CE

CONRAD

Table of Contents

1	**From installing the operating system to the first Python programme**		**5**
	1.1	What do I need?	5
	1.1.1	Micro USB mobile charger	6
	1.1.2	Memory card	6
	1.1.3	Keyboard	6
	1.1.4	Mouse	6
	1.1.5	Network cable	6
	1.1.6	HDMI cable	6
	1.1.7	Audio cable	6
	1.1.8	Yellow CVBS video cable	7
	1.2	Installing the Raspbian operating system	7
	1.2.1	Preparing the computer's memory card	7
	1.2.2	The software installer NOOBS	7
	1.2.3	The LEDs on the Raspberry Pi	8
	1.2.4	Starting the Raspberry Pi for the first time	9
	1.3	Almost like Windows – graphical user interface LXDE	9
	1.3.1	Saving your data on the Raspberry Pi	11
	1.4	The first programme using Python	12
	1.4.1	Guessing numbers with Python	14
	1.4.2	How does it work?	16
2	**The first LED on Raspberry Pi is illuminated**		**18**
	2.1	Components contained in the package	19
	2.1.1	Breadboards	20
	2.1.2	Connecting cable	20
	2.1.3	Resistors and color codes	21
	2.2	Connecting the LED	22
	2.3	GPIO and Python	26
	2.4	Switching the LED on or off	27
	2.4.1	How does it work?	28
	2.5	Starting Python with GPIO support without terminal	29
3	**Traffic light**		**32**
	3.1.1	How does it work?	34
4	**Pedestrian lights**		**36**
	4.1.1	How does it work?	38
	4.2	Button on the GPIO connector	39
	4.2.1	How does it work?	43
5	**Colorful LED patterns and chaser lights**		**45**
	5.1.1	How does it work?	48

6 Dimming the LED using pulse width modulation **53**
6.1.1 How does it work? 56
6.1.2 Dimming two LEDs independently of one another 57
6.1.3 How does it work? 59

7 Indicator with LEDs for free space on the memory card **60**
7.1.1 How does it work? 63

8 Graphical dice **65**
8.1.1 How does it work? 67

9 Analogue clock on-screen **72**
9.1.1 How does it work? 73

10 Graphical dialog fields for program control **77**
10.1.1 How does it work? 79
10.2 Controlling chaser by a graphical user interface 81
10.2.1 How does it work? 84
10.3 Setting the flashing rate 87
10.3.1 How does it work? 88

11 PiDance and LEDs **89**
11.1.1 How does it work? 93

1 From installing the operating system to the first Python programme

The Raspberry Pi has hit the headlines in recent months like no other electronic device in this price range. Although it does not look like that at first glance, the Raspberry Pi is a full-fledged computer about the size of a credit card - at a very reasonable price. Not only the hardware is cheap, but also the software: The operating system and all applications required for everyday use can be downloaded free of charge.

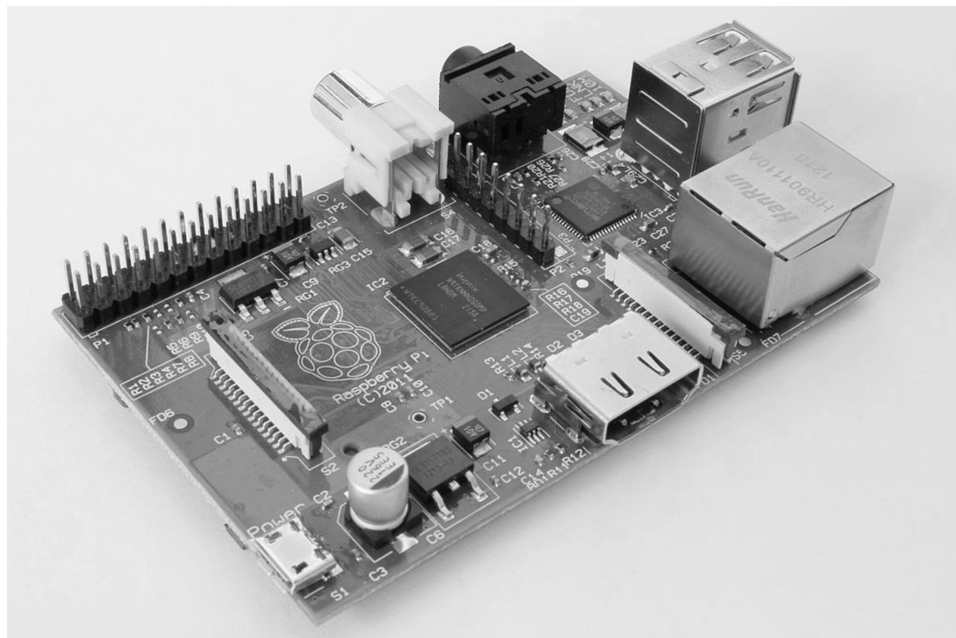


Fig. 1.1: The Raspberry Pi – a computer in mini format

The Raspberry Pi with its specially adapted Linux system with graphical user interface is a power-efficient and silent computer replacement. Its freely programmable GPIO interface makes the Raspberry Pi particularly interesting for hardware tinkers and the maker scene.

1.1 What do I need?

The Raspberry Pi is a full-fledged computer despite its tiny size. Like with any "normal" computer you still need some accessories in order to use it – an operating system, power supply, network, monitor, keyboard and various connecting cables.

1.1.1 Micro USB mobile charger

Any modern cell phone power supply is sufficient for the Raspberry Pi. However, older chargers from the early days of the USB charging technology are too weak. If you intend to connect a power-hungry USB devices such as external hard drives without a separate power supply you will need a stronger power supply. The power supply must provide 5 V and at least 700 mA, while 1000 mA is even better. The built-in power regulator prevents "burn out" in cases when the power supplies are too strong.

> **Signs of a weak power supply**
> The Raspberry Pi boots, but then the mouse pointer does not move or the system does not respond to keyboard commands these are the indications that the power supply is too weak. If you cannot access the connected USB sticks or hard drives you should also use a stronger power supply.

1.1.2 Memory card

The memory card in the Raspberry Pi acts as the hard drive. It contains the operating system. Your data and installed programs are also stored here. The memory card should have at least 4 GB in size and according to the manufacturer must support at least Class 4 standard. This standard specifies the speed of the memory card. The performance of a recent class 10 memory card is clearly noticeable.

1.1.3 Keyboard

Any common keyboard with USB port can be used. Sometimes wireless keyboards do not work because they require too much power or special drivers. If you do not have any other keyboard available, you need a USB hub with a separate power supply to operate a wireless keyboard.

1.1.4 Mouse

A mouse with USB connection is only required if on the Raspberry Pi an operating system with graphical user interface is used. Some keyboards feature additional USB ports for mice, so that you do not use any other ports. You can use the port then for a USB stick, for instance.

1.1.5 Network cable

A network cable is required to connect to the router on a local network. This is definitely required for the initial setup; later on you can also use Wi-Fi. Without Internet access many functions of the Raspberry Pi are not very useful.

1.1.6 HDMI cable

The Raspberry Pi can be connected via HDMI cable to a monitor or a TV. For the connection to computer monitors with DVI connector special HDMI cables or adapters are available.

1.1.7 Audio cable

The audio cable with 3.5 mm jacks can be used for headphones or PC speakers on the Raspberry Pi. The audio signal is also available via the HDMI cable. HDMI televisions or monitors do not require audio cables. If a computer monitor is connected via an HDMI cable to a DVI adapter, then typically at this point, the audio signal is lost, so that you will need again the analogue audio output.

1.1.8 Yellow CVBS video cable

If no HDMI monitor is available, you can connect the Raspberry Pi to an analogue composite video cable, with the typical yellow plugs and also to a typical a television; however, the image-screen resolution, will be very low. For TV sets without yellow composite input, converters from CVBS to SCART are available The graphical user interface works in analogue TV resolution only with restrictions.

1.2 Installing the Raspbian operating system

The Raspberry Pi comes without an operating system. Unlike personal computers,of which most of them use Windows, a specially adapted Linux system is recommended for the Raspberry Pi. With the economical hardware, Windows would not run at all.

Raspbian is the name of the Linux distribution that is recommended and supported by the manufacturer of the Raspberry Pi. Raspbian is based on Debian Linux, one of the most famous Linux distributions, which also provides the base for the popular Linux distributions Ubuntu and Knoppixi, among others. The hard drive for computers is here a memory card on the Raspberry Pi. It contains the operating system, and the Raspberry Pi also uses data from this memory card to boot.

1.2.1 Preparing the computer's memory card

The Raspberry Pi cannot boot yet by itself, so we will prepare the memory card using a PC. You will need a card reader on the computer. This may be an integrated one or it may be connected via USB.

It is best, if you use a brand new memory card, since those are already optimally pre-formatted by the manufacturer. But you can also use a memory card that has previously been used in a digital camera or for any other device. These memory cards should be reformatted before using them for the Raspberry Pi. In theory, you can use the formatting capabilities of Windows. The better option is the software "SDFormatter" of the SD Association. Thus, the memory cards are formatted for optimal performance. This tool can be downloaded for free from www.sdcard.org/downloads/formatter_4

If the memory card contains partitions from a previous operating system installation of a Raspberry Pi, the SDFormatter will not show the full size. In that case you will use the formatting option *FULL (Erase)* and also activate the option *Format Size Adjustment*. This will create a new partitioning to the memory card.

> **Memory card will be deleted**
> Your best choice is to use an empty memory card for the installation of the operating system. If there is any data on the memory card, these will be irreversible deleted in the process of reformatting during the installation of the operating system.

1.2.2 The software installer NOOBS

“New Out Of Box Software” (NOOBS) is a very simple installer for Raspberry-Pi-OS. Unlike previously it is not necessary any longer for the user to deal with image tools and boot blocks to set up a bootable memory card. NOOBS offers various operating systems of choice. When you start the Raspberry Pi for the first time, you can select your desired operating system directly from the Raspberry Pi which is then installed as a bootable version on the memory card. Download the NOOBS installation archive which has an approximate

size of 1.2 GB from the official download page *www.raspberrypi.org/downloads* and using the PC unpack it to the memory card of at least 4 GB.

Start the Raspberry Pi now from the memory card. Insert the memory card into the slot on the Raspberry Pi and connect keyboard, mouse, monitor and network cable. Finally it is time for the USB power port. It is used to switch the Raspberry Pi on. A separate power button does not exist.

After a few seconds a menu is displayed from which you can choose the desired operating system. We use the operating system Raspbian that is recommended by the Raspberry Pi Foundation.

Select your language (German) as the installation language at the very bottom, and select the pre-selected Raspbian operating system. After you have confirmed the safety message to overwrite the memory card the installation, which takes a few minutes, will start. Brief information about Raspbian are displayed during the process of installation.

1.2.3 The LEDs on the Raspberry Pi

Five LEDs with status indicators are located in one of the corners on the Raspberry Pi. The labels on older Raspberry Pi models may differ from newer models, but the functions are the same.

New board (Rev. 2)	Board (Rev. 1 older models)	Color	Meaning of the LED
ACT	OK	Green	Access to memory card
PWR	PWR	Red	Connected to power supply
FDX	FDX	Green	LAN in full duplex mode
LNK	LNK	Green	LAN access
100	10M	Yellow	LAN with 100 MBit/s

Tab. 1.1: The LEDs on the Raspberry Pi.

