Manual de programación con robots para la escuela

Guía de ejercicios para el robot Multiplo N6 + Python + Lihuen GNU/Linux

Javier Díaz
Claudia Banchoff
Sofía Martin
Joaquín Bogado
Damian Mel
Fernando López
Equipo de soporte y desarrollo de Lihuen GNU/Linux

Versión del manual 1.0

http://lihuen.linti.unlp.edu.ar | soportelihuen@linti.unlp.edu.ar

Manual de programación con robots para la escuela.
Claudia Banchoff, Joaquín Bogado, Damián Mel,
Sofía Martin, Fernando López.
Agosto 2012 - Versión 0.1.5
e-mail: soportelihuen@linti.unlp.edu.ar
web: http://robots.linti.unlp.edu.ar

Las imágenes que ilustran los capítulos son de dominio público y fueron obtenidas en los sitios

Agradecimientos:.
A Ariadna Alfano por el diseño gráfico.
A Andrea Gomez del Mónaco por la tabla con los comandos.
A Pablo Santamaría por la ayuda en LaTeX.

Índice general

Prefacio

Sobre este libro

Los jóvenes de hoy se socializan, se comunican e interactúan mediados por las nuevas tecnologías. Dichas tecnologías cumplen un papel fundamental en los actuales procesos de enseñanza/aprendizaje.

El aprendizaje a través de experiencias "del mundo real", mediadas por tecnología, puede aportar al desarrollo de individuos autónomos, críticos, creativos, capaces de resolver problemas, buscar alternativas, probar distintos caminos, etc. Este aprender en lo "real" les permite ver cómo se dan determinados procesos con los condicionantes propios de cada contexto, ensayando a través de prueba y error.

Introducir a docentes y jóvenes al mundo de la programación no sólo apunta a un aprendizaje técnico, sino que mediante la misma, se desarrollan una serie de habilidades, como el pensamiento analítico o de solución de problemas, que son muy requeridos en los trabajos que tienen que ver con tecnología y ciencia, pero que también se pueden aplicar a otras áreas.

El objetivo de este libro es introducir los primeros conceptos de la programación a través de la manipulación de unos pequeños robots, implementando algoritmos escritos en el lenguaje Python. Tanto Python, como las herramientas de desarrollo que usaremos son software libre y muy utilizados en entornos de desarrollo reales.

En este libro, concentramos las experiencias de dos años de trabajo con alumnos de escuelas secundarias en donde se promueven actividades que permiten trabajar colaborativamente con otras áreas y que son motivadoras tanto para los alumnos como para los docentes que las promueven.

Nos hemos inspirado en las experiencias presentadas el año 2008 por docentes del Instituto de Tecnología de Georgia a través de la iniciativa "Institute for Personal Robots for Education[1]" (IPRE).

Este libro, está publicado bajo licencia Creative Commons y puede ser utilizado, compartido y modificado tanto como sea necesario. Pueden descargarlo del sitio http://robots.linti.unlp.edu.ar/, en donde, encontrarán, también los fuentes escritos en Latex y los ejemplos mencionados.

¿A quiénes está dirigido el libro?

Si bien hemos planteado actividades pensadas para trabajarlas en la escuela, no es una condición necesaria ni obligatoria. Los robots que utilizamos se pueden adquirir en nuestro país a un costo razonable y todos aquellos que quieran iniciarse en el mundo de la programación pueden hacerlo a través de esta iniciativa.

No necesitamos contar con ningún conocimiento inicial sobre programación, dado que, justamente aquí se introducen. Próximamente, publicaremos una segunda parte, donde trabajaremos en el desarrollo de juegos sencillos, también utilizando Python.

[1] http://www.roboteducation.org/

Estructura del libro

En el capítulo 1, vamos a hablar sobre el **Software Libre** y por qué creemos que es importante utilizarlo y difundirlo, especialmente, en el ámbito educativo. Veremos por qué pensamos que es importante la enseñanza de la programación y por qué elegimos Python como opción.

En el capítulo 2 presentaremos a nuestro robot y nos ocuparemos de instalar y configurar las herramientas necesarias para comenzar a trabajar. Si bien, como veremos más adelante, Python es un lenguaje que puede utilizarse tanto el sistemas Microsoft Windows como GNU/Linux, nosotros enfocaremos las prácticas al uso sobre Linux, con lo cual hemos agregado al final de este libro el apéndice A donde mostramos una guía de cómo instalar aplicaciones en este sistema operativo.

En el capítulo 3 comenzaremos con las primeras actividades. Hemos elegido las más sencillas, donde el foco es familiarizarse con el entorno de trabajo y las funciones básicas del robot.

El capítulo 4 presenta los tipos básicos de valores con los que podemos trabajar en Python y los primeros mecanismos de interación con el usuario.

En los capítulos 5 y 6 daremos algunos conceptos de lógica sencillos que nos permitirán escribir programas utilizando las estructuras condicionales e iterativas del lenguaje.

En el capítulo 7 hacemos una pausa y mostramos formas de mejorar la organización de nuestros programas. Escribiremos módulos que contengan funciones y veremos cómo utilizarlos desde otros programas.

Por último, el capítulo 8, describe los distintos sensores que provee el robot básico y presenta actividades integradoras como culminación del curso.

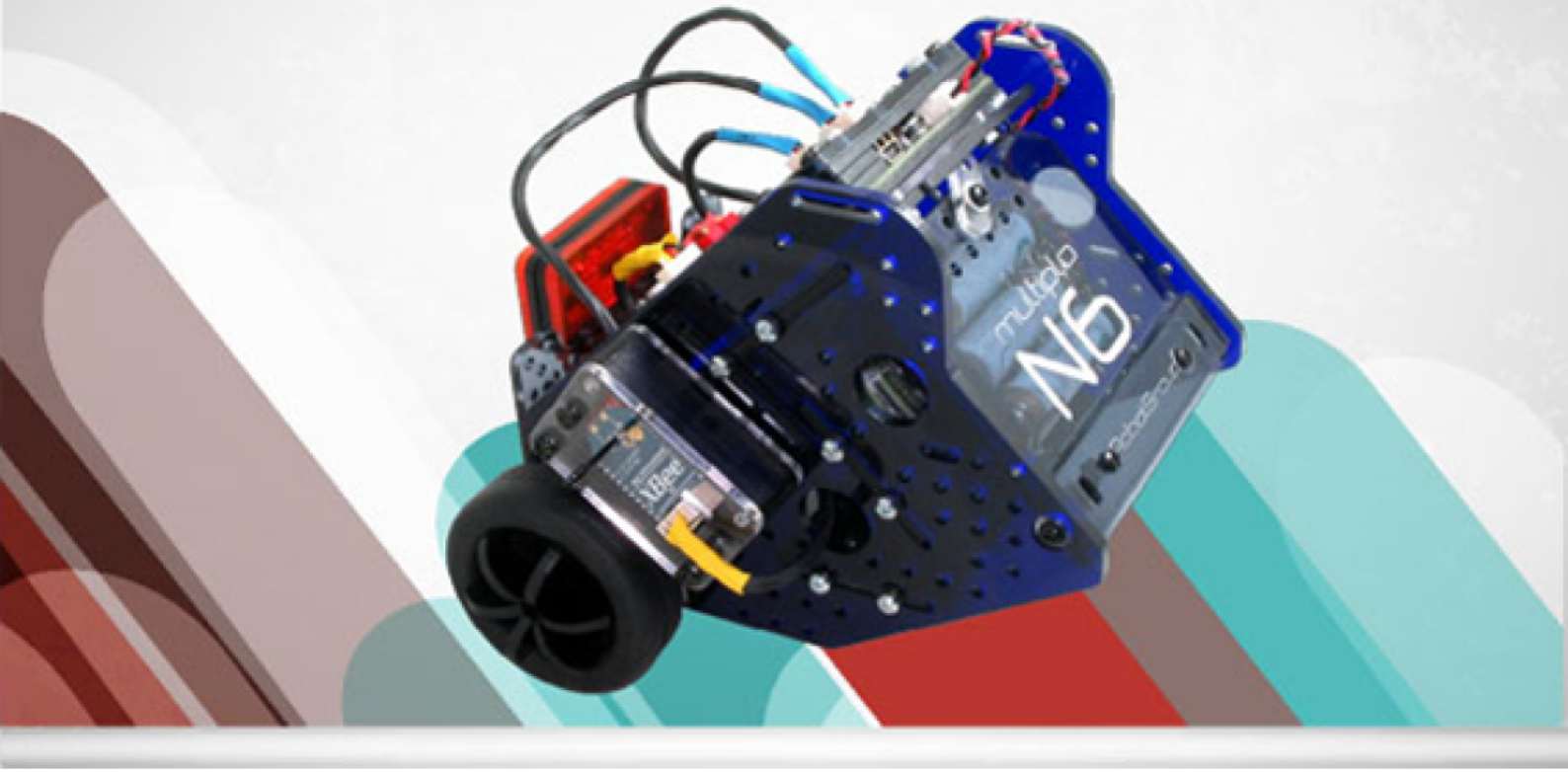
Sobre los autores

Los autores responsables de este libro, pertencecen al Laboratorio de Investigación de Nuevas Tecnologías Informáticas de la Facultad de Informática de la Universidad Nacional de La Plata. Este laboratorio mantiene a instancias del Lic. Javier Díaz,director del **LINTI**[2] desde el año 1994, una línea de trabajo permanente sobre Software Libre. Los proyectos más destacados son **Lihuen GNU/Linux**[3] y **Programando con Robots**[4]. La experiencia contenida en este libro resume la aspiración de este grupo de trabajo cuyo objetivo es llevar a las escuelas públicas el conocimiento sobre Software Libre.

[2] http://linti.unlp.edu.ar
[3] http://lihuen.info.unlp.edu.ar
[4] http://robots.linti.unlp.edu.ar/

Capítulo 1

Trabajar con Software Libre



En este capítulo vamos a hablar sobre el Software Libre y por qué creemos que es importante utilizarlo y difundirlo. Primero vamos a dar algunas definiciones básicas y luego nos introduciremos en las herramientas que utilizaremos a lo largo del libro.

1.1. Software libre en la escuela

El software libre no es una "nueva moda" o una nueva tendencia. Es una manera de pensar la tecnología, su construcción, uso y distribución. Cuando hablamos de software libre, decimos que los usuarios son libres de (les está permitido) utilizar, distribuir, estudiar y modificar el software que poseen. Una definición más precisa de este concepto la podemos encontrar en el sitio web del proyecto GNU[1]:

> *El software libre es una cuestión de la libertad de los usuarios de ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. Más precisamente, significa que los usuarios de programas tienen las cuatro libertades esenciales.*

En todos los casos, es necesario contar con el **código fuente** del software para poder modificarlo y estudiarlo. No se habla de un software que no tiene licencia de uso, sino que se distribuye con "licencias

[1] http://www.gnu.org/

libres", es decir, con licencias que le dan a los usuarios todas las libertades que enuncia su definición. Al tratarse de una comunidad muy amplia, de la que participan una gran diversidad de personas (no sólo programadores sino también usuarios) comienzan a surgir distintas posiciones y, hacia el año 1998, surge un movimiento denominado "código abierto", u "open source" en inglés. El problema es que el término "software libre", en inglés, "free software", trae algunas confusiones, dado que el término "free" también puede interpretarse como "gratis", y el software libre es mucho más que un software gratuito.

Es así que, en esa época, la corriente del software libre se abre en dos ramas: una más orientada a la "definición conceptual y filosófica" y otra, un poco más práctica. Las dos concepciones hacen referencia a casi el mismo tipo de aplicaciones, pero se basan en principios distintos. Richard Stallman, en su artículo *"Por qué el código abierto pierde el punto de vista del software libre"*[2], destaca las diferencias principales.

Para el movimiento del software libre, el software libre es un imperativo ético porque solamente el software libre respeta la libertad del usuario de elegir. En cambio, los seguidores del código abierto consideran los asuntos bajo los términos de cómo hacer «mejor» al software, en un sentido práctico operativo, instrumental solamente. Plantean que el software que no es libre no es una solución óptima. Para el movimiento del software libre, sin embargo, el software que no es libre es un problema social, y la solución es parar de usarlo y migrar al software libre.

Googleame: Software libre vs. código abierto

¿Qué opinan de esta diferencia?

Para evitar estas diferencias, se suele hablar también de **FLOSS**("Free/Libre/Open Source Software"), para designar este tipo de software sin entrar en las discusiones planteadas anteriormente.

A modo de síntesis, podemos decir que nosotros hablaremos de "software libre" porque creemos que es muy importante destacar que la gran ventaja de su adopción, especialmente en ámbitos educativos, se basa en las cuatro libertades de su definición.

Para poder elegir qué aplicación se adapta mejor a nuestras necesidades, es necesario conocer la mayor cantidad posible de opciones. Además, si contamos con acceso a los códigos fuentes, en caso de ser necesario, también podríamos adaptarlos y tener una versión propia del programa.

Por último, cuando usamos software libre, participamos de una gran comunidad dentro de la cual podemos jugar distintos roles, lo que fomenta el trabajo colaborativo.

1.2. ¿Por qué programar?

Introducir aspectos de programación no sólo apunta a un aprendizaje técnico, sino que permite desarrollar una serie de habilidades, como el pensamiento analítico o de solución de problemas, que son muy requeridos en los trabajos que tienen que ver con tecnología y ciencia, pero que también se pueden aplicar a otras áreas.

Estos programas que se ejecutan en nuestras computadoras, celulares, televisores, etc. fueron escritos por programadores en diferentes lenguajes que le indican al dispositivo cómo hacer una acción determinada, es por esto que es importante saber antes que nada qué es un programa:

Un **programa** es un conjunto de instrucciones u órdenes que le indican a la computadora cómo se realiza una tarea.

[2] http://www.gnu.org/philosophy/open-source-misses-the-point.es.html

En este libro, vamos a aprender nociones básicas de programación utilizando unos pequeños robots. Vamos a escribir algoritmos utilizando el lenguaje Python, que "instruirán" a nuestros robots a moverse, evitar obstáculos, etc.

Desde la perspectiva educativa, las características más importantes de estos robots es que los alumnos pueden aprender los conceptos básicos de programación en forma intuitiva y lúdica, explorando sobre instrucciones y sentencias del lenguaje para manipularlos, moverlos y darles órdenes para emitir sonidos, experimentando sus resultados en forma interactiva y mediante la observación directa del robot.

Vamos a utilizar los robots para organizar actividades artísticas (como pintar o bailar), sociales (por ejemplo, una obra de teatro) y lúdicas (carreras de obstáculos o batallas), de manera tal de trabajar con creatividad y en forma colaborativa en el desarrollo de programas.

1.3. Python

La primera pregunta que nos hacen siempre: ¿por qué Python? Es muy probable que conozcan otros lenguajes de programación como Pascal, C, C++, Java, etc. Nosotros elegimos trabajar con Python por varios motivos. En primer lugar, Python es un lenguaje interpretado, lo que simplifica el proceso de programación y uso por parte de personas con escasa experiencia y lo convierte en un lenguaje utilizado de manera extensa para la iniciación a la programación.

Además, Python provee una gran biblioteca de módulos que pueden utilizarse para hacer toda clase de tareas que abarcan desde programación web a manejo de gráficos y, dado que soporta tanto programación procedural como orientada a objetos sirve como base para introducir conceptos importantes de informática como por ejemplo abstracción procedural, estructuras de datos, y programación orientada a objetos, que son aplicables a otros lenguajes como Java o C++.

Googleame: Aplicaciones Python

Seguramente aparecerán varias aplicaciones implementadas en este lenguaje. ¿Reconocen alguna de ellas?

1.4. Resumen

En este capítulo hemos visto que el software libre nos permite **apropiarnos** del software, adaptando las aplicaciones a nuestras usos y costumbres, si es necesario y no en sentido inverso. Muchas veces hemos adaptado actividades en función de las aplicaciones. Si bien esto puede no ser una tarea sencilla, es posible y esto es muy importante.

En el ámbito educativo, y, cuando enseñamos a programar, trabajar con software libre nos da una ventaja adicional ya que también se puede aprender observando programas escritos por otros. Y esto es posible si podemos acceder a los códigos fuentes de los mismos.

1.5. Actividades

Ejercicio 1.1 Busquen el sitio oficial del proyecto [**?**] y revisen si tienen direcciones de contacto para reportar problemas, si los mismos son tenidos en cuenta, si podrían instalarlo en sus computadoras (revisen el tema del sistema operativo) y si la información que encuentran es suficiente para decidirse a instalarlo.

Ejercicio 1.2 Busquen en Internet el libro "Cultura libre" de Lawrence Lessig. Escriban algunas de sus principales ideas o la que más interesantes les parecieron y expliquen por qué.

Capítulo 2

Introducción

En este capítulo vamos a describir las herramientas que necesitamos para comenzar. En primer lugar vamos a describir el robot y los accesorios necesarios para trabajar con él y luego, vamos a ver la forma de interactuar con Python. Si bien no es la única manera de trabajar, comenzaremos con los mecanismos más sencillos y luego incorporaremos otros a medida que avancemos.

2.1. El robot Multiplo N6

El robot *Multiplo N6*, es un robot educativo extensible basado en la plataforma de prototipado Arduino. Tanto el hardware como el software utilizado son libres, es decir que tanto las especificaciones de la electrónica necesaria como la de los programas necesarios para su utilización están accesibles y pueden utilizarse libremente. En el Anexo D podrán encontrar las direcciones desde donde descargar todas las especificaciones y las aplicaciones utilizadas en este libro, como así también los códigos de los ejemplos presentados.

Googleame: Arduino

¿Qué es Arduino?

Como se ve en la Figura 2.1 nuestro robot cuenta con 3 ruedas que le permiten desplazarse: dos delanteras con tracción y una rueda trasera que le permite girar.

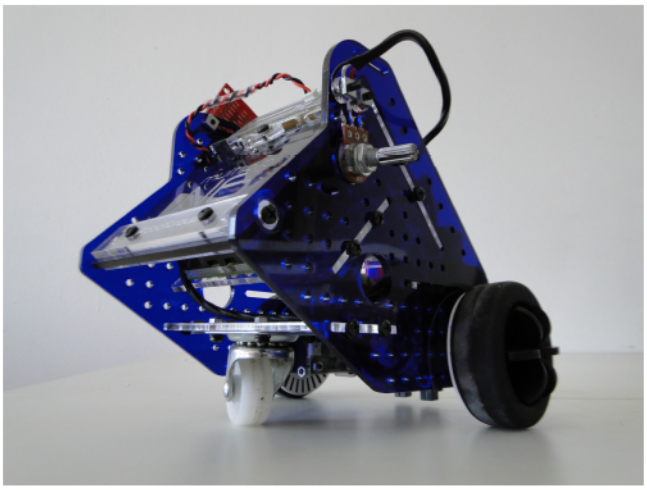


Figura 2.1. Robot Múltiplo N6

Figura 2.2. Módulo de comunicaciones XBee para la computadora

El robot se mueve gracias a dos motores de corriente continua, cuyas respectivas cajas de reducción permiten al robot alcanzar velocidades respetables.

Algo muy interesante es que este robot Multiplo N6 cuenta también con algunos sensores que nos permitirán realizar distintas actividades. El robot básico[1], posee dos sensores de línea que pueden desensamblarse y acoplarse para recibir información del movimiento de las ruedas, además de la posición clásica para hacer seguimiento de líneas basado en los cambios de contraste. Además de los sensores de línea, el robot tiene un sensor ultrasónico que permite detectar obstáculos u objetos con centímetros de precisión hasta una distancia de 6 metros. El robot se mueve de forma independiente, sin cable, utilizando tres pilas AA. La conexión para el manejo se realiza a través del protocolo XBee (IEEE 802.15.4) que permite la utilización del robot de manera inalámbrica. Esta conexión se realiza utilizando el módulo de comunicaciones XBee, similar al que se ve en la Figura 2.2. Este módulo se conecta a la computadora por medio de un conector USB, y permite la comunicación con uno o más robots al mismo tiempo (utilizando una misma computadora).

Más adelante, en este capítulo mostraremos cuáles son los pasos que debemos seguir para conectar el robot y la computadora. Antes veremos cuál será el entorno de trabajo. Recuerden que la forma de enviarle órdenes al robot es escribiendo programas en el lenguaje Python. Por lo tanto, veremos qué entornos podemos usar para escribir nuestros programas.

[1]Para una primer experiencia, hemos utilizado un robot básico, con unos pocos sensores que nos permitirán realizar actividades interesantes, pero sin introducir demasiada complejidad a la hora de empezar a programar. Pero es importante destacar que es posible agregarle más sensores y, de esta manera, plantear otro tipo de actividades más complejas.

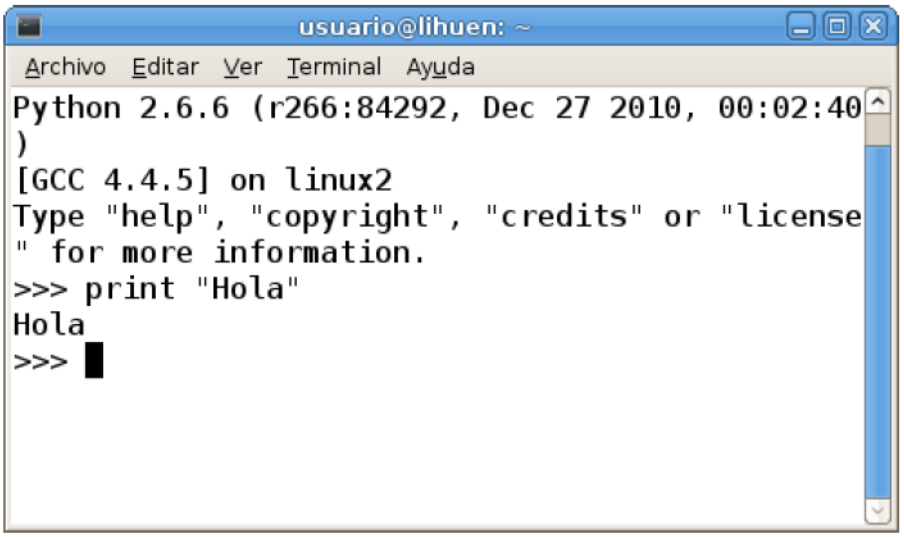


Figura 2.3. Entorno Interactivo

2.2. Introducción al entorno de Python

Como mencionamos al comienzo de este libro, Python es un lenguaje interpretado, con lo cual necesitamos de un intérprete que nos permita ejecutar nuestros programas. Para esto, tenemos dos opciones: si se trata de un programa sencillo, podríamos ir ejecutándolo línea a línea a través de las llamadas "sesiones interactivas", esto significa que cada orden que introducimos se ejecuta instantáneamente. La otra opción es utilizar un editor de textos cualquiera o un entorno de desarrollo (lo que se conoce como IDE, por sus siglas en inglés de Integrated Development Environment) y luego invocar al intérprete Python para que ejecute todo el código. Cualquier de las dos opciones son válidas, aunque nosotros vamos a sugerir comenzar por la primera, para ir acostumbrándonos al lenguaje y luego pasaremos a un entorno más completo. Respecto a la primer opción, los lenguajes interpretados suelen ofrecer una herramienta de ejecución interactiva, que permite dar órdenes directamente al intérprete e ir obteniendo respuestas inmediatas para cada una de ellas. Esta manera de ejecución nos permite ir probando cosas y viendo el resultado de manera inmediata sin necesidad de codificar todo el programa completo. El entorno interactivo es de gran ayuda para probar con fragmentos de programa antes de incluirlos en una versión definitiva.

Tanto para una manera u otra necesitamos de herramientas básicas como ser un intérprete Python y una consola interactiva, ya sea para poder ejecutar línea a línea el código como para contar con un editor de textos, en el caso de escribir todo el programa completo. Todo esto forma parte del llamado "entorno de programación". Como también mencionamos en varias ocasiones, Python es software libre, por lo que contamos con un intérprete para prácticamente cualquier plataforma en uso (computadoras con Linux, Windows, Macintosh, etc) sin pagar costos de licencias por ello. El intérprete estándar nos da también la posibilidad de trabajar en modo interactivo, pero podemos instalar programas como PyShell, Geany u otros que agregan funcionalidades extra al modo interactivo, como ser auto-completado de código, coloreado de la sintaxis del lenguaje, etc. Esto permite hacer más fácil la tarea de programar.

Como mencionamos antes, hay distintos entornos de trabajo. La Figura 2.3 muestra el entorno interactivo provisto por el lenguaje, mientras que la Figura 2.4 muestra un entorno un poco más amigable, denominado PyShell que será el que utilizaremos en un comienzo.

Se pueden observar claramente las ventajas que ofrece el uso de este tipo de ambientes. Para mostrar el ejemplo se ejecutó una sesión interactiva, se escribió la sentencia de impresión en la línea dónde aparece el "prompt" (en este caso la instrucción **print "hola"** y se ejecutó con el sólo hecho de presionar la tecla Enter. El resultado se pudo ver al instante.

