1. Introducción:

Ejemplo de mover brazo robótico:

```
from smart_grasping_sandbox.smart_grasper import SmartGrasper
from tf.transformations import quaternion_from_euler
from math import pi
import time

sgs = SmartGrasper()

sgs.pick()

sgs.reset_world()
```

2- Fundamentos de Python:

El comando `cd ~/catkin_ws/src/` cambia el directorio actual en la terminal al subdirectorio `src` dentro del espacio de trabajo de Catkin ubicado en tu directorio de inicio.

Hacemos uso de 2 códigos distintos:

robot_contol_class.py:

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
import time


class RobotControl():

    def __init__(self, robot_name="turtlebot"):
        rospy.init_node('robot_control_node', anonymous=True)

        if robot_name == "summit":
            rospy.loginfo("Robot Summit...")
            cmd_vel_topic = "/summit_xl_control/cmd_vel"
            # We check that sensors are working
            self._check_summit_laser_ready()
        else:
            rospy.loginfo("Robot Turtlebot...")
            cmd_vel_topic='/cmd_vel'
            self._check_laser_ready()

        # We start the publisher
        self.vel_publisher = rospy.Publisher(cmd_vel_topic, Twist, queue_size=1)
        self.cmd = Twist()
```

```
        self.laser_subscriber = rospy.Subscriber(
            '/kobuki/laser/scan', LaserScan, self.laser_callback)
        self.summit_laser_subscriber = rospy.Subscriber(
            '/hokuyo_base/scan', LaserScan, self.summit_laser_callback)

        self.ctrl_c = False
        self.rate = rospy.Rate(1)
        rospy.on_shutdown(self.shutdownhook)


    def _check_summit_laser_ready(self):
        self.summit_laser_msg = None
        rospy.loginfo("Checking Summit Laser...")
        while self.summit_laser_msg is None and not rospy.is_shutdown():
            try:
                self.summit_laser_msg = rospy.wait_for_message("/hokuyo_base/scan", LaserScan,
timeout=1.0)
                rospy.logdebug("Current /hokuyo_base/scan READY=>" + str(self.summit_laser_msg))

            except:
                rospy.logerr("Current /hokuyo_base/scan not ready yet, retrying for getting scan")
        rospy.loginfo("Checking Summit Laser...DONE")
        return self.summit_laser_msg

    def _check_laser_ready(self):
        self.laser_msg = None
        rospy.loginfo("Checking Laser...")
        while self.laser_msg is None and not rospy.is_shutdown():
            try:
                self.laser_msg = rospy.wait_for_message("/kobuki/laser/scan", LaserScan,
timeout=1.0)
                rospy.logdebug("Current /kobuki/laser/scan READY=>" + str(self.laser_msg))

            except:
                rospy.logerr("Current /kobuki/laser/scan not ready yet, retrying for getting scan")
        rospy.loginfo("Checking Laser...DONE")
        return self.laser_msg

    def publish_once_in_cmd_vel(self):
        """
        This is because publishing in topics sometimes fails the first time you publish.
        In continuous publishing systems, this is no big deal, but in systems that publish only
        once, it IS very important.
        """
        while not self.ctrl_c:
            connections = self.vel_publisher.get_num_connections()
            if connections > 0:
                self.vel_publisher.publish(self.cmd)
                #rospy.loginfo("Cmd Published")
                break
            else:
                self.rate.sleep()

    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def laser_callback(self, msg):
        self.laser_msg = msg

    def summit_laser_callback(self, msg):
        self.summit_laser_msg = msg
```

```
    def get_laser(self, pos):
        time.sleep(1)
        return self.laser_msg.ranges[pos]

    def get_laser_summit(self, pos):
        time.sleep(1)
        return self.summit_laser_msg.ranges[pos]

    def get_front_laser(self):
        time.sleep(1)
        return self.laser_msg.ranges[360]

    def get_laser_full(self):
        time.sleep(1)
        return self.laser_msg.ranges

    def stop_robot(self):
        #rospy.loginfo("shutdown time! Stop the robot")
        self.cmd.linear.x = 0.0
        self.cmd.angular.z = 0.0
        self.publish_once_in_cmd_vel()

    def move_straight(self):