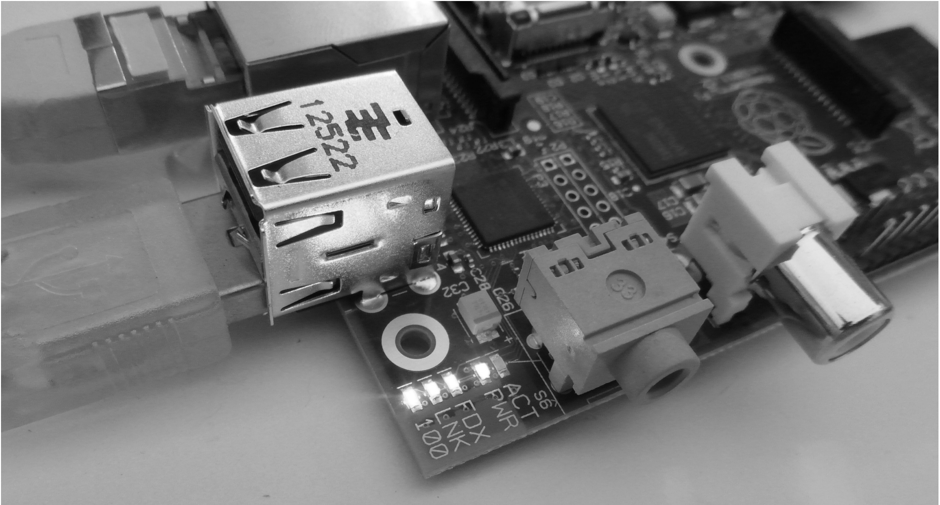


Fig. 1.2: The status LEDs on the Raspberry Pi.

1.2.4 Starting the Raspberry Pi for the first time

The Raspberry Pi will reboot and automatically launch the configuration `raspi-config` after the installation has completed. You only need to select the option *Desktop Log in as user 'pi'* under *Enable Boot to Desktop*. German and the German keyboard layout are automatically selected along with other important basic settings. The graphical LXDE desktop is available after rebooting.

1.3 Almost like Windows – graphical user interface LXDE

Many cringe when they hear the word "Linux"; they fear that they have to enter some cryptic command sequences into a command line, just like under DOS some 30 years ago. Far from it! Linux offers developers a free open operating system to develop their own graphical user interfaces. So the user is not bound to use a user interface that in its core is still a command-line based operation system.

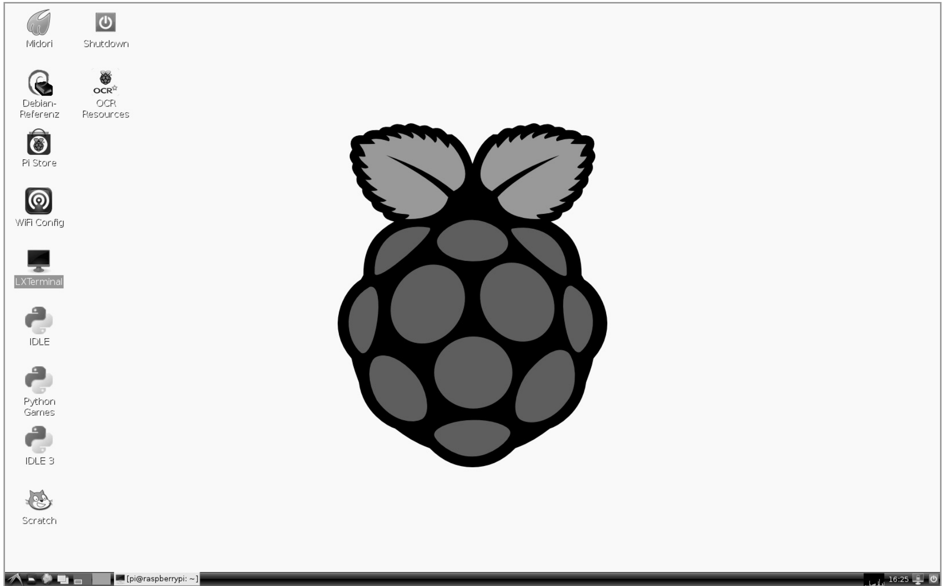


Fig. 1.3: The LXDE desktop environment on the Raspberry Pi is very similar to Windows XP.

Raspbian-Linux for the Raspberry Pi uses the interface LXDE (Lightweight X11 Desktop Environment), which on the one hand requires very little system resources and on the other hand its Start Menu and the File Manager are very much like the familiar Windows interface.

Linux Application
Even the typical Linux user registration happens in the background. However, if you should ever need it: The username is `pi` and the password `raspberry`.

The LXDE icon on the far left below opens the Start Menu, the icons next to it open the File Manager and the Web browser. Like in Windows, the Start Menu is a multi-level menu. You can add an icon for frequently used programs to your desktop by a right-click. Some of the preinstalled programs, the Midori web browser, Python development environments and the Pi Store are already there.

Switching the Raspberry Pi off
Actually, you may simply pull the plug of Raspberry Pi to turn it off. However, a clean system shut down like it is done on a computer is the better option. To do this, double-click the *Shutdown* icon.

1.3.1 Saving your data on the Raspberry Pi

File management on Linux runs a bit different than it does under Windows, but is not difficult at all. Raspbian features a file manager, which looks just like the Windows Explorer. There is an important difference to Windows: Linux does not strictly separate the drives, all files are located in a shared file system.

Basically on Linux, you store all your documents under the home directory. It is called here `/home/pi` according to the username `pi`. Linux uses the simple slash to separate the components of a path name(/) and not the backslash (\) as Windows does. You will store your Python programs in this directory, too. By default the file manager shows only this home directory. You can also access the file manager, like under Windows, using the key combination `Win`+`E`, by the way. Some programmes automatically create subdirectories there.

If you really want to see everything, even the files that should not concern the everyday user, you change the file manager on the top left from *Places* to *Directory tree*. Then choose in the menu *View* the option *Detailed view* and the display looks like what you would imagine from a Linux system.

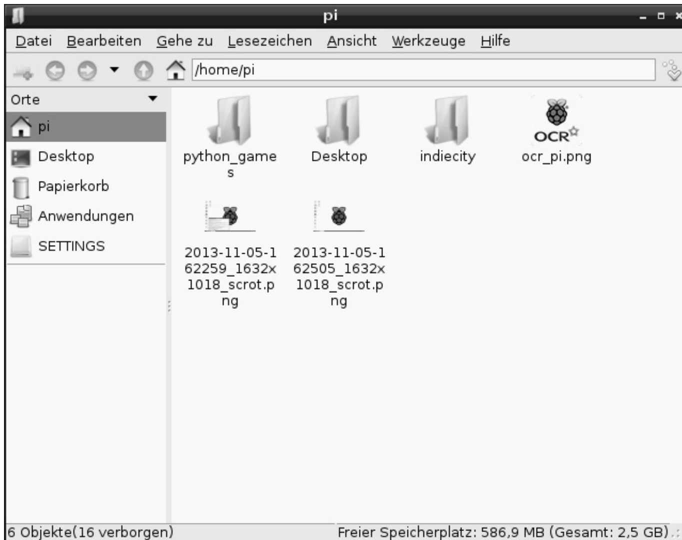


Fig. 1.4: The file manager on the Raspberry Pi can look like this ...

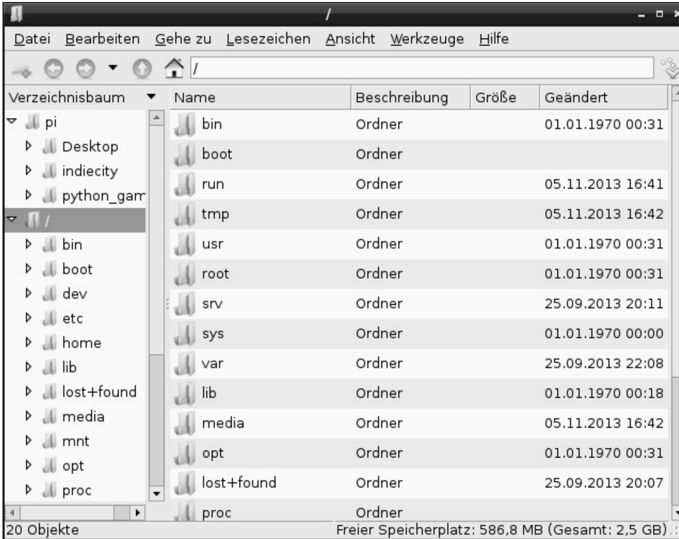


Fig. 1.5: ... or like that.

How much free space is on the memory card?
Not only hard drives of personal computers are quickly full - with the memory card of the Raspberry Pi this happens much faster. That is why it is important to always keep an eye on the free and the used space on the memory card. The status bar of the File Manager at the bottom on the right shows the memory card's free and used space.

1.4 The first program using Python

The programming language Python is preinstalled on the Raspberry Pi to help you getting started with programming. Python impresses with its clear structure that offers an easy entry point to programming, but it is also the ideal language to "quickly" automate the things you would otherwise do manually. Programming is really fun, because you don't have to observe variable declarations, types, classes or any complicated rules.

Python 2.7.3 or 3.3.0?
The Raspberry Pi comes with two preinstalled versions of Python. Unfortunately, the latest version of Python 3.x uses a partially different syntax than the proven version 2.x, so that the programs written in one version do not run with the other. Some important libraries, such as the famous PyGame for the programming of games and graphic display outputs are usually not available for Python 3.x. Because of this and also because most of the available programs on the Internet are written for Python 2.x, for the purpose of this book we are going to use the proven version Python 2.7.3. Our examples will work equally on a Raspberry Pi with an older Python 2.x version installed.

Python 2.7.3 is started with the icon *IDLE* on the desktop. A command window with a command prompt appears on the desktop, which looks simple at a first glance.

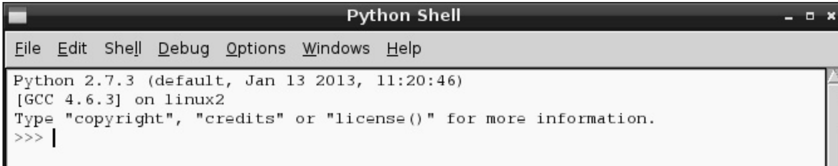


Fig. 1.6: The input window of the Python shell.

In this window you can open existing Python programs, write new ones or you can directly execute Python commands interactively, without having to write a real programme. For example, in the prompt type the following:

```
>>> 1+2
```

The correct answer is shown immediately:

```
3
```

Python can be used as a handy calculator, but that has nothing whatsoever to do with programming. Usually programming courses start with *Hello World* programs, which write the phrase "Hello World" on the screen. That is so simple in Python that it is not even worth to run it under its own heading. Simply type the following line in the Python shell window:

```
>>> print "Hello World"
```

This first "program" then writes `Hello World` in the next row on the screen.

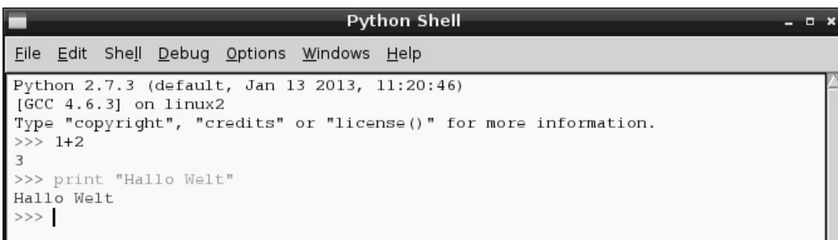


Fig. 1.7: »Hello World« in Python (further up you can still see the calculation output).

You can also see here that the Python shell automatically uses different text colors for reasons of clarity. Python commands are orange, strings are green and results are blue. Later on you will discover some more colors.

Python Flashcards
Python is the ideal programming language to introduce you to programming. It takes a bit to get used to the syntax and the layout rules. A brief description of the main syntax elements of the Python programming language in the form of a "cheat sheet" will help you with everyday programming. These are based on the Python Flashcards by David Whale. Find out what this is all about on *bit.ly/pythonflashcards*. These Flash Cards do not explain the technical background, but describe the syntax by giving very brief examples, telling you how exactly something is done.

1.4.1 Guessing numbers with Python

Rather than lingering on programming theory, algorithms and data types, let us write the very first small game in Python; a simple guessing game where the player tries to guess a random number, which is chosen by the computer, in as few attempts as possible.