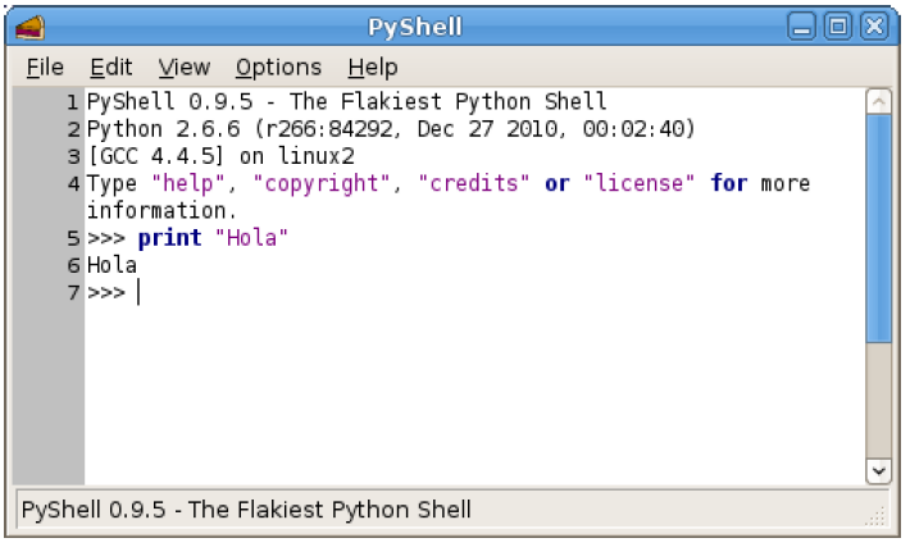


Figura 2.4. PyShell

✎ Se llama "prompt" al símbolo o conjunto de símbolos que usan algunos programas interactivos para indicar que están listos para recibir instrucciones, algunos "prompts" conocidos son:

usuario@maquina:~$ es el prompt de la terminal de GNU/Linux.

C:\> es el prompt de la terminal de Windows y DOS.

>>> y **...** son los prompts de Python, el primero indica que se puede escribir código, el segundo indica que el código debe estar indentado (debe contener espacios o una tabulación al principio).

2.2.1. Usando el intérprete

El intérprete de Python es un entorno sumamente sencillo pero que puede usarse sin problemas. La Figura 2.3 muestra su interfaz de usuario. Como pueden ver, simplemente es una aplicación donde debemos introducir las sentencias y Python las ejecutará directamente.

Para obtener ayuda desde la terminal o desde el intérprete de Python tenemos una serie de opciones. Desde la terminal podemos obtener una guía del uso de Python y las opciones al momento de invocar el intérprete con la opción `-h`:

```
$ python -h
```

Una vez dentro del intérprete podemos obtener ayuda sobre los mensajes y funciones propias de Python con la sentencia `help()` o bien incluir dentro de los paréntesis sobre lo que necesitamos conocer su uso:

```
>>> help()
>>> help(Board)
>>> help(Robot)
```

La sentencia `help` proporciona la lista de los mensajes que podemos utilizar, por ejemplo con el `Robot` muestra la siguiente información:

```
Help on class Robot in module robot:

class Robot
 |  Methods defined here:
```

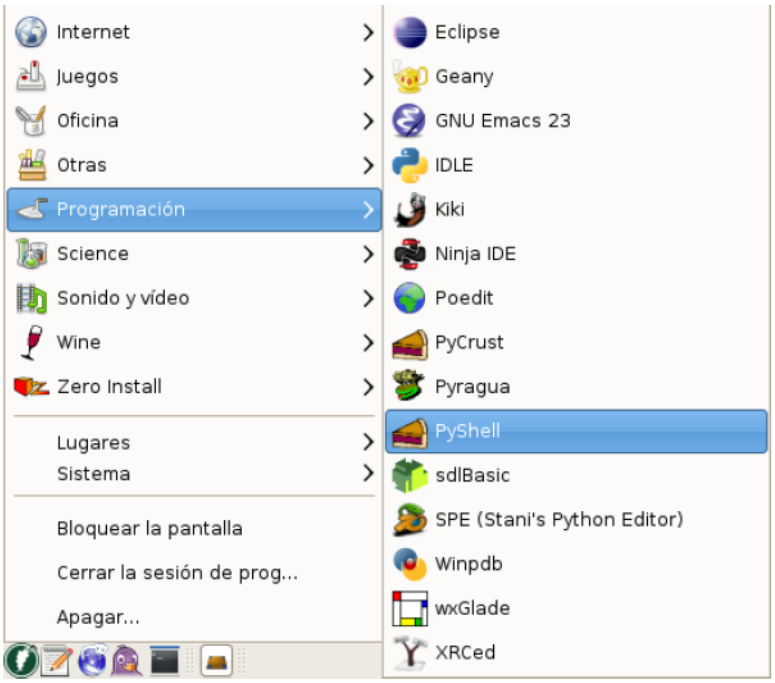


Figura 2.5. Icono de Inicio de PyShell

```
 |
 |  __del__(self)
 |
 |  __init__(self, board, robotid=0)
 |      Inicializa el robot y lo asocia con la placa board
 |
 |  backward(self, vel=50, seconds=-1)
 |      El robot retrocede con velocidad vel durante seconds segundos.
 |
 |  battery(self)
 |      Devuelve el voltaje de las baterias del robot.
.....
```

Mas adelante veremos el significado de cada uno y cuáles se pueden usar y cuáles no.

2.2.2. PyShell (del proyecto wxPython)

PyShell es un entorno de desarrollo implementado por el equipo de desarrollo de wxPython (un módulo de Python para escribir programas con interfaz gráfica). Como dijimos antes, se pueden crear y ejecutar programas en Python sin utilizar PyShell, pero con este entorno se realizan las tareas de una manera más sencilla.

Una de las primeras ventajas que podemos ver a simple vista es cómo muestra los textos con distintos colores de acuerdo a si se trata de una palabra clave (palabra perteneciente al lenguaje Pythpn) o no, o si estamos introduciendo un comentario o una cadena de caracteres (string). Otra característica muy útil es el auto-completado de texto, realizando una simple combinación de teclas como las que se pueden ver en la Tabla 2.1, etc..

Está claro que para poder trabajar con Python y su entorno debemos tenerlo instalado en nuestra computadora. Hay sistemas operativos que ya incluyen Python en su instalación, como ser algunas distribuciones de GNU/Linux[2], pero hay otros sistemas operativos que no lo incorporan, en cuyo caso

[2]Las netbooks del programa Conectar- Igualdad, cuentan ya con una versión de Python instalada en su partición de GNU/Linux

Acción	Combinación
Auto-completar texto	Shift+Enter
Repetir línea anterior del historial	Ctrl+Arriba
Repetir línea siguiente del historial	Ctrl+Abajo
Auto-completar después de un "."	Ctrl+Enter
Incrementar o decrementar el tamaño de la fuente	Ctrl y la ruedita del mouse

Tabla 2.1. Combinaciones de teclas importantes de PyShell

la instalación estará bajo nuestra responsabilidad.

En el caso de necesitar instalarlo por primera vez o actualizar la versión que tenemos instalada en nuestra máquina podemos acceder a la página oficial del lenguaje http://www.python.org, donde encontraremos versiones y documentación correspondiente a diferentes sistemas operativos.

Para ejecutarlo, simplemente debemos cliquear sobre el nombre del programa de la lista de programas que tengamos instalado. En la Figura 2.5 podemos ver como abrirlo desde el menú de Gnome en Lihuen GNU/Linux, en otras distribuciones GNU/Linux la forma de abrirlo es similar.

En distribuciones GNU/Linux basadas en Debian (como Lihuen y Ubuntu) el paquete que contiene PyShell se llama **python-wxtools**.

2.3. El ejemplo más sencillo

Cada vez que se aprende un lenguaje de programación el primer programa dado como ejemplo se conoce como "Hola Mundo". Este programa permite mostrar el mensaje "Hola Mundo" en la pantalla de la computadora. En nuestro caso, esto se realiza simplemente escribiendo la siguiente sentencia en el entorno y ejecutándola con el sólo hecho de presionar la tecla `Enter`.

```
>>> print "Hola Mundo"
```

El resultado, que puede verse al instante, se mostró en la Figura 2.3.

2.4. ¿Qué necesitamos para empezar?

Con respecto al hardware, necesitamos un computadora, preferentemente con un sistema GNU/-Linux[3], un robot y el módulo de comunicaciones XBee. Como dijimos antes, Python es un lenguaje multiplataforma, es decir, que puede usarse tanto en sistemas Microsoft Windows como en sistemas Linux o Mac, pero, como nosotros vamos a usar Linux como sistema operativo les detallamos qué aplicaciones deberemos instalar para empezar a trabajar. Igualmente, en cada caso, les vamos a indicar dónde pueden encontrar información si es que quieren trabajar sobre otra plataforma. Primero y principal, necesitamos el intérprete Python y las librerías para utilizar el robot. Después, aunque no menos importante, un entorno para desarrollar nuestros programas.

[3]Nosotros trabajamos con Lihuen GNU/Linux http://lihuen.info.unlp.edu.ar

✎ ¿Cómo puede saber si tengo instalado el intérprete Python? En muchas distribuciones de GNU/Linux Python ya se incluye en la instalación por defecto. Para saber si ya está instalado pueden probar con los siguientes comandos: en la consola [4].

```
# aptitude search python => muestra los paquetes instalados y posibles
de instalar que contengan la palabra python.
# dpkg -l | grep python => muestra solamente los paquetes instalados
que contengan la palabra python.
# python -V => muestra la versión de python instalada
```

La librería que vamos a utilizar para realizar la comunicación con el robot se llama `robot` y debemos instalarla junto con un paquete de software especial denominado `python-serial`. En el Apéndice I: Guía de instalación de paquetes. se muestra la forma de instalar paquetes en un sistema Linux basado e Debian, como lo son Ubuntu y Lihuen.

2.5. Conectando el robot

Antes de realizar cualquier acción con el robot, primero debemos "comunicarlo" con la computadora que estemos usando. Esto lo hacemos a través del módulo de comunicaciones XBee que mencionamos en secciones previas. Este módulo se conecta a la computadora por medio de una interfaz USB, y permite la comunicación con uno o más robots al mismo tiempo.

Primero es necesario conectar el módulo de comunicaciones a la computadora, utilizando para esto alguno de los puertos USB disponibles. Al hacerlo, se creará un dispositivo con un nombre similar a `/dev/ttyUSB0`[5] en el sistema, el número (en este caso 0) puede variar. Podemos comprobar esto desde una terminal de la siguiente manera:

```
usuario@host:~$ ls /dev/ttyUSB*
/dev/ttyUSB0
```

Luego es necesario encender el robot. Una vez presionado el botón I/O para encender el robot, es necesario esperar a que el LED de estado naranja de la esquina inferior derecha deje de parpadear. Luego, hay que presionar el botón "Run", cuando este LED se apague podremos empezar a usar el robot. Podemos ver la ubicación de estos botones en la Figura 2.6 y el LED de estado en la Figura 2.7.

✎ Algo muy importante a tener en cuenta es que se debe verificar que las pilas estén correctamente colocadas y bien cargadas. Esto es crítico para un correcto funcionamiento del robot.

Como mencionamos anteriormente para poder trabajar con el robot, primero debemos conectarlo con nuestra computadora. Para ello, debemos ingresar al entorno de Python (cualquier de los entornos mencionados) y, como vamos a trabajar con el robot en modo interactivo, todos los mensajes que le enviemos se ejecutarán inmediatamente.

Dentro del entorno, ahora debemos **importar** las funciones que nos permiten trabajar con el robot. Esto se debe a que las funciones propias del robot no vienen integradas con el lenguaje Python y, para poder usarlas, debemos incorporarlas al entorno. Esto se hace utilizando la sentencia `import` del lenguaje Python:

[4] Algunas veces, vamos a indicar que utilicen la consola o terminal de Linux para probar o ejecutar algunos comandos. La terminal, por lo general se encuentra en el menú Accesorios

[5] Recuerden que en este libro, estamos trabajando sobre una plataforma Linux, en Windows el dispositivo tendrá un nombre del estilo com0, el número correcto se puede ver en el 'Panel de control", dentro del "Administrador de dispositivos", en el desplegable "Puertos (COM y LPT)"

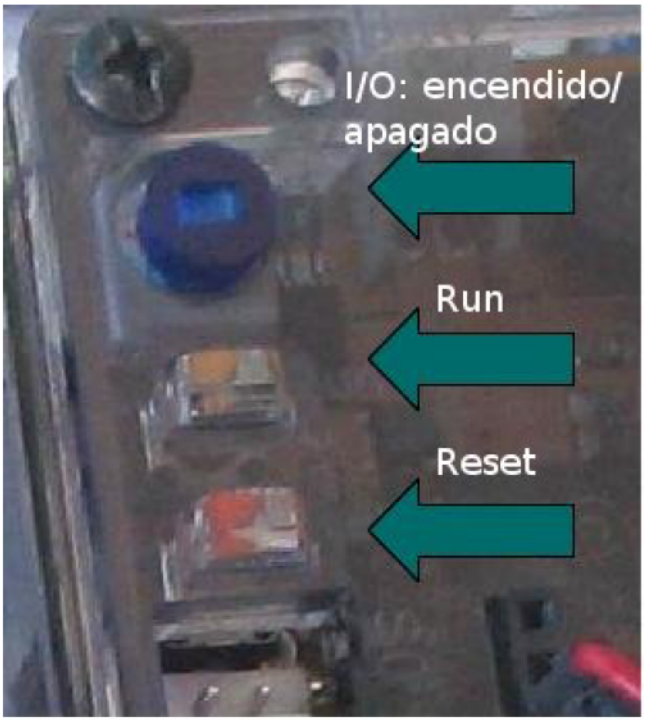


Figura 2.6. Botones del robot N6

Figura 2.7. LED de estado

```
>>> from duinobot import *
```

Una vez realizado lo anterior, ya podemos comenzar a utilizar el robot. Lo primero que debemos hacer, es identificarlo de alguna manera. Para eso, debemos ubicar cuáles robots se encuentran prendidos y conectados a nuestra computadora. Esto lo hacemos de la siguiente manera:

```
[1] >>> b = Board("/dev/ttyUSB0")
[2] >>> b.report()
[3] [1, 2, 3]
```

Analicemos un poco las instrucciones anteriores. En la línea [1] identificamos con el nombre **b** el módulo de comunicaciones que nos permitirá consultar qué robots están prendidos (línea [2]). Si volvemos a mirar la línea [1] vemos que tenemos que conocer la ruta del dispositivo que se crea cuando se inserta el módulo de comunicaciones al puerto USB de la computadora, por ejemplo "`/dev/ttyUSB0`". Aquellos robots que estén prendidos responderán diciendo su número de identificación. La línea [3] nos muestra todos los robots encendidos conectados al módulo de comunicaciones conectado.

Si hubiera más de un robot con el mismo número de identificación, será necesario cambiarlos para que todos tengan un número de identificación unívoco. En la sección 2.7 se puede ver como a cambiar el número de identificación del robot.

Si ningún robot responde, debemos probar otra vez teniendo en cuenta que es importante recordar que luego de prender el robot hay que presionar la tecla `Run` y esperar a que la luz de estado se apague.

Una vez que conocemos cuáles son los robots que se encuentran disponibles, tenemos que tomar uno de ellos para trabajar. Esto lo hacemos de la siguiente manera:

```
>>> mi_robot = Robot(b, 1)
```

`mi_robot` es un nombre que creamos para identificar al robot cuyo identificador es 1. Presten atención que para esto, debemos "crear" un robot, indicando el identificador que hace referencia al módulo de comunicaciones, en este caso, `b`; y el número identificador del robot que queremos utilizar, en este caso 1.

¡Ahora sí!, ya tenemos todo listo para empezar a trabajar con el robot...

2.6. La primera actividad

Como primer actividad, vamos a pedirle al robot que emita un sonido. Esto lo vamos a hacer con la orden `beep`. El robot N6 tiene un pequeño parlante interno con el que puede emitir frecuencias simples en el orden de 1 Hz a 15 Khz. El ejemplo siguiente indica al robot que emita un tono de 200 Hz durante medio segundo. Podemos variar la frecuencia y la duración del sonido invocando a `beep()` con otros argumentos.

```
>>> robot.beep(200,0.5)
```

Si escuchamos los sonidos, entonces estaremos seguros que nuestro robot está conectado y respondiendo a nuestras órdenes.

¿Le ponemos un nombre? En realidad, desde nuestros programas, debemos referenciar al robot a través del identificador con el que lo creamos, en este caso `mi_robot`. Pero, también podemos asignarle un nombre cualquiera. Para esto utilizamos la orden `setName`.

```
>>> robot.setName("R. Daneel Olivaw")
```

En nuestro ejemplo hemos utilizado el nombre "R. Daneel Olivaw". Observen que hemos utilizado comillas alrededor del nombre. Esto es porque en Python si queremos escribir un texto arbitrario (una frase, un nombre, o cualquier otro texto que no sea parte del lenguaje) debemos encerrarlo entre comillas.

A partir de ahora, nuestro robot se llama "R. Daneel Olivaw". Este nombre no es almacenado en la memoria interna del robot, a diferencia del número de identificación, por lo tanto si queremos que el robot siempre tenga nombre será necesario ponérselo cada vez que se reinicie el intérprete de Python.

Googleame: R. Daneel Olivaw

¿Descubrieron quién es este robot? ¿Lo conocían?

2.7. Cambiando el número de identificación del robot

Los robots Mutiplo N6 vienen con un número de identificación de fábrica, el problema es que si compramos más de uno es posible que los números se repitan, el número de identificación es lo que nos permite distinguir un robot de otro desde la PC, por lo que si hay repetidos no vamos a diferenciar un robot de otro. Si todos los robots tuvieran, por ejemplo, el número de identificación 0, el resultado de `report()` en una sala llena de robots prendidos sería:

```
>>> b.report()
[0]
```

Para solucionar esto debemos apagar todos los robots e ir cambiándoles su identificador uno a uno. Para cambiar el identificador de un robot hay que prenderlo, como vimos anteriormente y ver su identificador con `report()`:

```
>>> from duinobot import *
>>> b = Board("/dev/ttyUSB0")
>>> b.report()
[0]
```

Entonces nos conectamos al robot y le cambiamos su identificador con `setId()`, teniendo cuidado de ir asignando números que no se repitan:

```
>>> robot = Robot(b, 0)
>>> robot.setId(21)
```

Luego podemos prender otro robot y repetir este procedimiento hasta que todos tengan un número de identificación unívoco. La próxima vez que nos conectemos a los robots vamos a tener que usar estos nuevos números que asignamos.

2.8. Resumen

En este capítulo hemos aprendido a iniciar un entorno de ejecución de Python (utilizando el intérprete o un entorno de desarrollo) y hemos aprendido a conectar el robot a la computadora. Vimos que debemos definir una variable tanto para representar al robot (**mi_robot = Robot()**]) como para representar al módulo de comunicaciones (**b = Board("dispositivo")**) y, a través de ellas, comenzar a interactuar. Hasta ahora, sólo hemos visto cómo modificar el nombre del robot y hacer que emita sonidos.

También vimos que para utilizar las funciones del robot, debemos importar una librería usando la sentencia **from duinobot import *** de Python.

2.9. Actividades

Ejercicio 2.1 Verifiquen qué versión de Python tienen instalada en su computadora. Si tienen una computadora con doble booteo (es decir con dos sistemas operativos), hagan esta comprobación en ambos sistemas.

Ejercicio 2.2 Compruebe los dos entornos de trabajo descriptos en este capítulo. Si no los tiene instalado, instálenlos. Recuerden que pueden consultar el apéndice A con las guías de instalación en Linux. ¿Cuál le parece más cómodo?

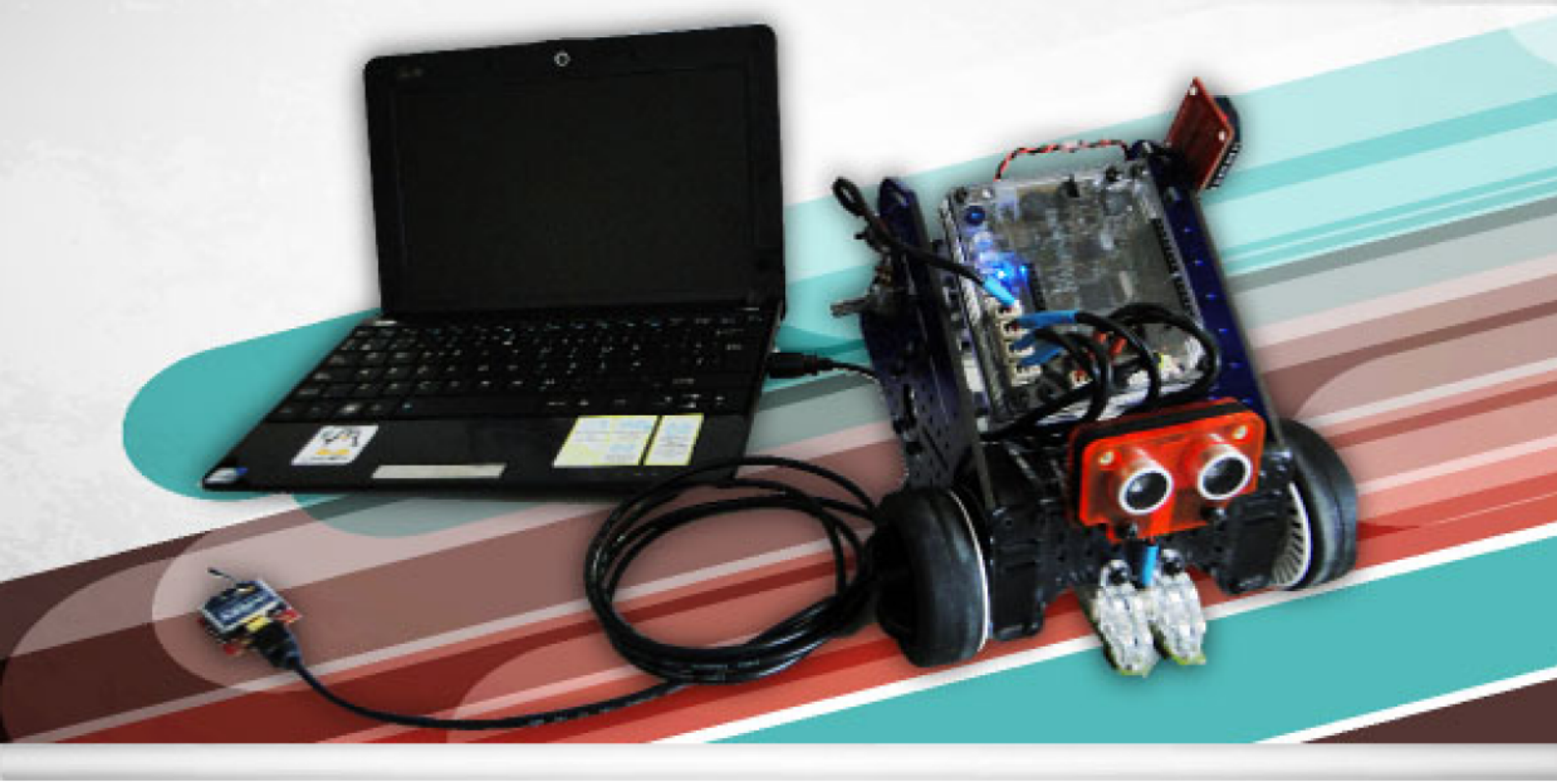
Ejercicio 2.3 Enumeren las aplicaciones y librerías que deben tener instalado en sus sistemas para comenzar a trabajar. Verifiquen cuáles están instaladas y cuáles no. Instalen las que hagan falta para comenzar a trabajar. Nuevamente, pueden consultar en apéndice A para obtener ayuda.

Ejercicio 2.4 Conecten el robot a la computadora. ¿Cómo nos damos cuenta que el robot está conectado? ¿Le ponemos un nombre?

Ejercicio 2.5 Intente que el robot emita una melodía. Prueben distintas frecuencias de sonidos con distintas duraciones.

Capítulo 3

En movimiento



En el capítulo anterior comenzamos a interactuar tanto con el entorno de Python como con nuestro robot. Ahora ya estamos listos para hacer algo más interesante con él. En este capítulo vamos a ver de qué manera podemos indicarle al robot cómo moverse, girar y parar. De esta manera, podremos abordar actividades un poco más entretenidas, a la par que nos introducimos en el mundo de la programación.

3.1. ¿Movemos al robot?

Hay 6 órdenes básicas de movimiento que permiten al robot avanzar, retroceder, doblar a la izquierda, doblar a la derecha, parar y controlar las ruedas independientemente.

Para indicarle al robot que realice cualquiera de las tareas anteriores, primero debemos conectarlo con la computadora de la forma detallada en el capítulo 2 y, recién ahí, podremos comenzar a darle las instrucciones.

3.1.1. Hacia adelante y hacia atrás

Si queremos que nuestro robot avance a velocidad media por un tiempo indeterminado, le debemos dar la orden `forward()`.

¡¡¡No prueben esto con el robot sobre una superficie elevada!!! Si lo hacen, deberán ser muy rápidos para evitar que caiga, dado que esta orden hace mover al robot por un tiempo indefinido.

Si queremos indicarle al robot que se detenga, debemos indicarle la orden `stop()`.

