

Manual de programación con robots para la escuela

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

Claudia Banchoff
Joaquín Bogado
Damian Mel
Sofía Martín
Fernando López

Equipo de soporte y desarrollo de Lihuen GNU/Linux
<http://lihuen.linti.unlp.edu.ar>
soportelihuen@linti.unlp.edu.ar

Versión del manual 1.0

27 de marzo de 2012

Índice general

1. Prefacio	5
1.1. Sobre este libro	5
1.2. Estructura del libro	5
1.3. Sobre los autores	5
2. Trabajar con Software Libre	7
2.1. Objetivo	7
2.2. Software libre en la escuela	7
2.3. ¿Por qué programar?	8
2.4. Python	8
2.4.1. Algo más sobre Python	9
2.5. ¿Qué necesitamos para empezar?	9
2.6. Por qué programar en Python	9
3. Introducción	11
3.1. El robot Múltiplo N6	11
3.2. Introducción al entorno de python	11
3.3. Por qué programar en Python	12
3.4. Requerimientos del sistema	12
3.5. Conectando el robot	12
3.6. Repaso del capítulo 1	15
3.7. Para pensar	16
4. Las funciones de movimiento	17
4.1. Dibujando figuras	19
4.1.1. Mi primer script	20
4.1.2. Actividades	21
4.2. Agrupando instrucciones en funciones	21
4.2.1. Nombres de función válidos	24
4.2.2. Funciones con argumento	24
4.2.3. Actividades	25
4.3. Agrupar funciones en módulos	26
4.3.1. Mi primer módulo	26
4.3.2. Uso de import	28
4.3.3. Actividades	28
4.4. Repaso del capítulo 4	28
5. Variables	29
5.1. Introducción	29
5.2. Utilizando variables	29
5.2.1. Guardando datos en variables	30
5.2.2. Tipos de variables	30
5.2.3. Variables en funciones	31
5.3. Listas en Python	32
5.4. Ingresar datos	32

ÍNDICE GENERAL

5.5. Actividades	33
5.6. Repaso del capítulo 5	33
6. Robots que deciden (valores de verdad y condicionales)	35
6.1. Introducción	35
6.2. Lo verdadero y lo falso	35
6.3. Símbolos y conectivas	36
6.4. Los valores de verdad en Python	38
6.4.1. Expresiones simples	38
6.4.2. Operadores lógicos	39
6.5. Condicionando nuestros movimientos	40
6.6. Resumen Python	42
6.7. Actividades	42
6.8. Lecturas Recomendadas	42
7. Simplificando	43
7.1. Objetivos	43
7.2. Sentencia de iteración <code>for</code>	43
7.3. Sentencia de iteración <code>while</code>	45
7.4. Actividades	47
7.5. Repaso del capítulo 7	47
8. Organizándonos	49
8.1. Estructura de un programa	49
8.2. Funciones	49
8.2.1. Definición de una función	49
8.2.2. Pasando argumentos	50
8.3. Cómo correr un programa	50
8.4. Actividades	51
8.5. Repaso del capítulo 8	51
9. Percibiendo el mundo: Sensores	53
9.1. Conociendo los sensores	53
9.2. Evitando choques	54
9.2.1. Primer aproximación	54
9.2.2. Como cambiar el rango de los valores de los sensores	55
9.2.3. Actividades	55
9.3. Velocidad proporcional a la distancia	56
9.3.1. Actividades	57
10. Invitando a la matemática	59
10.1. Trabajo con funciones de matemáticas	59
10.2. trabajar con diccionarios	59
10.3. Actividades	59
11. Apéndice 1: Guía de instalación de paquetes	61

Capítulo 1

Prefacio

1.1. Sobre este libro

El objetivo de este libro es introducir los primeros conceptos de programación a través de la manipulación de unos pequeños robots. El término “robótica” a veces aparece ligado a cualquier actividad que trata con robots. En nuestro caso, utilizamos los robots para aprender a programar. Al finalizar este libro, mencionaremos algunas líneas que están más relacionadas a la robótica. ¿A quiénes está dirigido el libro? Si bien está pensado para trabajar en un aula, 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 los docentes

A los alumnos

1.2. Estructura del libro

En los capítulos 1 y 2, bla bla bla El último capítulo bla bla bla

1.3. Sobre los autores

Todos los autores son docentes y/o investigadores que participan en proyectos de software libre desde hace varios años.

Claudia Banchoff Tzancoff es Licenciada en Informática de la Universidad Nacional de La Plata (UNLP). Actualmente es Secretaria de Extensión de la Facultad de Informática de la UNLP. Es profesora-investigadora del Laboratorio de Investigación de Nuevas Tecnologías Informáticas (LINTI). Coordina varios proyectos relacionados al software libre con distintos niveles de la educación (primario, secundario y universidad), fomentando Lihuen GNU/Linux y enseñar programación con robots personales y lenguaje Python.

Fernando López es alumno avanzado de la carrera de Licenciatura en Informática e integrante del equipo de desarrollo de Lihuen GNU/Linux desde 2006. Participa en foros de desarrollo de software libre de Debian y desarrolla aplicaciones de código libre con repositorio en sourceforge como LTSPConfig.

Sofía Martín es alumna avanzada de la carrera de Licenciatura en Informática e integrante del equipo de desarrollo de Lihuen GNU/Linux. Participa activamente en la comunidad de Software libre de Debian y desarrolla aplicaciones de código libre con repositorio en sourceforge como LTSPConfig. Aporta personalizaciones y regionalización de varias aplicaciones educativas.

Joaquín Bogado es ...

1. Prefacio

Capítulo 2

Trabajar con Software Libre

2.1. Objetivo

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.

2.2. 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 GNU¹. Allí se define:

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.

1. *La libertad de ejecutar el programa, para cualquier propósito (libertad 0).*
2. *La libertad de estudiar cómo trabaja el programa, y cambiarlo para que haga lo que usted quiera (libertad 1). El acceso al código fuente es una condición necesaria para ello.*
3. *La libertad de redistribuir copias para que pueda ayudar al próximo (libertad 2).*
4. *La libertad de distribuir copias de sus versiones modificadas a terceros (la 3^a libertad). Si lo hace, puede dar a toda la comunidad una oportunidad de beneficiarse de sus cambios. El acceso al código fuente es una condición necesaria para ello.*

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 libres”, es decir, con licencias que le dan a los usuarios todas las libertades que enumera 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

2. Trabajar con Software Libre

artículo "*Por qué el código abierto pierde el punto de vista del software libre*"¹, 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. Plantea 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 sus necesidades, es necesario que conozcan la mayor cantidad posible de opciones. Además, si cuentan con el acceso a los códigos fuentes, en caso de ser necesario, también podrían 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.

2.3. ¿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.

En este libro, vamos a aprender nociones 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.

2.4. Python

La primera pregunta que nos hace: ¿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 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++.

¹<http://www.gnu.org/philosophy/open-source-misses-the-point.es.html>

2.4.1. Algo más sobre Python

Antes que nada comencemos con un poco de historia. Este lenguaje fue desarrollado por Guido Van Rossum a principios de los años '90, en el centro de investigación en matemáticas CWI en Holanda, por lo que podemos decir que es un lenguaje bastante nuevo¹. Es una derivación del lenguaje ABC y su nombre fue inspirado en el grupo de cómicos ingleses "Monty Python"². Se ha utilizado y se utiliza ampliamente para programar tareas de administración de sistemas, para la producción de efectos especiales de películas, en varios sistemas informáticos de la NASA, para la gestión de grupos de discusión (ej. Yahoo lo utiliza), como parte de componentes de rastreadores Web y motores de búsqueda (por ejemplo, Google lo utiliza), para la realización de juegos de computadora, en la bioinformática, para enseñar a programar, etc. Como se puede ver es utilizado en diferentes ámbitos. Es un lenguaje que ha crecido de manera constante ya que el interés en su uso fue aumentado notablemente en los últimos años. Python es un lenguaje de alto nivel y muy fácil de aprender. Es muy expresivo y legible, esto significa que la sintaxis que utiliza permite escribir programas que se entiendan fácilmente. Con él se puede programar utilizando como lenguaje imperativo procedural o como lenguaje orientado a objetos, de ahí que decimos que es un lenguaje "multiparadigma". Respecto a la ejecución, Python es un lenguaje interpretado. Esto quiere decir que debemos contar con un intérprete para ejecutar nuestros programas.

2.5. ¿Qué necesitamos para empezar?

Como dijimos antes, Python es un lenguaje multiplataforma, es decir, que puede usarse tanto en sistemas Windows como en Linux o Mac, pero, como nosotros vamos a usar Linux como sistema operativo les detallamos qué aplicaciones deberemos instalar para empezar a trabajar.

Primero y principal, el intérprete Python y las librerías para utilizar el robot. Después, aunque no menos importante, un entorno para desarrollar nuestros programas.

En el siguiente capítulo, vamos a describir, paso por paso, cómo obtener e instalar cada una de estas cosas, tanto si usan sistemas Windows como si están usando Linux.

-ESTO NO IRIA ACA-

- Una computadora personal con Lihuen GNU/Linux instalado o algún sistema operativo similar basado en Debian GNU/Linux o Ubuntu.
- Los paquetes de software `python-serial` y `duinobot`
- Uno o más robots Múltiplo N6 con el módulo de comunicaciones XBee para el robot. Además el robot debe contar con un firmware compatible con `N6.XBee.PyFirmware.v1.0` o superior.
- El módulo de comunicaciones XBee con conexión para la computadora.
- Algún programa para editar código en python, con resaltado de sintaxis como Geany.

Si tienes dudas acerca de como instalar paquetes, no dudes en consultar el Apéndice 1: Guía de instalación de paquetes.

2.6. Por qué programar en Python

Contar sobre como interactuar con el interprete , ejemplos de uso de print, ayuda , explicar que hace import

Capítulo 3

Introducción

Una breve introducción acerca de cómo y porqué programar con robots interactivos

3.1. El robot Múltiplo N6

El robot *Múltiplo N6*, es un robot educativo extensible basado en la plataforma de prototipado Arduino. Como se ve en la figura 3.1 cuenta con 3 ruedas para su movilidad, dos delanteras con tracción y una trasera que gira libremente. Dos motores de corriente continua, sus respectivas cajas de reducción permiten al robot alcanzar velocidades respetables. El Múltiplo N6 cuenta también con 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. 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.