1. Select in the Python shell's menu *File/New Window*. This opens up a new window in which you type the following programming code:

```
import random
number = random.randrange(1000); guess = 0; i = 0
while tip != number:
    tip = input("Your guess:")
    if number < guess:
        print "The number sought is less than ",guess

    if number > guess:
        print "The number sought is greater than ",guess

    i += 1
print "You guessed the number in ",i,". Guesses"
```

2. Save the file with *File/Save As* as `spiel1.py`. Or download the ready program file from *www.buch.cd* and open it in the Python shell using *File/Open*. The color coding shows up in the source code automatically and helps you to find typos.

3. Before you start the game, note a special feature of the German language, namely the umlauts.. Python runs on various computing platforms which encode umlauts differently. To display these correctly, select *Options/Configure IDLE* in the menu and in the tab *General* select the option *Locale-defined* in the field *Default Source Encoding*.

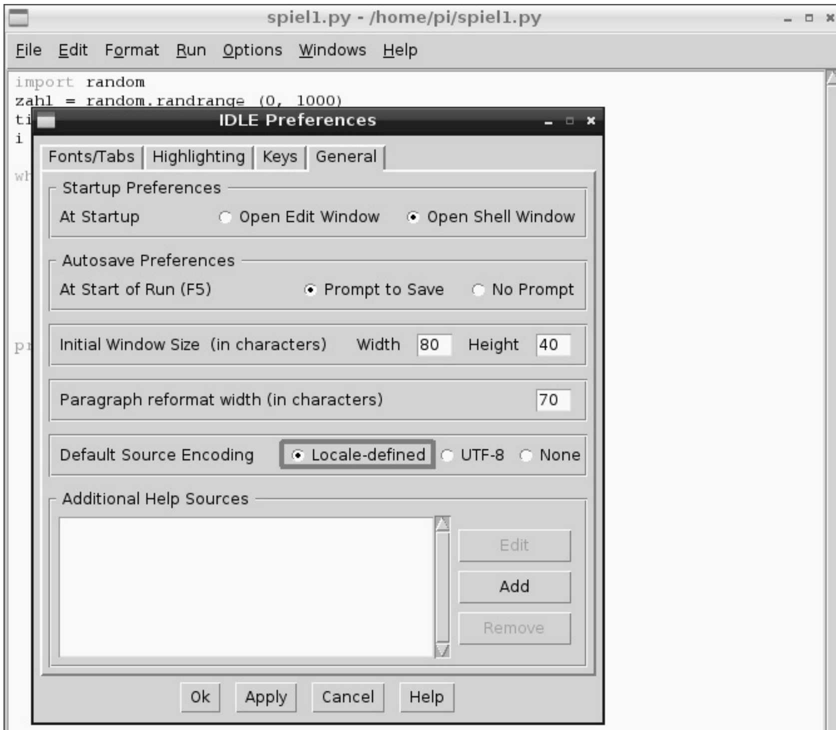


Fig. 1.8: The correct setting to display umlauts in Python.

4. Start the game using the key F5 or menu item *Run/Run Module*.

5. To simplify matters, the game forgoes any graphic interface and explanatory texts and does not query input parameters. The computer generates a random number between 0 and 1,000 in the background. Just enter a guess, and find out whether the number to be guessed is larger or smaller. Take more guesses until you have guessed the right number.

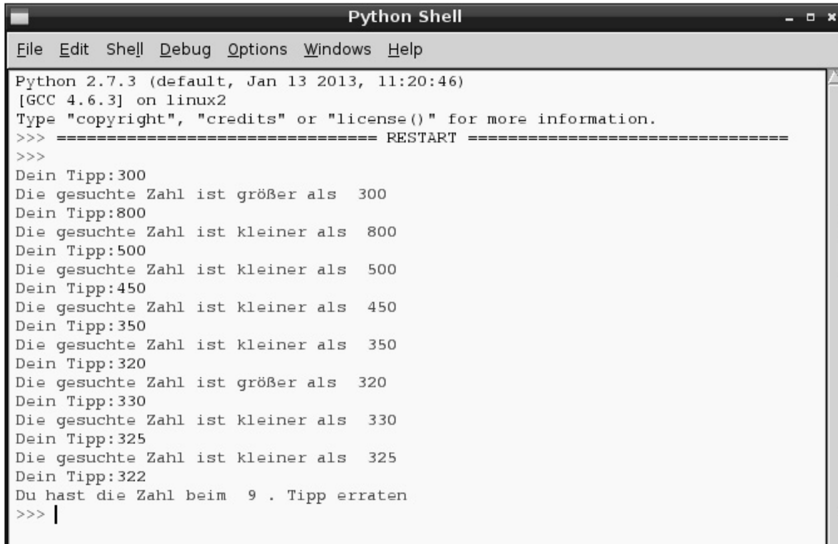


Fig. 1.9: 1.4.1 Guessing numbers with Python

1.4.2 How does it work?

Let's try and see if the game works. Naturally some questions are coming up: What is happening in the background? What are the individual command lines standing for in the programme?

`import random` To generate the random number, an external Python module called `random` is imported which contains various functions for random generators.

`number = random.randrange(1000)` The function `randrange` of the module `random` generates a random number in the number field which is defined by parameters, here a number between 0 and 999. The parameter of the function `random.randrange()` indicates the number of possible random numbers and beginning with 0, it is therefore always the first number not guessed right. The same goes for loops and similar functions in Python.

This random number is stored in the variable `number`. Variables in Python are memory locations which can have any name and can store numbers, strings, lists, or other data types. Unlike in some other programming languages, they do not have to be declared in advance.

How are random numbers generated?
Generally you would think that in a program nothing happens at random. How then is it possible that a program can generate random numbers? If you divide a large prime number by any value you will get umpteenth decimal digits that are barely predictable. These decimals change without any regularity, if the divisor is increased regularly. However, the outcome, which seems ostensibly at random, can be reproduced with an identical program or multiple calls of the same program at any time. If you are taking now any figure from these grouped figures and divide it by a number that is the result of the current time second or the contents of any storage position on the computer, an outcome will be produced that cannot be reproduced and it is therefore called a random number.

`guess = 0` The variable `guess` will later contain the number guessed by the player. It starts with `0`.

`i = 0` The variable `i` is commonly used among programmers as a counter for program loop cycles. Here it is utilized to count the number of guesses needed by the user to guess the secret number. Also this variable starts with `0`.

`while guess != number`: The word `while` initiates a program loop, which in our case is repeated as long as `guess`, the number guessed by the user, is unequal the secret number `number`. Python uses the character combination `!=` to express unequal. Following the colon is the actual program loop.

`guess = input("Your guess:")` The function `input` writes the text `Your guess:` and waits for the input, which is then stored in the variable `guess`.

Indentations are important in Python
In most programming languages the program loops or decisions are indented to make the code easier to read. In Python these indents serve not only for better clarity but are an absolutely necessity for the programming logic. However, specific punctuation is not needed to end a loop or decision.

`if number < guess`: If the secret `number` is less than the number guessed by the player `guess`, then ...

```
        print “The number guessed is less than “,guess
```

... this text is the output. The variable `guess` is written here at the end, so that the guessed number will be displayed in the text. If this condition is not true, the indented line is simply skipped.

`if guess < number`: If the secret `number` is greater than the number guessed by the player `guess`, then ...

```
        print “The number guessed is greater than “,guess
```

... a different text is displayed.

`i += 1` In each instance - not anymore indented - the counter `i` that counts the attempts is increased by one. The line with the operator += equals `i = i + 1`.

```
print “You guessed the number in ",i,". Guesses”
```

This line is more indented, which means that the `while` loop ends here. If the statement is not true any longer, that is, if the number `guess` guessed by the user is not unequal (but equal) the number of the secret `number`, then this text is displayed which consists of two parts of a sentence and the variable `i` and shows how many attempts were needed by the player. Python programs don't require a special statement to exit. They simply end after the last command is executed or after a loop that is no longer running and not followed by additional statements.

2 The first LED on Raspberry Pi is illuminated

The 26-pin block in the corner of the Raspberry Pi provides the option to connect hardware directly in order to make inputs via push buttons or to illuminate LEDs controlled by a program for instance. This pin header is referred to as GPIO. The abbreviation stands for "General Purpose Input Output".

17 of these 26 pins can be used for the digital input or output and as such can be utilized for a variety of hardware add-ons. The remaining pins are defined for the power supply and other purposes.

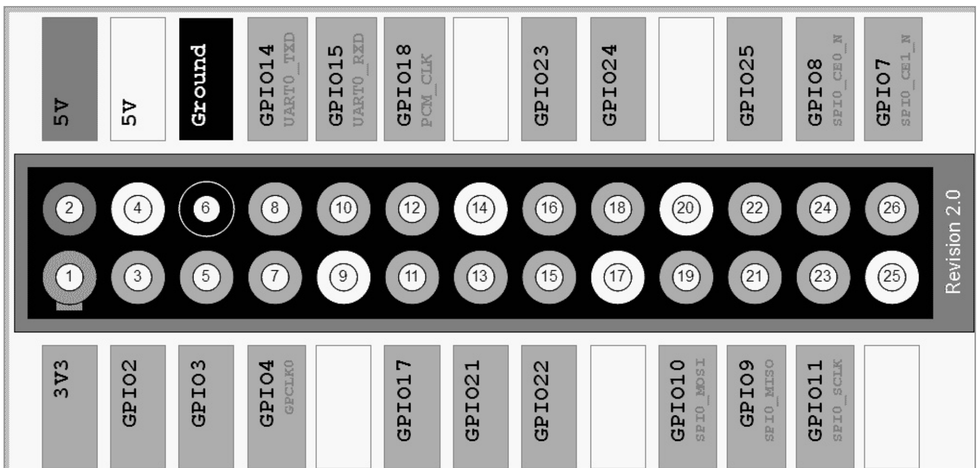


Fig. 2.1: Mapping the GPIO interface The gray line up and on the left indicates the border of the board. GPIO pin 2 lies therefore at the outer edge in the corner of the Raspberry Pi.

Caution
Under no circumstances do you connect any GPIO pins to one another and wait to see what will happen. Adhere to the following instructions:

Some GPIO pins are directly connected to terminals of the processor; a short circuit may totally ruin the Raspberry Pi. A series resistor must always be interposed when connecting two pins with each other using a switch or LED.

Pin 1, which supplies + 3.3V and a current draw up to 50 mA, is always used for logic signals. Pin 6 is the ground wire for logic signals. The others pins 9, 14, 17, 20, 25 which are named *Ground* or *3V3* are reserved for future add-ons. At present they can be used as labeled. However, you should not do that so that you can also use your own projects on future Raspberry Pi versions.

Each GPIO pin can be programmed as output (e. g. for LEDs) or as input (e. g., buttons). GPIO outputs in its logic state *1* supply a voltage of +3.3 V, in the logic state *0* 0 Volt. GPIO inputs supply at a voltage of up to + 1.7 V, the logic signal *0*, at a voltage between +1.7 V and +3.3 V, the logic signal *1*.

Pin 2 supplies +5 V as a power supply for external hardware. Here as much power can be drawn as the USB power supply of the Raspberry Pi is able to deliver. Do not connect this pin to a GPIO input.

2.1 Components contained in the package

The learning package contains various electronic components that allow you to build the experiments described (as well as your own). The components are only briefly introduced here. You will gain the essential practical experience in dealing with them through the actual experiments.

- 2x breadboard
- 1x LED red
- 1x LED yellow
- 1x LED green
- 1x LED blue
- 4x button
- 4x resistor 10 kOhm (brown-black-orange)
- 4x resistor 1 kOhm (brown-black-orange)

- 4x resistor 220 Ohm (red-red-brown)
- 12x connecting cables (breadboard – Raspberry Pi)
- ca. 1 m jumper wire

2.1.1 Breadboards

Two breadboards are included in the package so that electronic circuits can be built quickly. Here the electronic components can be plugged directly into a matrix with standard spacing, and without the need to solder. The outer longitudinal rows with contacts (X and Y) are all interconnected on these circuit boards.

