinates*) -- 目标坐标。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s^2]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 LinearSegment 对象：

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

或使用默认值：

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

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

创建一个 CircularSegment 对象。

参数:

- **intermediate_position** (*list*) -- 通向目标坐标路径上的中间位置 [x, y, z]。中间位置决定圆周运动的曲率。
- **coordinates** (*list, dict or Coordinates*) -- 目标坐标。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s^2]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 CircularSegment 对象：

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

或使用默认值：

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

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

创建 OrientationSegment 对象。

参数:

- **orientation** (*list*) -- 目标姿态。注意位置保持不变。
- **angular_velocity** (*float or None*) -- 期望的角速度 [°/s]。如果未定义角速度，将使用全局角速度值。
- **angular_acceleration** (*float or None*) -- 期望的角加速度 [°/s²]。如果未定义角加速度，将使用全局角加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。混合半径大于 0 时，TCP（工具中心点）可以在到达该段位置前旋转。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 OrientationSegment 对象：

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

或使用默认值：

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

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

创建 CoordinatesSegment 对象。

参数:

- **coordinates** (*list, dict or Coordinates*) -- 目标坐标。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s^2]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 CoordinatesSegment 对象：

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

或使用默认值：

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

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

创建 PoseSegment 对象。

参数:

- **pose** (*Pose*) -- 目标姿态。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s^2]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 PoseSegment 对象：

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

或使用数据库中的默认值和姿态：

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

get_coordinates(*pose*)

读取指定姿态的坐标。请注意，这些坐标与当前全局坐标系无关。

参数:

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

用于读取坐标的机器人位姿。可以是以下之一：

- 一个 Pose 对象
- 数据库中保存的位姿名称（str）。
- 数据库中已保存姿态的 ID（int）

返回:

该姿态的坐标。

返回类型:

Coordinates

示例:

读取数据库中先前保存的、名称为"grasping_pose"的姿态坐标。

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

读取数据库中先前保存的、ID 为 3 的姿态坐标。

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

get_configuration(*pose*)

读取指定姿态的构型。

参数:

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

要读取构型的姿态。可按以下方式指定：

- 一个 Pose 对象
- 预定义姿态的名称（已保存在数据库中）
- 预定义姿态的 ID（已保存在数据库中）

返回:

该姿态的构型（取值为'c1'到'c8'之一）。

返回类型:

str

示例:

读取数据库中名称为"grasping_pose"的姿态构型。

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

通过两种不同方式从 Pose 对象中读取构型。

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

parse_coordinates(*coordinates*)

创建 Coordinates 对象。

参数:

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

一组特定的坐标，可以是以下之一：

- 列表形式 [x, y, z, roll, pitch, yaw]，位置值单位为 [mm]，方向值单位为 [°]
- 字典形式 {'x': x, 'y': y, 'z': z, 'roll': roll, 'pitch': pitch, 'yaw': yaw}，位置值单位为 [mm]，方向值单位为 [°]
- 一个 Coordinates 对象（复制另一个 Coordinates 对象）

构造函数：

根据坐标值创建一个 Coordinates 对象。

```
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*)

通过相对工具坐标系和工作坐标系之间的差异在不同坐标系之间变换机器人坐标。如果既未指定相对工具坐标系也未指定相对工作坐标系，则不会进行坐标变换，并返回未修改的坐标对象。

参数:

- **coordinates** (*Coordinates, list, or dict*) --
  要转换的TCP坐标。请按如下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典
- **relative_tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **relative_work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

转换后的机器人坐标。

返回类型:

Coordinates

示例:

将一组坐标转换到另一个工作坐标系。如果工作坐标系的原点正好在要转换的坐标位置，转换将返回一个所有值均为零的Coordinates对象。

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

get_grid_points(*grid_name*)

读取数据库中预先保存的特定网格的所有网格点位置。

参数:

grid_name (*str*) -- 要提取点的网格名称。

返回:

网格点列表，按照网格属性中定义的迭代顺序。

返回类型:

list

示例:

移动到预定义网格中的所有点。

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

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

正向运动学 是机器人运动学中的一个概念，它根据给定的一组关节角计算TCP（工具中心点）的位姿。该函数的反函数是 *逆向运动学* 函数，它根据机器人的TCP位姿计算关节角。运动学计算通常在机器人内置的基座和法兰坐标系之间进行。不过，用户可以通过定义工作坐标系和工具坐标系来修改参考坐标系。默认情况下，使用全局定义的工作坐标系和工具坐标系。

参数:

- **joint_angles** (*list of float*) -- 六个关节角的列表 [度]
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

机器人位姿对象。

返回类型:

Pose

示例:

在奇异点中，多组关节角会导致相同的TCP位姿。在此示例中，两组关节角得到完全相同的位姿。

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

使用两种不同方法读取当前TCP位姿。

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

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

逆向运动学 是机器人运动学的一个概念，它根据TCP（工具中心点）的位姿计算一组关节角。该函数的反函数是 *正向运动学* 函数，它根据机器人的关节角计算TCP位姿。请注意，*逆向运动学* 通常有多个解，但只返回其中一个解。

参数:

- **pose** (*Pose, str, or int*) --
  要计算对应关节角的位姿。请按如下方式指定值：
  - 一个 Pose 对象
  - 预定义姿态的名称（已保存在数据库中）
  - 预定义姿态的 ID（已保存在数据库中）
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

六个关节角的列表 [度]

返回类型:

list of float

示例:

计算数据库中先前保存的位姿的关节角。

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

计算在特定配置下达到指定坐标集的关节角。

```
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*)

该函数计算最接近当前机器人位姿或特定位姿的 *逆向运动学* 解（详见函数 inverse_kinematics ）。

参数:

- **coordinates** (*Coordinates, list, or dict*) --
  要计算对应关节角的TCP坐标。请按如下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典
- **previous_angles** (*list or None*) --
  将计算最接近解的关节角度。请按如下方式指定该值：
  - None，用于使用当前机器人的关节角度
  - 关节角度列表 [度]
- **tool_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。
- **tool_frame** (*Coordinates, list, dict, or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。
- **work_offset** (*Coordinates, list, dict, or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。
- **work_frame** (*Coordinates, list, dict, or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表

- 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

六个关节角的列表 [度]

返回类型:

list of float

示例:

逆向运动学可用于简单检查特定坐标集是否返回运动学解。

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

为了获得同一姿态的多个解，可以相应地更改前一姿态。在此示例中，计算了一个直立机器人略微向前和向下移动的两个解。

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

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

该函数计算最接近当前机器人位姿或特定位姿的 *逆向运动学* 解（详见函数 inverse_kinematics ）。

参数:

- **coordinates** (*Coordinates*, *list*, *or dict*) --
  要计算对应关节角的TCP坐标。请按如下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

- **tool_offset** (*Coordinates*, *list*, *dict*, *or None*) --
  用于移动机器人工具中心点 (TCP) 的变换，相对于工具坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下没有工具偏移，因此 TCP 位于工具坐标系的原点。

- **tool_frame** (*Coordinates*, *list*, *dict*, *or None*) --
  用于改变机器人TCP的变换，相对于法兰框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，应用全局工具框架。如果工具框架仅包含零坐标，则它与法兰框架重合。

- **work_offset** (*Coordinates*, *list*, *dict*, *or None*) --
  用于移动机器人参考的变换，相对于工作框架给出。请按以下方式指定值：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下，没有工作偏移，使参考位于工作框架的原点。

- **work_frame** (*Coordinates*, *list*, *dict*, *or None*) --
  用于修改机器人的参考坐标的变换，相对于基坐标系给出。指定值如下：
  - 一个Coordinates对象
  - 以 [x, y, z, roll, pitch, yaw] 格式的笛卡尔坐标列表
  - 以 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw} 格式的笛卡尔坐标字典

  默认情况下应用全局工作坐标系。如果工作坐标系仅包含零值，则与基坐标系重合。

返回:

六个关节角的列表 [度] 及其配置，格式为 {'c1': [...], 'c2': [...], ...}

返回类型:

dict

示例:

在此示例中，机器人移动到所有能够达到相同 TCP 坐标的关节角集合。

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

如果我们知道一组能够达到期望姿态的关节角，则可以计算出达到相同姿态但关节配置不同的所有其他解。

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

wait(*seconds*)

等待，因此在定义时间内主动阻塞代码的进一步执行。

参数:

seconds (*float*) -- 等待时间，单位 [秒]。

示例:

此示例将机器人移动到预定义姿态，并在动作之间等待 3 秒。

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

此示例会无限重复您的部分代码。为确保执行速度受限，每次迭代结束时代码等待 100 毫秒。

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

此示例与上例类似，但等待时间不是固定的，而是动态选择，以便每秒以恒定间隔执行您的代码。

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

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

在主应用程序内运行一个（阻塞）应用程序。

参数:

- **script_name** (*str*) -- 要运行的应用程序名称。如果应用程序在文件夹中，请使用应用程序的完整路径（例如 *folder/subfolder/app* ）。
- **arguments** (*dict*) -- 传递给运行应用程序的参数。您可以在被调用的应用程序中使用内置变量 *script_arguments* 读取它们。

示例:

假设我们有一个单独的应用程序用于测量环境温度。此示例每分钟重复调用该应用程序一次。

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

start_parallel_script(*script_name*)

在主应用程序中并行运行一个（非阻塞）应用程序。请注意，主应用程序会立即继续，所有并行应用程序在主应用程序终止时都会中止。

参数:

script_name (*str*) -- 要启动的并行应用程序名称。如果应用程序在文件夹中，请使用应用程序的完整路径（例如 *folder/subfolder/app* ）。

示例:

此示例触发一个并行应用程序，用于监控某些温度传感器，以确保机器人仅在特定温度范围内运行。主应用程序会持续运行，如果操作温度条件不再满足，则中止。

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

stop_application()

停止当前正在运行的应用程序。这包括主应用程序以及由主应用程序触发的任何应用程序。请注意，正在运行的 *background* 应用程序将继续运行。

抛出:

ScriptInterrupt -- 如果从使用 *run_script* 函数触发的应用程序中运行此函数，则会在主应用程序中触发。

示例:

此代码示例可用于 *background* 应用程序，以暂时暂停并手动恢复正在运行的主应用程序。

```
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()

暂停当前运行的应用程序。这包括主应用程序以及由主应用程序触发的任何应用程序。请注意，正在运行的 *background* 应用程序会继续运行。

示例:

请查看 *stop_application* 的组合示例，其中使用了此函数。

resume_application()

恢复当前暂停中的应用程序。由于此函数只能在 *paused* 状态下运行，因此只能从 *background* 应用程序中使用。

示例:

请查看 *stop_application* 的组合示例，其中使用了此函数。

stop_and_release()

停止当前运行的应用程序，并释放所有关节（启用重力补偿模式）。请注意，用户需要手动固定所有关节（禁用重力补偿模式）才能继续运行后续应用程序。

is_status(*status*)

检查程序状态是否与给定状态匹配。

参数:

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

要检查的程序状态可以是以下之一或多个：

- 'ready'（机器人空闲，等待接收命令）
- 'processing'（机器人正在处理数据，可能需要一些时间）
- 'hand_guidance_control'（机器人处于手动引导控制模式）
- 'running'（机器人正在运行应用程序或执行运动）
- 'paused'（机器人正在暂停应用程序或运动）
- 'emergency_stop'（急停已触发）
- 'normal_stop'（正常停止已触发）
- 'protective_stop'（保护停止已触发）
- 'error'（发生错误或安全违规）
- 'position_limit'（发生了安全位置限制违规）
- 'safeguard'（触发了安全保护中断）
- 'collision'（触发了碰撞中断）
- 'proceed'（机器人正在从中断中继续）
- 'stopping'（机器人正在停止程序）

返回:

程序状态是否与给定状态匹配（或为给定状态之一）

返回类型:

bool

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

创建带按钮的对话框。此对话框会阻塞程序流程，直到用户选择其中一个选项。

参数:

- **text** (*str*) -- 呈现给用户的消息（或问题）。
- **options** (*list of str*) -- 可供选择的选项，即每个按钮上的标签。
- **title** (*str*) -- 对话框弹出窗口的标题
- **dialog_type** (*int*) --
  向用户显示的对话框类型。可以是以下之一：
  - 0：信息 / 蓝色
  - 1：成功 / 绿色
  - 2：警告 / 橙色
  - 3：错误 / 红色
- **dialog_value** (*str*) -- 如果对话框未被回答，则返回的默认值。
- **mandatory** (*bool*) -- 对话框是否必须回答才能继续程序流程。如果设置为 *True* ，则对话框不能被忽略，例如关闭弹出窗口，并会阻塞直到收到答案。

返回:

所选选项，即用户按下的按钮标签。

返回类型:

str

示例:

此示例展示了如何创建一个包含"继续"按钮的简单信息弹窗。按下按钮后，程序将继续执行其代码。

```
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*)

创建一个带有下拉列表选项的对话框。程序流程将阻塞，直到用户选择并确认其中一个选项。

参数:

- **text** (*str*) -- 呈现给用户的消息（或问题）。
- **options** (*list of str*) -- 可选择的选项，即下拉列表选项的标签。
- **title** (*str*) -- 对话框弹出窗口的标题
- **dialog_type** (*int*) --
  向用户显示的对话框类型。可以是以下之一：
  - 0：信息 / 蓝色
  - 1：成功 / 绿色
  - 2：警告 / 橙色
  - 3：错误 / 红色
- **dialog_value** (*str*) -- 如果对话框未被回答，则返回的默认值。
- **mandatory** (*bool*) -- 对话框是否必须回答才能继续程序流程。如果设置为 *True* ，则对话框不能被忽略，例如关闭弹出窗口，并会阻塞直到收到答案。

返回:

所选选项，即被选中的下拉列表标签。

返回类型:

str

示例:

此示例展示了如何创建警告弹窗，并让用户在多个选项中选择，以影响程序流程。

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

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

创建一个带文本输入框的对话框。程序流程将阻塞，直到用户输入文本并确认。

参数:

- **text** (*str*) -- 呈现给用户的消息（或问题）。
- **title** (*str*) -- 对话框弹出窗口的标题
- **dialog_type** (*int*) --
  向用户显示的对话框类型。可以是以下之一：
  - 0：信息 / 蓝色
  - 1：成功 / 绿色
  - 2：警告 / 橙色
  - 3：错误 / 红色
- **dialog_value** (*str*) -- 如果对话框未被回答，则返回的默认值。
- **mandatory** (*bool*) -- 对话框是否必须回答才能继续程序流程。如果设置为 *True* ，则对话框不能被忽略，例如关闭弹出窗口，并会阻塞直到收到答案。

返回:

文本输入框中输入的内容。

返回类型:

str

示例:

此示例展示了如何创建一个简单的带文本输入框的弹窗用于输入您的姓名。用户输入姓名后，可在程序中进一步使用，例如用于日志记录。

```
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*)

创建一个带'是'和'否'按钮的对话框。程序流程将阻塞，直到用户按下其中一个按钮。

参数:

- **text** (*str*) -- 呈现给用户的消息（或问题）。
- **title** (*str*) -- 对话框弹出窗口的标题
- **dialog_type** (*int*) --
  向用户显示的对话框类型。可以是以下之一：
  - 0：信息 / 蓝色
  - 1：成功 / 绿色
  - 2：警告 / 橙色
  - 3：错误 / 红色
- **dialog_value** (*str*) -- 如果对话框未被回答，则返回的默认值。
- **mandatory** (*bool*) -- 对话框是否必须回答才能继续程序流程。如果设置为 *True* ，则对话框不能被忽略，例如关闭弹出窗口，并会阻塞直到收到答案。

返回:

如果用户选择"是"，返回 True；如果选择"否"，返回 False。

返回类型:

bool

示例:

此示例使用一个简单的确认对话框，让用户选择是否继续或终止应用程序。

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

print(**args*)

将消息打印到应用程序输出。此功能用法类似于 Python 内置的 *print* 函数。它支持多个参数和不同类型的参数，参数会自动转换为字符串并用空格连接。

参数:

args (*tuple of various*) -- 要打印的参数。

示例:

以良好格式打印变量的值。第一次调用使用自动转换和连接，第二次调用使用 f-string，结果相同。

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

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

在界面中显示自定义通知消息。这对程序流程没有影响。用户可以在设置菜单中指定通知是淡出还是保留在界面中。

参数:

- **message_text** (*str*) -- 要在界面中显示的通知文本。
- **message_title** (*str*) -- 通知弹窗的标题。

raise_error(*message*)

在正在运行的应用程序中抛出自定义错误消息。该错误类型为 `ApplicationError` ，可以在应用程序中捕获。如果未被捕获，正在运行的应用程序将终止。该错误也会记录到日志系统中的 `error.log` 文件里。可通过界面按下 `CTRL + SHIFT + L` 下载日志。

参数:

message (*str*) -- 要在应用程序中抛出的错误消息。

抛出:

ApplicationError -- 包含自定义消息。

示例:

后台应用程序监控数字输入并检查是否触发了特定的错误输入。如果错误输入被触发，则在应用程序中抛出该错误。