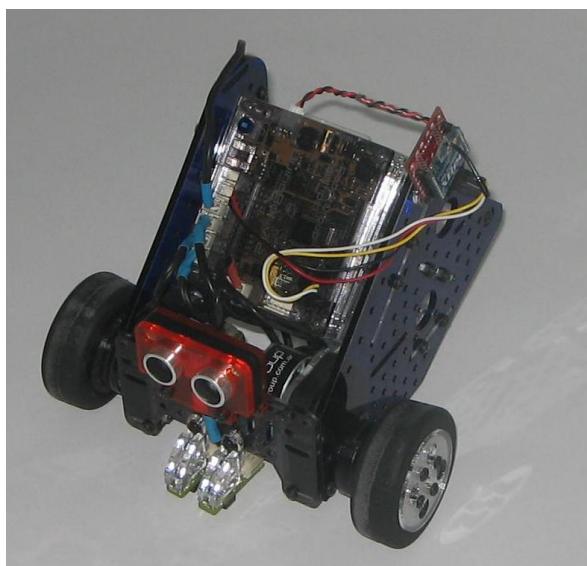


Figura 3.1. Robot Múltiplo N6 (PONER UNA FOTO DONDE SE VEAN LAS 3 RUEDAS?)

3.2. Introducción al entorno de python

sacar de las presentaciones de claudia

3. Introducción

*** Fernando: explicar algo de objetos, mensajes y métodos, y que en el curso no se verá *** esto en detalle

3.3. Por qué programar en Python

*****Contar sobre como interactuar con el intérprete , ejemplos de uso de print, ayuda , explicar que hace import*****

3.4. Requerimientos del sistema

Para el mejor aprovechamiento de esta guía, se recomienda contar con

- Una computadora personal con Lihuen GNU/Linux instalado o algún sistema operativo similar basado en Debian GNU/Linux o Ubuntu.
- Los paquetes de software `python-serial` y `duinobot`
- Uno o más robots Múltiplo N6 con el módulo de comunicaciones XBee para el robot. Además el robot debe contar con un firmware compatible con *N6.XBee.PyFirmware.v1.0* o superior.
- El módulo de comunicaciones XBee con conexión para la computadora.
- Algun programa para editar código en python, con resaltado de sintaxis como Geany.

Si tienes dudas acerca de como instalar paquetes, no dudes en consultar el Apéndice 1: Guía de instalación de paquetes.

3.5. Conectando el robot

Para empezar a realizar cualquier acción con el robot se debe establecer la comunicación entre la computadora que estamos usando y el robot. Para esto se utilizaremos un módulo de comunicaciones XBee, similar al que se ve en la figura 3.2.



Figura 3.2. Módulo de comunicaciones XBee para la computadora

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` en el sistema, el número puede variar. Puedes comprobarlo desde la terminal de *Geany* o desde algún otro emulador de terminal como `gnome-terminal` de la siguiente manera:

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

Luego es necesario prender el robot, el cual debe tener las pilas puestas y bien cargadas. 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. Puede verse la ubicación de estos botones en la figura 3.3 y el LED de estado en la figura 3.4.



Figura 3.3. Botones del robot N6

3. Introducción



Figura 3.4. LED de estado

Para conectarse al robot desde una consola interactiva de Python utilizando *Geany* o *gnome-terminal*, se pueden seguir los siguientes pasos:

```
1) 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.
>>>
```

El paso 1) invoca el intérprete de comandos de Python. Este intérprete permite conectarse con el robot para trabajar en modo interactivo. Todos los mensajes enviados al robot se ejecutarán inmediatamente.

```
2) >>> from robot import *
```

En el paso 2) se importan las funciones del robot al entorno. Esto permitirá acceder desde la consola interactiva de *Python* a las funciones que permiten conectar el robot y enviarle instrucciones a través del módulo de comunicaciones.

```
3) >>> b = Board("/dev/ttyUSB0")
>>> b
<robot.Board instance at 0xb743966c>
```

En la línea 3) se crea un objeto *Board* y se lo asigna a una variable llamada *b*. Este objeto representa el módulo de comunicaciones conectado a la computadora. Por eso el parámetro para crear el objeto que se le pasa a *Board()* es la ruta al dispositivo que se crea cuando se inserta el módulo de comunicaciones al puerto USB de la computadora, por ejemplo *"/dev/ttyUSB0"*.

Cuando escribe *b* seguido de enter en el intérprete, el mismo imprime información acerca del objeto. Para objetos de clase *Board* como *b* informa que es una instancia de la clase *Board* con memoria reservada o alocada a partir de la posición de memoria *0xb743966c*.

La clase *Board* implementa varias funciones de bajo nivel, que no usaremos directamente en este curso, pero que sin embargo conviene conocer. Uno de los mensajes de *Board* es *report()*, el cual

envía una señal a todos los robots que estén prendidos para que informen que efectivamente están prendidos y listos para recibir comandos. Además, los objetos del tipo Board pueden enviar comandos a los robots conectados directamente. Por ejemplo es posible, si el robot cuyo id es 1 está listo, enviar a través de un objeto Board el mensaje `b.motors(100, 100, 2, 1)`. Esto hará que el robot 1, avance a velocidad máxima con sus dos motores durante 2 segundos. Sin embargo, como veremos más adelante, es mejor en cuanto a claridad del código crear un objeto Robot y enviarle los mensajes al robot a través de este objeto que enviar mensajes al robot a través del objeto Board.

```
4)    >>> b.report()
      Robot 1 prendido
```

Con la variable `b` creada, es posible pedirle a los robots prendidos que se reporten. Aquellos robots que estén prendidos responderán diciendo su número de identificación como se indica en 4). Si hubiera más de un robot prendido y listo, todos contestarán con su número de identificación. Si hubiera más de un robot con el mismo número de identificación, será necesario cambiarlos para que todos tengan un número de identificación único. Mira la sección **Cambiando el número de identificación del robot** para aprender a cambiar el número de identificación del robot. Si ningún robot responde, prueba otra vez y no olvides prender el robot y presionar la tecla `run`.

```
5)    >>> robot = Robot(b, 1)
      >>> robot
      <robot.Robot instance at 0xb743c90c>
```

Una vez que sabemos el número de identificación de un robot listo, podemos crear un objeto robot. Para esto se escribe el nombre de la clase seguido de paréntesis en 5) y se guarda el resultado en una variable nueva `r`. Los parámetros pasados a `Robot(b, 1)` son: la variable que hace referencia al objeto `Board()`, `b`; y el número identificador del robot que queremos utilizar. Como en el caso de la variable `b`, si solo ponemos el nombre de la variable `r`, se imprime información acerca de lo que esta representa. En el ejemplo debajo de 5) podemos ver que `r`, luego de la invocación a `Robot()` es una instancia de la clase `Robot`, con memoria alocada a partir de la dirección `0xb743c90c`.

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

¡Genial!. Ahora es posible mandarle mensajes al robot. Una de las pruebas más simples es pedirle al robot que emita un sonido. El robot Múltiplo N6 tiene un pequeño parlante interno con el que puede emitir frecuencias simples en el orden de 1 Hz a 15 KHz. El ejemplo de la línea 6) indica al robot que emita un tono de 200 Hz durante medio segundo. Puedes variar la frecuencia y la duración del sonido invocando a `beep()` con otros argumentos.

Es posible asignarle un nombre al robot enviándole el mensaje `setName("nombre")`. El parámetro `nombre` debe ser un string (una cadena de caracteres) el cual se convertirá en el nuevo nombre del robot.

```
7)    >>> robot.setName("R. Daneel Olivaw")
8)    >>> robot.speak("Hola, mi nombre es " + r.getName())
      Hola, mi nombre es R. Daneel Olivaw
```

En este ejemplo, en 7) se le pone al robot el nombre de un famoso robot de la literatura enviándole el mensaje `setName()`. Este nombre no es almacenado en la memoria interna del robot a diferencia del número de identificación, por lo tanto será necesario ponerle un nombre cada vez que se reinicie el intérprete de Python. En 8) se le pide al robot que diga su nombre. Esto se logra enviándole el mensaje `speak()`, que al igual que `setName()`, recibe como parámetro un string. Este a su vez está compuesto por la concatenación de dos strings, "Hola, mi nombre es " y el resultado de la función `getName()`, que también es un string. Ambos strings están concatenados usando el operador de concatenación de strings, `+`.

3.6. Repaso del capítulo 1

En este capítulo hemos aprendido a conectar el robot a la computadora, creando objetos Board y Robot. Vimos como mandarle mensajes simples al robot utilizando las funciones `beep()`, `setName()`, `getName()` y `speak()`. Aprendimos a concatenar strings.

3.7. Para pensar

Si la variable b es un Board y robot un Robot, ¿por qué puedo hacer b.report() y no robot.report()?
¿?

¿Qué es una clase? ¿y un objeto? ¿y un mensaje?

¿Cómo puedo saber que mensajes o métodos puedo utilizar con un objeto?

Capítulo 4

Las funciones de movimiento

Ahora que sabemos conectarnos con el robot, podemos hacer algo interesante con él. Hay 6 mensajes básicos de movimiento que permiten al robot avanzar, retroceder, doblar a la izquierda, doblar a la derecha, parar y controlar las ruedas independientemente.

Para mover el robot, nos conectamos de la forma detallada en el capítulo 3 y luego comenzamos a darle instrucciones con los mensajes correspondientes. Por ejemplo, en la figura 4.1, en la línea 2) se invoca al mensaje `forward()` que hace que el robot avance a velocidad media por un tiempo indeterminado. Para detener el robot se puede invocar a `stop()` como en 3).

```
1)  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 robot import *
    >>> b = Board("/dev/ttyUSB0")
    >>> b.report()
    Robot 1 prendido
    >>> robot = Robot(b, 1)
2)  >>> robot.forward()
3)  >>> robot.stop()
```

Figura 4.1. Primeros movimientos

El mensaje `forward()` soporta cero, uno o dos argumentos. Si no se le indican argumentos, como en 2), `forward()` hace que el robot avance a velocidad media por un tiempo indeterminado. Para detener el robot se le envía el mensaje `stop()` como en 3). Si se envía el mensaje con un solo argumento, éste indicará la velocidad a la cual tiene que avanzar el robot, como se ve en la figura 4.2. En este caso al igual que con los demás mensajes de movimiento, el robot se moverá a la velocidad indicada hasta que se le indique al robot que debe detenerse enviándole el mensaje `stop()`. La velocidad es un valor numérico entre -100 y 100, donde 0 indica que el robot no debe morverse, 100 indica que el robot se moverá a velocidad máxima y -100, que el robot se moverá a velocidad máxima pero hacia atrás.

```
>>> robot.forward(100)
```

Figura 4.2. `forward()` solo con velocidad

Si se envía el mensaje `forward()` con dos argumentos, 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. En la figura 4.3, el ejemplo hace que el robot que avance a velocidad máxima durante 2 segundos.