En el siguiente ejemplo, pueden probar esto:

```
usuario@host:~$ python
Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from duinobot import *
>>> b = Board("/dev/ttyUSB0")
>>> mi_robot = Robot(b, 1)
>>> mi_robot.forward()
>>> mi_robot.stop()
```

La orden `forward()` soporta cero, uno o dos argumentos. Si, como en el ejemplo anterior, no se le indican argumentos, `forward()` hace que el robot avance a velocidad media por un tiempo indeterminado y sólo se detendrá cuando algún elemento obstaculice su camino o cuado se le acaben las baterías. Obviamente, esta opción no es muy práctica, por lo tanto podemos indicarle a qué velocidad queremos que se mueva y durante cuánto tiempo.

Algo muy importante de tener presente, es que detenemos al robot con la orden `stop()`. Esto va a ser muy útil al probar los siguientes ejemplos.

Para indicarle que avance a una velocidad determinada, lo haremos pasándole dicha información a la orden `forward()`. La velocidad es un valor numérico entre -100 y 100, donde 0 indica que el robot no debe moverse, 100 indica que el robot se moverá a velocidad máxima y -100, que el robot se moverá a velocidad máxima pero hacia atrás.

En el siguiente ejemplo, movemos el robot hacia adelante a la velocidad máxima:

```
>>> mi_robot.forward(100)
```

Si probaron el ejemplo anterior, seguramente se habrán dado cuenta que es muy importante poder indicarle al robot cuándo parar, ya que estas órdenes no lo hacen. Sólo lo hacen avanzar a distintas velocidades pero por un tiempo indefinido, y, seguramente más de uno se habrá apurado a tomar el robot antes que choque con algún obstáculo. Como mencionamos antes, la orden `forward()` permite que le enviemos un segundo argumento para indicar el tiempo que queremos que se mueva. Esto lo indicamos emitiendo la orden `forward()` con dos argumentos, donde el segundo argumento indica el tiempo durante el cual se moverá el robot a la velocidad indicada en el primer argumento. Luego de ese tiempo el robot se detendrá por si mismo.

El siguiente ejemplo hace que el robot que avance a velocidad máxima durante 2 segundos.

```
>>> mi_robot.forward(100, 2)
```

Si queremos ordenarle al robot que avance durante 5 segundos, pero sin indicarle la velocidad, debemos invocar la orden `forward()` de la siguiente forma:

```
>>> mi_robot.forward(seconds=5)
```

Para averiguar los nombres de los parámetros se puede usar `help(Robot.forward)`.

Incluso, si referenciamos los argumentos por su nombre, podemos indicar los valores sin recordar el orden en que el debemos escribirlos, como pueden ver en el siguiente ejemplo:

```
>>> mi_robot.forward(seconds=20, vel=10)
```

De la misma manera que le ordenamos avanzar con `forward()`, tenemos una orden que nos permite indicarle al robot que de desplace hacia atrás. Esto lo hacemos con la orden `backward()` que acepta los mismos argumentos que `forward()`, es decir, la velocidad y la cantidad de segundos que queremos que se mueva.

En el siguiente ejemplo le indicamos al robot que retroceda a velocidad 20 durante 5 segundos.

```
>>> mi_robot.backward(20, 5)
```

Recuerden que antes mencionamos que si indicamos una velocidad negativa a `forward()` hace que el robot retroceda, por lo tanto, las órdenes `forward()` y `backward()` pueden usarse indistintamente. Por ejemplo, `forward(-20)` y `backward(20)` son sinónimos y producen exactamente los mismos resultados.

Sin embargo, un buen programador utilizará `forward()` para indicar que el robot avance y `backward()` cuando el robot retroceda.

3.1.2. ¿Giramos?

Además de avanzar y retroceder, dijimos que nuestro robot puede girar. Para lograr que gire a izquierda le daremos la orden `turnLeft()`. Si no indicamos argumentos a esta orden, el robot girará a velocidad media por tiempo indeterminado. De la misma forma que con las funciones `forward()` y `backward()` el robot puede detenerse con `stop()`.

Para girar a derecha se puede usar `turnRight()`. Tanto `turnLeft()` como `turnRight()` funcionan de forma similar y soportan entre cero y dos argumentos de la misma forma que `forward()` y `backward()`.

Probemos el siguiente ejemplo:

Código 3.1. Moviéndonos en zig zag

```
>>> mi_robot.turnLeft()
>>> mi_robot.stop()
>>> mi_robot.turnRight()
>>> mi_robot.stop()
```

Si queremos indicarle al robot que gire durante 3 segundos hacia la izquierda a velocidad máxima y luego hacia derecha durante 2 segundos a velocidad 30 podríamos darle las siguientes órdenes:

```
>>> mi_robot.turnLeft(100, 3)
>>> mi_robot.turnRight(30, 2)
```

Si utilizamos el intérprete interactivo de Python veremos que, cuando usamos trabajamos con el robot indicándole que se desplace o gire, algunas veces no podemos dar la siguiente orden hasta que no finalice la actual. Más adelante, cuando empecemos a escribir y guardar programas notaremos que estas mismas variantes demoran la ejecución de las sentencias subsiguientes del programa hasta que terminan. Por eso es importante tenerlas en mente. En la Tabla 3.1 podemos ver las diferentes formas de utilizar estas órdenes y cuáles se bloquean demorando la ejecución de otras sentencias. Que una orden se bloquee hasta terminar su ejecución o no, no es ni bueno ni malo. A lo largo de este libro vamos a encontrar un uso a ambos tipos de métodos.

Recordamos que un programa es un conjunto de instrucciones u órdenes que le indican a la computadora cómo se realiza una tarea.

Utilización	Bloqueante
`forward(10,5)`	si
`forward(seconds = 2)`	si
`forward(20)`	no
`forward()`	no

Tabla 3.1. Funciones bloqueantes y no bloqueantes

Una orden que puede sernos de mucha utilidad es `wait()`. Esta orden hace que nuestro programa se detenga (bloquee) durante una cierta cantidad de segundos.

En el siguiente ejemplo usamos `wait()` para simular el comportamiento del ejemplo anterior pero usando las versiones no bloqueantes de `turnLeft()` y `turnRight()`.

Código 3.2. Muevo y espero

```
>>> mi_robot.turnLeft(100)
>>> wait(3)
>>> mi_robot.turnRight(30)
>>> wait(2)
\index{wait}
```

3.2. Dibujando figuras

Ya aprendimos cómo mover el robot en cualquier dirección. Si todavía no probaron los ejemplos que les mostramos, pruebenlos jugando y haciendo que el robot se mueva en distintas direcciones y con distintas velocidades.

Una vez que se familiaricen con las formas en que podemos girar, avanzar y retroceder pueden empezar a combinarlas para hacer cosas más interesantes, por ejemplo, podemos hacer que nuestro robot dibuje un cuadrado en el piso.

Para dibujar un cuadrado el robot tiene que avanzar una cierta distancia, girar 90 grados, repitiendo esto mismo cuatro veces. Un recurso que usamos los programadores para bosquejar un programa es lo que llamamos pseudocódigo. Esto es, básicamente, describir cuáles son las órdenes que debemos introducir en nuestro programa, pero descriptas con palabras en lenguaje natural, sin usar ningún lenguaje de programación en particular.

Pensemos entonces cómo deberíamos indicarle al robot que realice un cuadrado de 20x20 cm en este pseudocódigo:

```
avanzar 20 centímetros
girar 90°
avanzar 20 centímetros
girar 90°
avanzar 20 centímetros
girar 90°
avanzar 20 centímetros
girar 90°
```

Si intentaron probar esto en el intérprete Python, se habrán dado cuenta que se produjeron varios errores. ¿Por qué? Es muy fácil, estas órdenes no son sentencias propias de Python, de hecho es un lenguaje inventado, solamente utilizaremos esta estrategia para tener una idea de cómo quedará el programa una vez escrito. En este caso, se trata de un programa muy sencillo, pero esto puede sernos de mucha utilidad cuando tengamos que pensar algoritmos más complejos.

Revisemos las órdenes que desplazan y giran al robot. ¿Alguna de ellas, permite indicarle al robot que avance 20 cm y gire 90°? No. Por lo tanto, vamos a tener que experimentar el uso de `forward()` con distintas velocidades y tiempos para calcular el tamaño del lado del cuadrado que nos parezca adecuado. En general es recomendable que sea una figura chica así es más fácil darse cuenta si lo estamos dibujando bien o no. Lo mismo deberemos hacer con `turnRight()` o `turnLeft()`, para lograr un ángulo lo más parecido posible a 90 grados. Estos valores pueden variar según el robot que estemos utilizando y es recomendable que cada uno haga estas pruebas con el robot que usará para dibujar el cuadrado. No se preocupe porque las formas de las figuras sean perfectas, el robot no está diseñado para tener movimientos tan precisos como para hacer un cuadrado perfecto, lo importante es que se asemeje a la figura deseada.

3.2.1. Mi primer programa

Una vez que encontramos cuáles son los valores adecuados para indicarle al robot que avance tanto centímetros y gire 90 grados, debemos escribir nuestro programa en Python. Una posible versión podría ser parecida a la que les mostramos a continuación:

Código 3.3. Nuestro primer programa

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

b.exit()
```

Como pueden ver, en el código pueden notar que cada tanto usamos la función `wait()` para que detenga (o bloquee) el programa hasta que el robot termine de detenerse (normalmente las ruedas del robot se mueven unos milímetros de más por inercia). Existe una orden, denominada `motors()` que permite controlar las ruedas del robot de forma independiente. `motors()` puede recibir 2 argumentos indicando la velocidad de los motores derecho e izquierdo. Como en los casos anteriores, la velocidad es un valor en el rango entre -100 y 100, indicando con el signo la dirección en la que girará cada rueda.

Probemos el siguiente ejemplo:

```
>>> mi_robot.motors(-50, 50)
>>> mi_robot.stop()
```

¿Qué pasó? ¡El robot gira! También es posible indicar cuántos segundos debe moverse el robot con un tercer argumento a `motors()`. Probemos:

```
>>> mi_robot.motors(-50, 50, 1)
```

3.2.2. Guardando mis programas

Si por algún motivo decidimos continuar más tarde y apagamos nuestra computadora, ¿qué creen que pasa con el programita que hicimos recién? En realidad, como sólo fueron órdenes dadas al intérprete de Python y nunca fueron almacenadas en ningún lado, deberíamos volver a ingresarlas nuevamente si quisiéramos repetir nuestro cuadrado.

Como pueden ver, obviamente esto es muy poco práctico. Cuando escribimos programas, queremos que éstos puedan guardarse en nuestra computadora para que los podamos ejecutar cuantas veces queramos. Es más, podría copiar nuestro programa en otra computadora con Python y ejecutarlo allí. Ahora bien, ¿cómo hacemos esto? Un programa Python es simplemente un archivo de texto que contiene las órdenes escritas anteriormente.

En este punto, es interesante contar con un entorno que nos facilite estas tareas. Estos entornos se conocen como IDEs (por las siglas en inglés de Integrated Development Environment) y nosotros vamos a usar uno denominado Geany[1] , cuya interfaz podemos ver en la Figura 3.1, si bien utilizamos Geany para escribir el código cualquier otro editor de texto podría servir, como el Block de Notas de Windows o el Gedit de Linux.

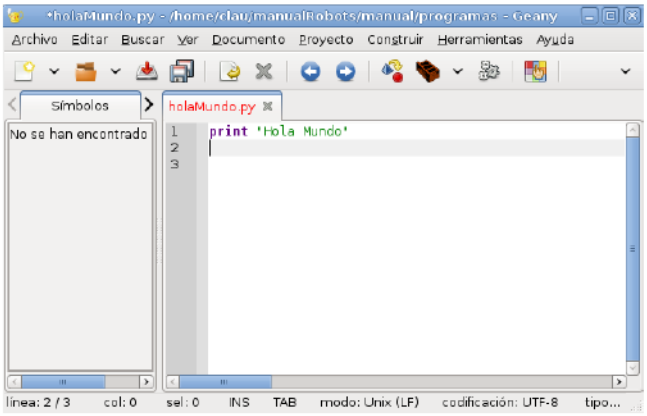


Figura 3.1. Entorno de Desarrollo Geany

Antes de avanzar vamos a darles algunas ayudas para trabajar.

- Las órdenes se introducen en la zona de edición.
- Debemos guardar esto en un archivo cuya extensión debe ser ".py". Ejemplo: `holaMundo.py`.
- Al terminar en py, Geany asocia estas órdenes con el lenguaje Python, con lo cual van a notar que algunas palabras aparecerán resaltadas y con distintos colores.
- Si prestan atención, en los ejemplos anteriores, al introducir una orden, la misma se ejecutaba en ese preciso momento y ahora no es así. Si queremos ejecutar nuestro programa debemos hacerlo a través del menú "Construir→Ejecutar" o usando el atajo en la tecla "F5". Al ejecutar la orden de construir se abre una ventana con la ejecución de nuestro programa, como se puede ver en la Figura 3.2.

✎ Pueden observar lo anterior en el vídeo http://vimeo.com/47518326 en el sitio web del proyecto[2].

Otra ventaja de escribir mi programa en un archivo es que lo podemos ejecutar directamente desde la terminal. Por ejemplo, podemos tipear el comando `python` seguido del nombre del archivo que acabamos de generar (`holaMundo.py`) como se muestra en el siguiente código:

[1] En el apéndice A les vamos a mostrar cómo instalarlo en sus máquinas.
[2] http://robots.linti.unlp.edu.ar

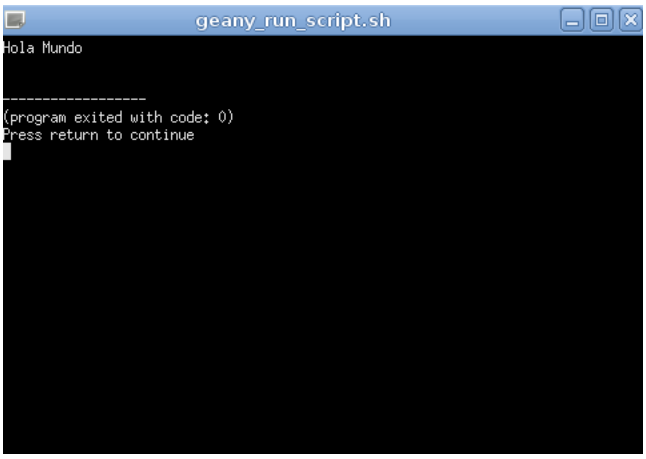


Figura 3.2. Entorno de Desarrollo Geany

```
usuario@host:~$ python holaMundo.py
```

Prueben ahora escribir un programa denominado `cuadrado.py` que le ordene al robot que dibuje un cuadrado. En la sección de actividades van a encontrar ejercicios propuestos para que nuestro robot pueda dibujar otras figuras.

✎ **Recordemos** Los archivos que agrupan código Python de esta manera se llaman `scripts` o simplemente `programas` y mientras el intérprete interactivo es lo más cómodo para probar fragmentos de código chicos, guardar el código en un archivo para luego probarlo es lo más adecuado para programas más largos.

3.3. Agrupando instrucciones en funciones

Hasta ahora con mayor o menor dificultad pudimos escribir un programa que le indica al robot que dibuje un cuadrado. Este programa realiza una sola acción (un cuadrado) pero normalmente los programas hacen muchas cosas diferentes.

¿Cómo haríamos ahora para hacer un programa que dibuje primero un cuadrado, luego un triángulo y finalmente otro cuadrado? Una posible solución es tomar el código que ya escribimos y copiar las partes que nos sirven. Visualicemos el resultado en el pseudocódigo de abajo.

✎ Usamos el símbolo **#** (al igual que en Python) para denotar un texto que no será ejecutado por el intérprete Python, pero que sirve de guía para los programadores que miren el código. Estos textos son **comentarios** y es una muy buena práctica de programación utilizarlos.

```
# Primer cuadrado
avanzar
girar 90°
avanzar
girar 90°
avanzar
girar 90°
avanzar
girar 90°

# Triángulo
avanzar
girar 120°
avanzar
girar 120°
avanzar
girar 120°

# Segundo cuadrado
avanzar
girar 90°
avanzar
girar 90°
avanzar
girar 90°
avanzar
girar 90°
```

Como podemos ver, el código es bastante largo y esto se debe, en parte, a que repetimos la porción de código que dibuja el cuadrado. Si miran atentamente las instrucciones que se corresponden con el primer y segundo cuadrado son exactamente iguales.

Afortunadamente existe una forma de escribir código una sola vez, y luego hacer que se ejecute las veces que necesitemos. Para esto, vamos a escribir una **función** en Python que se denomine, por ejemplo, `cuadrado`, y luego invocarla dos veces.

Función: En programación, una función es un conjunto de sentencias o instrucciones que nos permite abstraer una funcionalidad dada de mi programa y que se ejecuta al ser llamada o invocada, usualmente por su nombre.

Podemos definir dos funciones: una que dibuje un cuadrado y otra que dibuje un triángulo a las que podríamos invocar de la siguiente manera: **cuadrado**() y **triangulo**(). De esta manera, nuestro programa se reduciría a:

```
# Invocación a la función cuadrado
cuadrado()

# Invocación a la función  triangulo
triangulo()

# Segunda invocación a la función cuadrado
cuadrado()
```

De esta manera, no sólo evitamos repetir muchas instrucciones[3] sino que también nuestro programa

[3]Todavía no sacaremos todos los repetidos, seguiremos teniendo repetido el código para dibujar cada lado, más adelante veremos cómo resolver esto.

quedó más legible.

En Python una **función** se define usando el constructor **def** seguido por un nombre que identificará la función y que permitirá luego invocarla tantas veces como sea necesario.

La forma más simple de explicar esto es con un ejemplo:

Código 3.4. Función cuadrado

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

def cuadrado():
    mi_robot.forward(50, 0.5)
    wait(1)
    mi_robot.turnRight(35, 1)
    mi_robot.forward(50, 0.5)
    wait(1)
    mi_robot.turnRight(35, 1)
    mi_robot.forward(50, 0.5)
    wait(1)
    mi_robot.turnRight(35, 1)
    mi_robot.forward(50, 0.5)
    wait(1)
    mi_robot.turnRight(35, 1)

# Dibujamos 3 cuadrados
cuadrado()
cuadrado()
cuadrado()

b.exit()
```

En el ejemplo anterior, definimos una función llamada **cuadrado** y luego la invocamos tres veces. La primer línea de la definición de cualquier función comienza con la palabra clave **def**, seguida del nombre de la función y finalmente `():`. Luego, se escriben las instrucciones asociadas a la función. Cada línea tiene que comenzar con una tabulación o con dos espacios. Desplazar el texto hacia la derecha, se denomina "indentar" y, como veremos muchas veces es muy importante respetar esto en Python.

Se debe ser consistente con la forma elegida para indentar el código. Escoger una forma y hacerlo siempre así, de lo contrario el programa puede fallar.

3.3.1. Nombres de función válidos

En la sección anterior, le dimos un nombre a una función. En Python, estos nombres (tanto de funciones como de variables) deben respetar algunas normas:

- Los nombres pueden contener letras del alfabeto inglés (es decir no puede haber ñ ni vocales con tilde).
- Los nombres pueden contener números y guión bajo.
- Los nombres pueden empezar con una letra o un guión bajo (nunca con un número).

Nombres Válidos	Nombres Inválidos
mi_funcion	funcion_ñ
_mi_funcion	_mi_función
_3_mifuncion	3_mifuncion
mi3funcion	mi3 funcion

Tabla 3.2. Nombres válidos e inválidos de funciones

- No pueden usarse palabras reservadas como nombre. Estas palabras son utilizadas por Python para acciones específicas, por ejemplo: **def**.

En la Tabla 3.2 mostramos nombres válidos e inválidos de funciones válidos

3.3.2. Funciones con argumento

Volviendo al ejemplo sobre cómo dibujar un cuadrado de la sección 3.3, pensemos ¿qué tendríamos que hacer si ahora quisiéramos hacer cuadrados de distintos tamaños? Tendríamos que modificar la velocidad (y/o el tiempo) en cada invocación a `forward()`. Además de ser bastante incómodo, ya que hay que invocar a `forward()` cuatro veces para dibujar un cuadrado, no es recomendable porque cuando tenemos que modificar muchas partes distintas de un programa es muy fácil equivocarse y generar errores nuevos.

Ya vimos en la sección 3.3 que podemos escribir una sola vez el fragmento de código para dibujar cuadrados y usarlo muchas veces definiendo una función. El problema es que esa función hace siempre cuadrados del mismo tamaño, por lo tanto no nos sirve. Para lograr esto, podemos escribir una función e indicarle, a través de un parámetro, el dato necesario para dibujar cuadrados de distintos tamaños. De forma similar como le indicamos la velocidad o el tiempo a `forward()`.

Veamos el siguiente ejemplo:

Código 3.5. Cuadrado con argumento

```
def cuadrado(tiempo):
    mi_robot.forward(50, tiempo)
    wait(1)
    mi_robot.turnRight(35, 1)
    mi_robot.forward(50, tiempo)
    wait(1)
    mi_robot.turnRight(35, 1)
    mi_robot.forward(50, tiempo)
    wait(1)
    mi_robot.turnRight(35, 1)
    mi_robot.forward(50, tiempo)
    wait(1)
    mi_robot.turnRight(35, 1)

cuadrado(0.5)
cuadrado(1)
cuadrado(2)
```

En la primera línea podemos ver que entre los paréntesis de la definición de `cuadrado()` tenemos que escribir un nombre para el parámetro, en este caso usamos “tiempo”. El nombre “tiempo” representa el valor pasado al invocar la función. Dentro de la misma podemos usar “tiempo” como si fuera el valor que representa y luego, en cada invocación a `forward()`, reemplazamos el valor que usábamos para la duración del movimiento por el nombre del parámetro.

Como vimos anteriormente las órdenes `forward()` y `turnLeft()` pueden recibir más de un parámetro. Estas órdenes se implementan con funciones, así que una función cualquiera también puede recibir más de un parámetro. En nuestro caso,podemos mejorar aún más nuestra función `cuadrado()` de manera tal que podamos indicar también qué robot querríamos que dibujara el cuadrado. Pensemos que de la manera que está implementada hasta ahora, sirve para un único robot, que debe llamarse siempre "mi_robot". Por lo tanto, podemos pensar en agregar otro argumento que represente al robot. Esto nos va a permitir ponerle distintos nombres a la variable que hace referencia al robot e incluso usar la función cuadrado con distintos robots en el mismo programa. En el siguiente ejemplo se puede ver una implementación de la función cuadrado que recibe el robot como argumento, y en las últimas líneas de este ejemplo se puede ver que se usa cuadrado para 2 robots distintos.

Código 3.6. Dos robots haciendo cuadrados

```
from duinobot import *
def cuadrado(r, tiempo):
    r.forward(50, tiempo)
    wait(1)
    r.turnRight(35, 1)
    r.forward(50, tiempo)
    wait(1)
    r.turnRight(35, 1)
    r.forward(50, tiempo)
    wait(1)
    r.turnRight(35, 1)
    r.forward(50, tiempo)
    wait(1)
    r.turnRight(35, 1)

b = Board("/dev/ttyUSB0")
robot1 = Robot(b, 1)
robot2 = Robot(b, 10)
cuadrado(robot1, 0.5)
cuadrado(robot2, 1)
b.exit()
```

3.4. Agrupar funciones en módulos

En las actividades anteriores escribimos varias funciones que pueden combinarse de distintas formas para hacer programas distintos. Si juntamos todas estas funciones en un único archivo (teniendo cuidado de que cada función tenga un nombre distinto) tenemos, lo que en Python se conoce como un **módulo**.

Módulo: son archivos con instrucciones de Python guardados con extensión ".py", se los puede ejecutar cuantas veces se quiera, se pueden importar y usar desde otros módulos.

Durante el transcurso de este libro ya vimos y usamos el módulo **robot**. El objetivo de esta sección es que aprendamos a hacer y a utilizar nuestros propios módulos.

3.4.1. Mi primer módulo

Si copiamos las funciones que escribimos hasta ahora y las guardamos en un archivo ".py" podemos utilizarlas en muchos programas sin tener que copiarlas cada vez. Copien los siguientes ejemplos en un archivo denominado "movimientos.py".

Código 3.7. Definiendo un módulo

```
  def circulo(r, vel, diferencia):
      r.motors(vel, vel + diferencia)

  def zigzag(r):
      r.forward(50, 1)
      r.turnRight(100, 0.3)
      r.forward(50, 1)
      r.turnLeft(100, 0.3)
      r.forward(50, 1)
      r.turnRight(100, 0.3)
      r.forward(50, 1)
      r.turnLeft(100, 0.3)

  def bifuracion(r1, r2):
      r1.turnLeft(50, 0.5)
      r2.turnRight(50, 0.5)
      r1.forward(100, 1)
      r2.forward(100, 2)

  def carrera(r1, r2, tiempo):
      r1.forward(100)
      r2.forward(100)
      wait(tiempo)
      r1.stop()
      r2.stop()

  def mareado(r):
      r.turnRight(50, 5)
      r.turnLeft(50, 5)
      r.turnRight(50, 5)
      r.turnLeft(50, 5)
```

Hemos definido varias funciones que permiten realizar actividades con uno o dos robots. Ahora, ¿Cómo las podemos usar? De la misma manera que importamos el módulo que contiene las funciones para el manejo del robot, vamos a importar nuestro módulo. Antes de iniciar el intérprete de Python, debemos posicionarnos en el directorio donde guardamos nuestro módulo (o sea, el archivo "movimientos.py'). Si por ejemplo, lo guardamos en un directorio denominado "misModulos", entonces debemos seguir los siguientes pasos:

```
      usuario@host:~$ cd mis_modulos
      usuario@host:~/mis_modulos$ python
      Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
      [GCC 4.4.5] on linux2
      Type "help", "copyright", "credits" or "license" for more information.
      >>> from duinobot import *
(1)   >>> from movimientos import *
      >>> b = Board("/dev/ttyUSB0")
      >>> robotazo = Robot(r, 1)
(2)   >>> mareado(robotazo)
(3)   >>> zigzag(robotazo)
      >>> robotin = Robot(b, 2)
(4)   >>> carrera(robotazo, robotin, 3)
```

Prestemos atención a las líneas que numeramos. En la línea marcada con (1) se importa el módulo `movimientos` (que escribimos nosotros) para que el intérprete Python reconozca las funciones que definimos. En (2), (3) y (4) usamos estas funciones directamente. De esta forma las funciones definidas

en el archivo `movimientos` podrán ser usadas sin tener que definirlas cada vez que necesitemos su funcionamiento.