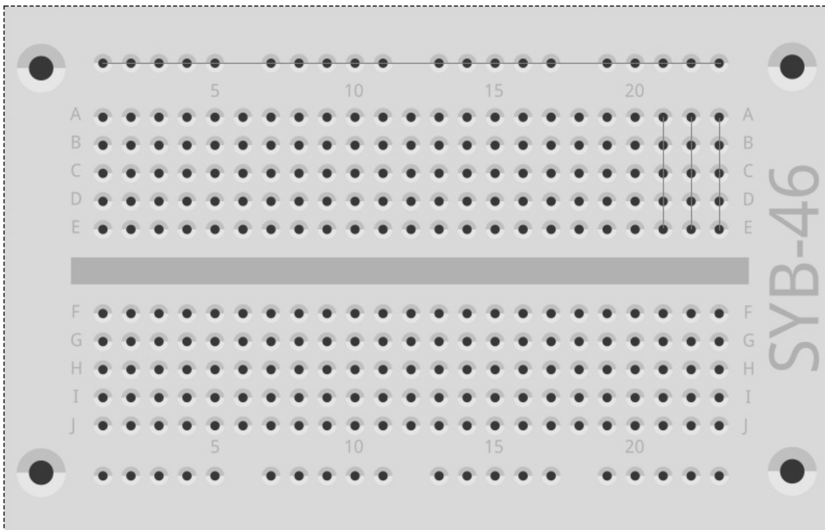


Fig. 2.2: The breadboard from the package with some sample connections indicated.

These contact rows are often used as the positive and negative terminals for the supply of power to the circuits. The other rows of contacts have each five contacts (A to E and F to J) that are cross-connected with each other, whereas in the centre of the board is a gap. That way, larger components can be plugged in here in the centre and are wired to the outward.

2.1.2 Connecting cable

All coloured connecting cables have a small wire connector on one side that can be plugged into the breadboard. On the other side you will find a socket that fits on a GPIO pin of the Raspberry Pi.

A jumper wire is also included in the learning package. It is used to build short connecting bridges to which the contact rows on the breadboard are connected. Cut the wire with a wire cutter to the appropriate length, as described in the individual experiments. We recommend cutting the wires slightly at an angle, so that they form a wedge-shape, which allows you to insert the wires into the breadboard more easily. Remove the insulation at both ends over a length of roughly half a centimeter.

2.1.3 Resistors and colour codes

Resistors are used in digital electronics primarily to limit the current at the ports of a microcontroller as well as series resistors for LEDs. Resistance is measured in Ohm. 1,000 ohms are one kilo Ohms, and is abbreviated kOhm.

The resistance values are indicated by colored rings on the resistors. Most resistors have four such colored rings. The first two colored rings stand for numerals, the third is a multiplier and the fourth is the tolerance. This tolerance ring has usually a gold or silver color - these are colors you will not find on the first rings, so the reading orientation becomes clear. The tolerance value itself has barely any significance in digital electronics.

Colour	**Resistance value in Ohm**			
	1. Ring (Tenth)	**2. Ring (Ones)**	**3. Ring (Multiplier)**	**4. Ring (Tolerance)**
Silver			10^{-2} = 0.01	±10 %
Gold			10^{-1} = 0.1	±5 %
Black		0	10^{0} = 1	
Brown	1	1	10^{1} = 10	±1 %
Red	2	2	10^{2} = 100	±2 %
Orange	3	3	10^{3} = 1,000	
Yellow	4	4	10^{4} = 10,000	
Green	5	5	10^{5} = 100,000	±0,5 %
Blue	6	6	10^{6} = 1,000,000	±0,25 %
Violet	7	7	10^{7} = 10,000,000	±0,1 %
Grey	8	8	10^{8} = 100,000,000	±0,05 %
White	9	9	10^{9} = 1,000,000,000	

Tab. 2.1: The table shows the significance of the coloured rings on the resistors.

The learning package contains resistors in three different values:

Value	1. Ring (Tenth)	2. Ring (Ones)	3. Ring (Multipl.)	4. Ring (Toleranc e)	Use
220	Red	Red	Brown	Gold	Series resistors for LEDs
1 kOhm	Brown	Black	Red	Gold	Protective resistors for GPIO inputs
10 kOhm	Brown	Black	Orange	Gold	Pull-down resistors for GPIO inputs

Tab. 2.2: Colour codes of resistors in the learning package

Pay attention in particular to the colours of the 1 kOhm and 10 kOhm resistors. These are mixed up easily.

2.2 Connecting the LED

LEDs (LED = Light Emitting Diode) for light signals and lighting effects can be connected to the GPIO ports. Hereby a 220 ohm series resistor (red-red-brown) will be installed between the appropriated GPIO pin and the anode of the LED in order to limit the current flow and thus preventing the LED from burning through. Besides, the series resistor protects the GPIO output of the Raspberry Pi, since the LED offers almost no resistance in the direction of the current flow, the GPIO port could become overcharged quickly when connected to ground. The cathode of the LED is connected to the ground wire on Pin 6.

Connecting LED plug but using which direction?

The two connecting wires of an LED differ in length. The longer one of the two is the positive pole, the anode, the shorter one is the cathode. Easy to remember: The plus sign has one bar more than the minus sign, thus making the wire a bit longer. Also most LEDs are flat on the minus end, just like a minus sign. Easy to remember: Cathode = short lead = flat spot

Connect first an LED via a 220-ohm series resistor (red-red-brown) to a +3.3 V connector (Pin 1), as shown in the image, and connect the negative terminal of the LED to the ground wire (Pin 6).

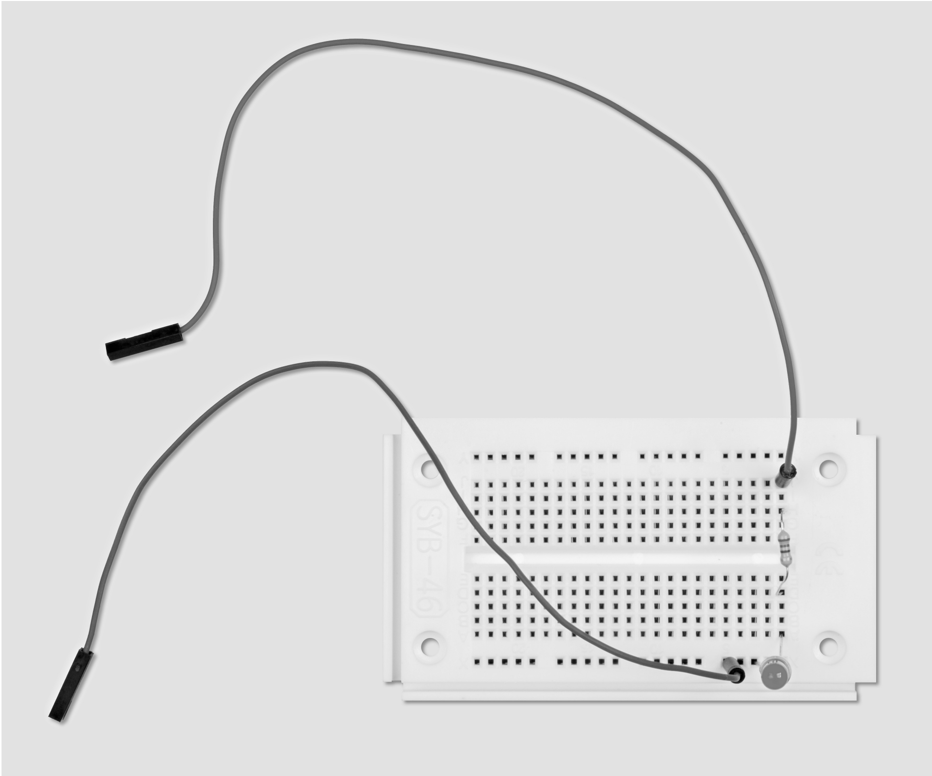


Fig. 2.3: Breadboard assembly to connect a LED.

Components required:
1x breadboard
1x LED red
1x 220-ohm series resistor
2x connecting wire

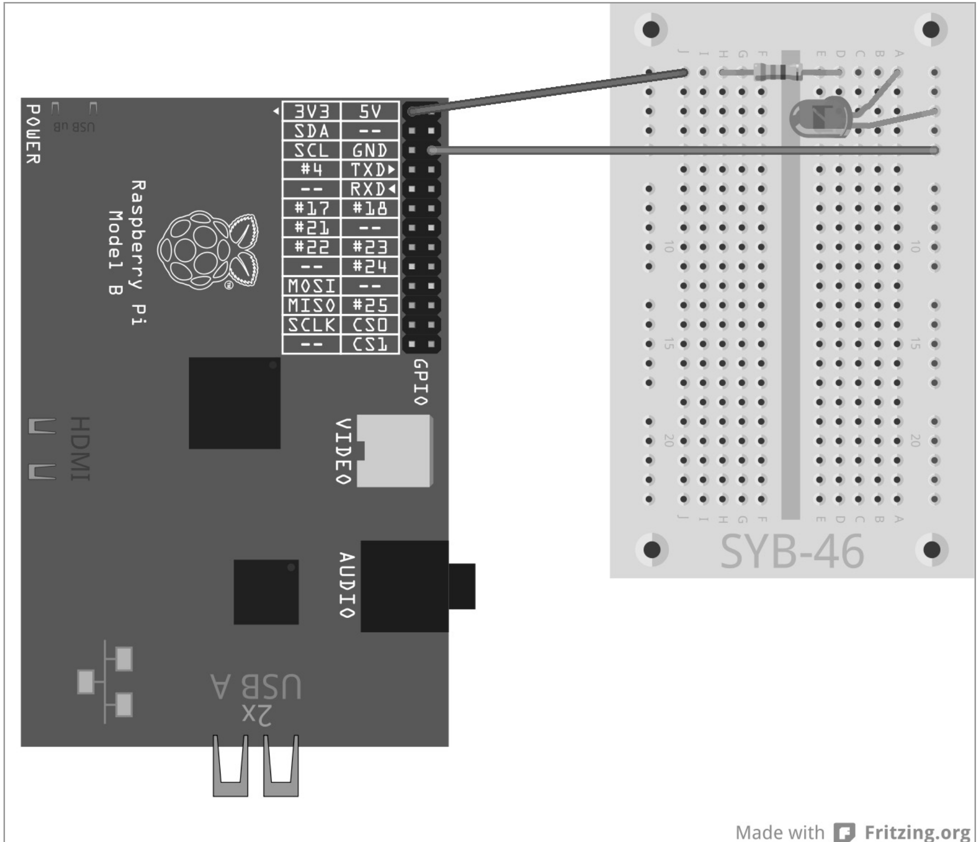


Fig. 2.4: The first LED on the Raspberry Pi.

In this first experiment we will use the Raspberry Pi only as the power supply for the LED. The LED is always on and there is no need for any software.

In the next experiment you add a button to the lead of the LED. The LED lights up only when this button is pressed. Here, too, no software is needed.