4. Las funciones de movimiento

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

Figura 4.3. forward() con dos parámetros

Es posible enviar el mensaje `forward()` indicando el tiempo pero no la velocidad, esto se hace enviándole un solo parámetro y usando su nombre, para averiguar los nombres de los parámetros se puede usar `help(Robot.forward)`, todo esto sirve para cualquier mensaje de un objeto y para cualquier función (veremos qué son y como crear funciones más adelante). En la figura 4.4 se puede ver como enviar el mensaje `forward()` al robot para que avance 5 segundos sin especificar la velocidad.

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

Figura 4.4. forward() con un parámetro por nombre

Incluso, si usamos los nombres de los argumentos los podemos enviar desordenados como se ve en la figura 4.5 aunque no sea ni práctico ni bonito, al menos en este ejemplo.

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

Figura 4.5. forward() con dos parámetros desordenados

La función para retroceder se llama `backward()` y acepta los mismos argumentos que `forward()`. En la figura 4.6 se indica al robot que retroceda a velocidad 20 durante 5 segundos.

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

Figura 4.6. backward() con dos parámetros

Los métodos `forward()` y `backward()` pueden usarse indistintamente, ya que 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 avanza y `backward()` cuando el robot retrocede.

Para girar a izquierda existe la función `turnLeft()`, si invocamos esta función sin argumentos el robot va a girar 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()`.

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

Figura 4.7. Uso de `turnLeft()` y `turnRight()`

En la figura 4.7 se muestra como enviar los mensajes `turnLeft()` y `turnRight()`.

Por ejemplo en la figura 4.8 se hace girar el robot durante 3 segundos hacia la izquierda a velocidad máxima y luego hacia derecha durante 2 segundos a velocidad 30.

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

Figura 4.8. Girar en una dirección y luego en otra

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

Cuadro 4.1. Funciones bloqueantes y no bloqueantes

Los mensajes de movimiento como ya vimos pueden enviarse con parámetros o sin ellos, la diferencia al momento de utilizarlos en el intérprete interactivo de Python es que algunas de estas variantes se bloquean hasta que terminan, impidiendo que podamos escribir otras sentencias. De la misma forma cuando empecemos a escribir y guardar programas notaremos que estas mismas variantes demoran la ejecución de las sentencias subsiguientes del programa hasta que terminan.

En la tabla 4.1 podemos ver las diferentes formas de utilizar estos mensajes y cuáles se bloquean demorando la ejecución de otras sentencias.

Que un mensaje se bloquee hasta terminar o no, no es bueno ni malo, a lo largo de este manual les vas a encontrar un uso a ambos tipos de métodos.

Existe también el mensaje `wait()`, el mismo detiene (bloquea) el programa por la cantidad de segundos que indiquemos. En el ejemplo de la figura 4.9 usamos `wait()` para simular el comportamiento del ejemplo de la figura 4.8 pero usando las versiones no bloqueantes de `turnLeft()` y `turnRight()`.

```
>>> robot.turnLeft(100)
>>> robot.wait(3)
>>> robot.turnRight(30)
>>> robot.wait(2)
```

Figura 4.9. Girar en una dirección y luego en otra con `wait()`

4.1. Dibujando figuras

Con las cosas que aprendimos ya podemos mover el robot en cualquier dirección, si todavía no probaste los mensajes del robot que vimos antes, probalos jugando y haciendo que el robot se mueva en distintas direcciones y con distintas velocidades.

Una vez que te familiarices con las funciones para girar, avanzar y retroceder podés empezar a combinarlas para hacer cosas más interesantes, por ejemplo podemos dibujar un cuadrado con el robot.

Para dibujar un cuadrado básicamente el robot tiene que avanzar y girar 90 grados, cuatro veces. Un recurso que usamos los programadores para bosquejar un programa es lo que llamamos pseudocódigo, es básicamente describir un programa en palabras, sin usar ningún lenguaje de programación en particular. En la figura 4.10 se puede ver el ejemplo del cuadrado en pseudocódigo, estas líneas no funcionan en el intérprete de Python ya que no son sentencias propias del lenguaje de programación Python, de hecho es un lenguaje inventado, solamente para tener una idea de como quedará el programa una vez escrito.

```
avanzar
girar 90°
avanzar
girar 90°
avanzar
girar 90°
avanzar
girar 90°
```

Figura 4.10. Programa para dibujar un cuadrado en pseudocódigo

Vamos a tener que experimentar el uso del mensaje `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. También es importante experimentar con alguno de los mensajes para girar, por ejemplo `turnRight()`, 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 te preocupes 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.

4.1.1. Mi primer script

Una vez calculados los valores necesarios para cada función, pasamos a escribirlos en el lenguaje Python, pero esta vez en lugar de usar el intérprete los invitamos a escribirlo en un editor de texto, en GNU/Linux pueden usar Geany o cualquier otro editor, sobre todo si resalta sintaxis. Escribir el programa en un archivo nos va a permitir por supuesto guardarlo, reutilizarlo y modificarlo en cualquier momento sin tener que escribir todo de nuevo, también tenemos que recordar poner las instrucciones necesarias para conectarnos con el robot como vimos en el capítulo [3](#).

De esta manera el programa debería quedar parecido al ejemplo de la figura [4.11](#), pero usando los valores que se calcularon para el robot que se está utilizando. En el código vas a 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).

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

robot.forward(50, 0.5)
robot.wait(1)
robot.turnRight(35, 1)
b.exit()
```

Figura 4.11. Dibujar un cuadrado con el Robot

Para probar el ejercicio hay que guardar este programa en un archivo, preferentemente con la extensión “.py”, por ejemplo “cuadrado.py”. Luego podemos ejecutarlo desde la terminal, como se ve en la figura [4.12](#) escribiendo `python` seguido del nombre del archivo que acabamos de generar.

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

Figura 4.12. Ejecutar cuadrado.py desde la terminal

Los archivos que agrupan código Python de esta manera se llaman **scripts** o a veces **programas** y

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

Figura 4.13. Girar usando `motors()` y `stop()`

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.

Los robots tienen un mensaje `motors()` que permite controlar las ruedas de forma independiente, este mensaje puede tener 2 argumentos indicando la velocidad de los motores derecho e izquierdo. La velocidad es un rango entre -100 y 100, indicando con el signo la dirección en la que girará cada rueda. En la figura 4.13 se hace girar al robot y luego se lo detiene con `stop()` igual que al resto de las funciones de movimiento.

También es posible indicar cuántos segundos debe moverse el robot con un tercer argumento.

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

Figura 4.14. Girar usando `motors()`

4.1.2. Actividades

1. Usando lo experimentado hasta ahora escriba un programa para dibujar un triángulo equilátero (los angulos internos miden 60 grados), no te preocupes porque el triángulo quede perfecto, es muy difícil lograrlo, solamente hacé algo que se parezca a un triángulo.
2. Escriba un programa que haga que el robot avance en zigzag.
3. Usando `motors()` se puede indicar que una rueda se mueva más lenta que la otra para dibujar círculos, controlando la diferencia de velocidad se pueden hacer círculos más grandes o más chicos. Experimente el mensaje `motors()` para crear círculos de distintos tamaños.
4. Usando las funciones de movimiento realice un programa que recorra un laberinto realizado sobre el piso con obstáculos físicos o con líneas.

4.2. Agrupando instrucciones en funciones

Hasta ahora con mayor o menor dificultad pudimos escribir programas para hacer cuadrados, triángulos y círculos. Estos programas hacen una sola cosa (un tipo de figura) 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 pseudocódigo en la figura 4.15, usamos el símbolo `#` (al igual que en Python) para denotar un comentario, un comentario es un texto que no es ejecutable y que sirve de guía para los programadores que miren el código.

4. Las funciones de movimiento

```
# 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°
```

Figura 4.15. Programa que dibuja 3 figuras en pseudocódigo

Es evidente que el código es demasiado largo para la tarea realizada, en parte esto es por repetir la porción de código que dibuja el cuadrado. Afortunadamente existe una forma de escribir la porción que se repite una sola vez, asignarle un nombre a esa porción de código e invocarlo escribiendo solamente su nombre, esta forma se llama función.

Nuestro objetivo es evitar repetir código siempre que sea posible¹, nuestro ejemplo debería quedar similar a la figura 4.16.

¹Todavía no sacaremos todos los repetidos, seguiremos teniendo repetido el código para dibujar cada lado, más adelante veremos como resolver esto.

```

# Invocación a cuadrado
cuadrado()

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

#Segunda invocación a cuadrado
cuadrado()

```

Figura 4.16. Pseudocódigo de la definición y uso de una función

Una función es, como se describió, un mecanismo que permite agrupar código bajo un nombre y luego invocar el código varias veces usando ese nombre. La forma más simple de explicar esto es con un ejemplo.

```

from robot import *
b = Board("/dev/ttyUSB0")
robot = Robot(b, 1)

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

# Dibujar cuadrado
cuadrado()
# Dibujar triángulo
robot.forward(50, 0.8)
robot.wait(1)
robot.turnRight(45, 1)
robot.forward(50, 0.8)
robot.wait(1)
robot.turnRight(45, 1)
robot.forward(50, 0.8)
robot.wait(1)
robot.turnRight(45, 1)
# Dibujar cuadrado
cuadrado()
b.exit()

```

Figura 4.17. Primer función

La figura 4.17 muestra como hacer una función para dibujar un cuadrado. La primer línea de la definición de cualquier función comienza con la palabra clave `def`, seguida del nombre de la función y finalmente `()`. Adentro, en el cuerpo de la función, cada línea tienen que comenzar con una tabulación o con dos espacios (hay que elegir una forma de hacerlo y hacerlo siempre así, sino el programa puede fallar).

4.2.1. Nombres de función válidos

Los nombres de variables, funciones y de argumentos en Python siguen las siguientes 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).

En las figuras 4.18 y 4.19 se pueden ver ejemplos de nombres de funciones válidas y no válidas, las mismas reglas se aplican a los nombres de los argumentos y a los nombres de las variables (más adelante veremos en detalle qué son y cómo usar variables).