✎ Es muy importante recordar que nos debemos ubicar en la carpeta o directorio donde están ubicados nuestros programas. De otra forma, al intentar importar el módulo (1) nos dará un error.

3.4.2. Uso de import

Existe otra forma de importar módulos, si tuviéramos el módulo, "funcionesSinParametros.py" y el módulo "funcionesConParametros.py", y si ambos tuvieran versiones distintas de la función `cuadrado()` existe una manera de importar ambos módulos y elegir qué versión de `cuadrado()` queremos. La forma es usar `import` y luego usar el nombre del módulo para invocar a la función que se quiera.

```
      Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
      [GCC 4.4.5] on linux2
      Type "help", "copyright", "credits" or "license" for more information.
      >>> from duinobot import *
1)    >>> import funcionesSinParametros
2)    >>> import funcionesConParametros
      >>> b = Board("/dev/ttyUSB0")
      >>> robot = Robot(b, 1)
3)    >>> funcionesSinParametros.cuadrado()
4)    >>> funcionesConParametros.cuadrado(robot, 2)
```

En las líneas 1) y 2) se puede ver cómo importar módulos con `import` y en las líneas 3) y 4) se usan las distintas versiones de `cuadrado()`. Dependiendo de si queremos nombres cortos o nombres que no sean ambiguos podemos elegir usar `from x import *` o `import x`.

3.5. Resumen

En este capítulo aprendimos a mover el robot con **forward()**, **backward()**, **turnLeft()**, **turnRight()** y **motors()**, aprendimos que a veces (cuando usamos estos mensajes sin indicar el tiempo) tenemos que detener los robots con **stop()** y que, para evitar repetir mucho código, podemos agruparlo en funciones utilizando el constructor **def** de Python.

Aprendimos que se puede bosquejar un programa sin preocuparnos por los detalles usando pseudocódigo, que podemos escribir comentarios con # y que podemos guardar el código en archivos (programas) y ejecutarlos más tarde, o bien guardar funciones en archivos llamados módulos y usarlas más tarde desde el intérprete o desde otros programas.

Técnicamente los programas y los módulos se hacen de la misma forma, la diferencia es que los programas realizan determinada tarea y los módulos almacenan funciones útiles para ser utilizadas desde programas.

Por otro lado podemos pausar la ejecución de un programa usando la función **wait(segundos)** donde "segundos" es un número que indica cuantos segundos debe durar la pausa.

También hemos aprendido a trabajar con el IDE Geany que nos facilita la programación mediante el resaltado de sintaxis y la ejecución del programa.

3.6. Actividades

Ejercicio 3.1 Usando lo experimentado hasta ahora escriba un programa para dibujar un triángulo equilátero (los ángulos internos miden 60 grados), no se preocupen porque el triángulo quede perfecto, es muy difícil lograrlo.

Ejercicio 3.2 Escriban un programa que haga que el robot avance en zigzag durante 1 minuto.

Ejercicio 3.3 Usando `motors()` se puede indicar que una rueda se mueva más lenta que la otra. Controlando la diferencia de velocidad se pueden hacer círculos más grandes o más chicos. Experimente el mensaje `motors()` para crear círculos de distintos tamaños.

Ejercicio 3.4 Implemente una función para dibujar triángulos, aprovechando el código que escribió en las actividades anteriores y ejecútela. Esta función no debe recibir parámetros.

Ejercicio 3.5 Modifique la función anterior para que el robot y la longitud de los lados del triángulo se pasen como argumentos.

Ejercicio 3.6 Implemente la función `mareado()` que haga girar el robot varias veces en una y otra dirección.

Ejercicio 3.7 Implemente una función que reciba 2 robots como argumentos, y envíe uno hacia la derecha y el otro hacia la izquierda a velocidad máxima durante un número de segundos pasados como parámetros.

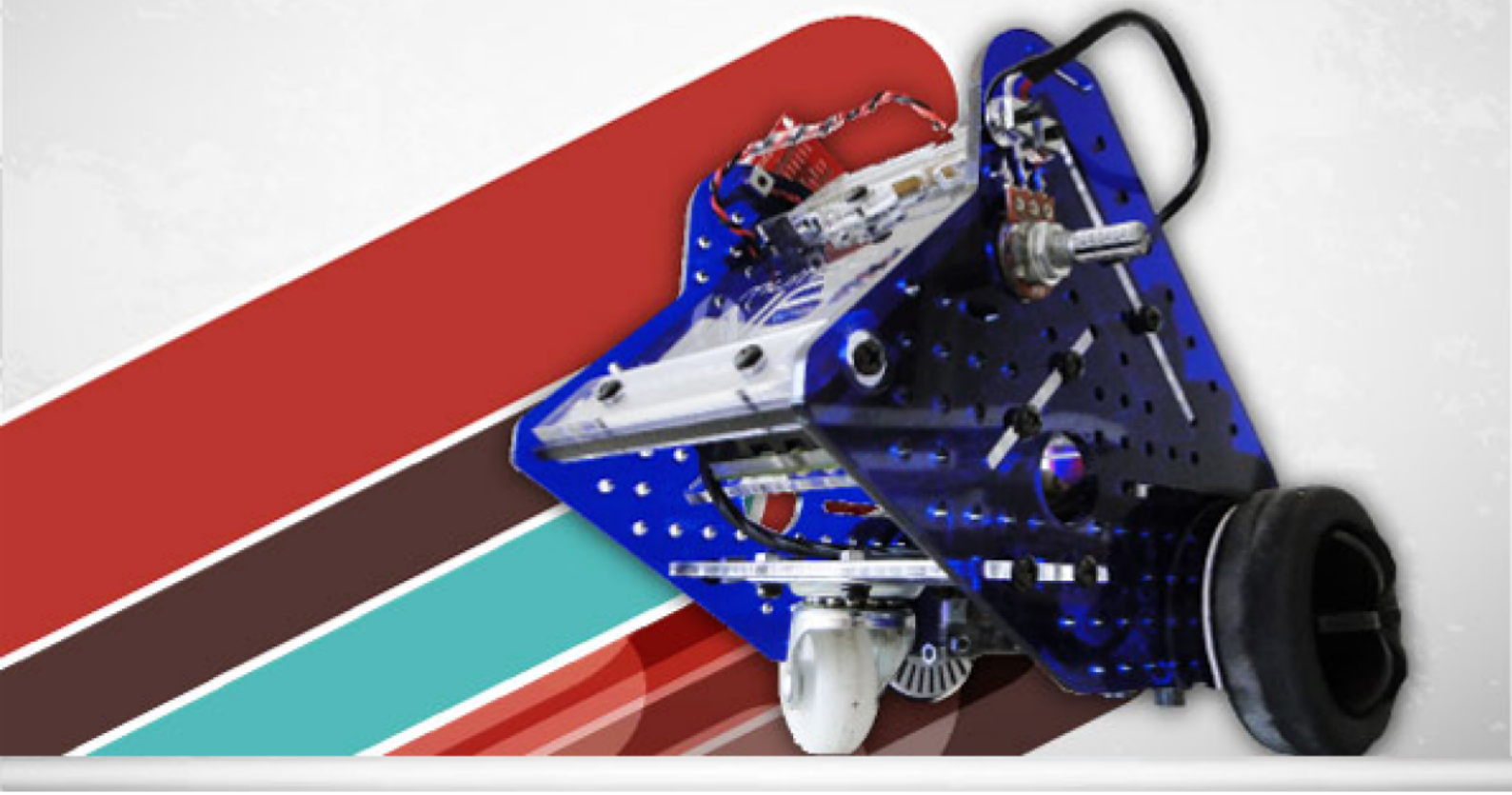
Ejercicio 3.8 Cree la función carrera que reciba 2 robots y los haga avanzar a velocidad máxima durante una cantidad de segundos que se recibe como parámetro. Ayuda: use `forward()` con un sólo argumento y el mensaje `wait()` de alguno de los 2 robots para que ambos avancen al mismo tiempo.

Ejercicio 3.9 Usando las funciones de movimiento realice un programa que recorra un laberinto realizado sobre el piso con obstáculos físicos o con líneas.

Ejercicio 3.10 Para hacer un paso de baile, un robot debe realizar alguna figura, por ejemplo, un círculo, un cuadrado, un triángulo, un 8, una vueltita, etc. Un baile completo entre 2 robots, consiste en una secuencia de varios pasos de baile. Realice un módulo baile. El módulo debe contener los pasos de diferentes bailes, además de algunos bailes predefinidos. Prueben, filmen y suban a nuestro canal de youtube el resultado de este ejercicio.

Capítulo 4

Los valores y sus tipos



Ya vimos cómo mover el robot dibujando algunas figuras. También vimos que, al incorporar el uso de funciones, podemos reutilizar código agrupando instrucciones y construir programas que sean más fáciles de leer. Ahora, en este capítulo, veremos la facilidad que nos aporta el uso de variables, de manera tal de poder utilizar nuestros datos en varias instancias de nuestro código sin necesidad de repetir sus valores, y cómo pedirle al usuario que ingrese datos desde el teclado para usarlos en nuestros programas.

4.1. Utilizando variables

Repasando lo que vimos en el capítulo 3, recordamos que cuando quisimos dibujar un cuadrado, necesitamos probar con distintos valores de velocidad y de tiempo hasta dar con los adecuados para que el robot dibuje la figura deseada. Debido a que debemos escribir estos datos en cada instrucción, cada vez que quisimos probar con valores distintos, debimos modificar los mismos en cada una de sus ocurrencias. Para evitar la repetición de escritura o modificación de datos podemos guardar estos valores para luego utilizarlos tantas veces como necesitemos. ¿Qué usamos para realizar esto? Usamos **variable**.

Una **variable** es un nombre que representa o refiere a un valor.

Seguramente han usado variables en matemáticas, cuando resuelven ecuaciones. ¿Cuál es la idea en ese caso? Las variables son cantidades desconocidas que se nombran con una letra (x, y, z, por ejemplo).

En Python, podemos utilizar variables para representar valores. Por ejemplo:

```
mi_robot = Robot(b, 1)
```

La instrucción anterior la hemos utilizado cada vez que conectamos el robot a la computadora. Para manejar el robot usamos dos variables: *mi_robot* y *b*. La primera, nos representa al robot y la segunda, al módulo de comunicaciones. De esta manera cada vez que necesitamos referenciar al robot, lo hacemos utilizando la variable `mi_robot`.

Usamos variables también cuando enviamos información a las funciones. ¿Se acuerdan? En el capítulo anterior, escribimos funciones que recibían parámetros. Estos parámetros se representan también con variables. Por ejemplo:

```
>>> from duinobot import *
>>> from movimientos import *
>>> b = Board("/dev/ttyUSB0")
>>> robotazo = Robot(b, 1)
>>> mareado(robotazo)
>>> zigzag(robotazo)
```

✎ ¿Cuáles son las variables que usamos en el código anterior? ¿Qué representan cada una?

Asociamos un valor a una variable a través de la **sentencia de asignación**. En Python, esto se representa con el =. Por ejemplo: velocidad=100.

Mediante la asignación podemos almacenar resultados de operaciones matemáticas u operaciones de otro tipo en variables, lo cual nos permite modificar valores rápidamente y de forma más ordenada:

Código 4.1. Guardando datos en variables

```
proporcion = 2
esperar = 2 * proporcion
velocidad = 25 * proporcion
vuelta = velocidad / 2
mi_robot.forward(velocidad, tiempo)
mi_robot.turnRight(vuelta, 1)
wait(esperar)
mi_robot.forward(velocidad, tiempo)
mi_robot.turnRight(vuelta, 1)
wait(esperar)
mi_robot.forward(velocidad, tiempo)
mi_robot.turnRight(vuelta, 1)
wait(esperar)
mi_robot.forward(velocidad, tiempo)
mi_robot.turnRight(vuelta, 1)
wait(esperar)
```

Miremos el ejemplo anterior. Podemos modificar alguna de las 4 variables definidas al principio y alterar así el comportamiento de todo el programa sin necesidad de modificar el código línea por línea.

La forma general de asignarle un valor a una variables en Python es:

```
<nombre_variable> = <valor>
```

Veamos los siguientes ejemplos:

```
>>> velocidad = 20
>>> texto = "Mi primer robot"
```

Los nombres que damos a las variables deben cumplir con las mismas reglas que vimos en la sección 3.3.1 cuando nos referimos a los nombres de las funciones. Estas son:

- Los nombres pueden contener letras del alfabeto inglés (es decir no puede haber ñ ni vocales con tilde).
- Los nombres pueden contener números y guión bajo.
- Los nombres pueden empezar con una letra o un guión bajo (nunca con un número).
- No pueden usarse palabras reservadas como nombre.

✎ Algo importante para tener en cuenta, es que, en Python, hay algunas convenciones que nos parece adecuadas ir adoptando:

- Los nombres de las variables por convención son en minúsculas, para usar varias palabras se unen con guión bajo: nombre_mas_largo o mi_robot
- Las variables con nombre significativos ayudan mucho a la interpretación del código. No es lo mismo utilizar v1 o velocidad como nombre de una variable que representa la velocidad a la que moveremos el robot.
- Como las palabras reservadas del lenguaje no pueden ser nombres de variables, para conocer cuáles son estas palabras reservadas, prueben con las siguientes sentencias:
  import keyword
  print keyword.kwlist

En Python existen distintos tipos de valores. Los más comunes son los números y las cadenas de caracteres (**strings** en inglés). En este último caso, debemos recordar que las cadenas de caracteres **siempre** se encierran entre comillas `""`), como lo hicimos al definir la variable texto. En Python existen otros tipos de datos que no veremos en este libro, pero, a modo de comentario, nosotros usamos objetos que representan al robot y al módulo de comunicaciones. Estos valores no son ni números ni cadenas y volveremos a ver esto en el siguiente curso.

4.1.1. Variables en funciones

Cuando trabajamos con funciones y utilizamos variables, tanto dentro como fuera de las mismas, vamos a definir como **variables globales** a aquellas que se encuentran fuera de las funciones y **variables locales** las que se crean dentro de la función. En el caso de las primeras, pueden utilizarse en cualquier punto de nuestro programa (incluyendo dentro de las funciones) pero, en el caso de las variables locales, solamente pueden utilizarse dentro de la función donde fueron definidas. Veamos algunos ejemplos:

Código 4.2. Alcance de las variables

```
mi_robot = Robot(b, 1)
tiempo = 3
def cuadrado(robot, t):
    esperar = 1
    lado = 50
    vuelta = 35
    robot.forward(lado, t)
```

```
        robot.turnRight(vuelta, 1)
        wait(esperar)
        robot.forward(lado, t)
        robot.turnRight(vuelta, 1)
        wait(esperar)
        robot.forward(lado, t)
        robot.turnRight(vuelta, 1)
        wait(esperar)
        robot.forward(lado, t)
        robot.turnRight(vuelta, 1)
        wait(esperar)

    cuadrado(miRobot, tiempo)
```

En este caso la variable **esperar** sólo se puede usar dentro de la función `cuadrado` y no se puede modificar fuera de la misma. Lo mismo sucede con las variables **lado** y **vuelta**. En cambio, la variable **tiempo** que representa la cantidad de segundos que se moverá el robot, se define fuera de la función y al igual que el robot se pasa como parámetro a la función `cuadrado`.

Si bien, en el ejemplo anterior, definimos dos variables globales, no las accedimos desde dentro de la función, sino que pasamos sus valores como parámetros a la misma. En el siguiente ejemplo, les mostraremos otros escenarios de uso de variables globales y locales.

Código 4.3. Variables globales y locales

```
personaje = "Pinocho"

def imprimirPersonaje():
    print personaje

imprimirPersonaje()

print "Desde afuera de la función también puedo imprimir a : " + personaje
```

Como vimos en este ejemplo, pudimos imprimir el nombre de nuestro personaje, Pinocho, tanto dentro como fuera de la función. ¿Se animan a probarlo?

Ahora veamos qué pasa en el siguiente ejemplo:

Código 4.4. Variables globales y locales

```
def imprimirPersonaje():
    personaje = "Pinocho"
    print personaje

imprimirPersonaje()

print "Desde afuera de la función NO puedo imprimir a : " + personaje
```

¿Lo probaron? ¿Qué pasó? Este código da error. ¿Por qué? El error es intentar utilizar la variable **personaje** fuera de la función. Ahora, esta variable no es más global, sino local a la función **imprimirPersonaje** y, por lo tanto, utilizarla fuera dará un error.

Si bien es totalmente válido acceder a una variable global desde cualquier lugar de nuestro código, es una mala práctica de programación y no es recomendable hacerlo.

4.2. Tipos de datos

Hasta ahora vimos que los valores que manejamos no son siempre iguales y, de acuerdo a estos valores, son las operaciones que podemos hacer con los mismos. Cada variable, de acuerdo al valor que le asignemos tendrá asociado un determinado **tipo de dato**.

Un **tipo de datos** define el conjunto de valores y las operaciones válidas que pueden realizarse sobre esos valores.

- El conjunto de valores representa **todos los valores posibles** que puede llegar a tomar una variable de ese tipo.
- Las operaciones permitidas, son todas aquellas operaciones válidas para los datos pertenecientes a dicho tipo.

Python maneja muchos tipos de datos. Algunos de ellos corresponden a valores simples como ser números y cadenas de caracteres (los tipos de datos que usamos hasta el momento) y otros, corresponden a colecciones de valores, como son las listas. A continuación, les mostramos los tipos de datos utilizados en Python:

- Tipos Básicos
  - Números: Enteros - Flotantes
  - Lógicos (Booleanos)
  - Cadenas de texto (Strings)
- Colecciones
  - Listas
  - Tuplas
  - Conjuntos
  - Diccionarios

Como habrán podido notar, en ningún momento hemos asignado o asociado un tipo a una variable. Simplemente le asignamos un valor y, a partir de este valor, inferimos su tipo.

Antes de utilizar una variables, primero debemos asignarle un valor. Si no es así, Python informa un error.

```
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

¿Qué pasó en este ejemplo? Como la variable x no tiene ningún valor asignado, Python no la conoce aún. Por este motivo, el mensaje de error es "`name 'x' is not defined`", que significa que el nombre "x" no está definido.

Como dijimos anteriormente, el tipo de datos también define qué operaciones están permitidas. Algunas operaciones tienen asociados símbolos que las representan. En el Tabla 4.1 podemos ver algunos de los operadores disponibles en Python y la operación que representan. Vamos a volver sobre estos operadores cuando veamos en detalle cada uno de los tipos con los cuales trabajaremos a lo largo de este primer curso.

Operador	Operación
+	Suma de números o concatenación de cadenas
-	Resta
*	Multiplicación e números o repetición de cadenas
/	División entera o no entera
%	Resto de la división entera
> , <	Comparación mayor - menor
==	Comparación por igualdad

Tabla 4.1. Operadores en Python

Ejemplos	Interpretación	Tipo
1230, -98, 0	Enteros normales	`int`
999999999999L	Enteros largos	`long`
1.45, 3.14e-10, 4E210	Punto flotante	`float`
0177, 0x9ff, 0XFF	Literales octales y hexadecimales	`int`
3+4j, 3.0+4.0j	Números literales complejos	`complex`

Tabla 4.2. Tipos de números enteros y flotantes

4.2.1. Tipos Básicos

Los **números** son los datos que utilizamos diariamente para referirnos a medidas, distancias, etc. Para representar los mismos necesitamos diferentes tipos. Para utilizar números que no necesitan valores decimales, Python posee tipos de números enteros que varían su rango según para lo que se esté empleando, en la Tabla 4.2 podemos ver los tipos de enteros y flotantes disponibles en el lenguaje.

Ya que en el lenguaje Python las variables no se declaran asociándoles un tipo de dato antes de su utilización (a diferencia de otros lenguajes) el tipo de una variable se define en el momento en que le asignamos un valor.

En la Tabla 4.1 se listaron los operadores que puede utilizarse con los datos numéricos. Muchos de estos operadores ya se conocen del área de las matemáticas, como +, -, * y /. Pero, hay algunas cosas interesantes de destacar. Por ejemplo, debemos prestar particular atención al momento de utilizar /, ya que el resultado depende del tipo de los datos que se están usando en el momento de la operación. Si se dividen dos números enteros, el resultado que se obtiene es la parte entera de la división, pero si en cambio uno de los números es del tipo `float` (es decir, que contiene una parte decimal), retorna un valor con punto decimal. Por lo tanto, el valor resultante tendrá tipo `float`. En la Tabla 4.3 veremos algunos ejemplos de uso de los operadores vistos entre números enteros y flotantes.

El orden en que usamos los operadores influyen en el resultado final de la operación. A este orden

Operación	Resultado
5+10	15
10 / (2+3)	2
10 > 5	True
10/3.5	2.8571428571
10/3	3
10%3	1

Tabla 4.3. Ejemplos Operadores matemáticos y comparación en Python

Operadores	Descripción
<, <=, >, >=, ==, !=, x in y	Operadores de comparación
x + y, x - y	Suma y resta
x * y, x % y, x / y	Multiplicación/repetición, resto, división
-x	Negación unitaria

Tabla 4.4. Orden de precedencia de operadores matemáticos

se lo conoce como **precedencia**. Como hemos visto en matemáticas cuando tenemos expresiones con cálculos combinados en los que intervienen operaciones de multiplicación, división y sumas y restas, si no se utilizan paréntesis, primero se resuelven las divisiones y multiplicaciones y luego las sumas y restas. En Python existe una precedencia que hay tener en cuenta al momento de realizar operaciones en la Tabla 4.4 se describe la precedencia de los operadores vistos. Esto es muy importante tener en cuenta, dado que debemos saber exactamente cómo Python resolverá un cálculo matemático.

Otro tipo de datos básico es el **string** o **cadena de caracteres**. Como mencionamos antes, se pueden definir cadenas encerrando el texto entre comillas simples(') o dobles ("). Las cadenas también tiene operaciones asociadas. Veamos el siguiente ejemplo:

```
>>> ciudad='La Plata, '
>>> provincia = "Buenos Aires"
>>> ciudad + provincia
'La Plata, Buenos Aires'
>>> cadena = ciudad*5
'La Plata, La Plata, La Plata, La Plata, La Plata, '
```

¿Ven algo extraño? Estamos usando operadores matemáticos con cadenas y ¡no hubo un error! Esto es porque el símbolo + se utiliza también para la concatenación de cadenas y el símbolo * para la repetición de las mismas. ¿Qué pasa si necesitamos saber, por ejemplo si una cadena comienza con cierta letra, por ejemplo, vamos a necesitar acceder a una posición exacta dentro de la cadena (en nuestro ejemplo, la primera). Para esto, podemos hacerlo utilizando corchetes y un número llamado índice que indica a qué carácter queremos acceder. Se puede pensar una cadena como una sucesión de cajas donde cada una guarda un carácter:

L	a		P	l	a	t	a
0	1	2	3	4	5	6	7

De esta manera, Python almacena los string, reservando una posición para cada carácter correspondiente a la cadena de texto. Por lo tanto, para acceder a un carácter indicado lo haremos indicando su posición.

En Python, el primer carácter de una cadena se encuentra en la posición 0 (CERO).

En nuestro ejemplo, **ciudad**[0] corresponde a la letra "L".

También es posible acceder a fragmentos de una cadena, especificando el principio y el fin de la subcadena. Veamos cómo funciona:

Operación	Resultado
"Hola"+ "!"	"Hola!"
"hola"*3	"holaholahola"

Tabla 4.5. Ejemplos Operadores usando strings

```
>>> ciudad[4]
'l'
>>> ciudad[4:8]
'lata'
>>> ciudad[0]
'L'
```

¿Notaron algo extraño? En la segunda sentencia, **ciudad[4:8]** hay dos cosas interesantes e importantes de recordar. Primero, recordemos que las posiciones comienzan desde 0, por lo tanto el carácter que se indica como índice 4 es el quinto elemento de la cadena. Segundo, el carácter correspondiente al último valor del rango que pasamos, en nuestro caso el 8, **no** se incluye en la subcadena. Veamos un par más de ejemplos:

```
>>> ciudad[1:4]
'a P'
>>> ciudad[3:8]
'Plata'
```

✎ Prueben: ciudad[:] - ciudad[1:] - ciudad[:4]

Si necesitamos conocer cuál es la longitud de una cadena, existe una función que nos la retorna: `len`:

```
>>> len(ciudad)
8
```

✎ Prueben print cadena[len(cadena)-1]. ¿Qué estoy tratando de obtener? ¿Lo obtengo? ¿Por qué?

En la Tabla 4.5 resumimos los operadores vistos con las cadenas.

4.2.2. Conversión de tipos de datos

Como vimos anteriormente las variables no tienen especificado su tipo de datos de forma explícita. Vimos también que algunas operaciones se comportan diferente de acuerdo al tipo de los operandos sobre los cuales se trabaja. Veamos el siguiente ejemplo:

```
>>> dato = 5
>>> divido = dato/2
>>> print divido
2
```

Es así que, en algunas ocasiones necesitamos convertir tipo de una variable a algún otro.

En caso que necesitemos convertir una variable de tipo entero a flotante o viceversa utilizamos las funciones `float()` e `int()`[1]:

[1]Les diremos funciones para no profundizar en el tema de programación orientada a objetos pero en realidad son constructores.