```
# 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)
```

随后该应用程序错误可由控制机器人的主应用程序捕获并处理。

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

raise_warning(*message*)

在界面中弹出自定义警告消息框。这不会影响程序流程。警告是自动消失还是持续显示，可由用户在设置菜单中指定。

参数:

message (*str*) -- 要在界面中显示的警告消息。

示例:

假设我们在模拟输入上连接了一个温度传感器。当数值超过某个阈值时，此示例会在界面中显示警告；当数值再次下降到该阈值以下时，警告将被重置。

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

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

将消息记录到自定义日志文件中。

参数:

- **message** (*str*) -- 要记录的自定义消息。
- **logger** (*str*) -- 要写入的日志文件名称。日志文件位于日志文件夹中的 `custom/<logger>.log` 。默认文件名为 `custom/custom.log` 。可通过界面按下 `CTRL + SHIFT + L` 下载日志。
- **include_timestamp** (*bool*) -- 是否在记录的消息中添加时间戳。

示例:

将传感器测量值记录到 *custom/measurements.log* 文件中。

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

clear_log()

清空并删除所有自定义日志文件。

set_shared_variable(*name*, *value*)

设置共享变量的值。

请注意，*共享变量* 只在主应用程序运行期间存在。主应用程序终止后，所有 *共享变量* 都会被自动删除。如果需要变量在应用程序运行期间甚至在机器人重启后仍保留其值，请改用 *全局变量* 。

参数:

- **name** (*str*) -- 要设置的变量名称。
- **value** (*various*) -- 该变量的新值。

示例:

该示例展示了主应用程序如何使用共享变量终止并行应用程序。并行应用程序只需监听一个终止标志，一旦该标志被设置为 *True* 就会立即终止。此外，当共享变量不再起作用时会将其删除。由于主应用程序终止后共享变量会被自动清除，这并非必须，但属于良好实践。

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

主应用程序确保终止标志存在，运行并行应用程序，并最终将终止标志设置为 *True* 以结束并行应用程序。

```
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*)

读取指定共享变量的值。默认情况下，该函数会读取所有共享变量的值。

请注意，*共享变量* 只在主应用程序运行期间存在。主应用程序终止后，所有 *共享变量* 都会被自动删除。如果需要变量在应用程序运行期间甚至在机器人重启后仍保留其值，请改用 *全局变量* 。

参数:

name (*str (or None)*) -- 要返回的变量名称（如果返回所有变量则为 None）。

返回:

要返回的变量值（或包含所有变量的字典）。

返回类型:

various (or dict)

示例:

请参阅 *set_shared_variable* 的示例，其中综合展示了所有 *共享变量* 相关函数的用法。

remove_shared_variable(*name*)

删除一个共享变量。删除后将无法再访问其值。

请注意，*共享变量* 只在主应用程序运行期间存在。主应用程序终止后，所有 *共享变量* 都会被自动删除。如果需要变量在应用程序运行期间甚至在机器人重启后仍保留其值，请改用 *全局变量* 。

参数:

name (*str*) -- 要删除的变量名称

示例:

请参阅 *set_shared_variable* 的示例，其中综合展示了所有 *共享变量* 相关函数的用法。

set_global_variable(*name*, *value*)

设置全局变量的值。

请注意，对 *全局变量* 的更改是永久性的，即使关闭并重新启动机器人后仍然存在。因此，强烈建议在 *全局变量* 不再有任何用途时将其删除。如果变量只需要在单个应用运行期间使用，请改用 *共享变量* 。

参数:

- **name** (*str*) -- 要设置的变量名称。
- **value** (*various*) -- 该变量的新值。

示例:

此示例展示了一个重复应用如何统计其成功迭代的次数。方法是在首次运行时初始化一个全局计数器，并在每次循环迭代结束时递增该计数器。

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

get_global_variable(*name=None*)

读取指定全局变量的值。默认情况下，此函数会读取所有全局变量的值。

请注意，对 *全局变量* 的更改是永久性的，即使关闭并重新启动机器人后仍然存在。因此，强烈建议在 *全局变量* 不再有任何用途时将其删除。如果变量只需要在单个应用运行期间使用，请改用 *共享变量* 。

参数:

name (*str (or None)*) -- 要返回的变量名称（如果返回所有变量则为 None）。

返回:

要返回的变量值（或包含所有变量的字典）。

返回类型:

various (or dict)

示例:

请查看 *set_global_variable* 的示例，其中综合演示了 *全局变量* 相关函数的用法。

remove_global_variable(*name*)

删除一个全局变量。删除后，将无法再访问其值。

请注意，对 *全局变量* 的更改是永久性的，即使关闭并重新启动机器人后仍然存在。因此，强烈建议在 *全局变量* 不再有任何用途时将其删除。如果变量只需要在单个应用运行期间使用，请改用 *共享变量* 。

参数:

name (*str*) -- 要删除的变量名称

示例:

此示例会删除所有全局变量。

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

set_state_variable(*name*, *value*)

设置状态变量的值。

请注意，*状态变量* 变量是针对每个应用单独定义的，并且仅在主应用中使用。它们可以在应用编辑器的 *variables* 选项卡中进行设置和自定义。每个 *状态变量* 都有特定的类型，并且只接受该类型的值。*状态变量* 的主要用途是在 *Panel Interface* 中实时显示其值，用于观察和控制正在运行的应用。

参数:

- **name** (*str*) -- 要设置的变量名称。
- **value** (*various*) -- 该变量的新值。

示例:

假设在此应用中定义了一个只读的 *percent* 类型状态变量 *progress* ，以及一个可动态编辑的 *boolean* 类型状态变量 *logging* 。*progress* 变量用于以百分比形式存储程序的当前进度；*logging* 标志用于决定是否记录进度更新。用户可以在任何时候切换 *logging* 的值，从而动态调整程序行为。

```
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*)

读取指定状态变量的值。默认情况下，此函数会读取所有状态变量的值。

请注意，*状态变量* 变量是针对每个应用单独定义的，并且仅在主应用中使用。它们可以在应用编辑器的 *variables* 选项卡中进行设置和自定义。每个 *状态变量* 都有特定的类型，并且只接受该类型的值。*状态变量* 的主要用途是在 *Panel Interface* 中实时显示其值，用于观察和控制正在运行的应用。

参数:

name (*str (or None)*) -- 要返回的变量名称（如果返回所有变量则为 None）。

返回:

要返回的变量值（或包含所有变量的字典）。

返回类型:

various (or dict)

示例:

请查看 *set_state_variable* 的示例，其中综合演示了 *状态变量* 相关函数的用法。

read_actuator_position(*actuator_ids=None*)

读取一个、多个或所有执行器的角度位置。

参数:

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

要读取的执行器ID。可按如下方式指定：

- 单个关节的ID（例如 *6* ）
- 多个关节的ID列表（例如 *[1, 4, 6]* ）
- 多个关节的关节名称集合（例如：*{1, 4, 6}* ）
- 使用常量 *ALL_KIN_JOINTS* 或 *None* 从所有关节读取

返回:

所请求执行器的角度位置（单位：[deg]），其形式可以是：

- 单个关节位置值 -> float
- 关节位置值列表 -> float 列表
- 关节ID与其位置值的映射 -> float 字典

返回类型:

float, or list of float, or dict of float

示例:

打印腕部关节执行器的当前位置信息。

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

read_actuator_velocity(*actuator_ids=None*)

读取一个、多个或所有执行器的角速度。

参数:

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

要读取的执行器ID。可按如下方式指定：

- 单个关节的ID（例如 *6* ）
- 多个关节的ID列表（例如 *[1, 4, 6]* ）
- 多个关节的关节名称集合（例如：*{1, 4, 6}* ）
- 使用常量 *ALL_KIN_JOINTS* 或 *None* 从所有关节读取

返回:

所请求执行器的角速度（单位：°/秒），可以是：

- 单个关节速度值 -> 浮点数
- 关节速度值列表 -> 浮点数列表
- 关节ID与其速度值的映射 -> 浮点数字典

返回类型:

float, or list of float, or dict of float

示例:

在运动过程中监控速度，以确定移动最快的执行器。

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

get_linear_velocity()

获取机器人TCP（工具中心点）的线速度。

返回:

工具的线速度 [mm/s]。

返回类型:

list

示例:

获取并打印当前线速度。

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

get_angular_velocity()

获取机器人TCP（工具中心点）的角速度。

返回:

工具的角速度 [°/秒]。

返回类型:

list

示例:

获取并打印当前角速度。

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

read_actuator_current(*actuator_ids=None*)

读取一个、多个或所有执行器的施加电流。

参数:

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

要读取的执行器ID。可按如下方式指定：

- 单个关节的ID（例如 *6* ）
- 多个关节的ID列表（例如 *[1, 4, 6]* ）
- 多个关节的关节名称集合（例如：*{1, 4, 6}* ）
- 使用常量 *ALL_KIN_JOINTS* 或 *None* 从所有关节读取

返回:

所请求执行器的电流值（单位：A），可以是：

- 单个关节电流值 -> 浮点数
- 关节电流值列表 -> 浮点数列表
- 关节ID与电流值的映射 -> 浮点数字典

返回类型:

float, or list of float, or dict of float

示例:

打印机器人在特定姿态下的执行器电流，两次测量：一次在抓取物体前，一次在抓取物体后。