```
def mi_funcion():
    pass
def _mi_funcion():
    pass
def _3_mifuncion():
    pass
def mi3funcion():
    pass
```

Figura 4.18. Nombres de función válidos

```
def funcion_ñ():
    pass
def _mi_función():
    pass
def 3_mifuncion():
    pass
def mi3 funcion():
    pass
```

Figura 4.19. Nombres de función no válidos

4.2.2. Funciones con argumento

Volviendo al ejemplo de la figura 4.11, pensemos qué tendríamos que hacer si ahora quisieramos hacer cuadrados de distintos tamaños: tendríamos que modificar la velocidad (y/o el tiempo) en cada invocación a `forward()`, esto es bastante incómodo ya que hay que invocar a `forward()` cuatro veces para dibujar un cuadrado. Esto 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 figura 4.17 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 y ahora queremos hacerlos de distintos tamaños, para lograr esto podemos usar funciones con parámetros de nuevo lo mostraremos con un ejemplo en la figura 4.20.

```

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

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

```

Figura 4.20. Función cuadrado con tiempo por parámetro

En la primer línea de la figura 4.20 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 función podemos usar “tiempo” como si fuera el valor que representa. Luego en cada invocación a `forward()` reemplazamos el valor que usabamos para la duración del movimiento por el nombre del parámetro.

Como vimos anteriormente los mensajes `forward()` y `turnLeft()` pueden recibir más de un parámetro, los mensajes se implementan con funciones (a las funciones de un objeto se las conoce como métodos) así que una función cualquiera también puede recibir más de un parámetro, en nuestro caso todavía queda algo feo en nuestra función “cuadrado()” y es que para que cuadrado funcione, la variable que representa a nuestro robot debe llamarse “robot”, además si tenemos más de un robot conectado al mismo tiempo podríamos usar `cuadrado()` sólo para uno de ellos.

Pasemos ahora como argumento también 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 la figura 4.21 se puede ver una implementación de la función cuadrado que recibe el robot como argumento, en las últimas líneas se puede ver que se usa cuadrado para 2 robots distintos.

4.2.3. Actividades

1. Cree una función para dibujar triángulos, aprovechando el código que escribió en las actividades anteriores y ejecutela. Esta función no debe recibir parámetros.
2. Cree la función `mareado()` que haga girar el robot varias veces en una y otra dirección.
3. Modifique la función para que el robot y la longitud de los lados del triángulo se pasen como argumentos.
4. Cree una función que reciba 2 robots como argumentos, y envíe uno hacia la derecha y el otro hacia la izquierda.
5. 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.

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

Figura 4.21. Función cuadrado con tiempo y robot por parámetro

4.3. Agrupar funciones en módulos

En las actividades anteriores escribiste varias funciones que pueden combinarse de distintas formas para hacer programas distintos. Si juntás todas esas funciones en un archivo (teniendo cuidado de que cada función tenga un nombre distinto) tenés lo que en Python se conoce como módulo. Durante el transcurso del manual ya viste y usaste el módulo `robot`, el objetivo de esta sección es que aprendas a hacer y a usar tus propios módulos.

4.3.1. 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. En la figura 4.22 hay un ejemplo de módulo con algunas de las funciones de las actividades anteriores, pueden copiarlo y guardarla como “movimientos.py” o, mejor aún, tomar las funciones que escribieron ustedes y ponerlas en un archivo con ese nombre.

Para usar un módulo hay que abrir una terminal en la carpeta que contiene el módulo y luego importarlo. En la figura 4.23 vemos como importar el módulo desde el interprete y usarlo, como pueden ver hay que hacer casi lo mismo que con el módulo `robot`. Para importar el módulo `robot` no hacía falta posicionarse en ningún directorio porque es un módulo del sistema.

```

def circulo(robot, vel, diferencia):
    robot.motors(vel, vel + diferencia)

def zigzag(robot):
    robot.forward(50, 1)
    robot.turnRight(100, 0.3)
    robot.forward(50, 1)
    robot.turnLeft(100, 0.3)
    robot.forward(50, 1)
    robot.turnRight(100, 0.3)
    robot.forward(50, 1)
    robot.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)
    r1.wait(tiempo)
    r1.stop()
    r2.stop()

def mareado(robot):
    robot.turnRight(50, 5)
    robot.turnLeft(50, 5)
    robot.turnRight(50, 5)
    robot.turnLeft(50, 5)

```

Figura 4.22. Módulo de movimientos: movimientos.py

```

usuario@host:~$ cd mis_modulos
usuario@host:~/mis_modulos$ ls
movimientos.py
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 robot 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)

```

Figura 4.23. Uso del módulo movimientos desde el intérprete de Python

En la línea marcada 1) de la figura 4.23 se importa el módulo movimientos, en 2), 3) y 4) se usan las funciones del módulo de forma directa. De esta forma las funciones definidas en el archivo movimientos

podrán ser usadas sin tener que definirlas cada vez que necesitemos su funcionamiento. Si quisieramos hacer algún cambio o corregir algún error en el archivo, para utilizar las funciones del archivo con las modificaciones realizadas, tenemos que VER!!! recargar el modulo

4.3.2. Uso de import

Existe otra forma de importar módulos, si tuvieramos 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 que 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 robot import *
1)   >>> import funcionesSinParametros
2)   >>> import funcionesConParametros
      >>> b = Board("/dev/ttyUSB0")
      >>> robot = Robot(b, 1)
3)   >>> funcionesSinParametros.cuadrado()
4)   >>> funcionesConParametros.cuadrado(robot, 2)
```

Figura 4.24. Uso de 2 versiones distintas de `cuadrado()`

En las líneas 1) y 2) de la figura 4.24 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`.

4.3.3. Actividades

1. 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.

4.4. Repaso del capítulo 4

En este capítulo aprendimos a mover el robot con los mensajes `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.

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 enviando el mensaje `wait()` a cualquier robot (si tenemos varios no importa de cuál).

Capítulo 5

Variables

5.1. Introducción

Ya hemos visto cómo mover el robot dibujando figuras geométricas, al incorporar el uso de funciones vimos cómo reutilizar código agrupando instrucciones, en este capítulo veremos la facilidad que nos aporta guardar nuestros datos para usarlos en varios instancias de nuestro código sin necesidad de repetir su valor, como así también, cómo ingresar datos a medida que se los necesite.

5.2. Utilizando variables

Repasando lo que vimos en el capítulo 4 para dibujar un cuadrado es necesario probar los valores de velocidad y tiempo que nos permitirán mover el robot de manera que nos salga la figura deseada. Debido a que debemos escribir estos datos en cada instrucción utilizada, cada vez que queremos modificar estos datos tenemos que cambiarlo en cada instancia. Para evitar la repetición de datos podemos definir variables que guarden los valores durante la ejecución del código y de esta forma en lugar de escribir la velocidad y el tiempo en cada instrucción, utilizamos los nombres de las variables definidas y cuando queramos modificar uno de sus valores lo cambiaremos una sola vez (en la asignación de la variable), veamos un ejemplo del uso de variables en el siguiente código:

```
tiempo = 2
r.forward(50, tiempo)
r.turnRight(35, tiempo)
r.wait(1)
```

Sin darnos cuenta ya hemos usado variables durante los ejemplos anteriores, por ejemplo al conectarnos con el robot, creamos una variable para manejarlo.

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

Mediante el almacenamiento de datos en variables se facilita el envío de datos a las funciones, un ejemplo de uso de variables para enviar parámetros se practicó en el capítulo 4. En la definición de la función cuadrado vista en la sección 4.2.2 se recibe como parámetro el tiempo durante el cual se va a mover el robot por cada lado. Definiendo el tiempo como variable podemos llamar a varias funciones utilizando la misma variable como parámetro, como ejemplo tomaremos las funciones cuadrado y zigzag definidas en la sección 4.3 y en la sección 4.3.1 respectivamente recordemos que podemos guardar en un archivo aquellas funciones que necesitamos usar, importándolo para su utilización en el intérprete.

```
usuario@host:~$ cd mis_modulos
usuario@host:~/mis_modulos$ ls
movimientos.py
usuario@host:~/mis_modulos$ python
Python 2.6.6 (r266:84292, Dec 27 2010, 00:02:40)
[GCC 4.4.5] on linux2
```

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División entera o no entera
%	Resto división entera
>, <	Comparación mayor - menor
==	Comparación igualdad

Cuadro 5.1. Operadores en Python

```
Type "help", "copyright", "credits" or "license" for more information.
>>> from robot import *
>>> from movimientos import *
>>> b = Board("/dev/ttyUSB0")
>>> robotazo = Robot(b, 1)
>>> tiempo = 0.5
>>> cuadrado(robotazo, tiempo)
>>> zigzag(robotazo, tiempo)
```

5.2.1. Guardando datos en variables

Para almacenar datos en variables usamos la sentencia de asignación de Python, los valores que pueden guardarse pueden ser números o strings (tener en cuenta que los strings deben estar encerrados entre comillas “”):

```
<nombre_variable> = <valor>
velocidad = 20
nombre = "MI primer robot"
```

Al definir una variable el nombre que debemos utilizar debe cumplir con una serie de reglas definidas por el lenguaje Python al igual que se vió en la sección [4.2.1](#):

- 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).

5.2.2. Tipos de variables

Los tipos de datos de las variables se asignan dinámicamente, por lo tanto al utilizar los operadores de Python entre variables de diferentes tipos puede generar resultados inesperados. En el cuadro [5.1](#) podemos ver los operadores disponibles en Pyhton.

Muchos operadores ya se conocen del área de las matemáticas, como +, -, * y /, hay que 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, si en cambio uno de los números es del tipo float (contiene parte decimal), retorna un valor con punto decimal.

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:

Operación	Resultado
5+10	15
“Hola”+ “!”	“Hola!”
10 / (2+3)	2
10 >5	True
10/3.5	2.8571428571
10/3	3
10 %3	1
“hola”*3	“holaholahola”

Cuadro 5.2. Ejemplos Operadores matemáticos en Python

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

Figura 5.1. Asignación de valores a variables

5.2.3. Variables en funciones

***** Fernando: A este parrafo habría que darle una vuelta de tuerca *****
Como ya vimos el uso de variables nos permite automatizar el cambio de un valor evitando la modificación individual en cada sentencia que lo utiliza. Otra alternativa es definir variables dentro de una función como vemos en la figura 5.2, en este caso la variable sólo se puede usar dentro de la función cuadrado y no se puede pasar por parámetro para modificar su valor, en cambio, el tiempo durante el cual se mueve se define fuera de la función y la igual que el robot se pasa como parámetro a la función cuadrado.