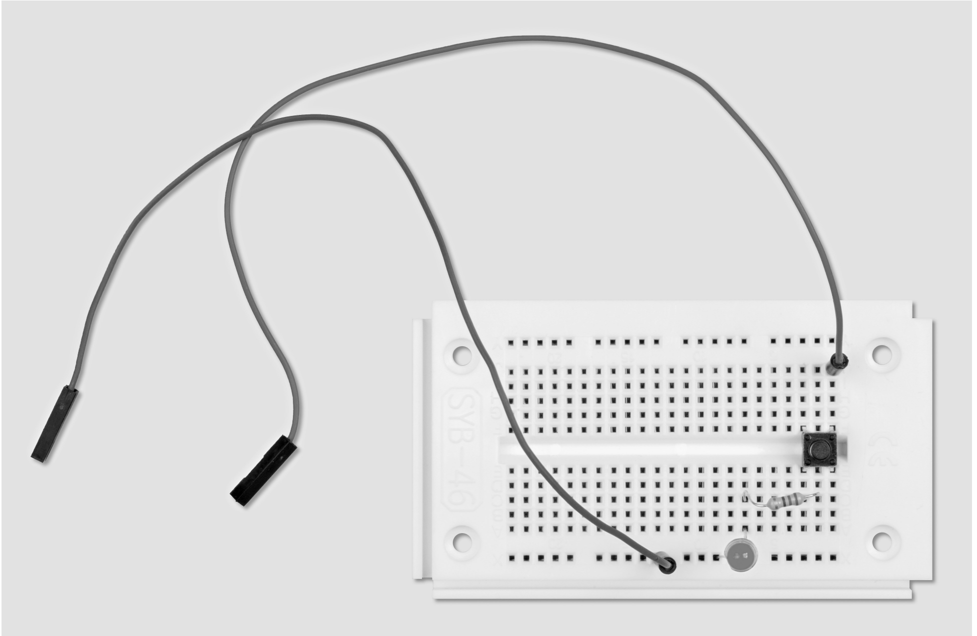


Fig. 2.5: Breadboard assembly for a LED that is controlled by a button.

Components required:
1x breadboard
1x LED red
1x 220-ohm series resistor
1x button
2x connecting wire

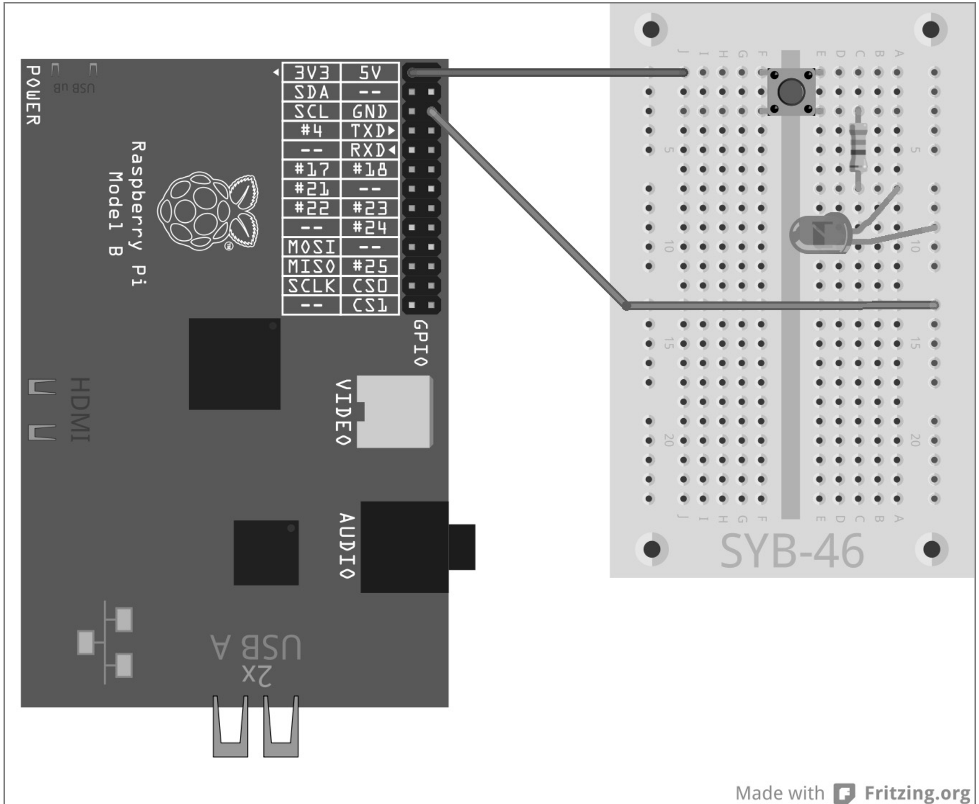


Fig. 2.6: LED with button on a Raspberry Pi.

2.3 GPIO and Python

In order to be able to use Python programs on the GPIO ports, the Python library for GPIO must be installed. If you are you not sure whether all necessary modules are installed, reinstall the up-to-date versions using the following the console commands:

```
sudo apt-get update
sudo apt-get install python-dev
sudo apt-get install python-rpi.gpio
```

GPIO ports are listed as data files in the directory structure, as is common practice for all devices under a Linux operating system. To access the file you will need root-level access privileges. Start the Python shell with root privileges using the LXTerminal: `sudo idle`

2.4 Switching the LED on or off

Connect a LED via a 220-ohm series resistor (red-red-brown) to the GPIO port 25 (Pin 22), and not anymore directly to the + 3.3 V connector, and connect the negative terminal of the LED via the ground rail on the breadboard to the ground wire of the Raspberry Pi (Pin 6), as shown in the next image.

Components required:
1x breadboard
1x LED red
1x 220-ohm series resistor
2x connecting wire

The next program `led.py` turns the LED on for 5 seconds and then off:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(25, GPIO.OUT)
GPIO.output(25, 1)
time.sleep(5)
GPIO.output(25, 0)
GPIO.cleanup()
```

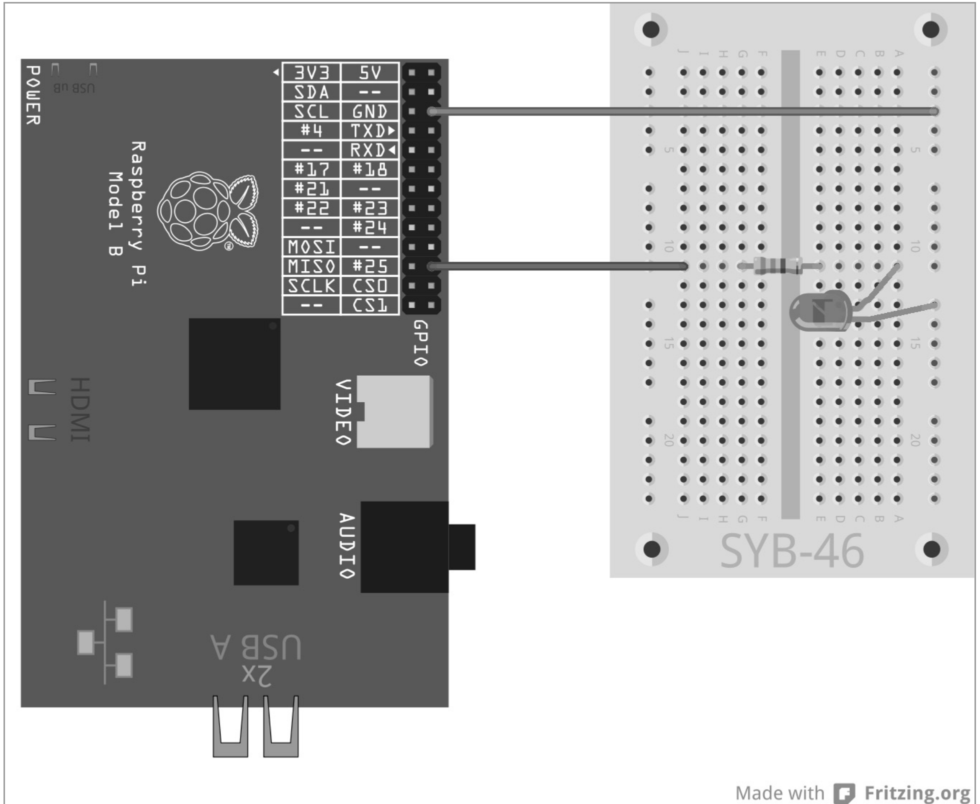


Fig. 2.7: An LED at GPIO port 25

2.4.1 How does it work?

The example shows the essential basic functions of the `RPi.GPIO` library.

`import RPi.GPIO as GPIO` The library `RPi.GPIO` must be imported into every Python program, that you are going to use. By using the prefix `GPIO` this notation can call all functions of the library.

`import time` The often used Python library `time` has nothing to do with GPIO programming. It contains functions in terms of time and date calculation, including the `time.sleep` function, which helps to implement waiting times in a program in a very simple way.

`GPIO.setmode (GPIO.BCM)` When you start with any program the designation of GPIO ports must be defined. Typically the default numbering `BCM` is used.

Numbering of the GPIO ports
The library `RPi.GPIO` supports two modes to identify the ports. In mode `BCM` the known GPIO port numbers are used, as they are also used on command line level or in shell scripts. In alternative mode `BOARD` the notation equal the pin numbers from 1 to 26 on a Raspberry Pi board.

`GPIO.setup(25, GPIO.OUT)` The function `GPIO.setup` initializes a GPIO port as output or as input. The initial parameter identifies a port depending on the preset mode `BCM` or `BOARD` by a GPIO number or a pin number. The second parameter can either be `GPIO.OUT` for an output or `GPIO.IN` for the input.

`GPIO.output(25, 1)` On the port that is just initialized, `1` is called out. The LED connected there lights up. Instead of `1` also the predefined values `True` or `GPIO.HIGH` may be called out.

`time.sleep(5)` This function is derived from the previously imported `time` library and triggers a waiting time of 5 seconds until the program continues to run.

`GPIO.output(25, 0)` To turn off the LED you call out the value `0` or `False` or `GPIO.LOW` on the GPIO port.

`GPIO.cleanup()` At the end of a programme the GPIO ports must be reset. This line is doing it for all of the GPIO ports that were initialized by the programme and at once. Ports that have been initialized by another programme remain unaffected. That way the process of any other, possibly parallel running programmes is not disrupted.

Intercepting GPIO warnings
If you configure a GPIO port that was not properly reset and which may be still open due to being used by another program or by an aborted program, you will receive a warning alert that, however, does not interrupt the program flow. These alerts can be very useful to detect errors during the process of developing a program. Yet in a ready-made programme an inexperienced user may get confused. Therefore there is an option in the GPIO library to suppress these warnings using `GPIO.setwarnings(False)`.

2.5 Starting Python with GPIO support without terminal

If you work a lot with Python and GPIO, you don't want to call up LXTerminal every time to start IDLE. There is an easier way. Add an icon to your desktop that calls the Python IDE using superuser privileges:

- Create a copy of the pre-installed desktop icon *IDLE*. Proceed as follows:

- Right-click the icon *IDLE* on the desktop using your mouse and select *Copy* in the shortcut menu.

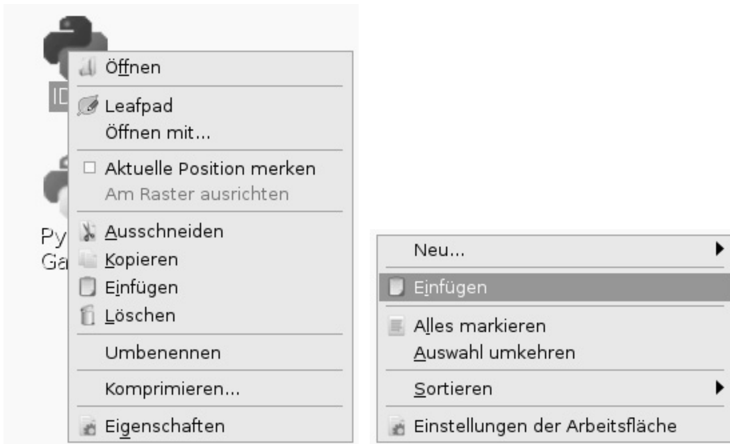


Fig. 2.8: Copying the *IDLE* desktop icon.

Then right-click on the desktop with your mouse and select *Paste* in the shortcut menu. A message appears when you attempt to create a copy that a desktop shortcut with the same name already exists.

Rename the copy here from *idle.desktop* to *idle_gpio.desktop*. Basically, the icon on the desktop will not be affected. The name shown is still *IDLE*.

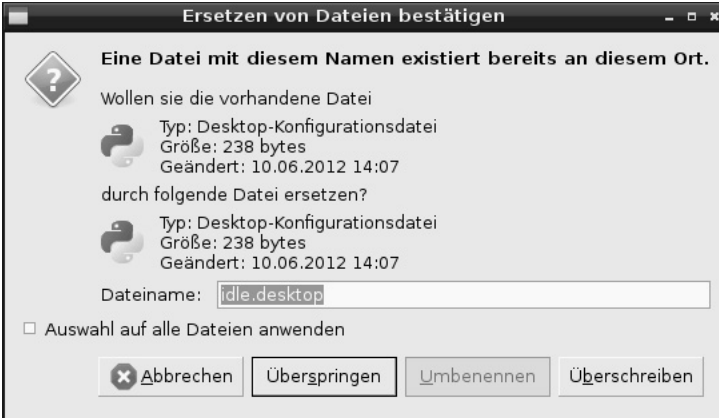


Fig. 2.9: A message of a desktop shortcut when duplicating.

Right-click now the copy of the desktop icon with your mouse and select *leafpad* from the shortcut menu. In Linux desktop shortcuts are mere text files, which can be edited using a text editor.

Fig. 2.10: The desktop shortcut in Leafpad editor.

Make the two modifications as illustrated here:

- Change the field `Name` to `IDLE GPIO`. This is the name shown on the screen.
- In the field `Exec` enter just before the actual command call the word `sudo`.

Close the editor and save the file. Double click on the new desktop icon and start the Python-IDE *IDLE* with superuser privileges. You can use now the GPIO functions without the need to calling up Python over the LXTerminal.

3 Traffic light

To turn on and off a single LED may be very exciting at first, but you do not really need a computer for this. A traffic light with a typical luminous cycle that is changing from green to yellow and then to red, and then from the light combination red and yellow back to green, can be easily built with three LEDs and hence, shows more programming techniques in Python.

Build the pictured circuit on the breadboard. In order to control the LEDs we are going to use three GPIO ports and a joined ground wire. The GPIO port numbers in BCM mode are shown in the drawing on the Raspberry Pi.

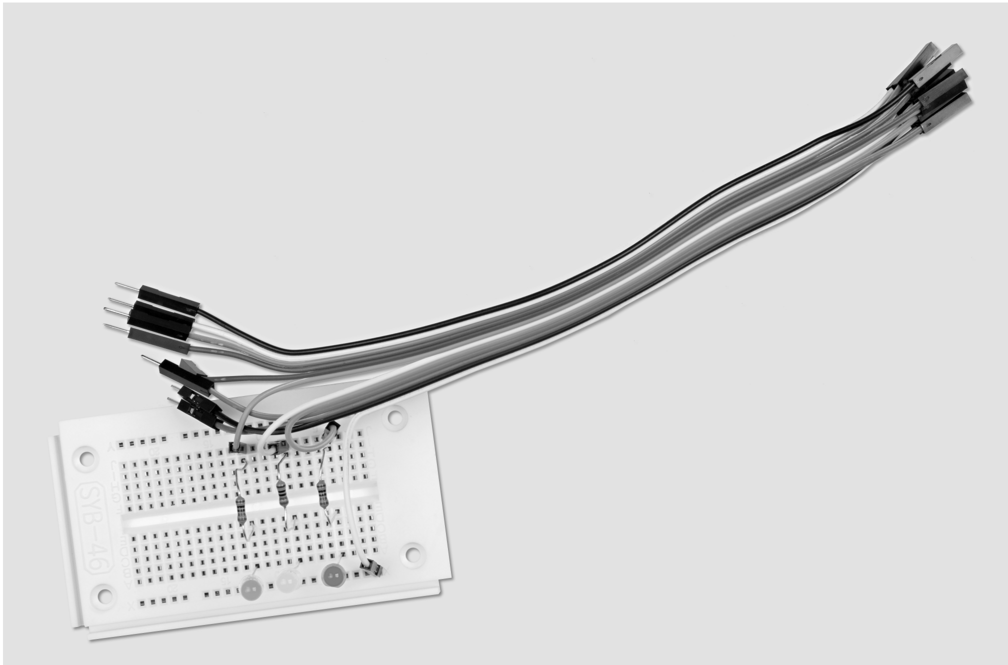


Fig. 3.1: Breadboard assembly for the traffic light.

Components required:
1x breadboard
1x LED red
1x LED yellow
1x LED green
3x 220-ohm series resistor
4x connecting wire

Fig. 3.2: A simple traffic light.

The program `ampel01.py` controls the light:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
red    = 0; yellow   = 1; green  = 2
Light=[4,18,23]
GPIO.setup(light[red], GPIO.OUT, initial=False)
GPIO.setup(light[yellow], GPIO.OUT, initial=False)
GPIO.setup(light[green], GPIO.OUT, initial=True)
print ("Strg+C exits the program")
try:
    while True:
        time.sleep(2)
        GPIO.output(light[green],False); GPIO.output(light[yelllow],True)
        time.sleep(0.6)
```

```
        GPIO.output(light[yellow],False); GPIO.output(light[red],True)
        time.sleep(2)
        GPIO.output(lightl[yellow],True)
        time.sleep(0.6)
        GPIO.output(light[red],False); GPIO.output(light[yellow],False)
        GPIO.output(lihtl[green],True)
except KeyboardInterrupt:
    GPIO.cleanup()
```