        # Initilize velocities
        self.cmd.linear.x = 0.5
        self.cmd.linear.y = 0
        self.cmd.linear.z = 0
        self.cmd.angular.x = 0
        self.cmd.angular.y = 0
        self.cmd.angular.z = 0

        # Publish the velocity
        self.publish_once_in_cmd_vel()

    def move_straight_time(self, motion, speed, time):

        # Initilize velocities
        self.cmd.linear.y = 0
        self.cmd.linear.z = 0
        self.cmd.angular.x = 0
        self.cmd.angular.y = 0
        self.cmd.angular.z = 0

        if motion == "forward":
            self.cmd.linear.x = speed
        elif motion == "backward":
            self.cmd.linear.x = - speed

        i = 0
        # loop to publish the velocity estimate, current_distance = velocity * (t1 - t0)
        while (i <= time):

            # Publish the velocity
            self.vel_publisher.publish(self.cmd)
            i += 1
            self.rate.sleep()

        # set velocity to zero to stop the robot
        self.stop_robot()

        s = "Moved robot " + motion + " for " + str(time) + " seconds at " + str(speed) + "  m/s"
        return s
```

```
    def turn(self, clockwise, speed, time):

        # Initilize velocities
        self.cmd.linear.x = 0
        self.cmd.linear.y = 0
        self.cmd.linear.z = 0
        self.cmd.angular.x = 0
        self.cmd.angular.y = 0

        if clockwise == "clockwise":
            self.cmd.angular.z = -speed
        else:
            self.cmd.angular.z = speed

        i = 0
        # loop to publish the velocity estimate, current_distance = velocity * (t1 - t0)
        while (i <= time):

            # Publish the velocity
            self.vel_publisher.publish(self.cmd)
            i += 1
            self.rate.sleep()

        # set velocity to zero to stop the robot
        self.stop_robot()

        s = "Turned robot " + clockwise + " for " + str(time) + " seconds at " + str(speed) + "
radians/second"
        return s


if __name__ == '__main__':

    robotcontrol_object = RobotControl()
    try:
        robotcontrol_object.move_straight()

    except rospy.ROSInterruptException:
        pass
```

pyscript1.py:

```
from robot_control_class import RobotControl

rc = RobotControl()

a = rc.get_laser(360)

print ("The distance measured is: ", a, " m.")
```

Explicación:

`from robot_control_class import RobotControl`

Allí, estamos importando una clase Python llamada RobotControl. Esta clase es la que hemos creado en Robot ignite Academy para ayudarle a interactuar con el robot ROS sin tener que utilizar realmente ROS.

Funciona así: `from <name_of_the_module> import <method_or_class_to_be_imported>`

```
a = rc.get_laser(360)
```

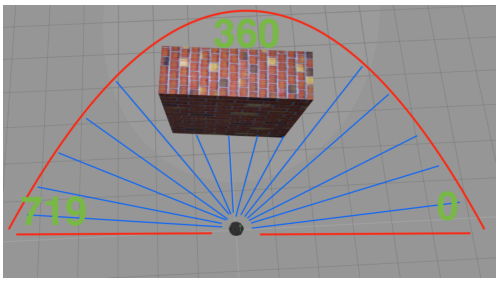
Tres cosas nuevas aquí:

- Estamos llamando al método `get_laser()` , que proporciona la RobotControl clase
- Estamos proporcionando un parámetro al método (el número 360).
- A continuación, almacenamos la salida del método en la nueva variable a.

¿Qué es este `get_laser()` método?

- El método **`get_laser (ray_number)`**: este método le permite obtener datos del láser del robot. Cuando llame a este método, le devolverá la distancia medida por el rayo láser que especifique como parámetro.
- El parámetro **`ray_number`**: aquí es donde se especifica un número entre 0 y 719, que indicará el rayo de la lectura láser del que desea obtener la distancia medida.

El 360 es porque la pared se encuentra en frente del TurtleBot, no al lado, y se mandan muchos rayos a la vez (son 720, del 0 al 719:



2. Variables:

Una variable puede verse como un contenedor que almacena algún dato: puede ser un número, un texto o tipos de datos más complejos.

En la mayoría de los lenguajes de programación, como C++ o Java, es necesario **declarar una variable** antes de poder utilizarla. Pero en Python, esto es mucho más fácil. **No hay necesidad de declarar una** variable en Python. Si quiere utilizar una variable, ¡sólo tiene que pensar en un nombre y empezar a asignarle un valor! Puede ver esto en el siguiente fragmento de código.