5. Variables

```
t = 3
def cuadrado(robot, tiempo):
    esperar = 1
    lado = 50
    vuelta = 35
    robot.forward(lado, tiempo)
    robot.turnRight(vuelta, 1)
    robot.wait(esperar)
    robot.forward(lado, tiempo)
    robot.turnRight(vuelta, 1)
    robot.wait(esperar)
    robot.forward(lado, tiempo)
    robot.turnRight(vuelta, 1)
    robot.wait(esperar)
    robot.forward(lado, tiempo)
    robot.turnRight(vuelta, 1)
    robot.wait(esperar)
cuadrado(r, t)
```

Figura 5.2. Definición de variables dentro de una función

5.3. Listas en Python

FALTA!!!!

```
a = ['Entre Ríos', 'Corrientes', 'Misiones']
```

5.4. Ingresar datos

Los valores que se utilizan como parámetro en las funciones pueden ser cambiados manualmente o bien se puede solicitar que se ingresen los valores en el momento de la ejecución. Para esto utilizaremos la función `input` que nos permite ingresar datos por teclado. Con esta función solicitamos los valores en el momento de ejecución.

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

Figura 5.3. Ingreso de datos por teclado

5.5. Actividades

Ejercicios: Modificar la función zigzag reemplazando los valores por variables Mejore la interacción con el usuario, cambiar los valores usados en las sentencias forward y turnRight definiendo variables Modificar

5.6. Repaso del capítulo 5

Aquí escribir el repaso de lo que vimos en este capítulo

5. Variables

Capítulo 6

Robots que deciden (valores de verdad y condicionales)

6.1. Introducción

Los robots pueden, no solamente seguir una secuencia de instrucciones detallada, sino tomar decisiones, llevandólos un paso más cerca de la inteligencia y la autonomía.

Durante este capítulo aprenderemos lo necesario para que un robot tome una decisión empezando con lógica booleana y terminando con las sentencias de Python que nos permiten evaluar condiciones y actuar en consecuencia.

Las secciones 6.2 y 6.3 son una introducción a la lógica proposicional, el alumno impaciente que tenga una base de lógica mínima puede leer secciones posteriores sin perderse nada.

6.2. Lo verdadero y lo falso

Nosotros “sabemos” intuitivamente que afirmaciones son verdaderas y cuales son falsas, coloquialmente combinamos proposiciones simples para formar proposiciones compuestas y “sabemos” si son verdaderas o falsas. A pesar de que la mayoría de las personas es bastante buena distinguiendo lo verdadero de lo falso, el método intuitivo no sirve para la ciencia, para decir que algo es una verdad científica tendremos que justificarlo formalmente siguiendo una serie de reglas y trabajando con una evidencia o suposición inicial.

El lenguaje coloquial tiene cierta ambigüedad que no puede ser tolerada por un programa, los intérpretes de lenguajes de programación como Python no pueden deducir el valor de verdad de una expresión que sea ambigua. Por esto Python (como el resto de los lenguajes de programación) provee una serie de reglas para expresar y evaluar el valor de verdad de determinadas proposiciones, estas reglas están basadas en la lógica proposicional.

Algunas definiciones de “proposición” extraídas de Wikipedia¹:

- [...] entidades portadoras de los valores de verdad.
- Un enunciado lingüístico (generalmente en la forma grammatical de una oración enunciativa) puede ser considerado como proposición lógica cuando es susceptible de poder ser verdadero o falso. Por ejemplo “Es de noche” puede ser Verdadero o Falso.
- Se llama proposición atómica, o simple, cuando hace referencia a un único contenido de verdad o falsedad; vendría a ser equivalente a la oración enunciativa simple en la lengua.
- Proposición molecular cuando está constituida por varias proposiciones atómicas unidas por ciertas partículas llamadas “nexos o conectivas”, que establecen relaciones sintácticas como función de coordinación y subordinación determinadas entre las proposiciones que la integran; tal ocurre en la función de las conjunciones en las oraciones compuestas de la lengua.

¹<http://es.wikipedia.org/w/index.php?title=Proposici%C3%B3n&oldid=54669817s>

6. Robots que deciden (valores de verdad y condicionales)

Definición de “lógica proposicional” extraída de Wikipedia ²: “Una lógica proposicional es un sistema formal cuyos elementos más simples representan proposiciones, y cuyas constantes lógicas, llamadas conectivas, representan operaciones sobre proposiciones, capaces de formar otras proposiciones de mayor complejidad.”

Veamos algunas proposiciones simples:

1. El cielo es azul
2. Los gatos tienen dos colas
3. Los gatos tienen cuatro patas
4. Los perros tienen tres ojos

Es muy fácil darse cuenta las respuestas para todas estas afirmaciones, nosotros sabemos que las afirmaciones **1** y **3** son verdaderas (al menos los días soleados y para la mayoría de los gatos respectivamente).

Construyamos ahora proposiciones compuestas (moleculares), para esto tomaremos las proposiciones simples (atómicas) y las uniremos con la conjunción “y” y con la disyunción “o”.

1. El cielo de la tierra es azul y los gatos tienen cuatro patas
2. El cielo de la tierra es verde y los gatos tienen dos colas
3. El cielo de la tierra es verde o los perros tienen tres ojos
4. Mañana lloverá o nevará o estará nublado o saldrá el sol

Ahora la cuestión es un poco más complicada, la complicación tiene que ver con la ambigüedad del lenguaje coloquial, pero si partimos desde los valores de verdad de las proposiciones atómicas es muy simple. Es obvio que **1** y **4** son sentencias verdaderas. **1** es verdadera porque sabemos que las dos sentencias que las componen son verdaderas (los días soleados y para la mayoría de los gatos), por lo tanto la conjunción de ambos es verdadera, **4** es verdadera porque sabemos que al menos una de las tres sentencias que la componen es verdadera, siendo verdadera una disyunción si al menos una de las sentencias es verdadera.

Son menos obvias las respuestas para **3** y **2** pero se pueden solucionar fácilmente si los analizamos cada proposición atómica y las conectivas que las unen. La proposición molecular **3** está compuesta de proposiciones atómicas unidas por una disyunción, por lo tanto con que una de sus proposiciones atómicas sea verdadera alcanza para que toda la proposición sea verdadera, teniendo en cuenta que el cielo de la tierra no es verde y que los perros no tienen tres ojos, podemos afirmar que toda la proposición es falsa. En cuanto a la proposición **2**, la misma está unida por una conjunción, con lo que para ser verdadera, ambas proposiciones atómicas deben ser verdaderas, es decir si determinamos que alguna es falsa podemos decir que toda la proposición molecular es falsa, cómo ya dijimos que el cielo de la tierra no es verde ese conocimiento es suficiente para firmar que toda la proposición molecular es falsa (sin necesidad de evaluar la otra proposición atómica).

6.3. Símbolos y conectivas

Las proposiciones atómicas pueden ser reemplazadas por símbolos que las representan (típicamente se usan las letras p, q, r, s y t) que se parecen mucho a las variables de los programas y por símbolos que representan funciones de verdad, las funciones de verdad son las conectivas como la conjunción, la disyunción y las negaciones que en los lenguajes de programación se conocen como operadores lógicos.

Veamos primero en la tabla **6.1**, los símbolos que se usan para representar las conectivas en la lógica proposicional (más adelante veremos como se escriben en Python).

Cada conectiva tiene una tabla de verdad que establece los valores que toma en base a sus operandos, vea las tablas **6.2**, **6.3** y **6.4** para guiarse cuando tenga que usar estas conectivas.

²http://es.wikipedia.org/w/index.php?title=L%C3%B3gica_proposicional&oldid=53612306

Conectiva	Nombre	Ejemplo	Se lee	Descripción
\wedge	Conjunción (y)	$p \wedge q$	p y q	Es verdadera si tanto p como q son verdaderos
\vee	Disyunción (o)	$p \vee q$	p o q	Es falsa si tanto p como q son falsas
\neg	Negación (no)	$\neg p$	no p	Es verdadera si p es falsa

Cuadro 6.1. Conectivas de la lógica proposicional

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

Cuadro 6.2. Tabla de verdad de la conjunción

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

Cuadro 6.3. Tabla de verdad de la disyunción

p	$\neg p$
V	F
F	V

Cuadro 6.4. Tabla de verdad de la negación

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

Cuadro 6.5. Operadores de comparación

6.4. Los valores de verdad en Python

En Python existen variables que representan valores de verdad, son las variables de tipo booleano, las mismas pueden tomar 2 valores `True` o `False` que representan los valores verdadero y falso.

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

6.4.1. Expresiones simples

Una forma de armar expresiones que tengan un valor de verdad (análogas a las proposiciones atómicas que vimos antes) es por ejemplo comparando otras expresiones o variables, veamos unos ejemplos:

```
>>> 5 < 10
True
>>> 5 > 10
False
>>> "Gómez" < "Pérez"
True
>>> "Gómez" == "Pérez"
False
>>> "Gómez" != "Pérez"
True
>>> def entre(valor, min, max):
...     return min <= valor <= max
...
>>> entre(25, 0, 10)
False
>>> entre(10, 0, 10)
True
>>> entre(4, 0, 10)
True
>>> 5 + 1 > 5
True
>>> 2 * 20 == 41
False
```

Las expresiones booleanas utilizadas para controlar la repetición del bloque debe devolver `true` o `false` en el momento su evaluación, los operadores utilizados para comparar variables o valores se pueden ver en la tabla 6.5.

Al momento de evaluación de las expresiones el lenguaje determina el valor de verdad de cada una para decidir si ejecuta el bloque o lo finaliza, en la tabla 6.6 se pueden observar algunos ejemplos del uso de los operadores y el valor de verdad que retornan al evaluarse.

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

Cuadro 6.6. Ejemplos operadores de comparación

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)

Cuadro 6.7. Operadores lógicos

6.4.2. Operadores lógicos

En el siguiente ejemplo se pueden ver las conectivas (de ahora en más operadores lógicos) de conjunción, disyunción y negación, en Python se escriben `and`, `or` y `not` respectivamente y son palabras reservadas (es decir no se pueden usar como nombres para variables, funciones, etc...):

```
>>> a = True
>>> b = False
>>> a and b
False
>>> a or b
True
>>> a and True
True
>>> False or True
True
>>> not False
True
```

Cuando se combinan varios operadores hay que tener en cuenta en qué 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:

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

Lo que sucede la última expresión del ejemplo anterior es que 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".