3.1.1 How does it work?

The first lines are already known, they import `RPi.GPIO` libraries for controlling the GPIO ports and the `time` for time delays. Then the numbering of the ports GPIO, as in the previous example, is set to BCM.

`red    = 0; yellow   = 1; geuen  = 2` These lines define the three variables `red`, `yellow` and `green` for the three LEDs. Thus the program does not require you to remember numbers or GPIO ports; you can be easily control the LEDs via its coloring.

`Light=[4,18,23]` To control the three LEDs, a list is created which contains the GPIO numbers in the sequence in which the LEDs are installed on the breadboard. Because the GPIO ports surface only in this part of the program, you can easily rebuild the program to use it with different GPIO ports.

```
GPIO.setup(light[red], GPIO.OUT, initial=False)
GPIO.setup(light[yellow], GPIO.OUT, initial=False)
GPIO.setup(light[green], GPIO.OUT, initial=True)
```

One by one, the three GPIO ports used will be initialized as outputs. We will not use the GPIO port numbers this time, but the list which we have previously defined. Within a list the individual elements are indexed using numbers starting from 0. `Light[0]` is therefore the first element in our case, it is `4`. The variables `red`, `yellow` and `green` contain the numbers `0`, `1` and `2`, which are required to classify the elements in the list as indices. That way the GPIO ports can be addressed by the colors used:

- `Light[red]` is GPIO port `4` with the red LED.
- `Light[yellow]` is GPIO port `18` with the yellow LED.
- `Light[green]` is GPIO port `23` with the green LED.

The `GPIO.setup`instructions may include an optional parameter `initial`, which assigns a logical status to the GPIO port the moment it is initialized. In this program we already turn on the green LED from start. The other two LEDs start the program in turned off state.

`print (“Ctrl+C exits the program")` Brief instructions are displayed on the screen. The program runs automatically. The key combination `Ctrl`+`C` is used to quit the program. We will use the query string `try...except` to ask the user, whether the program shall quit with `Ctrl`+`C`. The code assigned under `try:`is initially executed normally.

If in the meantime a system exception occurs – which can be due to an error or caused by the keyboard shortcut Ctrl + C – the code will abort and the statement `except` at the end of the programme is executed.

```
except KeyboardInterrupt:
  GPIO.cleanup()
```

The key combination will invoke a `KeyboardInterrupt` and the loop is automatically exited. The last line closes the GPIO ports used and thus switches off the LEDs. After that the program closes. Due to the controlled closing of the GPIO ports no system warnings or abort messages that may confuse the user are shown. The actual traffic light cycle runs in an infinite loop.

`while True` : Such infinite loops require always a termination condition, or the programme would otherwise never stop running.

`time.sleep(2)` The green LED lights up for 2 seconds at the beginning of the program and also at each fresh start of a loop.

```
        GPIO.output(light[green],False); GPIO.output(light[yelllow],True)
        time.sleep(0.6)
```

Now the green LED turns off and the yellow LED lights up. This one alone will be lit for 0.6 seconds.

```
        GPIO.output(light[yellow],False); GPIO.output(light[red],True)
        time.sleep(2)
```

Now the yellow LED turns off and the yellow LED lights up instead. This one alone will be lit for 2 seconds. The red phase of a traffic light usually lasts significantly longer than the yellow phase.

```
        GPIO.output(lightl[yellow],True)
        time.sleep(0.6)
```

To start the red-yellow phase the yellow LED is also turned on without switching off another LED. This phase lasts for 0.6 seconds.

```
        GPIO.output(light[red],False)
        GPIO.output(light[yellow],False)
        GPIO.output(lightl[green],True)
```

At the end of the loop the light jumps back to green. The red and yellow LEDs are turned off, the green is turned on. The loop starts afresh with a waiting time of 2 seconds within the green phase of the traffic lights. You can adjust all times of course as you like. In reality the phases of traffic lights depend on the size of the crossing and the traffic flow. The yellow and the red-yellow phase are usually 2 seconds long.

4 Pedestrian lights

In the next experiment we are going to extend the traffic light by adding a pedestrian light which will feature a flashing light for pedestrians during the traffic light's red phase, as is common practice in some countries. Obviously we could also integrate a pedestrian light into the program with red and green lights, which are commonplace in Central Europe. However, besides the LEDs used for the traffic light the learning package contains only one more LED.

For the following experiment incorporate an additional LED with series resistor into the circuit as illustrated. This will be connected to the GPIO port 24.

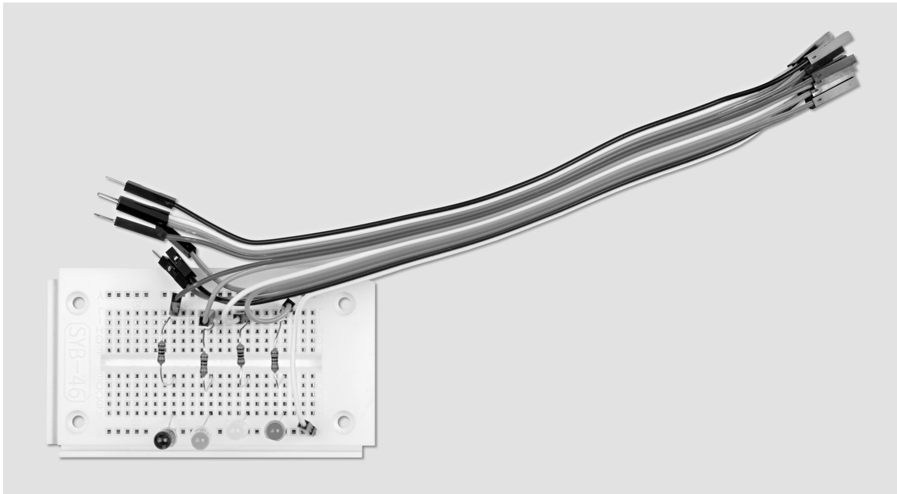


Fig. 4.1: Breadboard assembly for traffic light and flashing light for pedestrians.