```
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()

激活工具的抓取功能以抓取物体。这可能导致机械工具夹紧或气动工具吸附。

示例:

激活工具抓取功能。

```
tool_pick()
```

tool_place()

激活工具的放置功能以释放物体。这可能导致机械工具松开夹具或停止气动吸附。

示例:

激活工具放置功能。

```
tool_place()
```

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

在机器人操作过程中动态设置工具负载。

参数:

- **payload** (*float*) -- 工具负载（重量 + 抓取物品），单位：[kg]
- **center_of_mass** (*list of float*) -- 相对于法兰框架的质心 [x, y, z]，单位为米 [m]
- **inertia** (*list of float*) -- 惯性矩阵展开为长度为9的列表 [kg·m²]

示例:

在从一个位置抓取和放置物体时更改工具负载。

```
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*)

读取'数字主输入'类别中一个、多个或所有I/O的值。

数字I/O的两种状态为：

- *HIGH*，用布尔值 *True* 表示
- *LOW*，用布尔值 *False* 表示

参数:

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

指定要读取哪些 I/O。标识符可以是 I/O 编号，也可以是可分配给这些 I/O 的唯一自定义名称。可以是以下之一：

- 单个数字 (int) 或自定义名称 (str) -> 返回布尔值
- 数字列表 (int) 或自定义名称列表 (str) 或混合 -> 返回布尔列表
- 数字集合 (int) 或自定义名称集合 (str) 或混合 -> 返回布尔值字典，键为标识符

返回:

所请求数字I/O的布尔值 (HIGH = True, LOW = False)

返回类型:

bool, list, dict

示例:

假设将名称 'trigger' 分配给其中一个数字输入。此代码块将等待该触发器变为 HIGH 后再继续执行。

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

读取前8个数字I/O，并将布尔值的二进制列表转换为0到255之间的整数。

```
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*)

读取'数字工具输入'类别中一个、多个或所有I/O的值。

有关详细信息，请参阅用法类似的'read_digital_main_input'。

read_analog_main_input(*identifiers*)

读取'模拟主输入'类别中一个、多个或所有I/O的值。

模拟 I/O 可以处于电压模式或电流模式。数值以 0.1 为步进进行设置和读取，这意味着 I/O 值 55 表示实际电压值 5.5 V 或实际电流值 5.5 mA。模拟 I/O 的取值范围为：

- 0.0 至 10.0 V（电压模式）
- 4.0 至 20.0 mA（电流模式）

参数:

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

指定要读取哪些 I/O。标识符可以是 I/O 编号，也可以是可分配给这些 I/O 的唯一自定义名称。可以是以下之一：

- 单个数字 (int) 或自定义名称 (str) -> 返回布尔值
- 数字列表 (int) 或自定义名称列表 (str) 或混合 -> 返回布尔列表
- 数字集合 (int) 或自定义名称集合 (str) 或混合 -> 返回布尔值字典，键为标识符

返回:

所请求的模拟 I/O 的数值

返回类型:

float, list, dict

示例:

假设在电流模式下的某个模拟输入被命名为 'temperature'。一个温度传感器连接到该输入，并且已确定返回值 10.6 mA 表示一个不应被超过的临界温度阈值。随后请将以下程序与主应用程序并行运行，当达到该临界阈值时，该程序将中止运行。

```
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*)

读取'模拟工具I/O'类别中一个、多个或所有I/O的值。

有关详细信息，请参阅用法类似的'read_analog_main_input'。

write_digital_main_output(*identifiers*, *values*)

写入'数字主输出'类别中一个、多个或所有I/O的值。

数字I/O的两种状态为：

- *HIGH*，用布尔值 *True* 表示
- *LOW*，用布尔值 *False* 表示

参数:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) --
  指定要写入哪些 I/O。标识符可以是 I/O 编号，也可以是可分配给这些 I/O 的唯一自定义名称。可以是以下之一：
  - 单个数字 (int) 或自定义名称 (str) -> *values* 是布尔值
  - 数字列表 (int) 或自定义名称列表 (str) 或混合 -> *values* 是布尔值列表
  - 数字集合 (int) 或自定义名称集合 (str) 或混合 -> *values* 是布尔值字典，键为标识符
- **values** (*bool, list (of bool), dict (of bool)*) --
  要写入的数字I/O的布尔值 (HIGH = True, LOW = False)。根据 *identifiers*，可以是以下之一：
  - 单个布尔值 -> 如果 *identifiers* 是单个 I/O
  - 布尔值列表 -> 如果 *identifiers* 是 I/O 列表
  - 布尔值字典，键为标识符 -> 如果 *identifiers* 是 I/O 集合

示例:

假设将名称 'trigger' 分配给其中一个数字输出。此代码通过将其设为 HIGH 持续 100 毫秒，然后恢复为 LOW，向该数字输出发送一个脉冲。

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

将一个8位整数（0到255之间）写入前8个数字I/O，先将数字转换为布尔列表，然后同时写入全部8个值。

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

write_digital_tool_output(*identifiers*, *values*)

写入'数字工具输出'类别中一个、多个或所有I/O的值。

有关详细信息，请参阅用法类似的'write_digital_main_output'。

write_analog_main_output(*identifiers*, *values*)

写入'模拟主输出'类别中一个、多个或所有I/O的值。

模拟 I/O 可以处于电压模式或电流模式。数值以 0.1 为步进进行设置和读取，这意味着 I/O 值 55 表示实际电压值 5.5 V 或实际电流值 5.5 mA。模拟 I/O 的取值范围为：

- 0.0 至 10.0 V（电压模式）
- 4.0 至 20.0 mA（电流模式）

参数:

- **identifiers** (*int, str, list (of int or str), set (of int or str)*) --
  指定要写入哪些 I/O。标识符可以是 I/O 编号，也可以是可分配给这些 I/O 的唯一自定义名称。可以是以下之一：
  - 单个数字 (int) 或自定义名称 (str) -> *values* 是 float
  - 数字列表 (int) 或自定义名称列表 (str) 或混合 -> *values* 是 float 列表
  - 一组数字 (int) 或自定义名称 (str)，或混合 -> *values* 是一个以标识符为键、值为 float 的字典
- **values** (*float, list (of float), dict (of float)*) --
  要写入的模拟 I/O 的数值。根据 *identifiers*，可以是以下之一：
  - 单个 float -> 如果 *identifiers* 是单个 I/O
  - float 列表 -> 如果 *identifiers* 是 I/O 列表
  - 一个以标识符为键、值为 float 的字典 -> 如果 *identifiers* 是 I/O 集合

示例:

假设一个可控 LED 的 RGB 值分别直接连接到三个模拟 I/O，其自定义名称为 'red' 'green' 和 'blue'，使得最大数值范围 [0, 255] 直接映射到最大电压范围 [0.0 V, 10.0 V]。此示例同时写入这些数值。

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

write_analog_tool_io(*identifiers*, *values*)

写入'模拟工具I/O'类别中一个、多个或所有I/O的值。

有关详细信息，请参阅用法类似的'write_analog_main_output'。

wait_for_digital_main_input(*identifier*, *value*)

等待'数字主输入'类别中的某个I/O读取到特定值。

数字I/O的两种状态为：

- *HIGH*，用布尔值 *True* 表示
- *LOW*，用布尔值 *False* 表示

参数:

- **identifier** (*int, str*) -- 指定要等待的I/O。标识符可以是I/O编号 (int) 或分配给该I/O的唯一自定义名称 (str)。
- **value** (*bool*) -- 要等待的值。

示例:

假设将名称 'trigger' 分配给其中一个数字输入。此代码块将等待该触发器变为 HIGH 后再继续执行。

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

在某些罕见情况下，你可能希望能够暂停程序，并且在I/O被触发时等待函数仍然终止。此等待函数不能用于这种方法，但下面的示例可以实现这一点：

将此代码添加到后台应用程序中。它监视特定I/O的值，并在监控过程中达到指定值时设置标志。