En la tabla 6.7 se describen los operadores lógicos ordenados en orden decreciente de precedencia.

Por supuesto se pueden usar las expresiones de comparación que vimos antes y combinarlas con los operadores lógicos, en general no hace falta usar paréntesis porque la precedencia 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
```

6. Robots que deciden (valores de verdad y condicionales)

```
>>> 12 * 2 == 24 or False
True
>>> int("20") == 20
True
>>> "20" == 20
False
>>> def espar(numero):
...     return numero % 2 == 0
...
>>> espar(4)
True
>>> espar(7)
False
>>> espar(7) and 7 < 20
False
>>> espar(4) and 7 < 20
True
```

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, por eso en la segunda expresión del ejemplo se compara el resultado de evaluar `12 * 2` con el número 24 en lugar de comparar el 2 con el 24 y luego multiplicar el resultado por 12. Una forma equivalente de escribir esta expresión es:

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

Los paréntesis son innecesarios en este caso, pero si ayudan a la legibilidad del código es una buena práctica usarlos.

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

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

La expresión anterior compara `2 == 24` lo cuál obviamente es falso, al resultado de esto que es `False`, lo multiplica por 12 (no tienen importancia los detalles y probablemente no tenga mucho sentido multiplicar un booleano por un número pero a fines prácticos digamos que `12 * False` es igual a `False`), finalmente la expresión se reduce a `False or False` cuyo resultado es `False`.

6.5. Condicionando nuestros movimientos

En determinadas situaciones es necesario decidir qué acción realizar en base a un dato externo, como puede ser el teclado o los sensores propios del robot, el resultado de una operación, etc. En dichos casos contamos con la estructura de control `if`, la cual ejecuta un conjunto de instrucciones en caso que una expresión condicional sea verdadera u (opcionalmente) otro conjunto de instrucciones si la expresión es falsa. El conjunto de instrucciones que se ejecuta si la condición es falsa se escribe anidada en un `else`, veamos un ejemplo:

```
def menoresIgualesQueCien(cantidad):
    if(cantidad > 100):
        print(str(cantidad)+" no es un número válido")
    else:
        print str(cantidad)+" es un valor válido"

menoresIgualesQueCien(10)
menoresIgualesQueCien(100)
menoresIgualesQueCien(101)
```

En el contexto del robot podemos usar esto, por ejemplo, para validar los valores de velocidad que se le pasan al robot o como veremos más adelante para hacer que el robot reaccione a su entorno usando sus sensores.

Veamos como límitar el rango de velocidades del robot al que nosotros queramos, usando `if` y declarando una función:

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

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

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:

```
def avanzarLimitado(robot, velocidad, tiempo):
    if(tiempo > 10):
        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)
```

También es posible tener más de 2 caminos a seguir, podemos hacer algo si una condición se cumple, otra cosa si es otra la condición que se cumple y así siguiendo, para eso existe la palabra reservada `elif` que permite introducir otras condiciones y acciones, veamos un ejemplo sencillo:

```
def imprimeNumero(numero):
    if numero == 0:
        print "Cero"
    elif numero == 1:
        print "Uno"
    elif numero == 2:
        print "Dos"
```

La última condición, puede tener un `else` que nos permite considerar todas las posibilidades:

```
def imprimeNumero(numero):
    if numero == 0:
        print "Cero"
    elif numero == 1:
        print "Uno"
    elif numero == 2:
        print "Dos"
    else:
        print "Tengo programado hasta el dos"

imprimeNumero(0)
imprimeNumero(1)
imprimeNumero(2)
imprimeNumero(3)
```

Aplicando esto al robot podemos escribir una función que reciba un string con una orden y mueva el robot acorde a esta orden:

```
def mover(robot, accion):
```

6. Robots que deciden (valores de verdad y condicionales)

```
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")
```

6.6. Resumen Python

En este capítulo vimos como escribir condiciones simples y complejas. También vimos cómo hacer que un programa escrito en Python ejecute un conjunto de acciones si se cumple una condición u otro conjunto distinto si la condición no se cumple.

Por otro lado en las actividades se menciona el uso de la función `exit()` para hacer que un programa termine de inmediato, si bien no es común el uso de esta función, puede resultar necesaria para resolver algunos problemas.

6.7. Actividades

1. Implemente una función que reciba un número, 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.
2. Desarrolle una función que reciba un número, si el número es múltiplo de 9, el robot debe realizar un baile de la victoria.
3. Desarrolle una función que reciba dos números y retorne `True` si el valor de la izquierda es menor que el de la derecha.
4. Modifique el ejemplo que permite mover al robot ingresando la acción a realizar de forma interactiva, para que además de la acción reciba la velocidad a la que debe moverse, si la velocidad no es un valor entre 0 y 100, el programa debe imprimir un mensaje de error y luego terminar. Nota: para forzar la terminación del programa se puede invocar a la función `exit()`.

6.8. Lecturas Recomendadas

Capítulo 7

Simplificando

7.1. Objetivos

Cuando realizamos acciones repetitivas solemos tener que escribir la misma sentencia varias veces, en estos casos se puede automatizar la repetición recurriendo a las sentencias de repetición `for` y `while`, si bien ambas ejecutan el código que se encuentra dentro del bloque hay algunas diferencias de uso, las cuales detallaremos en este capítulo.

7.2. Sentencia de iteración `for`

Las secuencias de instrucciones dentro del bucle iterativo se repiten mientras se cumpla una condición, la cantidad de veces que se va a repetir, en el caso del uso del `for`, va a estar dado por la iteración sobre los ítems definidos. La forma general de utilización de la estructura de control `for` está dada por la forma:

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

La variable es la que va a tomar el valor de cada elemento de la lista hasta que finalice la ejecución del `for`. Veamos algunos ejemplos prácticos de utilización:

```
for x in [1,2,3]:  
    print x  
  
for x in ['a','b','c']:  
    print x  
  
for x in ['Entre Ríos', 'Corrientes', 'Misiones']:  
    print x
```

De esta forma definimos la estructura `for` e indicamos cuántas veces y qué valores va a tomar la variable utilizada para iterar. Se puede iterar sobre una lista de números o strings, o directamente sobre las letras que componen un string:

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

7. Simplificando

```
i  
o  
s
```

En caso que queramos generar listas de gran cantidad de números o por simplicidad no tener que escribir cada valor manualmente puede utilizarse la función primitiva `range()`, la cual genera una secuencia de números en el momento que definimos la estructura `for`:

***** ¿En este ejemplo no convendría poner los tres mayores y ... para que se vea que es ***** en modo interactivo? y después cuando imprime queda un número por línea que obviamente ocupa mucho

```
for x in range(10):  
    print x  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
for x in range(0, 10, 2):  
    print x  
[0, 2, 4, 6, 8]  
for x in range(-10, -100, -30):  
    print x  
[-10, -40, -70]
```

En los ejemplos vistos anteriormente se puede observar las diferentes posibilidades que hay para generar las listas con la función `range()`. En el segundo ejemplo se genera una lista que inicia en 0 , aumenta en 2 cada iteración en un rango de 10 valores, se dpuede advertir que el número 10 no lo toma, ya que cuando definimos 10 valores, al iniciar en 0, el último valor posible es 9, veamos en ejemplo que avance de a 3 valores:

```
for x in range(0, 10, 3):  
    print x  
[0,3,6,9]
```

Podemos utilizar las listas vistas en el capítulo 5 para iterar dentro de un for, en lugar de los valores enumerados :

```
a = ['Entre Ríos', 'Corrientes', 'Misiones']  
for x in a:  
    print x
```

Para el caso de las funciones vistas anteriormente podemos re-implementar la función `cuadrado()` utilizando la sentencia `for` para evitar la repetición de sentencias:

```
def cuadrado(r, tiempo):  
    for i in range(0, 4):  
        r.forward(50, tiempo)  
        r.turnRight(35, 1)  
        r.wait(1)
```

De esta manera, las instrucciones incluídas dentro del `for` se van a repetir 4 veces, durante el cual la variable `i` tomará los valores 0, 1, 2 y 3. Así mismo se pueden anidar los bloques `for` para controlar además la cantidad de figuras que queramos dibujar:

```
def hagofiguras(r):  
    tiempo = 2  
    lado = 30  
    esperar = 2  
    cantFiguras = input("Ingrese la cantidad de figuras: ")  
    for i in range(cantFiguras):  
        cantLados = input("Ingrese el numero de lados de la figura "+str(i+1)+" : ")  
        vuelta = input("Segun la figura entre la vuelta de la figura "+str(i+1)+" : ")  
        for j in range(cantLados):
```

```
r.forward(lado, tiempo)
r.turnRight(vuelta, 2)
r.wait(esperar)
```

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 robot import *
b = Board(device='/dev/ttyUSB0')
r = Robot(b, 1)
import movimientos
movimientos.hagofiguras(r)
```

En caso que necesitemos realizar algún cambio al módulo movimientos sin necesidad de reinciar el intérprete de python, podemos utilizar la sentencia `reload`, la cual recarga el módulo con los cambios realizados por nosotros:

```
movimientos = reload(movimientos)
movimientos.hagofiguras(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.

7.3. Sentencia de iteración while

Al igual que la sentencia `for` la `while` repite la ejecución del código dentro del bucle, con la diferencia que utiliza expresiones de tipo boolean para controlar la iteración. En cada repetición verifica que la expresión sea verdadera, luego ejecuta las sentencias que incluye dentro del bloque. La forma general de la iteración `while`:

```
while <expresión>:
    <instrucción/es>
```

En cada repetición se evalúa la expresión o expresiones utilizadas, en caso que sea correcto su resultado, se ejecutan las instrucciones que se encuentran incluidas en el bloque. Veamos un ejemplo en el cual el robot avanza mientras la distancia hasta un obstáculo sea mayor que 15, como se verá más adelante el mensaje `ping` devuelve la distancia hay existe entre el robot y un obstáculo.

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

Una de las aplicaciones para su uso es la interacción con el ingreso de datos por parte del usuario, como puede ser ingresar datos hasta que cierta condición se cumpla:

```
n = raw_input('Ingrese valores mayores que 10: ')
while (int(n) < 10):
    print n
    n = raw_input('Ingrese valores mayores que 10: ')
```

De esta forma manejamos contralamos la cantidad de veces que se repite la iteración, en caso que necesitemos agregar otra condición que actúe como límite superior para verificar el valor ingresado, se pueden suministrar varias condiciones:

```
n = input('Ingrese valores mayores que 20 y menor que 100: ')
Ingrese valores mayores que 20 y menor que 100: 45
while (n > 20) and (n < 100):
    print n
    n = input('Ingrese valores mayores que 20 y menor que 100: ')
```

7. Simplificando

Las expresiones necesarias para decidir la ejecución de un bloque pueden ser mas de una, la para cual necesitamos utilizar Los operadores para combinar más de una expresión son los denominados lógicos, nombrados en la sección 5.2.2, la evaluación de las diferentes expresiones se realizan de izquierda a derecha, por lo tanto si la primera expresión resulta `false` y se esta combinando dos expresiones con el operando `and` no se evaluará la que se encuentra a la derecha de la misma, veamos un ejemplo de esta situación:

```
n = input('Ingrese valor: ')
while (n > 5) and (n < 10):
    sumo = sumo + n
    n = input('Ingrese valor: ')
```

En este ejemplo si se ingresa un valor menor que 5, ya no se evaluará la segunda expresión que verifica si el valor es mayor que 10. Los operadores lógicos soportados por el lenguaje son utilizados para controlar la cantidad de veces que se repiten las instrucciones incluidas en el bloque, vemos en la tabla 6.7 su forma de uso. Para nuestro caso que estamos manejando el robot podríamos generar un módulo que pregunte al usuario cuánto quiere hacer avanzar el robot, el dato se solicita por teclado hasta que se ingrese 0 para detenerlo.