```
>>> dato = 5
>>> divido = float(dato)/2
>>> print divido
2.5
```

Recordemos que el operador / da como resultado un número entero si los dos operandos son enteros, en caso que uno de ellos sea de tipo float no es necesario la conversión para obtener el resultado buscado. Por lo tanto podríamos haber obtenido el valor en número flotante simplemente dividiendo por 2.0 en vez de 2.

```
>>> dato = 5
>>> divido = dato/2.0
>>> print divido
2.5
```

Otra conversión disponible es con los datos de tipo string, como ya vimos podemos guardar una cadena de texto compuesta por letras y números, en caso que se necesite un número ya almacenado con algún tipo de dato numérico podemos convertirlo a string o viceversa:

```
>>> cadena = str(divido)
>>> print ('Valor necesario '+cadena)
Valor necesario 2.5
>>> actual = '2012'
>>> dato = int(actual)-1976
>>> print 'Edad', dato
Edad 36
```

Ya veremos más adelante la importancia de estas funciones predefinidas en Python para convertir los datos ingresados por teclado.

4.2.3. Las Listas: las colecciones más simples

Como la mayoría de los lenguajes de programación, Python maneja varios tipos de colecciones de datos. Las listas son quizás las más sencillas de utilizar.

Una **lista** es una colección ordenada que puede contener cualquier tipo de datos, inclusive otras listas.

Miremos los siguientes ejemplos de listas en Python:

```
>>> lista_vacia = []
>>> lista_de_pcias = ['Entre Ríos', 'Corrientes', 'Misiones']
>>> lista_de_muchas_cosas = [23, 'Hola', [6,7,8,]]
```

En el ejemplo anterior, podamos notar que las listas pueden contener 0 elementos (lista_vacia), o pueden contener elementos del mismo tipo (lista_de_pcias) o de distintos (lista_de_muchas_cosas). En este último caso, la lista contiene un número, una cadena de caracteres y otra lista como elementos.

Al igual que en el caso de las cadenas de caracteres, para acceder a los elementos de la lista usamos los corchetes y la posición del elemento al que necesitamos acceder. Y, de igual manera que con las cadenas de caracteres, los elementos se numeran de 0 en adelante.

```
>>> print lista_de_pcias[1]
```

4.2.4. Trabajando con listas

Las listas son estructuras dinámicas. Es decir, que podemos agregar y quitar elementos a medida que así lo necesitemos. Supongamos que tenemos la lista con los siguientes elementos:

```
>>> lista = [43, 'Buenos Aires', [3,7,9]]
```

Para agregar un elemento al final de la lista utilizamos la sentencia `append`

```
>>> lista = [43, 'Buenos Aires', [3,7,9]]
>>> lista.append(5)
>>> print lista
[43, 'Buenos Aires', [3, 7, 9], 5]
```

En caso que necesitemos insertar un elemento en una posición determinada utilizamos la sentencia `insert` indicando como primer argumento la posición y como segundo argumento el elemento a insertar:

```
lista.insert(posicion,elemento_nuevo)
```

En nuestro caso veamos la inserción del número **333** en la posición 2 de la lista:

```
>>> lista = [43, 'Buenos Aires', [3,7,9]]
>>> lista.insert(2,333)
>>> print lista
[43, 'Buenos Aires', 333, [3, 7, 9]]
```

Para eliminar elementos de una lista Python provee la sentencia `remove`, la cual elimina el primer elemento de la lista que coincida con el valor pasado como argumento.

```
>>> lista.remove(333)
```

Otra forma de eliminar elementos es utilizando la sentencia `pop(i)`, pero a diferencia de `remove` elimina el elemento que se encuentra en la posición **i**, o bien se puede utilizar sin argumentos, con lo cual elimina el último elemento.

```
>>> lista = [43, 'Buenos Aires', 333, [3, 7, 9]]
>>> lista.pop(2)
333
>>> lista = [43, 'Buenos Aires', [3,7,9]]
>>> lista.pop()
[3, 7, 9]
>>> print lista
[43, 'Buenos Aires']
```

Para conocer la cantidad de veces que aparece un elemento en una lista contamos con la sentencia `count`

```
>>> print lista
[43, 'Buenos Aires', 5, 57, 5, 'era', 55, 'era']
>>> lista.count(5)
2
```

Para conocer la cantidad de elementos que contiene la lista recurrimos a la función `len` que vimos con las cadenas:

```
>>> lista = [43, 'Buenos Aires', 5, 57, 5, 'era', 55, 'era']
>>> len(lista)
8
```

A modo de resumen les dejamos la Tabla 4.6 con las operaciones básicas de listas.

Operación	Descripción	Ejemplo
append	Agrega un elemento a la lista	lista.append(10)
remove	Elimina la primera ocurrencia del elemento dado como parámetro	lista.remove(10)
pop	Elimina y retorna el elemento ubicado en la posición dada como parámetro	lista.pop(1)
count	Retorna la cantidad de ocurrencias del elemento dado como parámetro	lista.count(1)
len	Retorna la cantidad de elementos de la lista	len(lista)

Tabla 4.6. Operaciones básicas de listas

4.3. Ingreso datos desde el teclado

Hasta ahora, todos los valores que utilizamos en los ejemplos fueron escritos en el texto de nuestros programas. La realidad es que en muchos casos estos valores no se conocen de movida y es el usuario de nuestros programas quien los ingresa desde el teclado. Para ingresar un valor desde el teclado utilizaremos la función `raw_input`. Con esta función solicitamos los valores en el momento de ejecución.

Código 4.5. Solicitando valores por teclado

```
mi_personaje = raw_input("¿Cuál es tu personaje favorito?")
def imprimirPersonaje(personaje):
  print personaje

imprimirPersonaje(mi_personaje)
```

Como pudieron observar, la función `raw_input` puede recibir un texto como argumento que se muestra en el momento que se solicita el ingreso de datos. Este texto es opcional, puedo no querer mostrar ningún texto y, simplemente usarla de la siguiente manera:

```
print "¿Cuál es tu personaje favorito?"
mi_personaje = raw_input()
def imprimirPersonaje(personaje):
  print personaje

imprimirPersonajel(mi_personaje)
```

Algo muy importante es que `raw_input` retorna siempre una cadena de caracteres. Miren el siguiente ejemplo:

```
>>> mi_edad = raw_input("¿Cuál es tu edad?")
"¿Cuál es tu edad?" 23
>>> print mi_edad
23
>>> print "El doble de tu edad es ", mi_edad*2
El doble de tu edad es 2323
```

¿Qué pasó? El problema es que mi_edad es una cadena de caracteres y Python utilizó el operador * como operador de repetición de cadenas. Si necesito el valor numérico de esta cadena, deberé utilizar las funciones de conversión que mencionamos antes (**int**).

```
>>> mi_edad = raw_input("¿Cuál es tu edad?")
"¿Cuál es tu edad?" 23
>>> print mi_edad
23
```

```
>>> print "El doble de tu edad es ", int(mi_edad)*2
El doble de tu edad es 46
```

Otra función que podemos utilizar para ingresar datos desde el teclado es `input`, pero la misma genera en algunas ocasiones errores de lectura de lo ingresado por teclado. Para evitar esto, usaremos siempre `raw_input` y en caso que se esté solicitando un tipo de datos diferente a string se lo convertirá que ya se explicó en la sección 4.2.2. Veamos otro ejemplo:

Código 4.6. Ingresando un valor entero desde el teclado

```
print "Ingrese los valores necesarios para hacer el cuadrado"
velocidad = int(raw_input("velocidad: "))
vuelta = int(raw_input ("Velocidad para dar la vuelta: " ))
def cuadrado(robot, lado, esquina):
  tiempo = 3
  robot.forward(veloc, tiempo)
  robot.turnRight(esquina, 2)
  wait(esperar)
  robot.forward(lado, tiempo)
  robot.turnRight(esquina, 2)
  wait(esperar)
  robot.forward(lado, tiempo)
  robot.turnRight(esquina, 2)
  wait(esperar)
  robot.forward(lado, tiempo)
  robot.turnRight(esquina, 2)
  wait(esperar)

cuadrado(robot, velocidad, vuelta)
```

4.4. Imprimiendo por pantalla

Ya usamos en muchas ocasiones la función `print` para mostrar en la pantalla datos. En esta sección, veremos la sentencia `print` con mayor detalle. En forma general se indica la expresión luego de la sentencia:

```
>>> x = 3
>>> print x
3
```

Si estamos trabajando en el intérprete de Python, también podemos utilizar el nombre de la variable directamente sin la sentencia `print` delante:

```
>>> x
3
```

En caso que se necesite mostrar un texto determinado recurrimos a las `" "` o `' '` para abarcar el texto:

```
>>> print "x"
x
```

✎ En Python es indistinto el uso de las comillas simples `' '` o las comillas dobles `" "` dentro de la sentencia `print`

También podemos imprimir varias variables o expresiones con una única sentencia `print`:

```
>>> x = 3
>>> y = 4
>>> print "Los valores a sumar son:",x,y, " y es su suma es",x+y
Los valores a sumar son: 3 4  y es su suma es 7
```

En este caso separamos el texto que queremos mostrar de las variables utilizando la **coma**, como vemos en ejemplo. De la misma manera para agregar texto luego de las variables nuevamente utilizamos **coma**. Como vimos en el ejemplo anterior las variables se pueden concatenar con texto, realizar operaciones entre las mismas en la misma sentencia y, de ser necesario, cambiar el tipo de las variables utilizando las funciones de conversión.

```
>>> print "x"+str(y)
```

Si necesitamos mostrar una cadena que contenga múltiples líneas de texto, utilizaremos comillas triples `''' '''`,

```
>>> print('''Ingrese la acción que desea realizar
      f: hacia adelante
      b: hacia atrás
      r: doblar a la derecha
      l: doblar a la izquierda
      s: salir
     ''')
```

Se debe tener en cuenta que todos los espacios con las que comienzan las líneas también se mostrarán, ya que se interpreta el texto tal cual aparece.

✎ En la versión 3 de Python la sentencia `print` sólo puede utilizarse como función: `print()`, lo que implica invocarla con los datos a imprimir entre paréntesis.

4.5. Resumen

En este capítulo vimos el uso y asignación de variables como así también algunos de los diferentes tipos de datos que pueden utilizarse. Aprendimos que Python nos provee distintos tipos (**int**, **float** y **long**) para trabajar con valores numéricos y **str** para trabajar con cadenas de caracteres. Estos tipos, nos definen qué valores y qué operaciones tenemos disponibles para trabajar de acuerdo al tipo que usemos. Hemos utilizado la sentencia **raw_input()** para solicitar al usuario que nos ingrese datos desde el teclado y la sentencia **print** para mostrar datos en la pantalla.

4.6. Actividades

Ejercicio 4.1 Modificar la función zigzag realizada anteriormente reemplazando todos los valores por variables.

Ejercicio 4.2 Modifique el ejercicio anterior para que la velocidad de los movimientos sea ingresada por teclado.

Ejercicio 4.3 Analice el siguiente código y mencione si debería dar algún tipo de error

```
velocidad=raw_input("ingrese la  velocidad de marcha")
robot.forward(velocidad, 2)
```

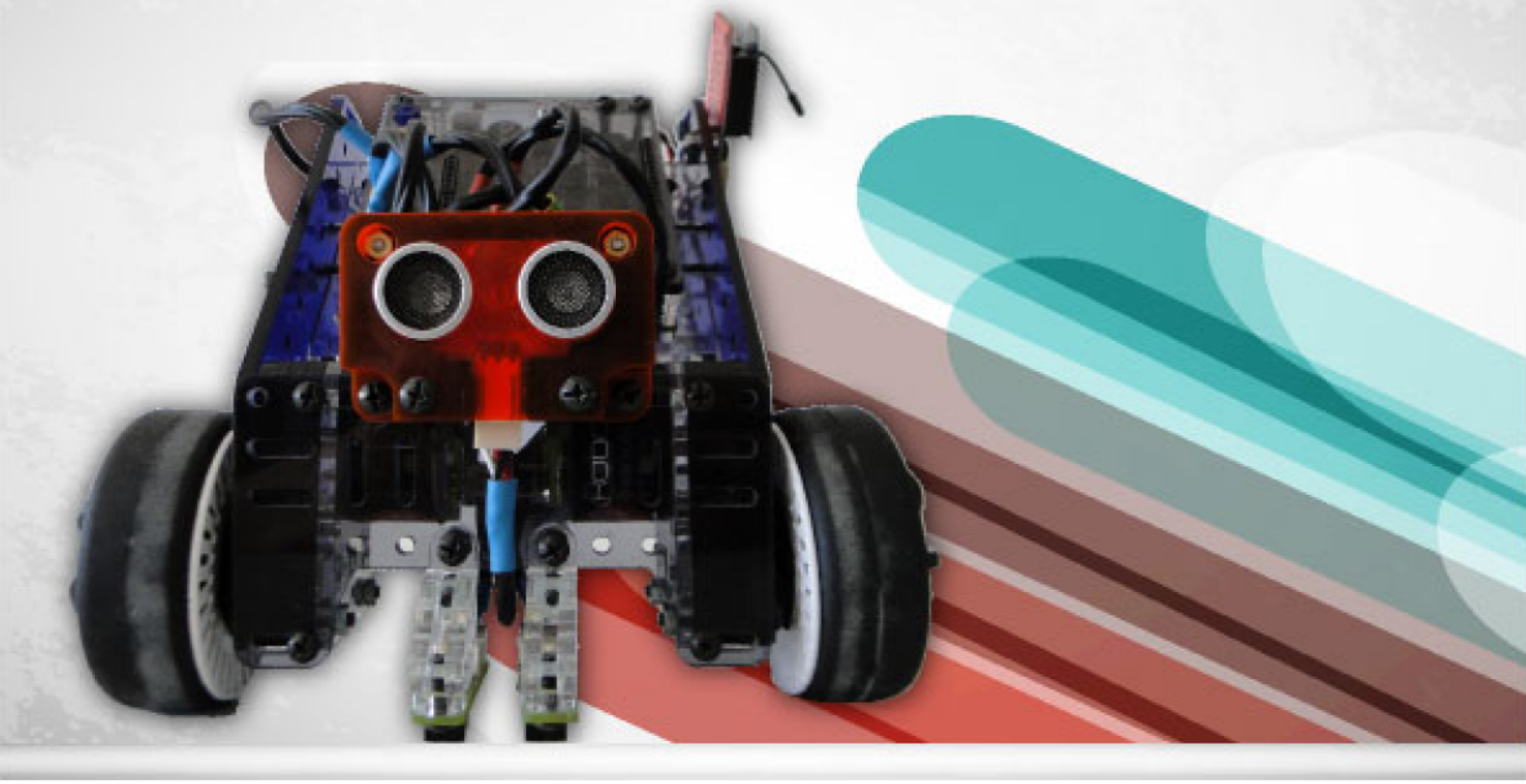
Ejercicio 4.4 Corrija el código anterior para que funcione correctamente.

Ejercicio 4.5 Volvamos al laberinto planteado en capítulos anteriores. Como aún no hemos utilizado los sensores del robot, para resolver el mismo, debíamos mover al robot por un camino conocido durante un cierto tiempo y a una velocidad dada. Arme una lista con los valores de velocidad y tiempo utilizados y modifique el programa realizado para que acceda a estos datos.En algunos capítulos más adelante volveremos sobre este ejercicio.

Ejercicio 4.6 Realicen un programa que pida al usuario que ingrese el nombre del robot por teclado y se lo asigne.

Capítulo 5

Robots que deciden



Hasta ahora, nuestros robots siguieron órdenes estrictas de acuerdo a una secuencia de instrucciones preestablecidas. En ningún momento fueron capaces de tomar ningún tipo de decisión en base al planteo de una situación dada. A partir de ahora, vamos trabajar para dotarlos de cierta "inteligencia" para que puedan ser capaces de decidir ante ciertas circunstancias.

Para poder decidir, es necesario que se puedan expresar distintas situaciones de manera tal que nos podamos preguntar su "valor de verdad". Esto es, poder escribir expresiones que representen preguntas tales como: ¿esta situación está pasando?, ¿es cierto que hay un obstáculo delante?, ¿ya recorrimos 10 cm?, etc. En este capítulo veremos cómo expresar estas situaciones o condiciones y, en base a si se cumplen o no, tomar las decisiones necesarias.

5.1. Los valores de verdad en Python

En Python existe un tipo cuyos valores representan **valores de verdad**. Este tipo se denomina Booleano y sus valores pueden ser `True` o `False` que representan a los valores verdadero y falso. Las variables booleanas son tan simples que sólo pueden tomar estos dos valores, sin embargo resultan muy útiles para indicar al robot qué decisiones tomar. Antes de profundizar en este tema, es muy importante conocer algunos aspectos básicos de lógica, dado que el robot deberá tomar decisiones de acuerdo a ciertas condiciones que se les presente y debemos saber exactamente cómo las evaluará.

Googleame: Buscá información sobre quién es George Boole y por qué se les dicen booleanos a los tipos de datos que representan valores de verdad.

En esta sección veremos cómo escribir expresiones que devuelven valores de verdad y cómo combinar distintos valores de verdad para crear otros más complejos.

5.1.1. Expresiones simples

Seguramente ya han tratado con este tipo de expresiones, aunque no sabían cómo se llamaban. Veamos los siguiente ejemplos:

```
>>> 5 < 10
True
>>> 5 > 10
False
>>> 5 + 1 > 5
True
>>> 2 * 20 == 41
False
>>> "Gato" < "Pato"
True
>>> "Gato" == "Pato"
False
>>> "Gato" != "Pato"
True
```

Cuando comparamos dos números, en nuestro ejemplo 5 y 10, el resultado es un valor de verdad: verdadero o falso, de acuerdo a la comparación realizada. Todos sabemos que es verdad (True) que 5 es más chico que 10. Cuando comparamos cadenas de caracteres, usamos el orden alfabético de los caracteres para compararlos. Así decimos que "Gato" es más chico que "Pato", dado que la letra "G" está antes en el abecedario que la letra "P".

✎ Prueben qué pasa con el siguiente ejemplo. Analicen el resultado.

```
>>> "gato" < "Pato"
False
```

¿Qué pasó?

Como habrán visto, hay varios operadores que me permiten comparar valores y, en base a dicha comparación, retornar el valor de verdad correspondiente. La Tabla 5.1. muestra la lista de operadores de comparación que pueden usarse en Python y en la Tabla 5.2 les mostramos algunos ejemplos del uso de dichos operadores y el valor de verdad que retornan al evaluarse.

5.1.2. Operadores lógicos

Hasta ahora, los operadores presentados sólo nos permiten evaluar o consultar por una única condición. Por ejemplo: ¿la velocidad es menor que 100? (velocidad<100) o el ¿tiempo es menor a 5? (tiempo<5). ¿Qué pasa si se nos plantea una situación donde el robot deba decidir actuar en función de que la velocidad no sea la máxima (es decir, que sea menor que 100) y el tiempo no supere los 5

Operador	Descripción	Ejemplo
==	Verifica si el valor de dos operadores son iguales o no, si los valores son iguales la condición es verdadera	(n == 4)
!=	Verifica si el valor de dos operadores son distintos o no, si los valores son distintos la condición es verdadera	(n != 0)
<>	Similar a != (no se puede usar en Python 3)	(n <> 0)
>	Verifica si el operando de la izquierda es mayor que el de la derecha	(n > 0)
<	Verifica si el operando de la izquierda es menor que el de la derecha	(n < 0)
>=	Verifica si el operando de la izquierda es mayor o igual que el de la derecha	(n >= 0)
<=	Verifica si el operando de la izquierda es menor o igual que el de la derecha	(n <= 0)

Tabla 5.1. Operadores de comparación

Operación	Valor de la expresión
'dos' > 'cero'	True
len('diez') < len('cinco')	True
3+5 > 10	False
3+5 == 8	True
'dos'=='dos'	True
'dos'!='tres'	True

Tabla 5.2. Ejemplos operadores de comparación

segundos (tiempo<5). En este caso, es evidente que con los operadores de comparación vistos hasta ahora no es suficiente dado que tenemos que verificar dos condiciones a la vez.

Sabemos resolver el problema por partes, pero nos faltan herramientas para combinar los resultados. Si evaluamos ambas partes de nuestra condición y ambas son verdaderas (True), entonces podemos decir que la condición general se cumpliría, pero si alguna (o ambas) no lo fuera, la condición general dejaría de serlo también.

Existe un operador lógico que nos permite evaluar situaciones de estas características: el `and` (o conjunción). Así, si queremos saber si dos condiciones son simultáneamente verdaderas, como en el caso anterior, podemos escribir:

```
>>> velocidad < 100 and tiempo<5
```

Así como existe el operador `and`, existe otro operador que nos permite expresar situaciones en las cuales sólo nos interesa saber si alguna (o ambas) de las expresiones son verdaderas, con que una de ellas lo sea, es suficiente. Este operador es el `or`. Veamos el siguiente ejemplo y las diferencias con el anterior.

```
>>> velocidad < 100 or tiempo<5
```

En este caso, si el robot va a una velocidad menor a 100 "o" el tiempo es menor a 5, el valor de verdad de la expresión general será verdadero. Sólo la consideramos falsa, cuando ambas condiciones sean falsas. En algunas situaciones queremos expresar una condición, negando una situación dada. Por ejemplo, podríamos querer expresiones tales como: ¿la velocidad no es máxima?, o ¿no llegué al final del camino? Para expresar estas condiciones, Python provee el operador **not**. Este operador, niega el valor de verdad del operando. Veamos el siguiente ejemplo:

Operador	Descripción	Ejemplo
not	revierte el valor de la expresión	not (n <> 0)
and	si ambas expresiones son verdaderas devuelve verdadero	(n == 4) and (n > 10)
or	si ambas expresiones son falsas devuelve falso	(n == 4) or (n > 10)

Tabla 5.3. Operadores lógicos

p	q	p and q
V	V	V
V	F	F
F	V	F
F	F	F

Tabla 5.4. Tabla de verdad de la conjunción

```
>>> fin_camino=False
>>> not fin_camino
True
```

Como pueden observar, "negar" un valor de verdad True, retorna el valor False y viceversa.

En la Tabla 5.3 se describen los operadores lógicos ordenados en orden decreciente de precedencia.

Con los operadores `and`, `or` y `not` podemos construir expresiones que representen cualquier pregunta que se nos ocurra. Para aplicar correctamente estos operadores, podemos valernos de sus tablas de verdad 5.4 5.5 5.6, que indican cuál es el resultado de cada operación para cada uno de los posibles valores de verdad de sus operandos.

Cuando se combinan varios operadores hay que tener en cuenta en que orden se aplican (es decir la precedencia de cada operador), en el siguiente ejemplo se puede observar como se puede cambiar el resultado de una expresión usando paréntesis:

```
>>> a = True
>>> b = False
>>> not a and b
False
>>> not (a and b)
True
>>> not b and a
True
```

En la última expresión del ejemplo anterior, se aplica primero el `not` a la variable "b" (porque tiene mayor precedencia) y luego el `and` entre el resultado del `not` y la variable "a".

Por supuesto, también pueden usarse los operadores de comparación que vimos antes y combinarlos con los operadores lógicos. En general no hace falta usar paréntesis porque la precedencia

p	q	p or q
V	V	V
V	F	V
F	V	V
F	F	F

Tabla 5.5. Tabla de verdad de la disyunción

p	not p
V	F
F	V

Tabla 5.6. Tabla de verdad de la negación

de las comparaciones es mayor que la de los operadores lógicos, por lo que primero va evaluar las comparaciones:

```
>>> 5 < 20 and "pepe" != "paco"
True
>>> 12 * 2 == 24 or False
True
>>> int("20") == 20
True
>>> "20" == 20
False

>>> def es_par(numero):
...     return numero % 2 == 0
...
>>> es_par(4)
True
>>> es_par(7)
False
>>> es_par(7) and 7 < 20
False
>>> es_par(4) and 7 < 20
True
```

✎ Recordemos que el operador `%` representa el resto de la división entera. Por lo tanto, la función `es_par` retornará `True` si el resto de dividir el numero pasado como parámetro por 2 es 0 o no. Con lo cual, deducimos que se trata de un número par o impar.

En algunas líneas del ejemplo anterior se introducen operaciones matemáticas mezcladas con operadores de comparación y operadores lógicos, las operaciones matemáticas tienen aún más precedencia que los operadores de comparación y, por lo tanto, se evalúan antes que los mismos. Analicemos el siguiente código:

```
>>> 12 * 2 == 24
True
```

Muchas veces, por una cuestión de claridad es mejor utilizar los paréntesis cuando queremos dejar expresadas nuestras condiciones. Quizás es más claro (aunque innecesario su uso en Python) escribir la expresión de la siguiente manera:

```
>>> (12 * 2) == 24
True
```

Y, como verán, el resultado es exactamente el mismo. Aunque los paréntesis sean innecesarios ayudan a la legibilidad del código y esto es una buena práctica de programación.

Si queremos expresar algo distinto, entonces sí es necesario usar los paréntesis:

```
>>> 12 * (2 == 24)
False
```

Veamos la expresión anterior. ¿Notan algo extraño? Si se aplican las precedencias que les presentamos antes, estamos comparando `2 == 24` lo cuál obviamente es falso y, al resultado de esto que es `False`, lo multiplica por 12. Aunque parezca raro, en Python esto no da un error (piensen que sería lógico pensar en un error ya que estamos multiplicando un valor booleano con un entero). En Python, el valor de verdad `False` es equivalente al entero 0 y el `true` a 1. Por lo tanto multiplicar `False` por cualquier número dará `False` (¡¡¡es como multiplicar por 0!!!).

5.2. Condicionando nuestros movimientos

Ahora que ya vimos cómo podemos plantear condiciones, podemos ver de qué manera nuestro robot podrá tomar decisiones. Estas decisiones pueden estar dadas por datos ingresados desde el teclado, o datos recibidos desde los sensores propios del robot. Para indicarle al robot que realice una u otra instrucción de acuerdo a si una condición es verdadera o falsa, utilizamos una sentencia de Python que se denomina `if`.

La sentencia `if` es una estructura de control que evalúa una condición, y ejecuta un conjunto de instrucciones en caso que condición sea verdadera u (opcionalmente) otro conjunto de instrucciones si la condición es falsa.

Veamos cómo utilizamos esta sentencia en Python. Vamos a escribir una función que reciba la velocidad a la que quiero que el robot se mueva, pero, siendo 20 la velocidad mínima de movimiento. Veamos cómo expresamos esto en un programa Python.

Código 5.1. Avanzando...