```
# 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)
```

并将此代码添加到主应用程序中，以触发后台监控并等待该标志被监控器设置。

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

wait_for_digital_tool_input(*identifier*, *value*)

等待'数字工具输入'类别中的某个I/O读取到特定值。

有关详细信息，请参阅用法类似的'wait_for_digital_main_input'。

wait_for_analog_main_input(*identifier*, *value_range*)

等待'模拟主输入'类别中的某个I/O读取到特定值。

模拟 I/O 可以处于电压模式或电流模式。数值以 0.1 为步进进行设置和读取，这意味着 I/O 值 55 表示实际电压值 5.5 V 或实际电流值 5.5 mA。模拟 I/O 的取值范围为：

- 0.0 至 10.0 V（电压模式）
- 4.0 至 20.0 mA（电流模式）

参数:

- **identifier** (*int, str*) -- 指定要等待的I/O。标识符可以是I/O编号 (int) 或分配给该I/O的唯一自定义名称 (str)。
- **value_range** (*list of float*) --
  正在等待该 I/O 达到此范围内的值。这是以下之一：
  - 同时设置下限和上限（例如：[5.5, 7.0]）
  - 仅设置下限（例如：[5.5, None]）
  - 仅设置上限（例如：[None, 7.0]）

示例:

我们考虑一个处于电压模式的模拟 I/O。此示例会等待其电压值增长到超过 8.0 V，这由数值 80 表示（以 0.1 V 为步进）。

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

wait_for_analog_tool_io(*identifier*, *value_range*)

等待'模拟工具I/O'类别中的某个I/O读取到特定值。

有关详细信息，请参阅用法类似的'wait_for_analog_main_input'。

get_runtime_position_offsets()

该函数返回机器人的位置偏移量。

返回:

机器人的位置偏移量

返回类型:

list of float

read_button_state(*channel=None*)

读取机器人界面中一个或所有可自定义按钮的当前状态（是否按下）。请注意，具有特定功能分配的按钮在运行时无法读取，因为它们正在使用中。

这些按钮位于机器人手臂的最上方关节。

参数:

channel (*int or None*) -- 按钮标识符，为1到按钮总数之间的数字。默认情况下，读取所有可自定义按钮的状态。

返回:

请求按钮的当前状态为布尔值，或每个按钮的当前状态，打包为形如 {1: bool, 2: bool, ...} 的字典。

返回类型:

bool or dict of bool

示例:

此示例在可自定义按钮状态变化时打印信息。

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

将此示例作为后台应用程序运行。它允许您使用按钮1和2暂停和恢复正在运行的主应用程序。

```
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*)

创建一个TCP套接字客户端用于发送消息。客户端会持续尝试连接到指定地址的服务器。消息使用 *UTF-8* 编码标准编码。此外，消息通过文本结束字符 *0x03*（Python中为 `\x03` ）分隔。

参数:

- **ip** (*str*) -- 服务器的IP地址（例如 `"192.168.80.2"` ）
- **port** (*int*) -- 服务器可访问的端口号（例如 `60001` ）。端口限制在 60000 到 61000 之间。

返回:

TCP 套接字客户端对象，用于发送和接收消息。

返回类型:

Socket

示例:

此示例演示应用程序中的 TCP 套接字客户端与同一网络中的 TCP 套接字服务器之间的握手。首先建立连接，然后客户端等待服务器发送初始消息。收到消息后，客户端发送自己的消息，最后关闭连接。

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

create_tcp_server(*port*)

创建一个 TCP 套接字服务器，可处理最多 20 个客户端。消息使用 *UTF-8* 编码，并用文本结束字符 *0x03*`（*Python 中为* `x03`` ）分隔。

参数:

port (*int*) -- 客户端可访问此服务器的端口号（例如 `60002` ）。端口限制在 60000 到 61000 之间。

返回:

TCP 套接字服务器对象，用于发送和接收消息。

返回类型:

Socket

示例:

此示例演示应用程序中的 TCP 套接字服务器与第一个连接的客户端之间的握手。首先，服务器在指定端口上打开连接并等待客户端连接。然后服务器等待客户端发送初始消息。收到消息后，服务器发送自己的消息，并最终关闭与所有已连接客户端的连接。

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

create_udp_socket(*port*)

创建一个 UDP 套接字用于传输消息。消息使用 *UTF-8* 编码，并用文本结束字符 *0x03*` *（Python 中为* `x03`` ）分隔。

参数:

port (*int*) -- 此套接字可访问的端口号（例如 `60003` ）。端口限制在 60000 到 61000 之间。

返回:

UDP 套接字对象，用于发送和接收消息。

返回类型:

Socket

示例:

此示例演示应用程序中的 UDP 套接字与第一个连接到它的对端之间的握手。首先，套接字在指定端口上打开连接并等待对端连接。然后套接字等待对端发送初始消息。收到消息后，套接字发送自己的消息，最后关闭连接。

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

send_message(*message*)

向打开的 TCP 连接发送带有键 'script_send' 的消息。

参数:

message (*str*) -- 要发送的消息。

示例:

通过 TCP 向任何可能监听的设备发送消息：

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

在连接的设备端，这些数据将以以下格式接收：

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

receive_message(*timeout=None*)

从调用 'script_receive' 动作的打开 TCP 连接接收消息。

参数:

timeout (*None or float*) -- 等待消息的最长时间。如果未指定，则无限等待。

示例:

外部设备通过 TCP 连接到机器人并发送以下数据：

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

在机器人端，此函数可用于接收并打印消息：

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

clear_messages()

清除 TCP 消息队列中可能尚未读取的先前消息。

示例:

在等待新消息之前，有时需要确保旧消息被拒绝。

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

class Coordinates

Coordinates 对象定义了一个数学变换，可用于运动函数中的目标坐标，或用于定义坐标系。

__init__(*coordinates*)

创建一个 Coordinates 对象，可以使用特定的坐标值，或包含当前机器人坐标。

参数:

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

一组特定的坐标，可以是以下之一：

- 列表形式 [x, y, z, roll, pitch, yaw]，位置值单位为 [mm]，方向值单位为 [°]
- 字典形式 {'x': x, 'y': y, 'z': z, 'roll': roll, 'pitch': pitch, 'yaw': yaw}，位置值单位为 [mm]，方向值单位为 [°]
- 一个 Coordinates 对象（复制另一个 Coordinates 对象）

构造函数：

根据坐标值创建一个 Coordinates 对象。

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

创建一个包含当前机器人坐标的 Coordinates 对象，使用 `get_current_coordinates` 函数。

```
current_coordinates = get_current_coordinates()
```

***property* position**

Coordinates 对象的位置部分。

返回:

位置，形式为 [x, y, z]，单位为 [mm]。

返回类型:

list of float

***property* orientation**

Coordinates 对象的方向部分。

返回:

方向，形式为 [roll, pitch, yaw]，单位为 [°]。

返回类型:

list of float

to_list()

将 Coordinates 对象转换为坐标值列表，形式为 [x, y, z, roll, pitch, yaw]，位置值单位为 [mm]，旋转值单位为 [°]。

返回:

坐标值的列表格式。

返回类型:

list of float

示例:

当需要以列表形式使用 Coordinates 时可调用此函数：

```
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()

将 Coordinates 对象转换为坐标值字典，形式为 {"x": x, "y": y, "z": z, "roll": roll, "pitch": pitch, "yaw": yaw}，位置值单位为 [mm]，旋转值单位为 [°]。

返回:

坐标值的字典格式。

返回类型:

dict of float

示例:

当需要以字典形式使用 Coordinates 时可调用此函数：

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

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

inverted()

计算此 Coordinates 对象的逆变换。

返回:

变换对象的逆

返回类型:

Transform

示例:

```
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 对象定义了机器人的姿态，由其坐标（位置和方向）以及配置组成（因为同一组坐标可以通过多种配置到达）。

__init__(*pose*)

创建一个 Pose 对象，可以通过名称或 ID 加载已保存的姿态，或使用机器人的当前姿态。

参数:

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

要加载的姿态引用，可以是以下之一：

- 数据库中保存的位姿名称（str）。
- 数据库中已保存姿态的 ID（int）
- 一个 Pose 对象（复制另一个 Pose 对象）

构造函数：

从数据库加载一个姿态。

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

创建一个 Pose 对象，包含机器人的当前姿态，使用 `get_current_pose` 函数。

```
current_pose = get_current_pose()
```