```
def avanzoMientras(r):
    print("Ingrese velocidad y tiempo - 0 para terminar")
    cantidad = input("ingrese velocidad a la que desea avanzar: ")
    while (cantidad != 0):
        tiempo = input("ingrese tiempo durante el que desea avanzar: ")
        r.forward(cantidad, tiempo)
        cantidad = input("ingrese velocidad a la que desea avanzar: ")
```

Así como vimos en la sección 7.3 los operadores lógicos se pueden utilizar para combinar expresiones que se evaluarán para la decisión de ejecutar el bloque. En caso que las decisiones se deben hacer dependen de varias cosas, podemos anidar las estructuras `if` para tomar decisiones en base a una expresión:

```
def decido_movimiento(r):
    avanco = 40
    vuelta = 30
    doblo = 1
    t = 2
    print('''Ingrese la accion que desea realizar
            f: hacia adelante
            b: hacia atras
            r: doblar a la derecha
            l: doblar a la izquierda
            s: salir
            ''')
    accion = raw_input("Accion: ")
    while (accion != 's'):
        if(accion == 'f'):
            r.forward(avanco, t)
        elif accion == 'b':
            r.backward(avanco, t)
        elif accion == 'r':
            r.turnLeft(vuelta, doblo)
        elif accion == 'l':
            r.turnRight(vuelta, doblo)
        else:
            print "No es una accion valida"
    accion = raw_input("Accion: ")
```

*****NOTA***** A diferencia de otros lenguajes de programación python no cuenta con una estructura de control similar al `case` ya que es un lenguaje orientado a objetos y se sugiere utilizar polimorfismo, con lo cual se identifica la acción deseada según la clase a la que pertenece el objeto.
*****nota*****

7.4. Actividades

7.5. Repaso del capítulo 7

7. Simplificando

Capítulo 8

Organizándonos

8.1. 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. Para nos que más claro la estructura cada componente de un programa veamos cómo se compone cada uno:

Programas están compuestos de módulos.
Módulos contienen sentencias.
Sentencias crean y procesan objetos.

8.2. Funciones

Como ya hemos visto a los largo del manual el uso de funciones simplifica la escritura de un algoritmo o programa ya permite realizar una acción determinada sin repetir código, por lo tanto es necesario entender el funcionamiento de las funciones y cómo definirlas. Hay dos roles importante que cumplen las funciones por los cuales es necesario su utilización:

- **Reuso de código:** una característica fundamental de la utilización de funciones es evitar la repetición de sentencias o bloques de código para su uso en diferente puntos de nuestro programa.
- **Descomponer programación procedural:** las funciones proveen también un mecanismo para separar en partes mas équeñas un programa. La idea de separar las acciones en partes mas pequeñas se conoce como **programación modular**. A través de la programación modular se simplifica y separa en partes mas pequeñas las acciones a realizar, lo cual permite acelerar la codificación. De esta manera cada porción de bloque encerrado en funciones o módulos resuelve una tarea específica y no interfiere con el funcionamiento de las otras funciones o el resto del programa.

8.2.1. Definición de una función

En forma general definimos la función con la palabra clave `def` y con la identación indicamos las instrucciones que se encuentran dentro de la misma.

```
def <nombre>(arg1, arg2, ... argN):  
    <instrucciones>  
    return <valor>
```

Hay varias opciones de definir las funciones según sea necesario el envío de parámetros o no, en la tabla [8.1](#) se puede ver cómo se pueden definir y cómo se hace el llamado en cada caso:

Las funciones cuentan con distintos componentes y palabras claves que se deben tener en cuenta y respetar. Veamos a continuación algunos conceptos sobre estos:

Definición	Llamada
def funcion()	func()
def funcion(parametro)	func(valor)
def func(nombre=valor)	func(nombre=valor)

Cuadro 8.1. Definición y llamadas de una función

- **def:** La palabra clave `def` identifica la declaración de una función, a continuación se debe especificar el nombre con el que se identifica. A diferencia de otros lenguajes de programación como C, `def` es una sentencia ejecutable, esto quiere decir que la función no es conocida hasta que Python llega y ejecuta la orden `def`.
- **return:** Otra capacidad de las funciones es retornar un valor resultante de las operaciones realizadas en la función.
- **Argumentos:** La funciones pueden recibir los parámetros como argumento o no. En caso que definamos la función con argumentos, podemos asignarle un valor por defecto. De esta manera no es necesario enviar la variable si la queremos usar con el valor asignado en la definición.

En caso que necesitemos retornar un valor desde la función, utilizaremos como vimos anteriormente, la sentencia `return`, la cual devuelve al que lo llama el valor de la variable:

```
def sacomax(num1, num2):
    if num1 >= num2:
        return num1
    return num2
```

Y luego para obtener el valor devuelto por la función la invoco de la siguiente manera:

```
max = sacomax(n1, n2)
print max
```

8.2.2. Pasando argumentos

8.3. Cómo correr un programa

Para ejecutar un programa simple desde la terminal sin utilizar el intérprete se puede hacer de diferentes maneras. Una forma de ejecutar es escribir el código en un archivo, como hemos visto anteriormente y guardarlo con extensión “.py”. Una vez finalizado la codificación de nuestro algoritmo para que se ejecute el programa lo haremos con el comando `python` delante. Veremos a continuación un ejemplo para utilizarlo desde la terminal, en este código se crea el objeto `Board` y `Robot` dentro del código como así también se importa la clase `Robot`, ya que no podemos ejecutar las sentencias en el intérprete y luego ejecutar el programa porque no reconoce los objetos creados fuera del programa.

```
def hagofiguras():
    tiempo = 2
    lado = 40
    esperar = 2
    nro = raw_input("Ingrese el nro de robot: ")
    b=Board('/dev/ttyUSB0')
    r = Robot(b, int(nro))
    cantFiguras = int(raw_input("Ingrese la cantidad de figuras: "))
    for i in range(1, cantFiguras+1):
        cantLados = int(raw_input("Ingrese el número de lados de la figura "+str(i)+" : " ))
        vuelta = int(raw_input("Ingrese velocidad para doblar "+str(i)+" : " ))
        for j in range(cantLados):
            r.forward(lado, tiempo)
```

```
r.turnRight(vuelta, 2)
r.wait(esperar)

from robot import *
hagofiguras()

$ python hagofiguras.py
```

Si se está trabajando en sistemas basado en Linux, se puede ejecutar el programa como archivo ejecutable conocidos como scripts. Este tipo de archivos si bien parecen comunes, cumplen con dos propiedades específicas:

- La primera línea es importante: los scripts comienzan con los caracteres especiales `#!` seguido del path donde se encuentra el intérprete de Python en la máquina.
- Los scripts necesitan ser marcados como ejecutables, para cual es necesario utilizar el comando `chmod +x archivo.py`.

Veamos un ejemplo de utilización de esta forma, supongamos que escribimos nuestro código, esta vez con la línea del path de ubicación de Python.

*****NOTA***** En los sistemas Linux para saber la ubicación de un programa podemos utilizar el comando whereis:

```
$ whereis python
python: /usr/bin/python2.5 /usr/bin/python2.6 /usr/bin/python3.1
/usr/bin/python2.6-config /usr/bin/python

*****
#!/usr/bin/python
print 'Usando robots en la escuela...'
```

Una vez definido el programa y cambiado los permisos del archivo lo ejecutamos directamente sin necesidad de anteponer el comando `python`

```
$ hagofiguras
```

8.4. Actividades

8.5. Repaso del capítulo 8

8. Organizándonos

Capítulo 9

Percibiendo el mundo: Sensores

En capítulos anteriores hicimos que el robot haga distintos movimientos y aprendimos a agrupar el código de distintas formas para hacer nuestro trabajo más eficiente. Pero el robot no percibía el entorno, hacía literalmente lo que indicabamos sin importar si existía algún obstáculo que lo impidiera por ejemplo.

En el capítulo 3 les contamos que el robot tiene sensores, ¡ya es hora de usarlos para que el robot pueda percibir su entorno y actuar en consecuencia!.

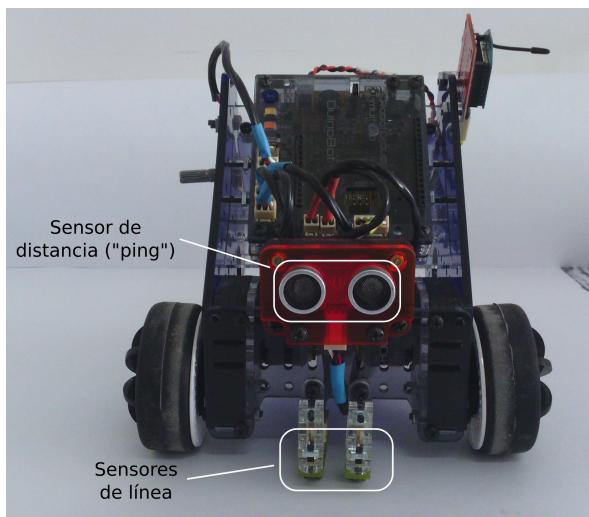


Figura 9.1. Sensores del robot

En la figura 9.1 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 (ya que los robots tienen que ser por definición autónomos) usando estos sensores.

9.1. Conociendo los sensores

– ini NO ESTÁ IMPLEMENTADO EN LA API OFICIAL

Para saber qué valores podemos esperar de los sensores podemos usar la función `senses()` como se ve en las figuras 9.2 y 9.3, esta función muestra una ventana con los valores de los sensores, probá poner la mano frente al robot para ver como cambia el valor de `ping`, apoyá el robot en distintas superficies para ver como varían los valores de los sensores de línea.