Components required:
1x breadboard
1x LED red
1x LED yellow
1x LED green
1x LED blue
4x 220-ohm series resistor
5x connecting wire

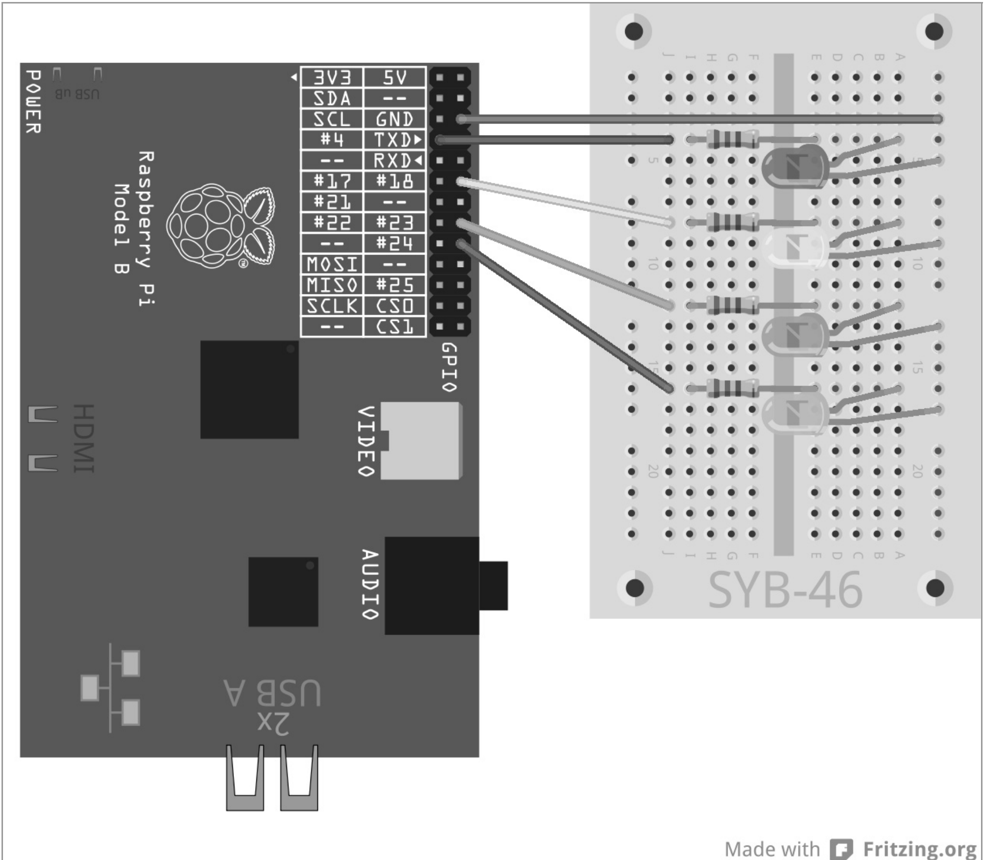


Fig. 4.2: Traffic light with flashing light for pedestrians.

The programme `ampel02.py` controls the traffic light system. Compared to the previous version the program has been slightly expanded.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
red    = 0; yellow   = 1; green  = 2; blue   = 3
Light=[4,18,23,24]
GPIO.setup(light[red], GPIO.OUT, initial=False)
GPIO.setup(light[yellow], GPIO.OUT, initial=False)
GPIO.setup(light[green], GPIO.OUT, initial=True)
GPIO.setup(light[blue], GPIO.OUT, initial=False)
print (“Ctrl+C exits the program")
try:
    while True:
```

```
        time.sleep(2)
        GPIO.output(light[green],False); GPIO.output(light[yelllow],True)
        time.sleep(0.6)
        GPIO.output(light[yellow],False); GPIO.output(light[red],True)
        time.sleep(0.6)
        for i in range(10):
            GPIO.output(light[blue],True); time.sleep(0.05)
            GPIO.output(light[blue],False); time.sleep(0.05)
        time.sleep(0.6)
        GPIO.output(light[yellow],True); time.sleep(0.6)
        GPIO.output(light[red],False)
        GPIO.output(light[yellow],False)
        GPIO.output(lightl[green],True)
except KeyboardInterrupt:
    GPIO.cleanup()
```

4.1.1 How does it work?

The programme sequence is pretty much known. During the red phase, now slightly prolonged, the blue pedestrian light should blink rapidly.

`blue = 4` A new variable in the list defines the LED for the pedestrian lights.

`Light=[4,18,23,24]` The list will add four elements in order to control the four LEDs.

`GPIO.setup(light[blue], GPIO.OUT, initial=False)` The new LED is initialized and at first switched off. This is the default setting during the traffic lights' green phase.

```
        time.sleep(0.6)
        for i in range(10):
            GPIO.output(light[blue],True); time.sleep(0.05)
            GPIO.output(light[blue],False); time.sleep(0.05)
        time.sleep(0.6)
```

The traffic light's cycle starts a loop during which the blue LED will flash, 0.6 seconds after the start of the red phase. To realize this, we use a `for` loop here, which contrary to the `while` loops used in the previous experiments always uses a certain number of loop runs until a certain termination condition is met.

`for i in range(10):` Each `for` loop needs a loop counter, that is, a variable, that adopts a new value with each loop run. All programming languages have accepted the variable name `i` for simple loop counters. Optionally, any other name is also possible, of course. This value can be queried like any other variable within the loop, but in our case this is not necessary. The parameter `range` in the loop indicates how many times the loop runs through, or more precisely, the values of the loop counter. In our example, the loop runs ten times. The loop counter `i` will have a value from `0` to `9`. Within the loop cycle the new blue LED is switched on and then off after 0.05 seconds. More 0.05 seconds and a loop run comes to an end; the next one starts again when the LED is switched on. In this manner it flashes ten times; overall it lasts for a total of one second.

`time.sleep(0.6)` The normal switching cycle of the traffic light is continued with a delay of 0.6 seconds after the last loop run by switching on the yellow LED in addition to the already lit red LED. Nothing new

here so far. But it gets interesting when the pedestrian light, instead of running automatically, is started by pressing a button as is the case with many pedestrian lights. In the next experiment a button is connected to a GPIO port that will simulate the push button on a real traffic light.

4.2 Button on the GPIO connector

GPIO ports not only output data, for example by means of LEDs, but they can also be used to input data. Therefore they must be defined as input in the programme. In the next project we are going to use a button for the input, which is directly plugged into the breadboard. The button has four connector pins, whereby each two pins that lie opposite to each other (long distance) are interconnected. As long as the button is pressed, all four connectors are linked to each other. Unlike a switch, a pushbutton does not lock. The connection breaks as soon as the button is released. If at a GPIO port which is defined as input a +3.3 V signal is present, it is evaluated as logical `True` or `1`. In theory, you could connect an appropriate GPIO port via a button to the +3.3 V port on the Raspberry Pi. Don't ever do it! It will overcharge the GPIO port. Always interpose a 1kohm protective resistor between a GPIO input and the + 3.3V terminal to protect the GPIO port from too much current that would also be flowing to the processor.

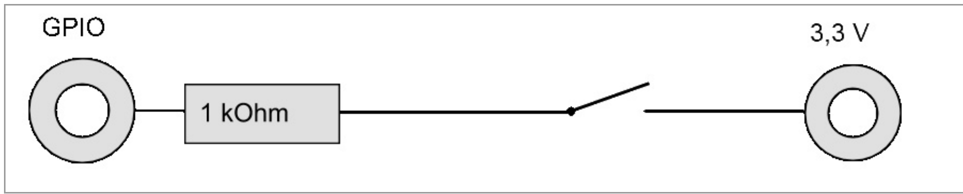


Fig. 4.3: Pushbutton with protective resistance on a GPIO input.

In most cases, this simple circuit works, but the GPIO port has no clearly defined state when the button is open. If a programme queries this port the results could be random. As a measure of prevention a relative high resistor - usually 10 ohms - is connected to ground. This so-called pull-down resistor pulls the status of the GPIO ports at open button back down to 0 V. As resistance is very high as long as the button is pressed, there is no risk of a short circuit. When the button is pressed, the + 3.3V and the ground wire are connected directly via this resistor.

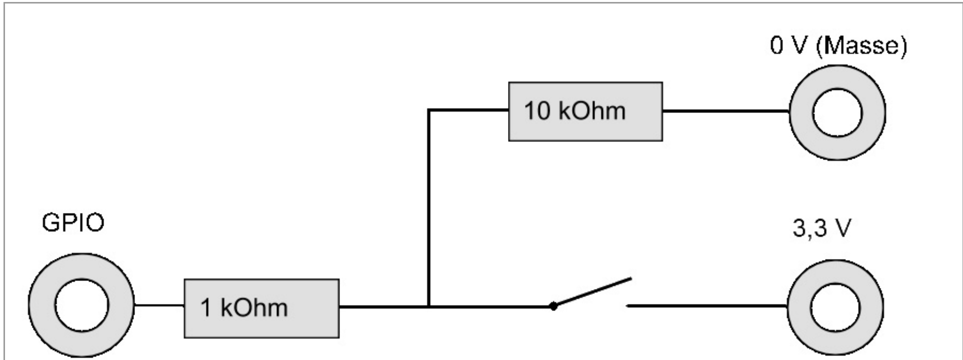


Fig. 4.4: Pushbutton with protective resistance and pull-down resistor on a GPIO input.

Build a button with its two resistors into the circuit according to the following illustration.

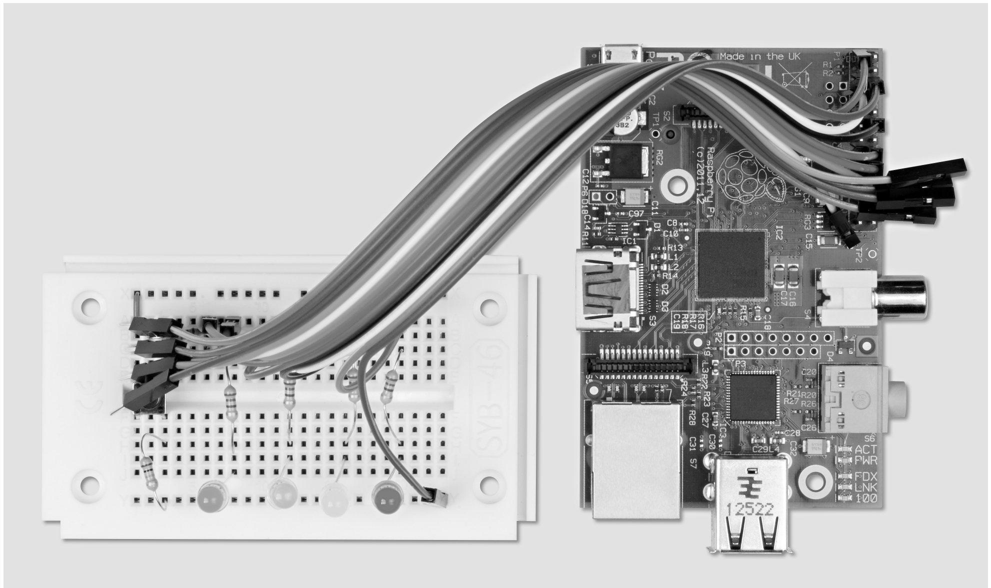


Fig. 4.5: Breadboard assembly for the pedestrian lights “on-demand”.

Components required:
1x breadboard
1x LED red
1x LED yellow
1x LED green
1x LED blue
4x 220-ohm series resistor
1x button
1x 1-kOhm resistor
1x 10-kOhm resistor
7x connecting wire
1x short jumper bridge

This figure shows the lower contact strip of the button connected via the positive rail of the breadboard to the +3.3 V wire of the Raspberry Pi (Pin 1). To connect the button to the positive rail, we will use a short jumper to keep the drawing clear. Alternatively, you can also connect one of the lower contacts of the button directly with a connecting cable to Pin 1 of the Raspberry Pi.

The upper contact strip of the pushbutton featured in this figure is connected via a 1 Kohm protection resistor (brown-black-red) to the GPIO port 25 and a 10-kohm pull-down resistor (brown-black-orange) to the ground wire.

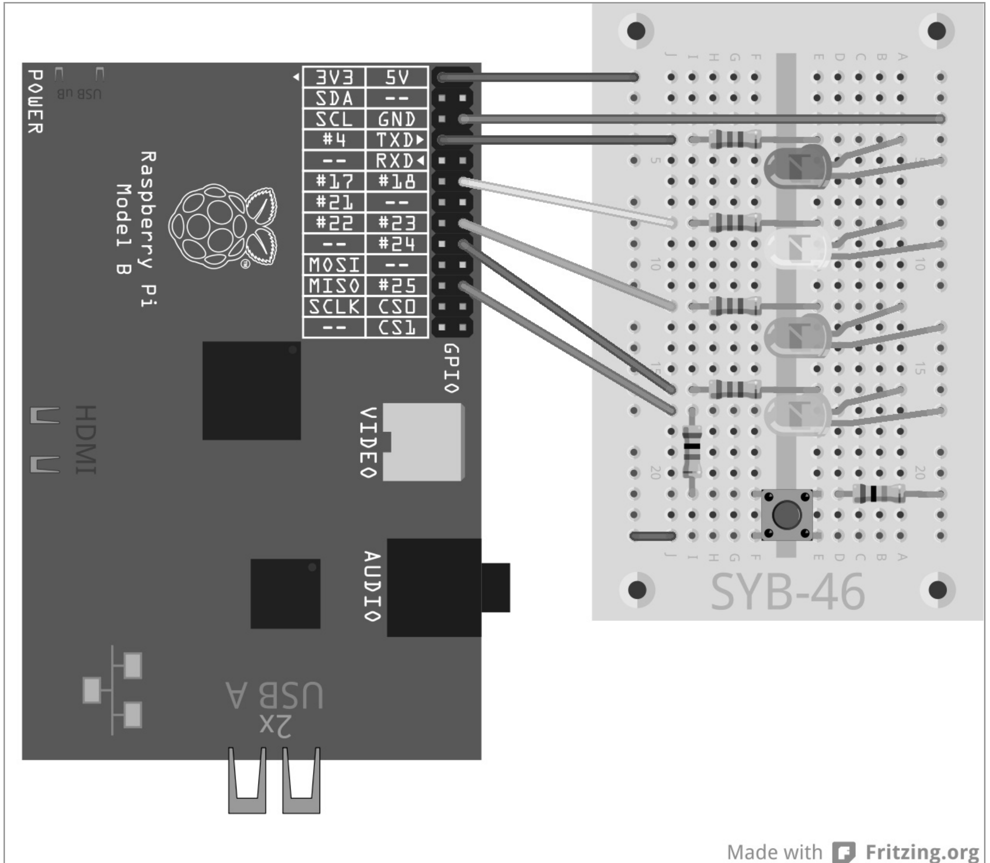


Fig. 4.6: Flashing pedestrian light with button.

The programme `ampel02.py` controls the new traffic light system with button for the flashing pedestrian light.