创建一个 Pose 对象，包含指定的坐标和配置，使用 `create_pose` 函数。

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

***property* coordinates**

该姿态的坐标，是一个 Coordinates 对象，包含位置和方向的值。

返回:

该姿态的坐标。

返回类型:

Coordinates

***property* configuration**

由于一组姿态坐标可以通过机器人关节的最多 8 种可能配置到达，因此配置定义为一个数字（0 到 7）或字符串（'c1' 到 'c8'）以唯一确定姿态。

返回:

姿态配置的索引（从 0 到 7）。

返回类型:

int

class LinearSegment

LinearSegment 对象描述 TCP（工具中心点）到目标坐标的线性路径段。此外，它包含所需的线速度、线加速度以及段开始处的混合半径信息。该混合半径允许从线性路径偏离，以确保段之间平滑过渡。

__init__(*linear_segment*)

创建一个 LinearSegment 对象。

参数:

- **coordinates** (*list, dict, or Coordinates*) -- 目标坐标。
- **linear_velocity** (*float or None*) -- 所需线速度 [mm/s]。如果未定义线速度，将使用全局线速度值。
- **linear_acceleration** (*float or None*) -- 所需线加速度 [mm/s^2]。如果未定义线加速度，将使用全局线加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 LinearSegment 对象：

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

或使用默认值：

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

***property* coordinates**

此段的目标坐标，是一个 Coordinates 对象，包含位置和方向值。

返回:

此段的目标坐标。

返回类型:

Coordinates

***property* blend_radius**

此段开始处的混合半径。

返回:

起始混合半径。

返回类型:

float

***property* linear_acceleration**

此段所需的线加速度。如果为 None，则使用全局线加速度值。

返回:

所需的线加速度。

返回类型:

float or None

***property* linear_velocity**

此段所需的线速度。如果为 None，则使用全局线速度值。

返回:

所需的线速度。

返回类型:

float or None

class CircularSegment

CircularSegment 对象描述 TCP（工具中心点）通过中间位置到目标坐标的圆周运动。请注意，该中间姿态的方向由算法确定。此外，它包含所需的线速度、线加速度以及段开始处的混合半径信息。该混合半径允许偏离圆周路径，以确保段之间的平滑过渡。

__init__(*circular_segment*)

创建一个 CircularSegment 对象。

参数:

- **intermediate_position** (*list*) -- 通向目标坐标路径上的中间位置 [x, y, z]。中间位置决定圆周运动的曲率。
- **coordinates** (*list, dict or Coordinates*) -- 目标坐标。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s^2]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 CircularSegment 对象：

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

或使用默认值：

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

***property* intermediate_position**

该段的中间位置。包括 x, y, z 坐标：[x, y, z]

返回:

该段的中间位置。

返回类型:

list

***property* coordinates**

此段的目标坐标，是一个 Coordinates 对象，包含位置和方向值。

返回:

此段的目标坐标。

返回类型:

Coordinates

***property* blend_radius**

此段开始处的混合半径。

返回:

起始混合半径。

返回类型:

float

***property* linear_acceleration**

此段所需的线加速度。如果为 None，则使用全局线加速度值。

返回:

所需的线加速度。

返回类型:

float or None

***property* linear_velocity**

此段所需的线速度。如果为 None，则使用全局线速度值。

返回:

所需的线速度。

返回类型:

float or None

class OrientationSegment

OrientationSegment 对象描述了一个纯姿态路径段，其中 TCP（工具中心点）根据所需工具姿态旋转，同时保持位置不变。此外，它还包含期望的角速度、角加速度和段开始处的混合半径信息。如果混合半径大于 0，则 TCP 的旋转将在到达目标位置前开始。

__init__(*orientation_segment*)

创建 OrientationSegment 对象。

参数:

- **orientation** (*list*) -- 目标姿态。注意位置保持不变。
- **angular_velocity** (*float or None*) -- 期望的角速度 [°/s]。如果未定义角速度，将使用全局角速度值。
- **angular_acceleration** (*float or None*) -- 期望的角加速度 [°/s^2]。如果未定义角加速度，将使用全局角加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。混合半径大于 0 时，TCP（工具中心点）可以在到达该段位置前旋转。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 OrientationSegment 对象：

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

或使用默认值：

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

***property* orientation**

该段的目标姿态。以航向角 [roll, pitch, yaw]（度）表示

返回:

该段的目标姿态。

返回类型:

list

***property* angular_velocity**

该段期望的角速度。如果为 None，则使用全局角速度值。

返回:

期望的角速度。

返回类型:

float or None

***property* angular_acceleration**

该段期望的角加速度。如果为 None，则使用全局角加速度值。

返回:

期望的角加速度。

返回类型:

float or None

***property* blend_radius**

此段开始处的混合半径。

返回:

起始混合半径。

返回类型:

float

class CoordinatesSegment

CoordinatesSegment 对象描述了关节空间中的线性运动以达到目标坐标。相比之下，TCP（工具中心点）通常不沿线性路径移动。此外，它还包含期望的线速度、线加速度以及段开始处的混合半径信息。该混合半径允许路径偏离，以确保段之间平滑过渡。

__init__(*coordinates_segment*)

创建 CoordinatesSegment 对象。

参数:

- **coordinates** (*list, dict or Coordinates*) -- 目标坐标。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s^2]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 CoordinatesSegment 对象：

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

或使用默认值：

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

***property* coordinates**

此段的目标坐标，是一个 Coordinates 对象，包含位置和方向值。

返回:

此段的目标坐标。

返回类型:

Coordinates

***property* blend_radius**

此段开始处的混合半径。

返回:

起始混合半径。

返回类型:

float

***property* linear_acceleration**

此段所需的线加速度。如果为 None，则使用全局线加速度值。

返回:

所需的线加速度。

返回类型:

float or None

***property* linear_velocity**

此段所需的线速度。如果为 None，则使用全局线速度值。

返回:

所需的线速度。

返回类型:

float or None

class PoseSegment

PoseSegment 对象描述了关节空间中的线性运动以达到目标姿态。相比之下，TCP（工具中心点）通常不沿线性路径移动。此外，它还包含期望的线速度、线加速度以及段开始处的混合半径信息。该混合半径允许路径偏离，以确保段之间平滑过渡。

__init__(*pose_segment*)

创建 PoseSegment 对象。

参数:

- **pose** (*Pose*) -- 目标姿态。
- **linear_velocity** (*float or None*) -- 期望的线性速度 [mm/s]。如果未定义线性速度，将使用全局线性速度值。
- **linear_acceleration** (*float or None*) -- 期望的线性加速度 [mm/s²]。如果未定义线性加速度，将使用全局线性加速度值。
- **blend_radius** (*float or None*) -- 段开始处的混合半径 [mm]。如果未定义混合半径，将使用 0.0。

构造函数：

通过定义所有参数创建 PoseSegment 对象：

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

或使用数据库中的默认值和姿态：

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

***property* pose**

该段的目标姿态，是一个Pose对象，包含位置、方向和配置值。

返回:

该段的目标姿态。

返回类型:

Pose

***property* blend_radius**

此段开始处的混合半径。

返回:

起始混合半径。

返回类型:

float

***property* linear_acceleration**

此段所需的线加速度。如果为 None，则使用全局线加速度值。

返回:

所需的线加速度。

返回类型:

float or None

***property* linear_velocity**

此段所需的线速度。如果为 None，则使用全局线速度值。

返回:

所需的线速度。

返回类型:

float or None

class Path

Path对象定义了机器人在两个或多个姿态之间的运动方式。连续路径仅在工具空间中由多个段定义，如线性段或圆弧段。如果路径的离散解不需要机器人重新定位（配置变化），则称该路径为无缝路径。如果沿路径的加速度连续，则称路径为连续路径。路径从指定的姿态开始（而不是坐标）。路径的各段通常定义坐标，而路径的配置由初始姿态定义。

__init__(*path*)

创建Path对象，可以通过名称或ID加载已保存的路径。

参数:

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

要加载的路径引用，可以是以下之一：

- 数据库中已保存路径的名称（str）
- 数据库中已保存路径的 ID（int）
- 一个Path对象（复制另一个Path对象）

构造函数：

从数据库加载路径。

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

创建Path对象，包含指定的初始姿态和定义的路径段，使用 `create_path` 函数。注意，路径段也可以在后续添加。

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

***property* initial_pose**

路径的机器人初始姿态。

返回:

初始机器人姿态。

返回类型:

Pose

***property* segments**

路径中包含的段。注意，段的类型不必全部相同。

返回:

包含的段。

返回类型:

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

add_segment(*segment*)

将段添加到路径末尾。

参数:

segment (*LinearSegment*, *CircularSegment*, *OrientationSegment*, *CoordinatesSegment*, *PoseSegment*) -- 要添加的段。

示例:

可以按如下方式添加段：

```
# creating and adding a segment
lin_seg = LinearSegment([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_segment(lin_seg)
```

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

将线性段添加到路径末尾。

参数:

- **coordinates** (*list, dict or Coordinates*) -- 路径的目标坐标。
- **linear_velocity** (*float*) -- 允许的最大线性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允许的最大线性工具加速度 [mm/s^2]。
- **blend_radius** (*float*) -- 段开始处允许的混合半径 [mm]。

示例:

可以按如下方式添加线性段：

```
# add a linear segment with different argument types for the "coordinates" parameter
path.add_linear([700.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_linear({"x": 700.0, "y": -125.0, "z": 500.0,
                 "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_linear(Coordinates([700.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
# first linear segment is added
coord = Coordinates([700.0, -250.0, 500.0, 180.0, 0.0, -90.0])
path.add_linear(coord, linear_velocity=500.0, linear_acceleration=1000.0)

# second linear segment is added
path.add_linear([700.0, 250.0, 500.0, 180.0, 0.0, -90.0],
                linear_velocity=250.0, linear_acceleration=500.0, blend_radius=100.0)
```

第一个段定义了一个朝向指定目标坐标的线性运动，最大速度为500 mm/s，最大加速度为1000 mm/s^2。注意，这些坐标不会被精确达到，因为后续段定义了100 mm的混合半径，这允许偏离这些坐标以确保更平滑和高效的过渡。第二个段随后将线性运动添加到下一个坐标，最大速度为250 mm/s，最大加速度为500 mm/s^2。

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

将圆弧段添加到路径末尾。

参数:

- **intermediate_position** (*list*) -- 路径中到目标坐标的中间位置 ([x, y, z])。中间位置决定圆弧运动的曲率。
- **coordinates** (*list, dict or Coordinates*) -- 路径的目标坐标。
- **linear_velocity** (*float*) -- 允许的最大线性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允许的最大线性工具加速度 [mm/s^2]。
- **blend_radius** (*float*) -- 段开始处允许的混合半径 [mm]。

示例:

可以按如下方式添加圆弧段：

```
# add a circular segment with different argument types for the "coordinates" parameter
path.add_circular([0.0, -500.0, 500.0], [-500.0, -125.0, 500.0, 180.0, 0.0, -90.0])
path.add_circular([0.0, -500.0, 500.0], {"x": -500.0, "y": -125.0, "z": 500.0,
                           "roll": 180.0, "pitch": 0.0, "yaw": -90.0})
path.add_circular([0.0, -500.0, 500.0],
                           Coordinates([-500.0, -125.0, 500.0, 180.0, 0.0, -90.0]))
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
# first circular segment is added
coord = Coordinates([-500.0, 250.0, 500.0, 180.0, 0.0, -90.0])
path.add_circular([-700.0, 0.0, 500.0], coord,
                           linear_velocity=500.0, linear_acceleration=1000.0)

# second circular segment is added
path.add_circular([-700.0, 0.0, 500.0], [-500.0, -250.0, 500.0, 180.0, 0.0, -90.0],
                           linear_velocity=250.0, linear_acceleration=500.0, blend_radius=0.0)
```

第一个段定义了通过指定的中间位置向目标坐标进行圆周运动，最大速度为 500 mm/s，最大加速度为 1000 mm/s^2。请注意，这些坐标不会被完全达到，因为后续段定义了 100 mm 的混合半径，允许偏离这些坐标以确保更平滑和高效的过渡。第二个段则通过下一个中间位置向下一个目标坐标进行圆周运动，最大速度为 250 mm/s，最大加速度为 500 mm/s^2。

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

在路径末尾添加一个方向段。

参数:

- **coordinates** (*list, dict or Coordinates*) -- 路径的目标坐标。
- **angular_velocity** (*float*) -- 允许的最大工具角速度 [°/s]。
- **angular_acceleration** (*float*) -- 允许的最大工具角加速度 [°/s²]。
- **blend_radius** (*float*) -- 段开始处允许的混合半径 [mm]。值大于 0 表示在到达此段目标位置前，工具可以提前旋转。

示例:

方向段可以按如下方式添加：

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

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
# first orientation segment is added
path.add_orientation([180.0, 0.0, -90.0],
                     angular_velocity=250.0, angular_acceleration=500.0)

# second orientation segment is added
path.add_orientation([150.0, 0.0, -90.0], angular_velocity=500.0,
                     angular_acceleration=1000.0, blend_radius=100.0)
```

第一个段定义了到指定目标方向的旋转，最大速度为 250°/s，最大加速度为 500°/s²。请注意，在执行该路径时，第一个旋转将被跳过，因为后续段的混合半径大于 0（参见 `OrientationSegment.__init__` ），并将直接执行第二段。第二段定义了到目标方向 [150.0, 0.0, -90.0] 的旋转，最大速度为 500°/s，最大加速度为 1000°/s²。相反，如果第二段的混合半径设为 0.0，则两个旋转将按顺序执行。

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

在路径末尾添加一个坐标段。

参数:

- **coordinates** (*list, dict or Coordinates*) -- 路径的目标坐标。
- **linear_velocity** (*float*) -- 允许的最大线性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允许的最大线性工具加速度 [mm/s^2]。
- **blend_radius** (*float*) -- 段开始处允许的混合半径 [mm]。

示例:

坐标段可以按如下方式添加：

```
# 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]))
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
# first coordinates segment is added
coord = Coordinates([700.0, -250.0, 500.0, 180.0, 0.0, -90.0])
path.add_coordinates(coord, linear_velocity=500.0, linear_acceleration=1000.0)

# second coordinates segment is added
path.add_coordinates([700.0, 250.0, 500.0, 180.0, 0.0, -90.0], linear_velocity=250.0,
                     linear_acceleration=500.0, blend_radius=100.0)
```

第一个段定义了朝指定目标坐标的运动，最大速度为 500 mm/s，最大加速度为 1000 mm/s^2。请注意，这些坐标不会被完全达到，因为后续段定义了 100 mm 的混合半径，允许偏离这些坐标以确保更平滑、更高效的过渡。第二个段则添加了朝下一个坐标的运动，最大速度为 250 mm/s，最大加速度为 500 mm/s^2。

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

在路径末尾添加一个姿态段。

参数:

- **pose** (*Pose*) -- 路径的目标坐标。
- **linear_velocity** (*float*) -- 允许的最大线性工具速度 [mm/s]。
- **linear_acceleration** (*float*) -- 允许的最大线性工具加速度 [mm/s²]。
- **blend_radius** (*float*) -- 段开始处允许的混合半径 [mm]。

示例:

姿态段可以按如下方式添加：

```
# 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)
```

在这些示例中，没有指定可选参数。因此，将使用全局参数。参数也可以在函数调用中明确指定。

```
# first pose segment is added
new_pose = create_pose([700.0, -125.0, 500.0, 180.0, 0.0, -90.0], "c8")
path.add_pose(new_pose, linear_velocity=500.0, linear_acceleration=1000.0)

# second pose segment is added
path.add_pose(get_pose("pick_pose"), linear_velocity=250.0,
                    linear_acceleration=500.0, blend_radius=100.0)
```

第一个段定义了朝指定目标姿态的运动，最大速度为 500 mm/s，最大加速度为 1000 mm/s²。请注意，这个姿态不会被完全达到，因为后续段定义了 100 mm 的混合半径，允许偏离相应坐标以确保更平滑、更高效的过渡。第二段则添加了朝下一个姿态的运动，最大速度为 250 mm/s，最大加速度为 500 mm/s²。

class Socket

Socket 是由 `create_tcp_client` 、 `create_tcp_server` 和 `create_udp_socket` 函数返回的对象。使用此对象通过套接字发送和接收数据。

receive(*timeout=None*)

通过此套接字接收消息。

参数:

timeout (*None or float*) -- 等待消息的最长时间。如果未指定，则无限等待。

返回:

接收到的消息，如果发生超时则为一个空字符串。

返回类型:

str

示例:

等待 3 秒以接收答案，并在接收时打印。

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

send(*message*)

通过此套接字发送消息。

参数:

message (*str*) -- 要发送的消息。

示例:

向所有已连接的端发送消息。

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

close()

关闭套接字连接。

示例:

关闭套接字。