```
a = 5
b = 'This is a string'
c = ['This', 'is', 'a', 'list', 1, 2, 3]
```

Tipos de datos:

1. **Números**: int y float
2. **Cadenas**: Las cadenas son básicamente secuencias de caracteres. Podemos utilizar comillas simples (`' '`) o comillas dobles (`" "`) para representar cadenas:

```
s1 = 'This is a string'

s2 = "This is also a string"

print (s1[0])
print (s2[8])
```

Saldría como en C++: T y a
Se pueden sumar:

```
s1 = 'This is'

s2 = " a string"

s3 = s1 + s2

print (s3)
```

Saldría: This is a string

3. **Listas:** Una Lista Python es una secuencia ordenada de elementos, en la que no es necesario que todos los elementos de la lista sean del mismo tipo. Para declarar una Lista Python, basta con poner todos los elementos entre corchetes [ ], y separar cada elemento mediante comas.

```
l = [1, 2, 3, 'This', 'is', 'a', 'list']

print (l[2])
print (l[0:3])
print (l[3:])
```

- En la 1ª impresión, estamos obteniendo el 3er elemento de la lista (recuerde que el 1er elemento es la posición 0).
- En la 2ª impresión, obtenemos todos los artículos entre el 1º y el 3º.
- En la 3ª impresión, estamos recibiendo todos los artículos desde el 4º hasta el último.

4. **Tuplas:** Las tuplas constan de una serie de valores separados por comas, que se encierran entre paréntesis `( )`. La particularidad de las tuplas es que no pueden actualizarse, son **de sólo lectura**. Así que, básicamente, son iguales que las listas, con la única diferencia de que los valores dentro de una tupla no pueden actualizarse.
5. **Diccionarios:** Los diccionarios también son similares a las listas, en el sentido de que contienen una lista de valores y que pueden actualizarse. Pero la principal diferencia es que a los elementos de los diccionarios se accede mediante claves y no mediante su posición. Así que, básicamente, los diccionarios son una lista de elementos, en la que cada elemento es un par formado por una clave y un valor:

```
dict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}

dict["Jon"] = 10
print (dict["Jon"])
```

Con todos estos nuevos tipos de datos introducidos, ¡vamos a hacer un par de ejercicios para ponerlos en práctica! Pero, para los siguientes ejercicios, vamos a introducir un nuevo método que podrá llamar en sus programas:

- `get_laser_full()`: Como su propio nombre indica, este método le permitirá obtener todos los datos de TODOS los rayos láser del robot. Como he dicho antes, se trata de un total de 720 lecturas diferentes. Así que, cuando sea llamado, este método devolverá una **LISTA** que contendrá todas las 720 lecturas diferentes de los rayos láser.

Ejercicios:

```
from robot_control_class import RobotControl

rc = RobotControl()

l = rc.get_laser_full()

print ("Position 0: ", l[0])
print ("Position 360: ", l[360])
print ("Position 719: ", l[719])
```

```
from robot_control_class import RobotControl

rc = RobotControl()

l = rc.get_laser_full()

dict = {"P0": l[0], "P100": l[100], "P200": l[200], "P300": l[300], "P400":
l[400], "P500": l[500], "P600": l[600], "P719": l[719]}

print (dict)
```

Entradas y salidas:

```
a = 5

print ("Simple print")
print ("Now we print the variable a = " + str(a))
print ("Now we print the variable a = %d" % a)
print ("This is an example of a \n new line")
```

- La 1ª impresión es sencilla, como puede ver.
- La segunda impresión combina una impresión normal con una variable. Observe que estamos convirtiendo la variable en un valor de cadena.
- La tercera impresión también combina una impresión normal con una variable. Observe que en este caso no necesitamos convertir la variable en un valor de cadena, porque estamos utilizando el formato `%`.
- La 4ª impresión muestra un ejemplo de cómo saltar a una nueva línea utilizando la `/n` expresión.

Como ya hemos visto, la salida en este caso será el `print()` mientras que la entrada la realizará el método `input()` método. Eche un vistazo al siguiente ejemplo:

```
age = int(input("What's your age? ")) #tenerlo en cuenta porque si no se toma como cadena

age2 = age + 1

print("So next year you will be %d years old!" % age2)
```

Operadores:

Operadores aritméticos:

Operador	Nombre	Ejemplo
+	Adición	1 + 1 = 2
-	Substracción	2 - 1 = 1
*	Multiplicación	2 * 2 = 4
/	División	5 / 2 = 2
%	Módulo	5 % 2 = 1

Operadores de asignación:

Operador	Ejemplo	Igual que
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Operadores de comparación:

Operador	Significa	Igual que
==	Igual	5 == 5
!=	No es igual	4 != 5
>	Mayor que	5 > 4
<	Menos de	4 < 5
>=	Mayor o igual que	5 >= 4
<=	Inferior o igual a	4 <= 5

3. Sentencias condicionales y bucles:

Como ya habrá notado, las sentencias que están dentro de la condición están **indentadas**. Estos bloques de código sangrados se ejecutan sólo si la "condición" se evalúa como Verdadera.
else if pasa a ser elif

```
movie = input("What's your favorite movie? ")

if movie == "Avengers Endgame" or movie == "Titanic":
    print("Good choice!")
elif movie == "Star Wars":
    print("Also a good choice!")
else:
    print("You really are an interesting specimen")
```

Dos nuevos métodos que deberá utilizar para completar el siguiente ejercicio:

- `move_straight()`: Como su nombre indica, este método le permitirá empezar a mover el robot en línea recta.
- `stop_robot()`: Como su nombre indica, este método le permitirá detener el movimiento del robot.

```
from robot_control_class import RobotControl

robotcontrol = RobotControl()

a = robotcontrol.get_laser(360)

if a < 1:
    robotcontrol.stop_robot()

else:
    robotcontrol.move_straight()

print ("The laser value received was: ", a)
```

Los while funcionan igual que en C++:

```
from robot_control_class import RobotControl

robotcontrol = RobotControl()

a = robotcontrol.get_laser(360)

while a > 1:
    robotcontrol.move_straight()
```

```
    a = robotcontrol.get_laser(360)
    print ("Current distance to wall: %f" % a)

robotcontrol.stop_robot()

print ("Wall is at %f meters! Stop the robot!" % a)
```

Los bucles for son un poco más distintos:

`for` Los bucles se utilizan para iterar sobre una secuencia. Recorre los elementos de listas, cadenas, las claves de diccionarios y otros iterables. El bucle for Python comienza con la palabra clave **`for`**seguida de un nombre de variable arbitrario, que contendrá los valores del siguiente objeto secuencial recorrido. La sintaxis general es la siguiente:

```
for variable in sequence:
    statement
```

Los elementos del objeto secuencia se asignan uno tras otro a la variable del bucle. A continuación, para cada elemento de la secuencia, se ejecuta el cuerpo del bucle:

```
names = ["Yoda", "ObiWan", "Anakin", "Palpatine"]

for x in names:
    print(x)
```

Más ejemplos:

```
from robot_control_class import RobotControl

robotcontrol = RobotControl()

l = robotcontrol.get_laser_full()

maxim = 0

for value in l:
    if value > maxim:
        maxim = value

print ("The higher value in the list is: ", maxim)
```

Es muy común utilizar la **`for`** junto con el método **`range()`**:

```
for i in range(5):
    print (i)
```

En cada ejemplo que ha visto hasta ahora, el cuerpo completo del bucle se ejecuta en cada iteración. Python proporciona dos palabras clave que terminan una iteración de bucle prematuramente:

- `break`: Finaliza inmediatamente un bucle por completo. La ejecución del programa procede a la primera sentencia que sigue al cuerpo del bucle.
- `continue`: Termina inmediatamente la iteración actual del bucle. La ejecución salta a la parte superior del bucle, y la condición se vuelve a evaluar para determinar si el bucle se ejecutará de nuevo o terminará.