```
def avanzar(robot, velocidad, tiempo):
    if velocidad < 20:
        robot.forward(20, tiempo)
    else:
        robot.forward(velocidad, tiempo)
```

En el ejemplo anterior, si se usa la función `avanzar()` para mover el robot, no se lo podrá hacer avanzar a una velocidad inferior a 20 aunque lo invoquemos con una velocidad menor.

La forma general de la sentencia `if` es:

Código 5.2. Definición del if

```
if condición:
  instrucciones por evaluación verdadera
else:
  instrucciones por evaluación falsa
```

La condición la expresamos usando expresiones booleanas (como las que vimos al comienzo de este capítulo) que pueden contener operador relacionales y lógicos.

Cada "rama" del `if` termina con **:** y las instrucciones asociadas se encuentran indentadas. Ambas características son obligatorias. En caso de no respetarlas, Python indicará un error de sintaxis.

Podemos usar varios `if` para tomar decisiones que no estén relacionadas, y como en este caso en particular no precisamos ninguno de los bloques `else` directamente no los escribiremos:

Código 5.3. Avanzar controlando la variable tiempo

```
def avanzar_limitado(robot, velocidad, tiempo):
  if tiempo > 1):
    print  "Más de 10 segundos es demasiado, asumo que elegiste 10"
    tiempo = 10
  if velocidad > 100:
    print str(velocidad) + " no es una velocidad válida, asumo que elegiste
        100"
    velocidad = 100

  robot.forward(velocidad, tiempo)
```

Hasta ahora, sólo podríamos plantear dos posibilidades o dos caminos seguir. Muchas veces esto no nos alcanza. Supongamos que queremos dejar que el usuario elija qué hacer con el robot a través de un menú de opciones. Por ejemplo:

Indicanos hacia qué dirección te gustaría mover el robot:
1.- Girar a la derecha
2.- Girar a la izquierda
3.- Avanzar
4.- Retroceder
5.- Dejarlo en la misma posición
Opción:

Una forma de plantear esto en Python es utilizando una versión de la sentencia `if` que permite plantear múltiples caminos. La forma general es:

```
if <condicion1>:
  Instrucciones
elif <condicion2>:
  Instrucciones
elif <condicion3>:
  Instrucciones
elif <condicion4>:
  Instrucciones
else:
  Instrucciones
```

El `else` final nos permite considerar todas las posibilidades que no fueron contempladas en las anteriores condiciones. De esta manera, nuestro ejemplo podría implementarse de la siguiente manera:

Código 5.4. Elijo hacia donde avanzar

```
print "Indicanos hacia qué dirección te gustaría mover el robot:"
print "1.- Girar a la derecha"
print "2.- Girar a la izquierda"
print "3.- Avanzar"
print "4.- Retroceder"
print "5.- Dejarlo en la misma posición"
opcion=raw_input("Opción:")
if opcion==1:
  robot.turnRight(100,1)
elif opcion==2:
```

```
    robot.turnLeft(100,1)
  elif opcion==3:
    robot.forward(100,1)
  elif opcion==4:
    robot.backward(100,1)
  else:
    robot.stop()
```

O podríamos haber definido una función que reciba la orden (de acuerdo a la opción elegida) y mueva el robot acorde a dicha orden:

Código 5.5. Elijo hacia donde avanzar con función

```
def mover(robot, accion):
    if accion == "avanzar":
        robot.forward(100, 1)
    elif accion == "retroceder":
        robot.backward(100, 1)
    elif accion == "derecha":
        robot.turnRight(100, 1)
    else:
        robot.turnLeft(100, 1)

mover("avanzar")
mover("izquierda")
mover("derecha")
mover("retroceder")
```

¿Cuáles son las todas las órdenes posibles? ¿Qué pasa si introduzco la orden: “Doblar”?

5.3. Resumen

En este capítulo aprendimos a trabajar con otro tipo de valores: los valores lógicos o **booleanos** y las formas de combinar estos valores para expresar condiciones más complejas.

También vimos cómo utilizar la sentencia **if** (con sus distintas variantes) que permite que un programa escrito en Python ejecute un conjunto de acciones si se cumple una determinada condición u otro conjunto distinto si la condición no se cumple.

5.4. Actividades

Ejercicio 5.1 Implemente la función `bailar` que recibe el tipo de baile que el robot realizará. Para esto, defina tres tipos de baile (con sus movimientos asociados) implementándolos en tres funciones distintas. La función `bailar` deberá invocar las otras funciones de acuerdo al parámetro recibido.

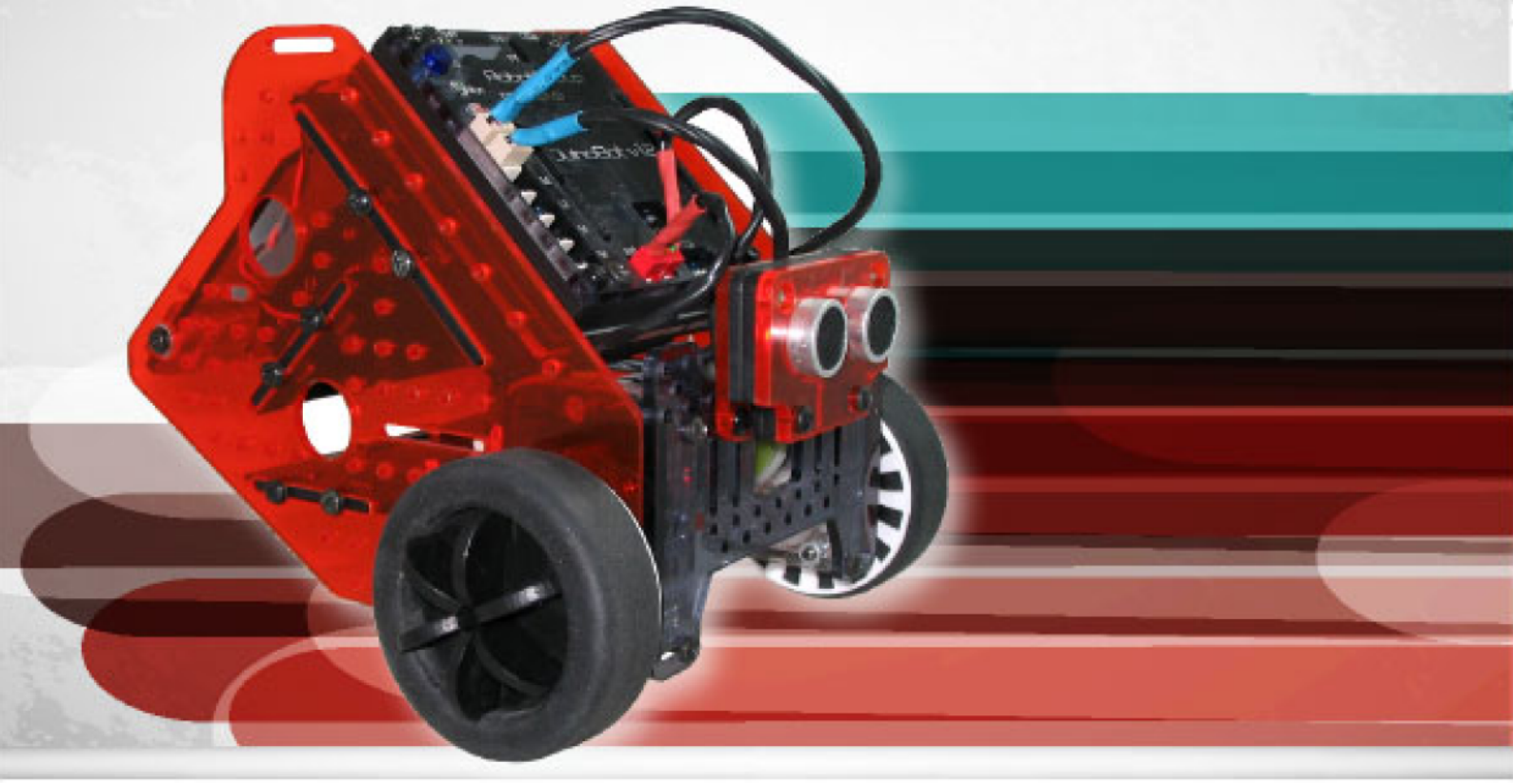
Ejercicio 5.2 Implemente la función `entonando_raro` que recibe un número y si el número es par o es mayor que 100 el robot debe hacer un beep agudo, caso contrario el robot debe hacer un beep grave.

Ejercicio 5.3 Vamos a plantear distintos estados de ánimo para nuestro robot. Vamos a suponer que si el robot está contento, avanzará a velocidad máxima y dará dos vueltas, si se encuentra enojado, hará esto mismo pero hacia atrás y si se encuentra deprimido sólo avanzará hacia adelante a na velocidad mínima. Escriba las tres funciones que implementen estos tres estados y un menú de opciones por el cual el usuario elegirá qué estado de ánimo tiene el robot. De acuerdo a la opción elegida es cómo se moverá.

Ejercicio 5.4 Modifique la función `mover` del último ejemplo, para que sólo se acepten las siguientes órdenes: avanzar, retroceder, izquierda y derecha. En caso de recibir una orden no incluida en la lista, el robot debe emitir un sonido y se debe indicar en la pantalla que la orden no es correcta.

Capítulo 6

Simplificando pasos



Hasta ahora hemos aprendido a codificar funciones y a escribir programas de manera tal que nuestro robot pueda tomar decisiones. Hay mucho más que aprender, tanto sobre el uso del robot, como de Python. En este capítulo vamos a trabajar con otras estructuras de control de Python que nos permitirán mejorar los programas que realizamos hasta el momento y plantear otros nuevos que se acercarán más a la programación real. En particular, vamos a trabajar con las sentencias iterativas o de repetición. Recordemos nuestro primer programa con el robot. ¿Se acuerdan? Hicimos que el robot dibuje un cuadrado. El programa que planteamos era el siguiente:

Código 6.1. Haciendo un cuadrado

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

mi_robot.forward(50, 0.5)
wait(1)
```

```
mi_robot.turnRight(35, 1)

mi_robot.forward(50, 0.5)
wait(1)
mi_robot.turnRight(35, 1)

b.exit()
```

¿Notan algo con respecto a este código? Hay varias secciones que se repiten. En realidad, es lógico que sea así, dado que para construir un cuadrado se debe dibujar cada lado exactamente de la misma manera.

En muchas ocasiones vamos a trabajar con iteraciones. Pensemos también en el menú de opciones del capítulo anterior:

Código 6.2. Elijo hacia donde va

```
print "Indicanos hacia qué dirección te gustaría mover el robot:"
print "1.- Girar a la derecha"
print "2.- Girar a la izquierda"
print "3.- Avanzar"
print "4.- Retroceder"
print "5.- Dejarlo en la misma posición"
opcion=raw_input("Opción:")
if opcion==1:
  robot.turnRight(100,1)
elif opcion==2:
  robot.turnLeft(100,1)
elif opcion==3:
  robot.forward(100,1)
elif opcion==4:
  robot.backward(100,1)
else:
  robot.stop()
```

En realidad el código anterior sería mucho más útil y real si permitiéramos que el usuario utilice este menú varias veces, inclusive ante un ingreso de una opción equivocada, que tenga la opción de volver a ingresarla. ¿Cómo podríamos escribir esto? Obviamente este caso es más complicado que el anterior. En el ejemplo del cuadrado, sabemos exactamente cuántas veces queremos repetir nuestro código en cambio en este ejemplo no lo sabemos.

Para plantear estas situaciones contamos con dos sentencias que nos permiten codificar iteraciones. Éstas son: la sentencia `for` y la sentencia `while`. A lo largo de este capítulo veremos ejemplos de usos de cada una.

6.1. Sentencia de iteración `for`

Pensemos en el ejemplo del cuadrado. Aquí sabemos exactamente cuántas veces tenemos que repetir la secuencia de instrucciones que nos permite dibujar un lado y girar. Esto es, debemos repetir cuatro (4) veces ese código. Para implementar esto usaremos la sentencia `for`.

En general, la forma de usar esta sentencia es la siguiente:

```
for <variable> in <lista>:
  <instrucciones>
```

La *variable* tomará cada uno de los valores de la *lista* y en cada paso, ejecutará el conjunto de *instrucciones*. Esto se repetirá hasta que no queden más valores en la lista.

✎ Algo muy importante a tener en cuenta que las instrucciones que queremos que se repitan deben estar indentadas (con el mismo margen) dentro del `for`.

En el ejemplo anterior, podríamos escribir nuestro programa de la siguiente forma:

Código 6.3. Usando for para dibujar un cuadrado

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

for i in [1,2,3,4]:
  mi_robot.forward(50, 0.5)
  wait(1)
  mi_robot.turnRight(35, 1)

b.exit()
```

Como pueden ver, esto es muy práctico y nos permite simplificar nuestros programas. Veamos otro ejemplo donde es muy útil esta sentencia:

```
for x in 'Entre Ríos':
  print x
E
n
t
r
e

R
i
o
s
```

En este caso, la variable `x` toma todos los valores de la cadena de caracteres "Entre Ríos".

Igualmente, seguimos teniendo un problema con esta estructura: **la cantidad de veces que se itera depende de la lista de valores que planteemos**.

Pensemos en el siguiente problema: vamos a inventar un nuevo baile para nuestro robot. En el mismo, el robot debe ir hacia adelante y hacia atrás y luego girar, repitiendo esto 3 veces. Una posible implementación de esto, usando la sentencia `for` que recién aprendimos sería:

Código 6.4. Usando for para bailar

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

for i in [1,2,3]:
  mi_robot.forward(50, 0.5)
  mi_robot.bakckward(50, 0.5)
  mi_robot.turnRight(35, 1)
  mi_robot.turnLeft(35, 1)
  wait(1)

b.exit()
```

Uso	Descripción	Ejemplo
range(vf)	Retorna la lista desde 0 hasta vf-1	range(3): [0,1,2]
range(vi, vf):	Retorna la lista desde vi hasta vf-1	range(1,4): [1,2,3]
range(vi, vf, inc):	Retorna la lista desde vi hasta vf-1 pero sumando inc valores	range(1,5,2): [1,3]

Tabla 6.1. Ejemplos de Usos de la función range

Bien, ¿qué pasaría si queremos modificar el baile para que el robot repita esto 20 veces? Lo primero que se nos ocurre es modificar la lista, en vez de que contenga los número de 1 a 3, podríamos agregarle los números restantes de manera tal que contenga los números de 1 a 20. Ya esto es poco práctico... pero, ¿qué pasa si ahora queremos repetir muestro paso 50 veces? ¿Y si pensamos en 100 veces?

Python cuenta con una función que nos permite generar listas, ideal para estas situaciones. La función se llama `range` y tiene varias formas de uso. Aquí les mostramos algunos de ellos:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(3, 8)
[3, 4, 5, 6, 7]

>>> range(0, 10, 2)
[0, 2, 4, 6, 8]

>>> range(7, 3, -1)
[7, 6  5, 4]

>>> range(-10, -100, -30)
[-10, -40, -70]
```

Lo primero que vemos es que podemos utilizar esta función con distinta cantidad de argumentos. La forma general se muestra en la Tabla 6.1.

Volviendo a nuestro ejemplo, si queremos que el robot baile repitiendo 100 veces un movimiento, podríamos escribir el siguiente código:

Código 6.5. Usando el range dentro del for

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

for i in range(1,101):
  mi_robot.forward(50, 0.5)
  mi_robot.backward(50, 0.5)
  mi_robot.turnRight(35, 1)
  mi_robot.turnLeft(35, 1)
  wait(1)

b.exit()
```

Algo muy importante de recordar es que si pasamos un única argumento a la función range, la lista generada comienza desde 0.

Podemos anidar sentencias `for` tantas veces como necesitemos. Veamos el siguiente ejemplo:

Código 6.6. Sentencias for anidadas

```
def hago_figuras(r):
  tiempo = 2
  lado = 30
  esperar = 2
  cant_figuras = input("Ingrese la cantidad de figuras: ")
  for i in range(cant_figuras):
    cant_lados = input("Ingrese el numero de lados de la figura "+str(i+1)+"
        : " )
    vuelta = input("Según la figura entre la vuelta de la figura "+str(i+1)+"
         : " )
    for j in range(cant_lados):
      r.forward(lado, tiempo)
      r.turnRight(vuelta, 2)
      wait(esperar)
```

Como pueden ver, es el usuario quien va a determinar cuántas figuras quiere dibujar y de cuántos lados cada una. Prueben ejecutar este ejemplo. Recuerden que, como ya hemos visto, podemos guardar nuestro código en un archivo ".py" y utilizarlo varias veces, en nuestro caso podemos guardarlo como "movimientos.py":

```
from duinobot import *
b = Board(device='/dev/ttyUSB0')
r = Robot(b, 1)
import movimientos
movimientos.hago_figuras(r)
```

✎ Algo muy importante a tener en cuenta cuando usamos la sentencia import, el intérprete de Python sólo la ejecuta una única vez por sesión. Es decir que si modificamos el módulo que importamos debemos o bien reiniciar el intérprete y volver a importar el módulo o utilizar la sentencia `reload`, la cual recarga el módulo con los cambios realizados por nosotros:

```
>>> movimientos = reload(movimientos)
>>> movimientos.hago_figuras(r)
```

De esta manera no es necesario cerrar el intérprete de Python, cada vez que necesitemos realizar una modificación en nuestro código, con la sentencia `reload` el intérprete carga el módulo nuevamente con los cambios guardados.

6.2. Sentencia de iteración `while`

Si pensamos en el ejemplo del menú de opciones, la sentencia `for` no nos permite implementarlo. ¿Por qué creen esto? En realidad, no sabemos cuántas veces el usuario querrá utilizar nuestro menú de opciones. En este caso necesitamos una sentencia que nos permita repetir instrucciones mientras así lo necesitemos. La sentencia `while` repite la ejecución un código mientras se cumpla una determinada condición.

La forma general de la iteración `while`:

```
while <condicion>:
  <Instrucciones>
```

En cada repetición se evalúa la *condición* y, si es verdadera, se ejecutan las *instrucciones*.

Veamos un ejemplo sencillo en el cual el robot avanza mientras se encuentre a más de 15 cm. de un obstáculo. Cuando detecta que se acercó a un obstáculo, se detendrá. Para esto vamos a utilizar la orden `ping`, que veremos más adelante con más detalle, que nos devuelve la distancia que existe entre el robot y un obstáculo.

```
while (mi_robot.ping() > 15):
  mi_robot.forward()
mi_robot.stop()
```

Al igual que en el caso de la sentencia `for`, las instrucciones que queremos que se repitan deben estar indentadas dentro del `while`.

Volvamos a nuestro menú de opciones. Conociendo esta sentencia, podríamos modificar nuestro ejemplo para que el menú se presente **mientras** el usuario no elija finalizar el programa. Veamos cómo quedaría:

Código 6.7. Decido movimientos...

```
def decido_movimiento(robot):

  print('''Indicanos hacia qué dirección te gustaría mover el robot:
        1.- Girar a la derecha
        2.- Girar a la izquierda
        3.- Avanzar
        4.- Retrocede
        5.- Salir
       ''')

  opcion=raw_input("Opción:")
  while (opcion!= '5'):
    if opcion==1:
      robot.turnRight(100,1)
    elif opcion==2:
      robot.turnLeft(100,1)
    elif opcion==3:
      robot.forward(100,1)
    elif opcion==4:
      robot.backward(100,1)
    else:
      print "Ingresaste una opción no válida."
    opcion = raw_input("Opción: ")
```

Antes que nada, cambiamos la última opción del menú para incluir la opción "Salir" para controlar cuándo termina nuestro programa. La condición planteada en el `while` es una expresión booleana. Es decir que puede incluir todos los operadores vistos en el capítulo anterior cuando trabajamos con la sentencia `if`. La evaluación de las diferentes expresiones se realiza de izquierda a derecha, por lo tanto si la primera expresión resulta `False` y se están combinando dos expresiones con el operando `and` no se evaluará la que se encuentra a la derecha de la misma. Esto se denomina "evaluación de circuito corto". Veamos un ejemplo que muestra esta situación:

```
n = input('Ingresá un valor entre 6 y 9: ')
while (n > 5) and (n < 10):
  sumo = sumo + n
  n = input('Ingresá un valor entre 6 y 9: ')
print sumo
```

En este ejemplo si se ingresa un valor menor que 5, ya no se evaluará la segunda expresión que verifica si el valor es menor que 10.

Podemos usar una sentencia `while` para implementar la función que dibuja un cuadrado. Veamos cómo quedaría:

Código 6.8. Usando while para dibujar un cuadrado

```
from duinobot import *
b = Board("/dev/ttyUSB0")
mi_robot = Robot(b, 1)

cant_lados=4
lado
while lado<=cant_lados]:
  mi_robot.forward(50, 0.5)
  mi_robot.wait(1)
  mi_robot.turnRight(35, 1)

b.exit()
```

En este caso, usar una u otra sentencia es indistinto. No así en el caso del menú. En el caso que puedan usarse cualquiera de las dos sentencias, dependerá de cada uno de uds. elegir la que más adecuada.

6.3. Resumen

En este capítulo hemos usado las dos estructuras de control que nos permite repetir instrucciones. Vimos que usando **while**, podemos repetir un conjunto de instrucciones mientras una condición lógica se cumpla y usando el **for** podemos repetir las instrucciones un número fijo de veces. Usamos la función **range()** para generar una lista de valores automática que nos facilitó el uso de **for**, aunque su uso no es exclusivo de este contexto.

6.4. Actividades

Ejercicio 6.1 Reescriban los ejercicios 1 y 3 del capítulo anterior, planteando los distintos en un menú que el usuario puede elegir. No se olviden de incluir una opción “Salir” para concluir con el programa.

Ejercicio 6.2 Arme una lista con distintas frecuencias de sonidos y escriba una función que le indique al robot que “entone” esa melodía. Sugerencia: Pueden pasar la lista como parámetro a la función.

Ejercicio 6.3 Usando `ping` realice una función que obligue al robot a avanzar durante 10 segundos a una velocidad dada, pero que se detenga si es que encuentra un obstáculo a 20 cm.

Capítulo 7

¿Hacemos una pausa?

En este capítulo vamos a formalizar algunos conceptos que venimos trabajando y vamos ver cómo organizamos nuestros códigos para poder compartirlos y reutilizarlos cada vez que sea necesario. Vamos a utilizar este capítulo para afianzar algunos conceptos y profundizar en otros.

7.1. Estructura de un programa

Dado que ya hemos construido un conjunto de módulos y hemos definido las variables necesarias para su funcionamiento, podemos agruparlos en un programa.

Un **programa** no es ni más ni menos que el conjunto de órdenes o instrucciones que resuelven un problema dado.

Podemos pensar la estructura de un programa de la siguiente forma:

- Los programas están compuestos de módulos.
- Los módulos contienen sentencias.
- Las sentencias crean y procesan datos.

Definición	Llamada
def funcion1():	funcion1()
def funcion2(nombre):	funcion2(20)
def funcion3(nombre=100):	funcion3(nombre=10) o funcion3(23) o funcion3()
def funcion4(arg1, arg2):	funcion4(22,11) o funcion4(arg2=12, arg1=22)

Tabla 7.1. Definición y llamadas de una función

Recordamos que un **módulo** es un archivo de texto con instrucciones en Python cuya extensión es ".py". Más adelante veremos algunas formas más complejas de trabajar con módulos pero, en principio, vamos a retomar con la definición de funciones.