```
socket.close()
```

插件功能

Cognex 插件功能

class Cognex

Cognex 模块支持使用 Cognex 相机。请注意，配置 Cognex 相机需要安装 Cognex In-Sight Explorer。

有关如何正确设置并将 Cognex 相机连接到机器人，请参阅用户手册。

此插件的所有功能均使用 `cognex.` 前缀调用。

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

设置 TCP 客户端，通过 ISE 端口连接已连接的 Cognex 相机以发送本地命令。

参数:

- **ip** (*str*) -- 相机的 IP 地址，例如 `'10.0.0.210'`。
- **user** (*str*) -- 登录相机的用户名（如在 Cognex In-Sight Explorer 中先前设置）。默认用户名为 `'admin'`。
- **password** (*str*) -- 登录相机的密码（如在 Cognex In-Sight Explorer 中先前设置）。默认密码为空字符串。

抛出:

MinorError -- 如果凭证错误，连接过程中被中止，或连接耗时过长。

示例:

连接到刚交付的相机，使用默认 IP 和凭据。

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

连接到具有先前修改的 IP 地址和凭据的相机。

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

trigger_camera(*ip*)

设置相机触发事件（"SW8"），拍摄新图像并返回由当前 Cognex 任务处理的数据。

参数:

ip (*str*) -- 相机的 IP 地址，例如 `'10.0.0.210'` 。

返回:

来自相机的响应消息，包含检测到的实体。

返回类型:

str

change_job(*ip*, *job_name*)

通过原生相机命令 "LF" 更改 Cognex 任务，该命令会加载已存储在相机上的任务文件。请注意，此命令可能需要几秒钟才能执行完成。

参数:

- **ip** (*str*) -- 相机的 IP 地址，例如 `'10.0.0.210'`。
- **job_name** (*str*) -- 要切换的任务名称（如在 Cognex In-Sight Explorer 中先前设置）。

抛出:

MinorError -- 如果相机不存在，或更改任务失败。

示例:

切换到之前存储的任务 'CircleDetection.job'，然后拍摄图像以检测当前相机视图中的圆。

```
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*)

触发相机上的事件，并请求 Cognex 端的特定关键字套接字消息。

参数:

- **ip** (*str*) -- 相机的 IP 地址，例如 `'10.0.0.210'`。
- **event_id** (*int*) -- 要触发的 Cognex 事件的 ID。
- **keyword** (*str*) -- 要等待的关键字。
- **timeout** (*float or None*) -- 等待请求的超时时间（单位：[s]）。使用 `None` 表示无限等待。

抛出:

TimeOutError -- 如果相机在超时之前未响应有效字符串。

示例:

请求包含字符串 'my_message' 的消息，该消息与 Cognex In-Sight 任务的事件 1 (SW1) 相关联。

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

disconnect_camera(*ip*)

断开特定相机的连接。

参数:

ip (*str*) -- 相机的 IP 地址，例如 `'10.0.0.210'` 。

Modbus 插件功能

class Modbus

Modbus 模块包含用于向外部 REST 服务器发送请求的功能。

此插件的所有功能都使用 `modbus.` 前缀调用。

write_user_register(*address*, *values*)

写入 Modbus 输出寄存器。

参数:

- **address** (*int*) -- 开始写入的寄存器地址。可选值范围为 0x10 (16) 到 0xFF (255)。
- **values** (*int or list of int*) -- 要写入的值。每个寄存器最多可存储 2 字节数据。因此最大值为 65535 (=2^{16}-1)。如果提供多个值，这些值将写入连续寄存器。

示例:

将单个值写入 Modbus 输出寄存器的特定地址。

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

使用单个命令将多个值写入连续的输出寄存器。

```
# write 555 to the address 0x30, and 777 to the address 0x31
modbus.write_user_register(0x30, [555, 777])
```

read_user_register(*address*, *count=1*)

从 Modbus 输入寄存器读取数据。

参数:

- **address** (*int*) -- 开始读取的寄存器地址。可选值范围为 0x10 (16) 到 0xFF (255)。
- **count** (*int*) -- 要读取的连续寄存器数量。默认情况下只读取一个寄存器。

返回:

从寄存器读取的值。列表长度等于 *count* 参数。

返回类型:

list of int

示例:

从 Modbus 输入寄存器的特定地址读取单个值。

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

使用单个命令从连续输入寄存器读取多个值。

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

wait_for_event(*index*)

等待指定索引的事件被触发。事件由 Modbus 客户端从外部设置。

参数:

index (*int*) -- 要等待的事件索引。可用索引为 0 到 15。

示例:

等待事件被触发。

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

REST 插件功能

class Rest

REST 模块包含用于向外部 REST 服务器发送请求的函数。

此插件的所有功能均使用 `rest.` 前缀调用。

setup_client_connection(*target_url*)

设置客户端连接以向 REST 服务器发送请求。

参数:

target_url (*str*) -- 要连接的 REST 服务器的 URL（通用 API 端点）。

返回:

REST 客户端连接对象。

返回类型:

RestClientConnection

示例：

建立与 REST 服务器的连接。

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# ... this is similar to ...
connection = rest.setup_client_connection("https://some_url:5000/endpoint/")
```

class RestClientConnection

RestClientConnection 是 `rest.setup_client_connection` 返回的对象，用于向已连接的 REST 服务器发送请求。

get_default_headers(*method*)

读取每个请求发送的默认头信息。

参数:

method (*str*) -- 要获取头信息的方法（"get" 或 "post"）。

返回:

此请求方法当前设置的默认头信息。

返回类型:

dict

示例：

读取 GET 和 POST 方法的默认头信息。

```
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*)

更改每个请求发送的默认头信息。

参数:

- **method** (*str*) -- 要设置头信息的方法（"get" 或 "post"）。
- **new_headers** (*dict*) -- 要覆盖的新默认头信息。

示例:

建立与 REST 服务器的连接。

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
# resetting the headers of the GET method
connection.set_default_headers("get", {"Accept": "application/json"})
# resetting the headers of the POST method
connection.set_default_headers("post", {"Accept": "application/json", "Content-Type":
"application/json"})
```

get(*resource*, *headers_override=None*, *query_params=None*, *secure_connection=True*)

向 REST 服务器发送 GET 请求。

参数:

- **resource** -- 要访问的资源名称。资源名称会添加到 URL (<url>/<resource>)，并可选择性地包含前导 '/'。
- **headers_override** (*dict*) -- 为此单个请求设置头信息。如果未指定，则使用默认头信息。
- **query_params** (*dict*) -- 指定查询（如果有）。
- **secure_connection** (*bool*) -- 是否使用安全连接（验证证书）。

返回:

响应代码（HTTP 代码，例如 `200` 表示请求成功）和响应负载（响应数据）（如果有）。

返回类型:

(int, dict)

示例：

向 REST 服务器发送 GET 请求并检查响应代码。

```
connection = rest.setup_client_connection("https://some_url:5000/endpoint")
code, payload = connection.get("my_resource")  # or connection.get("/my_resource")
# print received data if request was successful:
if code == 200:
    print(payload)
```

post(*resource*, *data*, *headers_override=None*, *query_params=None*, *secure_connection=True*)

向 REST 服务器发送 POST 请求。

参数:

- **resource** (*str*) -- 要访问的资源名称。资源名称会添加到 URL (<url>/<resource>)，并可选择性地包含前导 '/'。
- **data** (*str or dict*) -- POST 请求的主体/数据。
- **headers_override** (*dict*) -- 为此单个请求设置头信息。如果未指定，则使用默认头信息。
- **query_params** (*dict*) -- 指定查询（如果有）。
- **secure_connection** (*bool*) -- 是否使用安全连接（验证证书）。

返回:

响应代码（HTTP 代码，例如 `200` 表示请求成功）和响应负载（响应数据）（如果有）。

返回类型:

(int, dict)

示例：

向 REST 服务器发送 POST 请求并检查响应代码。

```
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 插件功能

class ROSHandler

ROS 模块为其他插件和用户直接访问添加 ROS 功能。

此插件的所有功能均使用 `ros_handler.` 前缀调用。

ros_node()

ROS 模块创建自己的 ROS 节点，可以通过此函数直接访问。可用于在脚本中创建发布者、订阅者等对象。同时，`rclpy` 也可以直接在应用程序中导入以实现此目的。

> **警告**
>
> 不要在此节点上运行 `rclpy.spin()`，因为它已经在后台运行！

示例：

打印机器人正在发布的所有主题的名称。

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

返回:

ROS 模块的 ROS 节点。

返回类型:

rclpy.node.Node