```
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 robot import *
>>> b = Board("/dev/ttyUSB0")
>>> b.report()
Robot 1 prendido
>>> robot = Robot(b, 1)
>>> robot.senses()
```

Figura 9.2. Invocación a `senses()`

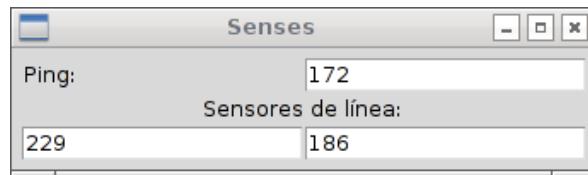


Figura 9.3. Ventana de información de sensores

– fin NO ESTÁ IMPLEMENTADO EN LA API OFICIAL

9.2. Evitando choques

9.2.1. Primer aproximación

Sabiendo que podemos detectar obstáculos con `ping()` podemos hacer un programa en el que el robot pueda avanzar sin chocar.

Para poder realizarlo tenemos que poder evaluar el valor devuelto por `ping()` mientras el robot avanza, como vimos en el capítulo 4 usar la función `forward()` sin especificar el tiempo puede resultar útil para esto. Además hay que evaluar el valor devuelto por `ping()` de forma periódica, para esto podemos usar la estructura de control de repetición `while`. Finalmente cuando se detecte un obstáculo podremos detener el robot con la función `stop()`.

En la figura 9.4 se muestra en pseudocódigo la forma planteada para resolver este problema.

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

Figura 9.4. Pseudocódigo: Avanzar hasta detectar obstáculo

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, estructuras de control o clases vacías. La instrucción de Python que no hace nada se llama `pass`.

También en la figura 9.4 aparece `hayObstáculo`, una sentencia que devuelve `True` cuando hay un obstáculo y `False` si no lo hay, pero lo más parecido que tiene el robot a `hayObstáculo` es la función `ping()` que devuelve la distancia en centímetros al objeto más cercano, para no chocar podríamos considerar obstáculos a los objetos que estén a 15 centímetros o menos del robot, de esta forma `hayObstáculo` se podría escribir en Python como `robot.ping() <= 15`.

Pasando en limpio todo lo anterior en la figura 9.5 podemos ver la implementación en Python del pseudocódigo anterior.

```
robot.forward()
while not robot.ping() <= 15:
    pass
robot.stop()
```

Figura 9.5. Programa: Avanzar hasta detectar obstáculo

Pensando un poco se le puede poner una condición equivalente pero más elegante al `while`. En castellano se podría leer la condición de la siguiente forma: “`robot.ping()` no es menor igual que 15”, aplicando un poco de lógica, que `robot.ping()` no sea menor igual que 15 quiere decir que “`robot.ping()` sea mayor que 15”, por lo que podemos escribir nuevamente la condición como se ve en la figura 9.6, si bien este programa hace lo mismo que el anterior es un poco más fácil de ser leído y entendido por un humano.

```
robot.forward()
while robot.ping() > 15:
    pass
robot.stop()
```

Figura 9.6. Programa: Avanzar hasta detectar obstáculo 2

9.2.2. Como cambiar el rango de los valores de los sensores

Como viste con `senses()` los sensores devuelven distintos rangos de valores, en las próximas actividades vas a tener que usar los valores devueltos por los distintos sensores como entrada para distintas funciones de los robots. Para esto vas a tener que adaptar los valores de los sensores al rango de las funciones.

Por ejemplo el sensor de distancia devuelve valores entre 0 y 601, si quisieramos pasar ese valor por ejemplo a `forward()` deberíamos normalizarlo para que esté en el rango de entre 0 y 100, esto se puede hacer con una formula matemática simple. Básicamente se trata de (siempre asumiendo que todos los rangos tanto origen como destino empiezan en 0) dividir el valor original por el máximo del rango de origen, esto devuelve un valor con decimales entre 0 y 1 y luego multiplicar este valor por el máximo del valor destino, la fórmula sería $n(x) = (x/maxOrigen) \cdot maxDestino$.

Si no me creen que esto funciona, veamos unos ejemplos, para $maxOrigen = 601$ y $maxDestino = 100$:

- $n(0) = 0$
- $n(20) = 3,3277870216306153$
- $n(200) = 33,277870216306155$
- $n(560) = 93,178036605657226$
- $n(601) = 100$

9.2.3. Actividades

1. Programe la función `normalizar1()` que reciba 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 $normalizar1(20, 601, 100) \approx 3,33$.

2. La función del ejercicio 1 no tiene en cuenta que pasa cuando el valor está fuera de rango, programe la función `normalizar()` donde 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:

- $\text{normalizar}(-20, 601, 100) = 0$
- $\text{normalizar}(-1, 601, 100) = 0$
- $\text{normalizar}(602, 601, 100) = 100$
- $\text{normalizar}(1000, 601, 100) = 100$
- $\text{normalizar}(600, 601, 100) \approx 99,83$

```

while robot.ping() < 100:
    if robot.ping() < 10:
        robot.backward(50, 0.5)
        robot.turnLeft(50, 1)
        robot.forward(50)

robot.stop()

```

9.3. Velocidad proporcional a la distancia

Podemos modificar un poco el programa anterior 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.

Podemos solucionarlo de la siguiente forma, mientras los objetos estén a más de 55 centímetros simplemente avanzamos, cuando estén a menos de 55 y más de 15 centímetros vamos bajando la velocidad, finalmente a los 15 centímetros 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()`.

El programa que hace esto se puede ver en la figura 9.7. Notá que se hace referencia al módulo `mis_funciones` que contiene la función `normalizar()` tal cuál se definió en el ejercicio 1, vos usá el nombre de módulo que hayas elegido.

Para probar si funciona correctamente puede servir levantar el robot y acercarlo y alejarlo lentamente de un obstáculo para ver como cambia la velocidad de las ruedas.

```
from robot 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()
```

Figura 9.7. Programa: Velocidad proporcional a la distancia

9.3.1. Actividades

1. Escriba un programa que haga que el robot se escape cuando se le acerque algo.
2. Escriba un programa que recorra 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.
3. Si el piso es de un color claro escriba un programa que haga que el robot avance hasta encontrar una superficie oscura, el piso es oscuro haga lo contrario.

9. Percibiendo el mundo: Sensores

Capítulo 10

Invitando a la matemática

10.1. Trabajo con funciones de matemáticas

10.2. trabajar con diccionarios

para emular un switch explicar usando un diccionario, segun lo que se ingrese la accion deseada:
F:forward B:backward L:left R:right para hacer un recorrido guiado

10.3. Actividades

Capítulo 11

Apéndice 1: Guía de instalación de paquetes

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

Para ejecutar Synaptic vamos a Sistemas → Administración → Gestor de Paquetes Synaptic como vemos en la figura 11.1

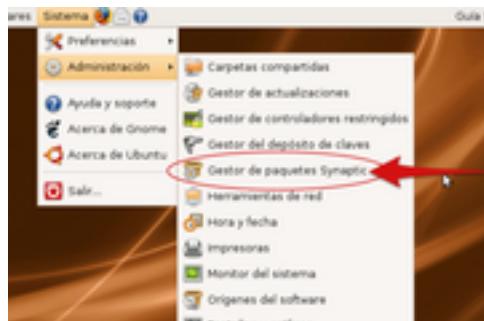


Figura 11.1. Arrancando el gestor

Podemos notar en la figura 11.2 que se encuentra dividida en 4 partes. En el margen izquierdo tenemos distribuido por categoría (Nº 1 Lista de categorías), abajo se encuentran los filtros (Nº 2), a nuestra derecha arriba se encuentran todos los paquetes que pertenecen a esa categoría (Nº 3 paquetes para instalar) y abajo una breve descripción del paquete seleccionado(Nº 4 Descripción de la aplicación).

11. Apéndice 1: Guía de instalación de paquetes

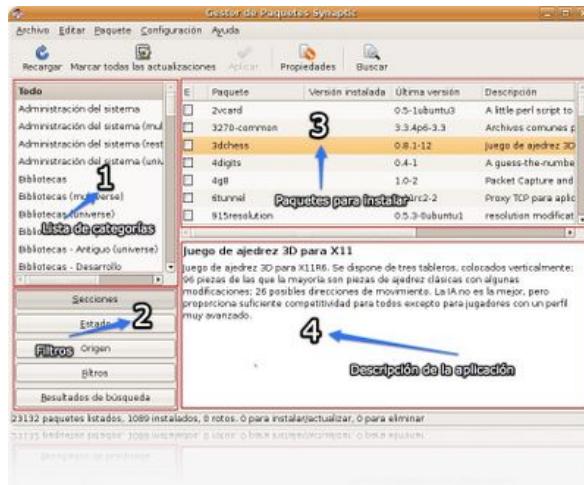


Figura 11.2. Secciones de la Synaptic

Para instalar un paquete puedes seleccionar una categoría y luego hacer doble clic sobre el o clic derecho, “Marcar para instalar”. De esta forma, puedes marcar todos los paquetes que deseas 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.

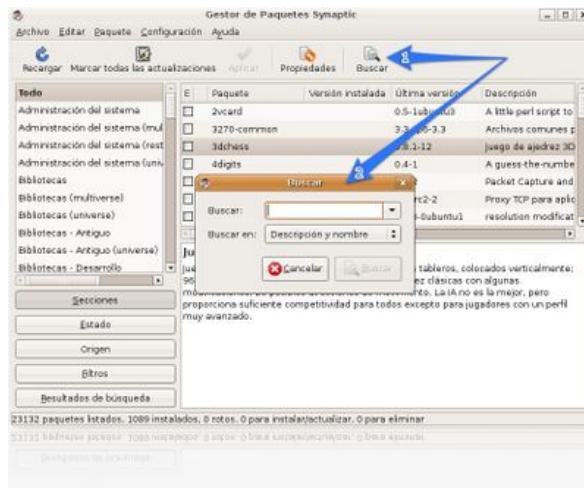


Figura 11.3. Buscar un paquete