7.2. Algo más sobre funciones

Como ya hemos visto a los largo de este libro, el uso de funciones simplifica la escritura de un algoritmo. No sólo permite realizar una acción determinada varias veces sin repetir su código, sino que hace nuestros programas más legibles y fáciles de mantener y modificar. En resumen, podemos decir que las funciones cumplen dos roles muy importantes en el momento de escribir nuestros programas:

- **Reuso de código**: Al definir una función, como mencionamos anteriormente, evitamos repetir sentencias o bloques de código para su uso en diferente puntos de nuestro programa. Si pensamos en funciones parametrizadas esta característica se potencia aún mucho más.
- **Abstracción de código**: Con las funciones podemos dividir nuestro programa en partes más pequeñas, cada una llevando a cabo una acción específica. De esta manera, invocamos a una función para realizar un conjunto de instrucciones, y, desde nuestro programa sólo nos importa qué es lo que hace la función y no cómo lo hace. Cada porción de código encerrado en una función resuelve una tarea específica y no interfiere con el funcionamiento de las otras funciones o el resto del programa.

Googleame: Abstraer

¿Pueden pensar en un ejemplo donde se ponga en práctica este concepto?

7.2.1. Definición de una función y argumentos

Si bien ya las hemos usado en capítulos previos, en ejemplos sencillos, en forma general vamos recordar cómo definimos una función en Python. Presten atención ya que estamos incluyendo algunos elementos que no hemos visto hasta ahora.

```
def nombre_funcion(arg1, arg2,... argN):
    instrucciones
    return valor_de_retorno
```

Hay varias formas de definir las funciones según sea necesario el envío de parámetros o no, en la Tabla 7.1 se puede ver cómo se pueden definir y cómo se hace el llamado en cada caso:

Como pueden ver, en la Tabla 7.1 si una función posee parámetros, los mismos pueden o no tener valores por defecto. La `función3` posee un único argumento y, por defecto tiene el valor `100`. Esto significa que podemos invocar a esta función sin ningún argumento y, en este caso tomará el valor por defecto (100) o, enviando un valor cualquiera.

En el caso que la función posea más de un parámetro, al invocarla, los mismos se asocian en el mismo orden en que fueron definidos. Esto es, en el caso de la `funcion4`, cuando invocamos con los valores `22` y `11`. se asume que `22` será el valor asociado al parámetro `arg1` y `24` será el valor de `arg2`.

Si conocemos el nombre de los parámetros, entonces podemos invocar a la función enviando los valores independientemente del orden en que la función los espera asociando los valores a los correspondientes nombres de argumentos, como se muestra en la última invocación a la `funcion4`. Si bien esto último no es lo más adecuado, muchas veces es útil cuando queremos que la función utilice alguno de los valores por defecto de sus parámetros, y sólo enviarle los que necesitemos[1]

Los parámetros **que poseen valores por defecto deben** ir al final de la lista de parámetros.

Veamos un ejemplo donde invocamos a una misma función de distintas formas. Presten especial atención a las tres últimas, ya que son incorrectas. ¿Pueden explicar la causa del error?

Código 7.1. Funciones con argumento

```
# Sea f una función cualquiera con 3 argumentos de la forma: f(a, b, c)
def f(a, b, c):
   return [a, b, c]

# Podemos enviar todos los argumentos en orden
f(23, 10, 100)
# Lo anterior es equivalente a:
f(a=23, b=10, c=100)
# pero cuando usamos nombres el orden no importa:
f(c=100, a=23, b=10)
# Podemos mezclar algunos con nombre, con otros que no tienen:
f(23, c=100, b=10)
f(23, b=10, c=100)
# Pero todas las líneas que siguen son INCORRECTAS:
f(a=23, 10, 100)
f(23, c=10, 100)
f(100, a=23, c=100)
```

Veamos otro ejemplo donde definimos una función con varios argumentos:

```
def imprimir_datos_vehiculo(patente, marca, color, tipo_motor, ruedas):
    print("Patente: " + patente)
    print("Marca: " + marca)
    print("Color: " + color)
    print("Tipo de motor: " + tipo_motor)
    print("Cantidad de ruedas: " + str(ruedas))
    print("-" * 80)  # Separador
```

Podemos invocar a esta función enviando los parámetros por posición, es decir, el primer parámetro se corresponde con patente, el segundo parámetro se corresponde con marca, etc...

```
imprimir_datos_vehiculo("xcz 001", "Chevrord", "rojo", "naftero", 4)
imprimir_datos_vehiculo("bsc 120", "Fioen", "blanco", "naftero", 4)
imprimir_datos_vehiculo("xzc 100", "Forvrolet", "azul", "gasolero", 4)
imprimir_datos_vehiculo("czx 010", "ScanBenz", "blanco", "gasolero", 6)
imprimir_datos_vehiculo("aaa 023", "Mercenia", "negro", "gasolero", 10)
```

[1] Recuerden en el capítulo 3 cuando definimos las funciones forward y backward

En este caso, hay que recordar el orden de los 5 argumentos al invocar a la función. De no hacerlo, se asociarán en forma incorrecta. Si conocemos el nombre de los parámetros, entonces podría utilizar sus nombres y esto me permitiría ponerlos en cualquier orden:

```
imprimir_datos_vehiculo("xcz 001", tipo_motor = "naftero",
                        marca = "Chevrord", ruedas = 4, color = "rojo")
```

Como se mencionó anteriormente cuando mezclamos parámetros con nombre y sin nombre en la invocación de una función, los que no tienen nombre tienen que estar en orden y deben situarse al principio de la lista de argumentos

Teniendo en cuenta lo anterior, es **incorrecto** invocar a `imprimir_datos_vehiculo()` de la siguiente forma:

```
# Lo siguiente es un ERROR y no funciona:
imprimir_datos_vehiculo("xcz 001", tipo_motor = "naftero",
                        marca = "Chevrord", ruedas = 4, "rojo")
```

¿Se dan cuenta cuál es el error?

7.2.2. Retornando valores

En muchos casos, definimos funciones que nos deben retornar valores. En este caso, utilizaremos la sentencia `return`, que nos permite especificar qué valor retornará la ejecución de la función. Si bien no es obligatorio usar `return` (sino, piensen en todos los ejemplos que les presentamos hasta el momento), es muy útil en muchos casos.

Veamos el siguiente ejemplo:

Código 7.2. ¿Qué movimiento nos gusta más?

```
def movimiento_mas_elegido(r):
  avanzo=0
  retrocedo=0
  print('''Indicanos hacia qué dirección te gustaría mover el robot:
        1.- Avanzar
        2.- Retroceder
        3.- Salir
       ''')

  opcion=raw_input("Opción:")
  while (opcion!= '3'):
    if opcion==1:
       robot.forward(100,1)
       avanzo=avanzo+1
    elif opcion==2:
       robot.backward(100,1)
       retrocedo=retrocedo+1
    else:
       print "Ingresaste una opción no válida."
    opcion = raw_input("Opción: ")
    if avzano>retrocedo:
       return "más avances que retrocesos"
    else:
       return "más retrocesos que avances"
```

La función que definimos, además de permitirle al usuario que elija la dirección en la que queremos mover al robot, nos retorna qué movimiento fue el que más elegimos. ¿Cómo deberíamos invocar y usar este valor retornado?

Código 7.3. Invocación a `movimiento_mas_elegido()`

```
...
mas_elegido=movimiento_mas_elegido(mi_robot)
print "Al final de las pruebas, hubo "+ mas_elegido
...
```

La sentencia `return` corta la ejecución de la función que la contiene. Esto significa que si detrás de esta sentencia escribimos otras, las mismas no se ejecutarán. En el siguiente ejemplo lo único que hace `funcion_poco_util()` es retornar el valor `True`, nunca realiza ninguna cuenta ni imprime nada en pantalla:

```
def funcion_poco_util(a, b):
    return True
    print("Esto no se ejecuta nunca")
    x = a * b
    x = x / 3
    print("esto tampoco")
```

7.3. Importando módulos

Lo primero que vimos cuando comenzamos a trabajar con el robot tenía que ver con "agregar" al entorno de Python el módulo que contiene las funciones y los elementos para trabajar con los robots. Esto lo hacíamos utilizando la sentencia `import`. Hay dos formas de importar un módulo. La forma vista hasta el momento:

```
from duinobot import *
```

O, también podemos indicar:

```
import duinobot
```

Si bien con ambas sentencias logramos incorporar las funcionalidades necesarias para manejar el robot en nuestros programas, las diferencias están en cómo voy a utilizar esas funciones.

Antes de seguir, veamos un concepto importante que debemos tener claro: los **espacios de nombres**.

Un **espacio de nombres** es una tabla que indica a qué objetos hace referencia cada nombre.

En el caso de la función del ejemplo último, `funcion_poco_util()`, el espacio de nombres asociado a la función tendrá una entrada para cada variable y parámetro que define, indicando que el nombre a, representa a tal o cual elemento (según sea el parámetro con el que invoque a la función).

Supongamos que definimos el siguiente módulo:

```
#Archivo: bailes.py

def baile_lento():
    # Implementación de la función...
    .....
def baile_rock():
    # Implementación de la función...
```

```
    .....
def baile_samba():
    # Implementación de la función...
    .....
```

Si desde mi programa quiero utilizar estas funciones, puedo importar todas o sólo la que quiero utilizar. Si quiero importar todas, puedo hacerlo de dos formas:

```
import bailes
```

O bien:

```
from bailes import *
```

La diferencia entre ambas tiene que ver en cómo se altera el espacio de nombres de mi programa, es decir, qué nombres se agregan y a qué harán referencia. En el primer caso, si bien se agregan todas las funciones que representan a los distintos bailes, sólo se incorpora a mi espacio de nombres el nombre del módulo (`bailes`) por lo cual para utilizar alguna de las funciones debo antes anteponer el nombre del módulo.

```
import bailes
....
bailes.baile_rock()
```

En cambio, si las importamos de la segunda manera los nombres de las funciones son las que se incorporan a nuestro espacio de nombres directamente.

```
from bailes import *

..
baile_rock()
```

¿Cuál es la mejor forma? En realidad utilizar la segunda opción puede ser un riesgo ya que si en mi programa ya tengo una función o una variable que se llama `baile_rock()` la misma será reemplazada en nuestro espacio de nombres por la referencia a la función del módulo `bailes`.

También podemos importar sólo algunas funciones o variables de un módulo. Esto lo podemos hacer de la siguiente forma:

```
from bailes import baile_rock, baile_lento

..
baile_rock()
```

De esta manera, sólo agrego al espacio de nombres de mi programa los nombres de las funciones importadas y puedo invocarlas sin anteponer el nombre del módulo.

7.4. Resumen

En este capítulo trabajamos con funciones. Vimos cómo podemos pasar argumentos a una función y que los mismos pueden tener valores por defecto, simplificando la invocación.

También vimos de qué manera podemos retornar valores desde una función y utilizarlos desde los programas que las invocan.

7.5. Actividades

Ejercicio 7.1 Escriba un programa que defina una lista con una secuencia de instrucciones de la forma: “avanzar”, “retroceder”, etc..; y luego, recorriendo la lista mueva al robot siguiendo las instrucciones codificadas en la misma.

Ejercicio 7.2 Vamos a hacer una pequeña variación al ejercicio anterior. Modifique la función `movimiento_mas_elegido` presentada en este capítulo de manera tal de ir completando la lista con los movimientos elegidos por el usuario. Esta lista será retornada por la función y a partir de ahí podremos repetir la secuencia en forma similar a lo realizado en el ejercicio 1.

Ejercicio 7.3 Vamos a hacer una obra de teatro. En esta primer etapa, pensemos en cuatro o cinco actos. Para eso, en grupos piensen el tema, el contexto, los personajes y, por último el guión. Piensen de qué manera pueden organizar las funciones que muevan a los robots intervinientes en la obra y cómo las importaremos y utilizaremos en el programa.

Ejercicio 7.4 Filmen la obra y súbanla a nuestro canal de youtube/vimdeo.

Capítulo 8

Utilizando los sensores

En capítulos anteriores escribimos programas para que el robot realice distintos movimientos y aprendimos a organizar nuestros códigos de distintas formas para hacer nuestro trabajo más eficiente y prolijo.

Pero el robot aún no percibe el entorno, realiza literalmente lo que le indicamos sin importar si existen obstáculos o si, por ejemplo, alguna situación del contexto no le permite moverse.

El robot que utilizamos posee varios sensores que podrían ser de suma utilidad para incorporarle algún tipo de "inteligencia" que le permita tomar decisiones en función del contexto.

En la Figura 8.1 se puede observar que el robot tiene un sensor para detectar obstáculos y dos sensores para superficies claras y oscuras (que normalmente usaremos para detectar líneas). Veremos cómo hacer a nuestro robot más inteligente y autónomo usando estos sensores.

8.1. Conociendo los sensores

Para saber qué valores podemos esperar de los sensores podemos usar la orden `senses()` que, en una ventana separada muestra los valores retornados por los sensores. Esto lo podemos ver en el siguiente código:

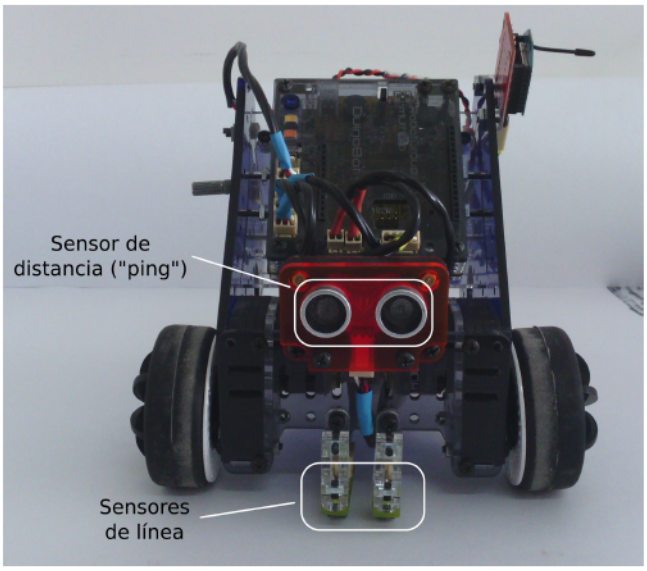


Figura 8.1. Sensores del robot

```
usuario@host:~$ python
Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from duinobot import *
>>> b = Board("/dev/ttyUSB0")
>>> b.report()
Robot 1 prendido
>>> robot = Robot(b, 1)
>>> robot.senses()
```

La Figura 8.2 nos muestra cómo es esta ventana y qué información presenta. Como pueden ver, además de mostrar los valores de los sensores de distancia y de línea, muestra el estado de las baterías. Esta información es importante tener en cuenta dado que, cuando las baterías están descargadas, el robot puede no funcionar correctamente.

Prueben acercar su mano frente al robot (delante de los sensores de distancia) para ver cómo cambia el valor de `ping` o apoyen el robot en distintas superficies para ver cómo varían los valores de los sensores de línea.

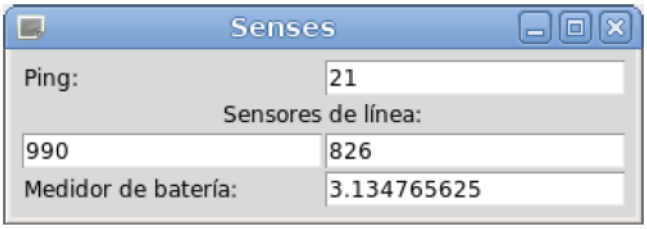


Figura 8.2. Ventana de información de sensores

La Tabla 8.1 muestra los rangos de valores retornados por los sensores.

8.2. Evitando choques

A partir de esta sección vamos a utilizar la información obtenida a través de los sensores para indicarle al robot que él decida si puede o no avanzar, retroceder o girar. En realidad, intentamos

Sensor	Valor Mínimo	Valor Máximo
distancia	0	601
linea	0	1000
batería	0	≈ 4.5

Tabla 8.1. Valores límites de los sensores

que, en base a la información de su contexto, pueda decidir qué hacer. Obviamente para todo esto, seguiremos escribiendo programas en Python.

8.2.1. Midiendo distancias

En algún capítulo anterior mencionamos la orden `ping()`. Trabajando con el valor retornado por esta orden vamos a poder escribir programas que permitan que el robot pueda avanzar sin chocar.

Pensemos un poco en términos de programación cómo planteamos un programa que permita que nuestro robot avance hasta que encuentre un obstáculo. En el siguiente código se muestra (en pseudocódigo) una forma posible de resolver este problema:

```
avanzar()
mientras no hayObstáculo:
    noHacerNada
detener()
```

Para trasladar este pseudcódigo a Python, primero analizamos cada parte. `avanzar()` y `detener()` no son un problema, ya hemos utilizado muchas veces las órdenes para ejecutar estas acciones. El problema radica en plantear la condición "no hay obstáculo". Dijimos que `ping()` devuelve la distancia del robot a un obstáculo, con lo cual podríamos pensar es escribir una función que devuelva `True` si el valor retornado por `ping()` es menor que cierto valor (por ejemplo 15 cm.) y `False` en caso contrario. Por lo tanto, podríamos escribir:

Código 8.1. Función que me retorna si hay o no obstáculos

```
def hay_obstaculo(robot):
  if robot.ping()<15:
     return True
  else:
     return False
```

En este pseudocódigo aparece la instrucción `noHacerNada`, en Python vamos a precisar una instrucción que no haga nada porque no se pueden escribir bloques vacíos, es decir, no se pueden escribir funciones o estructuras de control vacías. La instrucción de Python que no hace nada se llama `pass`. Pasando en limpio todo lo anterior en el código 8.2 podemos ver la implementación en Python del pseudocódigo anterior.

Código 8.2. Detectando obstáculos

```
robot.forward()
while not hay_obstaculo(robot):
    pass
robot.stop()
```

8.2.2. Detectando obstáculos

Hasta el momento usamos `ping()` para ejercitar el uso de las comparaciones, pero existe la orden `getObstacle()` que se comporta de manera similar a nuestra función `hay_obstaculo`. Por defecto, `getObstacle()` considera que algo a menos de 10 centímetros es un obstáculo, pero podemos indicarle que modifique esa cantidad pasándole la distancia como argumento. Veamos un ejemplo:

```
# Consideremos que el robot realmente tiene un obstáculo a 8 cm.

>>> robot.getObstacle() # True si hay algo a menos de 10 cm.
True
>>> robot.getObstacle(5) # True si hay algo a menos de 5 cm.
False
>>> robot.getObstacle(100) # True si hay algo a menos de 1 metro.
True
```

8.3. Sensores de línea

Con los sensores de línea es posible notar cambios de contraste en la superficie sobre la que está el robot, los valores de estos sensores también cambian si levantamos al robot del piso. Con la orden `getLine()` se obtienen por separado los valores del sensor izquierdo y el derecho. Podemos aprovechar esto si, por ejemplo, estamos siguiendo una línea dibujada en el piso, y queremos saber de qué lado el robot se salió de la línea primero a fin de compensar girando en la dirección contraria.

En realidad `getLine()` devuelve una tupla de dos elementos, donde el primer elemento es el valor del sensor izquierdo y el segundo elemento es el resultado del sensor derecho.

Googleame: tupla

¿Trabajaron alguna vez con estos elementos en matemática?

Veamos un ejemplo para analizar cómo trabajar con los sensores de línea.

```
>>> robot.getLine()
(250, 253)
>>> izq, der = robot.getLine()
```

8.4. Normalizando valores

Como vimos con la orden `senses()` los sensores devuelven distintos valores que se encuentran en cierto rango de valores. En muchas ocasiones vamos a queres utilizar estos valores para, por ejemplo, controlar la velocidad del robot. En estas situaciones, vamos a necesitar "adaptar" los valores retornados por los sensores al rango de valores esperados por el resto de las funciones.

Veamos un ejemplo concreto. El sensor de distancia devuelve valores entre 0 y 601. Si queremos utilizar este valor para controlar la velocidad que le enviamos a `forward()`, deberíamos "normalizarlo" para pasar este valor al rango entre 0 y 100 que es el rango de valores aceptados por `forward()`.

Esto que parece complicado, puede resolverse con una fórmula matemática simple. Si asumimos que todos los rangos tanto origen como destino empiezan en 0, podemos plantear la fórmula:

$$n(x) = (x/maxOrigen) \cdot maxDestino$$

Donde:

- **maxOrigen**: es el valor máximo original
- **maxDestino**: es el valor máximo al cual queremos convertir
- **x**: es el valor que queremos convertir

Básicamente al dividir el valor original por el máximo del rango de origen, nos da un valor con decimales entre 0 y 1. Si luego multiplicamos este valor por el máximo del valor destino obtenemos el valor que queremos. Si no nos creen, escriban una función que realice este cálculo y prueben con los siguientes valores:

maxOrigen = 601 y maxDestino = 100:

- $n(0) = 0$
- $n(20) = 3.3277870216306153$
- $n(200) = 33.277870216306155$
- $n(560) = 93.178036605657226$
- $n(601) = 100$

8.5. Velocidad proporcional a la distancia

Podemos modificar un poco el programa visto al comienzo del capítulo para frenar gradualmente cuando se acerca un obstáculo, podemos usar el valor del sensor de obstáculo para determinar la velocidad.

Si seguimos con el criterio anterior cuando estemos a 15 centímetros la velocidad debería ser cero, podemos hacer que el robot comience a frenar a 55 centímetros del obstáculo por ejemplo para que el efecto sea más visible.

Una posible solución sería: mientras los objetos estén a más de 55 cm. simplemente avanzamos pero, cuando estén a menos de 55 y más de 15 centímetros deberíamos ir bajando la velocidad para que, cuando finalmente lleguemos a los 15 cm. le indicamos al robot que se detenga.

Para que frene gradualmente podemos normalizar el valor de `ping()` a un valor entre 0 y 100 para poder utilizarlo como argumento de `forward()`. En realidad, esto será una aplicación de lo visto en la sección anterior.

El código 8.3 muestra una implementación en Python.

Código 8.3. Programa: Velocidad proporcional a la distancia

```
from duinobot import *
from mis_funciones import normalizar

board = Board("/dev/ttyUSB0")
robot = Robot(board, 1)
```

```
robot.forward(100)
while robot.ping() > 55:
    pass
while robot.ping() > 15:
    robot.forward(normalizar(robot.ping(), 55, 100))
robot.stop()

board.exit()
```

Asumimos que la función `normalizar()` está almacenada en el módulo `mis_funciones`, por eso la sentencia[1]:

```
from mis_funciones import normalizar
```

Para probar si esto funciona correctamente pueden levantar el robot y acercarlo y alejarlo lentamente de un obstáculo para ver cómo cambia la velocidad de las ruedas.

8.6. Resumen

En este capítulo hemos trabajado con los sensores del robot.

Vimos cómo detectar obstáculos utilizando las órdenes **robot.getObstacle()** que nos devuelve `True` si hay algún objeto cerca y `False` en otro caso y la orden **robot.ping()** que devuelve la distancia en centímetros al objeto más cercano al frente del robot.

También vimos cómo obtener los valores de los sensores de línea con la orden **robot.getLine()** y que podemos visualizar los valores de todos los sensores utilizando la orden **robot.senses()**.

8.7. Actividades

Ejercicio 8.1 Vamos a sumar un nuevo estado a nuestro robot: robot_miedoso. En este caso, el robot se escapará cuando algún objeto se le acerque a menos de 20cm. Para esto, agregue una nueva función que implemente esto en el módulo tipos_de_robots

Ejercicio 8.2 Escriban un programa que permita al robot recorrer una habitación sin chocar. En caso de encontrar un obstáculo el robot debe retroceder, girar y luego avanzar en la nueva dirección. El programa debe terminar luego de encontrar 5 obstáculos.

Ejercicio 8.3 Escriban un programa que permita al robot avanzar hasta encontrar una superficie oscura. Ubiquen al robot en una superficie clara para probar esto. (pueden probar esto al revés, es decir, sobre una superficie oscura, buscar una clara).

Ejercicio 8.4 Implementen la función `normalizar1()` que recibe como argumentos el valor, el máximo del rango origen y el máximo del rango destino y retorne el valor normalizado. Por ejemplo, para el caso anterior $normalizar1(20, 601, 100) \approx 3.33$.

Ejercicio 8.5 Modifique la función anterior, para contemplar valores fuera de rango. Es decir que si el valor es negativo debe hacer las cuentas como si el valor fuera 0 y si el valor es mayor que el máximo permitido debe hacer las cuentas como si el valor fuera el máximo permitido. Ejemplos:

- $normalizar(-20, 601, 100) = 0$
- $normalizar(-1, 601, 100)) = 0$
- $normalizar(602, 601, 100) = 100$

[1] Si uds. usan otros nombres, simplemente adecúen este código.

- $normalizar(1000, 601, 100) = 100$
- $normalizar(600, 601, 100) \approx 99.83$

Apéndice A

Instalación de paquetes

En distribuciones GNU/Linux basadas en Debian existe un gestor de paquetes llamado Synaptic, es un avanzado sistema para instalar o eliminar aplicaciones. Con Synaptic tenemos el control completo de los paquetes (aplicaciones) instalados en el sistema.