```
counter = 0

while counter < 10:
    counter += 1
    if counter == 3:
        break
    print (counter)

print ("Outside the loop!")
```

dfvd

4. Métodos (funciones):

Un método es un elemento estructurador en los lenguajes de programación, utilizado para agrupar un conjunto de sentencias de forma que puedan utilizarse más de una vez en un programa. La única forma de conseguirlo sin métodos sería reutilizar el código copiándolo y adaptándolo a un contexto diferente. El uso de métodos suele mejorar la comprensibilidad y la calidad del programa. También reduce el coste de desarrollo y mantenimiento del software.

- Definición:

```
def mymethod():
            print ("Python")
```

- LLamarlo:

```
def mymethod():
            print ("The method mymethod() has been called")

mymethod()
```

- Parámetros del método: Los parámetros se especifican después del nombre del método, dentro de los paréntesis. Puede añadir tantos parámetros como desee, sólo tiene que separarlos con una coma.

```
def add(a,b):
    res = a + b
    print (res)
```

```
add(2,2)
```

Los parámetros opcionales (normalmente conocidos como **parámetros por defecto**) deben seguir a los parámetros obligatorios, por ejemplo si a=2 y b=2 y llamar a add().

a) Cree un método que, dado un número entero, haga que el robot se mueva en línea recta durante esa cantidad de tiempo. Por ejemplo, si el número dado es un 5, el robot se moverá recto durante 5 segundos.

NOTA: Para este ejercicio, puede utilizar el método Python sleep() de Python. Para utilizarlo, sólo tiene que importar el módulo time y luego llamar al método sleep() método, de esta forma:

```
from robot_control_class import RobotControl
import time

robotcontrol = RobotControl(robot_name="summit")

def move_x_seconds(secs):
    robotcontrol.move_straight()
    time.sleep(secs)
    robotcontrol.stop_robot()


move_x_seconds(5)
```

- Declaración de retorno:

```
from robot_control_class import RobotControl

robotcontrol = RobotControl(robot_name="summit")

def get_laser_values(a,b,c):
        r1 = robotcontrol.get_laser_summit(a)
        r2 = robotcontrol.get_laser_summit(b)
        r3 = robotcontrol.get_laser_summit(c)

        return [r1, r2, r3]

l = get_laser_values(0, 500, 1000)

print ("Reading 1: ", l[0])
print ("Reading 2: ", l[1])
print ("Reading 3: ", l[2])
```

- Variables globales de manera explícita:

```
    def f():
      global gv
  gv = "Global Variable"

gv = "Empty"
print (gv)
f()
print(gv)
```

Para los siguientes ejercicios, vamos a introducir dos nuevos métodos que se encuentran en `robot_control_class.py`:

- `move_straight_time (motion, speed, time)`: Como su propio nombre indica, este método le permitirá mover el robot en línea recta. Necesitará pasarle tres parámetros.
  - `motion`: Especifique aquí si desea que su robot se mueva hacia delante (`"forward"`) o hacia atrás (`"backward"`).
  - `speed`: Especifique aquí la velocidad a la que desea que se mueva su robot (en m/s).
  - `time`: Especifique aquí el tiempo que desea que su robot se mantenga en movimiento (en segundos).
- `turn (clockwise, speed, time)`: Como su propio nombre indica, este método le permitirá girar el robot. Deberá pasarle tres parámetros.
  - `clockwise`: Especifique aquí si desea que su robot gire en el sentido de las agujas del reloj (`"clockwise"`) o en sentido contrario a las agujas del reloj (`"counter-clockwise"`).
  - `speed`: Especifique aquí la velocidad a la que desea que gire su robot (en m/s).
  - `time`: Especifique aquí el tiempo que desea que su robot siga girando (en segundos).

Ambos métodos **devolverán una cadena**, indicando el movimiento que han realizado:

```
from robot_control_class import RobotControl

rc = RobotControl(robot_name="summit")

rc.move_straight_time ("forward", 0.3, 5)
a = rc.turn("clockwise",0.3,7)
print(a)
```

5. Clases y Programación Orientada a Objetos:

La programación orientada a objetos se basa en el concepto de "objetos". Estos objetos suelen estar definidos por dos cosas:

- Atributos: Son los datos asociados al objeto, definidos en campos.
- Métodos: Son los procedimientos o métodos asociados al objeto.

La característica más importante de los objetos es que los métodos asociados a ellos pueden acceder a los campos de datos del objeto al que están asociados y modificarlos.

Una clase Python es, básicamente, una plantilla de código para crear un objeto.

Por ejemplo, la clase *Jedi* que aparece a continuación describe cómo crear un objeto de tipo Jedi, de modo que puede utilizar ese objeto de ese tipo para almacenar y manipular datos de la forma que define la clase.
Por ejemplo, si mi código ha creado la clase *Jedi*, entonces puedo crear un objeto de ese tipo haciendo:

```
my_object = Jedi("Qui-Gong-Jin")
```

Cuando creamos una variable de la clase `X`, decimos que hemos **creado una** instancia de la clase **`X`. En el código, arriba hemos creado una instancia de la clase** `Jedi`.

- Definición de una clase en Python:

```
class Jedi:
    def __init__(self, name):
        self.jedi_name = name

    def say_hi(self):
        print('Hello, my name is ', self.jedi_name)

j1 = Jedi('ObiWan')
j1.say_hi()

j2 = Jedi('Anakin')
j2.say_hi()
```

El código anterior tiene dos partes:

1. Donde definimos la clase
2. Donde utilizamos la definición para crear y utilizar variables de tipo *`Jedi`* (*`j1`* y *`j2`*)

Observe que cada uno de los métodos contiene la **`self`** como parámetro. Esa palabra clave indica que **los métodos pertenecen a la clase**. Incluir la *`self`* es obligatoria para cada método de la clase. (es como el this en C++)

- El constructor de clase:

```
def __init__(self, name):
    self.jedi_name = name
```

Para cualquier clase que cree (la `Jedi` clase `RobotControl` clase, o cualquier otra que pueda inventar), el método llamado `__init__` se denomina constructor de la clase.

¿Qué tiene de especial el constructor? El constructor **es llamado automáticamente (por el sistema Python) cada vez que se crea una nueva instancia de la clase**.

El constructor de una clase es un método muy importante porque es el que se utiliza para inicializar la instancia de la clase con los valores adecuados, basándose en los parámetros proporcionados. Por ejemplo, en el código anterior:

```
j1 = Jedi('ObiWan')
```

Estamos creando una instancia de la clase `Jedi`proporcionándole el parámetro `ObiWan`. Que es utilizado por el `__init__` método de la `Jedi` clase (el constructor de la `Jedi` clase) para inicializar esa `self.jedi_name` variable. Eso es lo único que hacemos en el constructor, pero podríamos añadir cualquier otra cosa necesaria para tener la `Jedi` instancia correctamente configurada.

IMPORTANTE: Para crear una variable de una clase, siempre tiene que empezar por el prefijo `self.` y a continuación el nombre de la variable. RECUERDE añadir ese `self.` prefijo en cualquier lugar de la clase.

¡Las variables de la clase son como variables *globales* para esa clase! ¿Qué significa esto? Significa que las variables se pueden utilizar en cualquier método que pertenezca a la clase, como cuando utilizábamos la `global` palabra clave en la Unidad 4.

IMPORTANTE 2: Este es uno de los puntos principales del uso de clases. No tendrá que utilizar variables globales a las que se pueda acceder en cualquier parte del código, contribuyendo a que se forme un lío. Al **encapsular** las variables dentro de la clase, sólo los métodos de la clase podrán acceder a esas variables, por lo que tendrá menos posibilidades de liarla.

¡Los métodos de la clase pueden modificar cualquier variable de la clase! Pueden acceder a esos valores para leerlos o incluso para modificarlos.

Cree un programa Python que haga que el robot realice 2 cuadrados. El segundo tendrá el doble de tamaño que el primero:

from robot_control_class import RobotControl

```
class MoveRobot:
    def __init__(self, motion, clockwise, speed, time):
        self.robotcontrol = RobotControl(robot_name="summit")
        self.motion = motion
        self.clockwise = clockwise
        self.speed = speed
```

```
        self.time = time
        self.time_turn = 7.0 # This is an estimate time in which the robot
will rotate 90 degrees

    def do_square(self):

        i = 0

        while (i < 4):
            self.move_straight()
            self.turn()
            i+=1

    def move_straight(self):
        self.robotcontrol.move_straight_time(self.motion, self.speed,
self.time)

    def turn(self):
        self.robotcontrol.turn(self.clockwise, self.speed, self.time_turn)


mr1 = MoveRobot('forward', 'clockwise', 0.3, 4)
mr1.do_square()
mr2 = MoveRobot('forward', 'clockwise', 0.3, 8)
mr2.do_square()
```