A.1. Instalador de Paquetes Synaptic

Para ejecutar Synaptic vamos a Sistemas → Administración → Gestor de Paquetes Synaptic como vemos en la Figura A.1

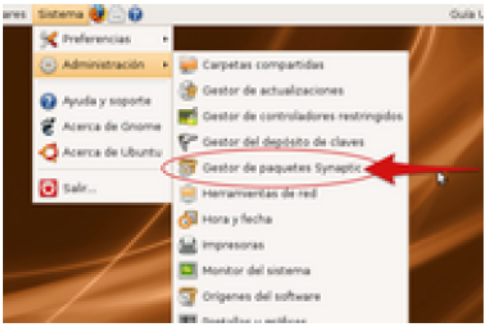


Figura A.1. Arrancando el gestor

Podemos notar en la Figura A.2 que se encuentra dividida en 4 partes. En el margen izquierdo tenemos distribuido por categoría (N° 1 Lista de categorías), abajo se encuentran los filtros (N° 2), a nuestra derecha arriba se encuentran todos los paquetes que pertenecen a esa categoría (N° 3 paquetes para instalar) y abajo una breve descripción del paquete seleccionado(N° 4 Descripción de la aplicación).

Figura A.2. Secciones de la Synaptic

Para instalar un paquete podemos seleccionar una categoría y luego hacer doble clic sobre él o clic derecho, "Marcar para instalar". De esta forma, podemos marcar todos los paquete deseados y luego pulsamos en el botón "Aplicar para instalar". Synaptic comenzará la descarga de los paquetes necesarios desde los repositorios en Internet o desde el CD de instalación.

También podemos realizar una búsqueda rápida. En la barra de menú, tenemos un buscador donde podemos ingresar el nombre del paquete que queremos instalar, abajo nos muestra el resultado de la búsqueda. Para instalar los paquetes repetimos el proceso ya mencionado.

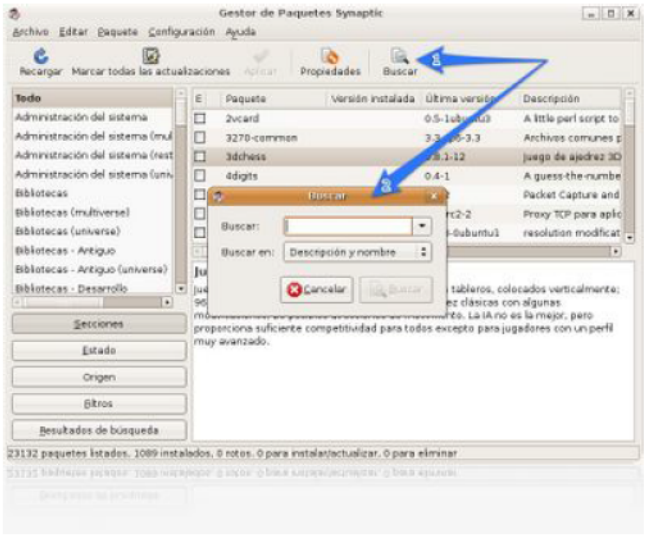


Figura A.3. Buscar un paquete

A.2. Instalación de la API del Robot

La API del robot puede ser instalada en distintas distribuciones GNU/Linux, el método de instalación varía dependiendo de la distribución que usemos, en particular el paquete de la API fue probado en distintas versiones de Lihuen, Debian, Ubuntu y RxArt.

Para tener un funcionamiento completo de la API recomendamos la versión de Python 2.6 o 2.7, aunque la API está preparada para funcionar en un modo con funcionalidad reducida en Python 2.5 y posiblemente 2.4.

Esta guía es solo ilustrativa y hace referencia a la versión del paquete al momento de ser escrita, es muy probable que cuando lean esto existan versiones más nuevas y mejores de los paquetes. Les recomendamos leer la versión actual de la guía en la página del proyecto (si en enlace anterior está roto buscá "instalación" con el buscador de la página del proyecto).

A.2.1. En distribuciones basadas en Debian con Python 2.5

Dentro de esta categoría entran:

- Debian Lenny
- Lihuen 3
- Ubuntu 8.04
- Algunas versiones de RxArt

En estas distribuciones se debe descargar nuestro paquete para Python 2.5 y abrirlo con la aplicación GDebi que nos permitirá instalarlo.

A.2.2. En distribuciones basadas en Debian con Python 2.6 o superior

Dentro de esta categoría entran:

- Debian Squeeze a Wheezy
- Lihuen 4.01 (ver A.2.3)
- Ubuntu 10.04 a 12.04
- Algunas versiones de RxArt

En estas distribuciones hay que descargar el paquete para Python 2.6 e, igual que en el otro caso, hay que abrirlo con la aplicación GDebi que nos permitirá instalarlo o bien con Software Center.

A.2.3. En Lihuen 4.x

En Lihuen 4 el paquete robot se encuentra en nuestros repositorios oficiales, así que basta con seguir la guía de uso de Synaptic, buscar el paquete y marcarlo para instalar.

A.2.4. En otras distribuciones GNU/Linux y otros sistemas operativos

Si bien no está del todo probada no tendría que haber ningún inconveniente con instalar la API en cualquier otra distribución GNU/Linux.

En el caso de otros sistemas operativos como FreeBSD o en sistemas propietarios es posible que la API funcione ya que fue desarrollada con la intención de ser multiplataforma (excepto la función `boards()` que no es portable y seguramente no funcione en otros sistemas).

El método de instalación genérico consiste en usar Python Distutils, para esto hay que tener instalado:

- Python 2.6 (recomendado, pero puede ser cualquier versión mayor a 2.4 y menor que 3.0)
- PySerial (python-serial)
- PyGame (python-pygame)
- Python-Tk (python-tk)

Es necesario descargar el paquete con el "código fuente" del módulo DuinoBot, descomprimirlo, entrar a la carpeta creada y ejecutar el siguiente comando en una terminal como usuario administrador:

```
python setup.py install
```

En sistemas Unix-like se puede usar el siguiente script:

```
#!/bin/sh
wget http://repo.lihuen.linti.unlp.edu.ar/lihuen/pool/lihuen4/main/r/robot/
    robot_0.10.tar.gz
tar -xzf robot_0.10.tar.gz
cd duinobot
python setup.py install
```

Apéndice B

Especificaciones de los robots

B.1. Multiplo N6

En este curso se utilizan robots de especificaciones abiertas basados en el sistema constructivo Multiplo [1] y fabricados en Argentina por la empresa *RobotGroup* [2]. La carcaza del robot está diseñada para poder ser replicada usando una cortadora láser [3] y la lógica del robot es compatible con Arduino.

El robot es controlado por la placa *DuinoBot v1.2* basada en el micro-controlador *AVR ATMega32U4* compatible con *Arduino*.

Es muy importante tener en cuenta que este libro fue escrito para ser usado con una versión modificada del robot *Multiplo N6* que tiene conectados los sensores de determinada forma y un programa grabado para poder recibir los comandos de forma inalámbrica. Los robots tal cual los venden RobotGroup precisan estas modificaciones para ser utilizados con Python con el módulo duinobot.

La versión original está planteada para ser programada por el cable USB usando C++ con el entorno *DuinoPack* o con programación por bloques con el entorno *Minibloq*. La versión que utilizamos con Python es entonces un Multiplo N6 al que se le añade un módulo de comunicaciones inalámbrico *XBee*, un conjunto determinado de sensores y se le graba un programa en C++ que permite al robot recibir instrucciones a través de ese módulo.

[1] http://multiplo.org/
[2] http://www.robotgroup.com.ar
[3] Cortadora láser en Wikipedia

Las especificaciones del Hardware pueden verse "Robot N6: Guía de programación con Minibloq y DuinoPack" en el sitio de RobotGroup. Es muy importante tener en cuenta que **para programar uno de estos robots con Python es necesario comprar como accesorios las placas *XBee* y cargarle un firmware especial en su placa desde una versión especial de *DuinoPack***, este firmware junto al *DuinoPack* modificado pueden ser descargados desde el sitio del proyecto "Programando con Robots" [4].

Es posible agregar otros sensores y acceder a ellos desde la API de Python, para esto hay que leer la disposición y números del los pines desde la página de RobotGroup y utilizar los métodos `analog()` y `digital()` del objeto Board. Los conectores S0 a S5 pueden leerse de los pines analógicos 0 a 5 respectivamente.

Por ejemplo para leer desde un script en Python un sensor analógico conectado en S5 en el robot 3, usando 4 muestras por cada invocación:

Código B.1. Lectura de sensor analógico desde un script Python

```
from duinobot import *
b = Board()
sensor = 5
for i in range(10):
  print(b.analog(sensor,samples=4,robotid=3))
  wait(0.5) # Esperar medio segundo

b.exit()
```

También existe la función digital, se puede ver su documentación (de la misma manera que la de cualquier función) con `help`:

```
help(Board.digital)
```

B.1.1. Otras formas de programarlo

En la página de RobotGroup se pueden descargar manuales para programar el robot en C++ con DuinoPack o con Minibloq usando bloques.

El siguiente es un ejemplo de un programa en C++ que lee un sensor de luz (una fotorresistencia[5]) conectado en S0 y en base a la intensidad de la luz genera un tono de determinada frecuencia.

Código B.2. Generar beeps dependiendo de la intensidad de la luz

```
#define BUZZER_PIN 23

void setup()
{
}

void loop()
{
   tone(BUZZER_PIN, analogRead(0), 20);
   delay(100);
}
```

Este ejemplo hace que el robot se mueva a velocidad máxima por 1 segundo.

Código B.3. Mover robot por 1 segundo

[4] http://robots.linti.unlp.edu.ar
[5] http://es.wikipedia.org/wiki/Fotorresistencia

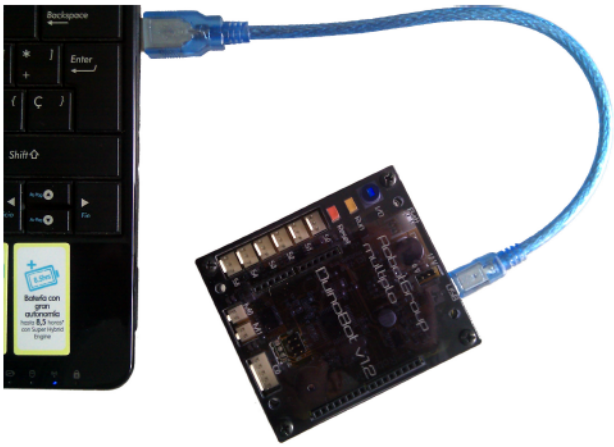


Figura B.1. Conexión al board usando el cable USB

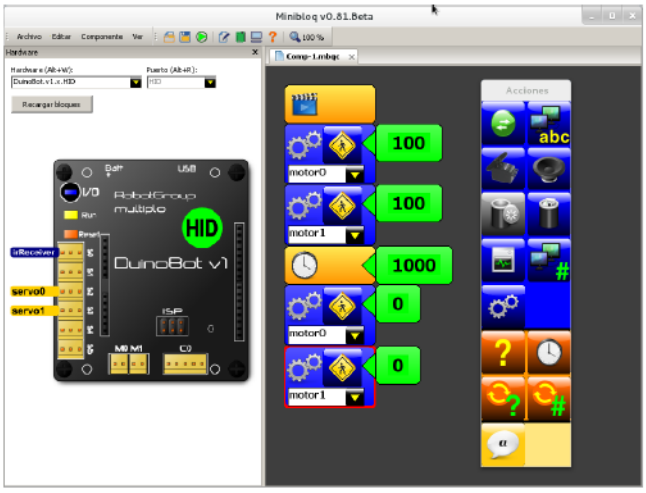


Figura B.2. Mover el robot con Minibloq

```
void setup()
{
   motor0.setSpeed(100);
   motor1.setSpeed(100);
   delay(1000);
   motor0.setSpeed(0);
   motor1.setSpeed(0);
}

void loop()
{
}
```

Estos programas se pueden compilar y cargar con DuinoPack (una versión modificada de Arduino IDE) conectando el robot a la computadora usando un cable USB.

También se puede hacer lo mismo con Minibloq como se ve en la figura B.2.

B.2. Otros robots similares

Existen otros robots similares que tienen el objetivo de ser herramientas para enseñar a programar, algunos de ellos tienen especificaciones abiertas, algunos están planteados como kits de construcción,

algunos deben ser construidos por alguien con nociones de electrónica y otros vienen ya armados.

B.2.1. Robots artesanales

Existen distintos modelos de microcontroladores que pueden ser utilizados para crear robots que tengan una funcionalidad similar a los robots de este curso, entre los más populares están los PIC [6] y los AVR [7], algunos de estos dispositivos precisan montarse temporalmente en placas especiales (programadores[8]) para ser programados, pero actualmente existen muchos que se pueden programar estando ya montados en la placa en la que funcionarán sin usar programadores especiales.

La plataforma Arduino [9] y proyectos derivados como Freeduino [10] actualmente son muy usados para este tipo de proyectos, Arduino y Freeduino están compuestos de software libre y hardware abierto.

El sitio http://hackaday.com es un gran recurso para encontrar información sobre construcción de robots.

B.2.2. Icaro

El proyecto argentino Icaro [11] plantea la construcción de placas fáciles de usar al estilo de Arduino y robots controlados por esas placas, también provee distintas modalidades de programación:

- Con bloques, con el robot funcionando de forma autónoma, con "icaro-bloques".
- Con bloques, si el robot está conectado a una notebook con "turtleart" ("tortucaro").
- Con C++ con "ICARO C++".
- A modo control remoto con la interfaz gráfica "pycaro".

Los robots de Icaro se programan usando una API en español y en el caso de usar "icaro-bloques" los nombres de las estructuras de control también están en español.

El software de Icaro es libre y el hardware es de especificaciones abiertas.

El hardware de Icaro es fácil de construir con herramientas al alcance de cualquier aficionado a la electrónica y está basado en microcontroladores PIC en lugar de los AVR de Arduino.

B.2.3. Myro Hardware

Anteriormente utilizamos los robots propuestos por *Institute for Personal Robots in Education* (IPRE), en particular los robots *Scribbler* con una placa conocida como *IPRE Fluke* que permite controlarlos a través de bluetooth y le agrega sensores extra, como ser más sensores de obstáculos y una cámara para tomar fotografías. Para controlarlos desde el intérprete de Python usamos el paquete *Myro* que provee una interfaz de comandos similar a la propuesta por el curso pero más orientada a la programación procedural con un solo robot (si bien es posible instanciar varios robots y controlarlos desde un solo programa).

Actualmente el Scribbler 2 fabricado por *Parallax* es hardware de especificaciones abiertas y el *Institute for Personal Robots in Education* está promoviendo otra API para reemplazar a Myro conocida como *Calico Myro* o simplemente *Calico*.

[6] http://es.wikipedia.org/wiki/Microcontrolador_PIC
[7] http://es.wikipedia.org/wiki/AVR
[8] http://es.wikipedia.org/wiki/Programador_(dispositivo)
[9] http://www.arduino.cc/
[10] http://www.freeduino.org/
[11] http://roboticaro.org/

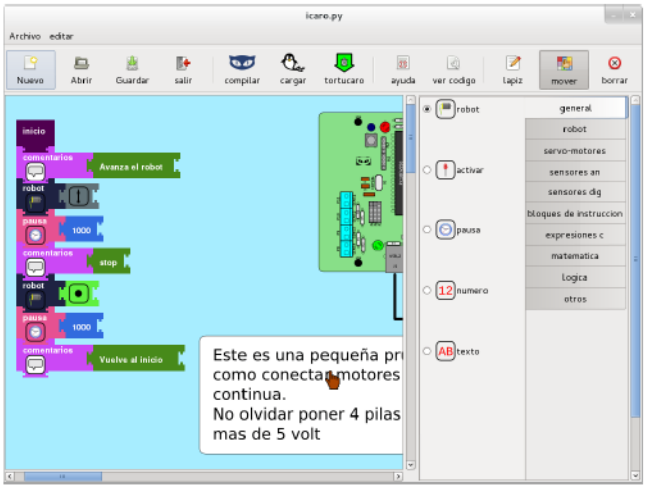


Figura B.3. Icaro-bloques

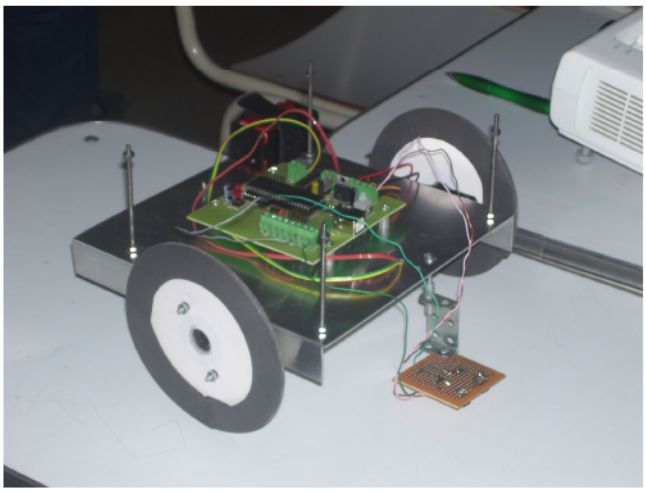


Figura B.4. Robot del proyecto Icaro

El *IPRE* es un esfuerzo conjunto entre *Georgia Tech* y *Bryn Mawr College* con el sponsor de *Microsoft Research*.

El Scribbler también puede ser programado a través de un puerto serie usando programación por bloques o con distintos lenguajes de programación dependiendo de la versión del robot, se pueden ver ejemplos en la página de *Parallax* [12].

[12] http://www.parallax.com/

Figura B.5. Robot scribbler y placa IPRE Fluke

Apéndice C

Resumen de instrucciones

En este apéndice hemos resumido todas las órdenes que podemos darle a nuestro robot. En este cuadro, veamos la forma general, una breve descripción, si la orden es bloqueante o no, y un ejemplo de uso.

Valor de retorno	Mensaje	Cant. de arg.	Descripción	Bloqueante	Ejemplo
-	setId(valorEntero)	1	Permite cambiar el número de identificación del robot.	-	robot.setId(21)
-	beep(frecuencia, tiempo)	2	El robot emite un sonido en el valor de la frecuencia y durante el tiempo ingresado como parámetro	Sí	robot.beep(200,0.5)
-	setName(nombre)	1	Asigna el contenido de la variable `nombre` como nuevo nombre del robot	-	robot.setName("Juancito")
str	getName()	0	Devuelve una cadena de caracteres con el nombre del robot	-	robot.getName() - Juancito

Valor de retorno	Mensaje	Cant. de arg.	Descripción	Bloqueante	Ejemplo
-	speak()	1	Imprime en pantalla el valor que se ingrese como parámetro.	-	robot.speak("Hola, mi nombre es " + r.getName()) - Hola, mi nombre es Juancito
-	forward()	0	Genera un movimiento del robot a una velocidad media por un tiempo indeterminado.	No	robot.forward()
-	forward(velocidad)	1	Genera un movimiento del robot a una velocidad enviada como parámetro por tiempo indeterminado	No	robot.forward(100)
-	forward(velocidad, tiempo)	2	Genera un movimiento del robot a una velocidad enviada como parámetro por el lapso de tiempo que indique el valor del segundo argumento. Pasado ese tiempo, el robot se detendrá por si mismo, sin necesidad de enviarle el mensaje stop().	Sí	robot.forward(100, 20)
-	forward(seconds= tiempo, vel=velocidad)	1-2	Si se referencia los argumentos por su nombre, se los puede utilizar en cualquier orden ambos o individualmente.	Sí	robot.forward(seconds=10, vel=90)
-	backward()	0	Genera un movimiento del robot hacia atrás a una velocidad media por un tiempo indeterminado.	No	robot.backward()
-	backward(velocidad)	1	Genera un movimiento del robot hacia atrás a una velocidad enviada como parámetro por tiempo indeterminado	No	robot.backward(20)
-	backward(velocidad, tiempo)	2	Genera un movimiento del robot hacia atrás a una velocidad enviada como parámetro por el lapso de tiempo que indique el valor del segundo argumento. Pasado ese tiempo, el robot se detendrá por si mismo, sin necesidad de enviarle el mensaje stop()	Sí	robot.backward(20, 5)
-	backward(seconds= tiempo, vel=velocidad)	1-2	Si se referencia los argumentos por su nombre, se los puede utilizar en cualquier orden a ambos o individualmente.	Sí	robot.backward(20)
-	turnLeft()	0	Permite que el robot gire hacia izquierda a velocidad media y por tiempo indeterminado.	No	robot.turnLeft()
-	turnLeft(velocidad)	1	Permite que el robot gire hacia izquierda a una velocidad enviada como parámetro por un tiempo indeterminado.	No	robot.turnLeft(50)

Valor de retorno	Mensaje	Cant. de arg.	Descripción	Bloqueante	Ejemplo
-	turnLeft(velocidad, tiempo)	2	Permite que el robot gire hacia izquierda a una velocidad enviada como parámetro durante el valor de tiempo ingresado como segundo parámetro.	Sí	robot.turnLeft(50, 20)
-	turnLeft(vel= velocidad, seconds=tiempo)	1-2	Si se referencia los argumentos por su nombre, se los puede utilizar en cualquier orden ambos o individualmente.	Sí	robot.turnLeft(50, 20)
-	turnRight()	0	Permite que el robot gire hacia la derecha a velocidad media y por tiempo indeterminado.	No	robot.turnRight()
-	turnRight(velocidad)	1	Permite que el robot gire hacia la derecha a una velocidad enviada como parámetro por un tiempo indeterminado.	No	robot.turnRight(50)
-	turnRight(velocidad, tiempo)	2	Permite que el robot gire hacia la derecha a una velocidad enviada como parámetro durante el valor de tiempo ingresado como segundo parámetro.	Sí	robot.turnRight(50, 20)
-	turnRight(seconds= tiempo, vel=velocidad)	1-2	Si se referencia los argumentos por su nombre, se los puede utilizar en cualquier orden ambos o individualmente.	Sí	robot.turnRight(seconds=20)
-	stop()	0	Detiene al movimiento del robot. Es utilizado cuando después de enviar los mensajes forward(), backward(), turnRight() y turnLeft(), con 0 ó 1 argumentos	-	robot.stop()
	motors(velDerecha, velIzquiera, tiempo)	3	Permite asignarle velocidades diferentes a cada rueda durante el valor de tiempo ingresado como parámetro.	Sí	robot.motors(100, 50, 20)
-	senses()	0	Muestra una interfaz gráfica con la información de los sensores y de la carga de la batería del robot.		
int	ping()	0	Devuelve la distancia en centímetros entre el robot y un objeto que se encuentre frente a él. Éste valor varía entre 0 y 601.		
(int,int)	getLine()	0	Retorna la información de los sensores de línea mediante una tupla. El primer elemento es el valor del sensor izquierdo y el segundo elemento es el resultado del sensor derecho.		robot.getLine()

Valor de retorno	Mensaje	Cant. de arg.	Descripción	Bloqueante	Ejemplo
bool	getObstacle()	0	Devuelve True si existe un objeto a 10 centímetros de distancia del robot.		robot.getObstacle()
bool	getObstacle(distancia)	1	Devuelve True si existe un objeto a una distancia (ingresada como parámetro) del robot.	-	robot.getObstacle(20)
-	wait(tiempo)	1	Pausa la ejecución de instrucciones del programa durante la cantidad de segundos que se ingrese como parámetro.	Sí	wait(5)

Velocidad: Es un valor en el rango entre -100 y 100.

Frecuencia: En el orden de 1 Hz a 15 Khz.

Tiempo: Expresado en segundos.

Apéndice D

Enlaces útiles

Enlaces del proyecto

- Sitio del Proyecto Robots con información útil, actividades, vídeos y enlaces:
  http://robots.linti.unlp.edu.ar
- Vídeos relacionados con el proyecto:
  http://vimeo.com/user12885626
- Guía para configurar y agregar repositorios de software en distribuciones basadas en Debian:
  http://lihuen.linti.unlp.edu.ar (...)
- Lihuen GNU/Linux una distribución que cuenta, en su repositorio de software, con los paquetes necesarios para las actividades planteadas en este libro:
  http://lihuen.linti.unlp.edu.ar
- Laboratorio de Investigación en Nuevas Tecnologías Informáticas:
  http://linti.unlp.edu.ar
- Facultad de Informática - UNLP:
  http://info.unlp.edu.ar

Documentación de Python

- PyAr, comunidad Python de Argentina:
  http://python.org.ar
- Recopilación de PyAr de enlaces a tutoriales para aprender Python:
  http://python.org.ar/pyar/AprendiendoPython
- Manual de Python traducido al español:
  http://docs.python.org.ar/tutorial/contenido.html
- Sitio oficial de Python:
  http://python.org (En inglés)
- Recopilación del proyecto Python de documentación en español:
  http://wiki.python.org/moin/SpanishLanguage

Hardware abierto y otros entornos de programación para robots

- Sitio de los fabricantes de los robots usados en el curso (tiene además propuestas de enseñanza de programación usando el entorno MiniBloq y DuinoPack):
  http://www.robotgroup.com.ar
- Arduino, plataforma de hardware abierta en la cual están basada la placa controladora de los robots:
  http://www.arduino.cc (En inglés)
- Freeduino, plataforma compatible con Arduino que permite utilizar la marca Freeduino en productos derivados:
  http://www.freeduino.org (En inglés)
- Proyecto de educación con robots y Python del "Institute for Personal Robots in Education":
  http://www.roboteducation.org (En inglés)

Índice alfabético