```
# -*- coding: utf-8 -*-
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

red    = 0; yellow   = 1; green  = 2; blue   = 3; button = 4

Light=[4,18,23,24,25]
GPIO.setup(light[red], GPIO.OUT, initial=False)
GPIO.setup(light[yellow], GPIO.OUT, initial=False)
GPIO.setup(light[green], GPIO.OUT, initial=True)
```

```
GPIO.setup(light[blue], GPIO.OUT, initial=False)
GPIO.setup(light[button], GPIO.IN)

print (“Press button to turn on pedestrian light,
“Ctrl+C exits the program")

try:
    while True:
        if GPIO.input(light[button])==True:
            GPIO.output(light[green],False)
            GPIO.output(lightl[yellow],True)
            time.sleep(0.6)
            GPIO.output(light[yellow],False)
            GPIO.output(light[red],True)
            time.sleep(0.6)
            for i in range(10):
                GPIO.output(light[blue],True); time.sleep(0.05)
                GPIO.output(light[blue],False); time.sleep(0.05)
            time.sleep(0.6)
            GPIO.output(lightl[yellow],True)
            time.sleep(0.6)
            GPIO.output(light[red],False); GPIO.output(light[yellow],False)
            GPIO.output(lightl[green],True); time.sleep(2)
except KeyboardInterrupt:
    GPIO.cleanup()
```

4.2.1 How does it work?

Some features have been added to the program with respect to the previous version.

`# -*- coding: utf-8 -*-` To display for instance the German umlauts of `Fußgängerblinklicht` that is pedestrian flashing light -correctly in the program output - regardless of how the user's IDLE surface is set up - we start with defining an encoding for the representation of special characters. This line should be included in all programs that will display texts which contain umlauts or other country-specific special characters.

ASCII, ANSI and Unicode

The alphabet consists of 26 letters plus a few special characters, all in uppercase and lowercase, plus ten digits, and some punctuation; that adds up to about 100 different characters. 256 different characters can be represented by one byte. That should be sufficient - so they thought at the beginning of computer history when the basics of today's technology were defined.

Very soon it turned out that the inventor of the ASCII character set (American Standard Code for Information Interchange), which is based on 256 characters, was wrong. They were Americans who had not looked beyond the English-speaking world. In all major languages of the world, that goes without even mentioning the Asian and Semitic languages with their own writing systems, feature hundreds of letters that need to be represented. Few of those found a free space in the list consisting of 256 characters.

Later, when in parallel with the ASCII character the ANSI character set was introduced, which is used by older versions of Windows, the very same mistake happened again. When the German umlauts and other accented characters were allocated other places in the character set than in the standard of ASCII, the linguistic chaos reached perfection.

As a solution to the problem, Unicode, which can represent all possible languages, including Egyptian hieroglyphics, cuneiform and Vedic Sanskrit, the oldest recorded written language in the world was introduced in the 90s. UTF-8, an encoding that works across platforms and that is congruent with the first 128 characters in ASCII - and therefore backward compatible with almost all text performing systems - is the most commonly used form to encode Unicode characters in plain text files. The coding is shown in a comment line. All lines beginning with the character # will not be evaluated by the Python interpreter. The encoding which must always stand at the very beginning of a program, tells the Python shell, how to represent characters; it is no real program statement. In the same way, you can also enter your own comments into the programming code.

Comments in programmes

When you are writing a programme, you sometime don't remember what was on your mind when you wrote certain programme statements. Programming is one of the most creative activities, because you are creating something merely based on your ideas and you are not limited by any materials and tools. Comments are important, especially when you are writing programmes that also need to be understood by another person, or which need further editing. In the sample programmes we did not include comments to keep the programming code clear. All programme statements are described in detail.

There is always one question that arises for programmes that are published by non-English speakers: Comments in English or German, for instance? If comments are in German, the French complain about the incomprehensible language; sometimes you yourself don't understand your own English comments anymore, and the British laugh at your poor English.

```
button = 4
Light=[4,18,23,24,25]
```

For reasons of simplicity, we are adding an additional element with the number `4` and the GPIO port 25 to the list for the button. That let's you also easily select another GPIO port for the button, because its number like the GPIO ports of LEDs appears only here in the program.

`GPIO.setup (traffic light [button], GPIO.IN)` The GPIO port of the button is defined as input. These definitions are also done via `GPIO.setup`, yet this time with the parameter `GPIO.IN`.

```
print (“Press button to turn on pedestrian light,
“Ctrl+C exits the program")
```

When starting the programme, a message is displayed that notifies the user to press the button

```
    while True:
        if GPIO.input(light[button])==True:
```

Within the loop, a query is now set up. The following statements are only executed when the GPIO port 25 has the value `True`, that is, the user presses the button. Until then, the traffic light will be in the green state. The rest of the loop's process basically corresponds to the previous programme. The traffic light turns to yellow and then to red, the flashing light flashes ten times. Then the lights returns from red to yellow and to green.

`time.sleep(2)` However there is a tiny deviation in this program. The green phase of 2 seconds is now built in at the end of the loop and no longer at the beginning. Nevertheless it is applied once for each loop run, with the difference that the traffic light cycle starts immediately and without delay as soon as the button is pressed. This delay is now included at the end of the loop to prevent the green phase from failing when the button is immediately pressed again after the yellow phase.

5 Colorful LED patterns and chaser lights

Chaser lights are popular effects in order to arouse attention, whether it is used in a party room in the basement or in professional illuminated signage. This can be easily realized using the Raspberry Pi and a few LEDs.

Set up four LEDs with series resistors for the following experiment as illustrated. This circuit corresponds to the pedestrian lights without button used in the previous experiment.

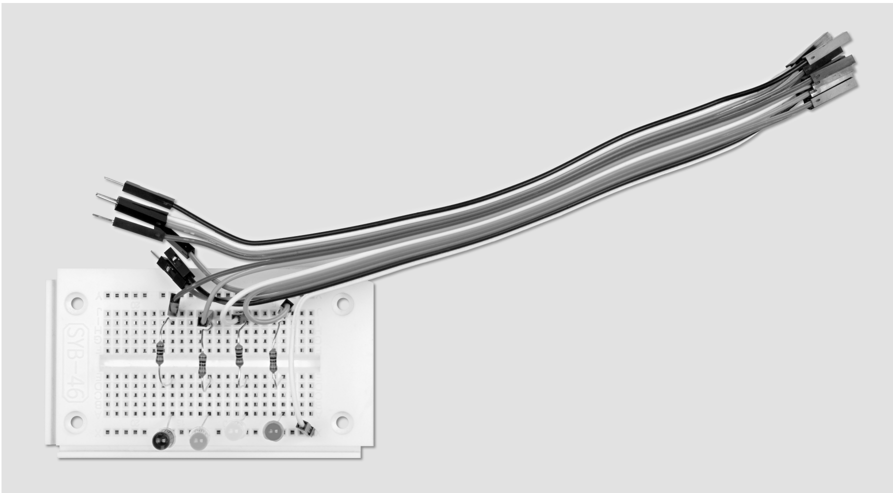


Fig. 5.1: Breadboard assembly for the samples and the chaser lights.

Components required:
1.x breadboard
1x LED red
1x LED yellow
1x LED green
1x LED blue
4x 220-ohm series resistor
5x connecting wire

Fig. 5.2: Four LEDs with series resistors.

We will explain now more loops and programming methods in Python by using different LED flashing patterns. The next programme offers different LED samples that the user can select by making keyboard entries.

The programme `ledmuster.py` allows the LEDs to use different flashing patterns.

```
# -*- coding: utf-8 -*-
import RPi.GPIO as GPIO
import time
import random

GPIO.setmode(GPIO.BCM)

LED = [4,18,23,24]
for i in LED:
    GPIO.setup(i, GPIO.OUT, initial=0)

z = len(LED); w = 5; t = 0.2

print (“Light effects selection"); print ("1 – cyclical chaser")
print ("2 – chase bidirectional"); print ("3 – ascending and descending")
print ("4 – all flashing simultaneously"); print ("5 – all flashing at random")
print (“Ctrl+C exits the program")

try:
    while True:
        e = raw_input (“Please select a pattern: ")
        if e == "1":
            for i in range(w):
                for j in range(z):
                    GPIO.output(LED[j], True); time.sleep(t)
                    GPIO.output(LED[j], False)
        elif e == "2":
            for i in range(w):
                for j in range(z):
                    GPIO.output(LED[j], True); time.sleep(t)
                    GPIO.output(LED[j], False)
                for j in range(z-1, -1, -1):
                    GPIO.output(LED[j], True); time.sleep(t)
                    GPIO.output(LED[j], False)
        elif e == "3":
            for i in range(w):
                for j in range(z):
                    GPIO.output(LED[j], True); time.sleep(t)
                time.sleep(2*t)
                for j in range(z-1, -1, -1):
                    GPIO.output(LED[j], False)
                    time.sleep(t)
                time.sleep(2*t)
        elif e == "4":
            for i in range(w):
                for j in range(z):
                    GPIO.output(LED[j], True)
                time.sleep(2*t)
```

```
                for j in range(z):
                    GPIO.output(LED[j], False)
                time.sleep(t)
        elif e == "5":
            for i in range(w*z):
                j = random.randint(0,z-1)
                GPIO.output(LED[j], True); time.sleep(t)
                GPIO.output(LED[j], False)
        else:
             print (“Invalid entry")

except KeyboardInterrupt:
    GPIO.cleanup()
```

5.1.1 How does it work?

Through previous experiments you are already familiar with the first lines of the program that is, the definition of the UTF-8 encoding and the import of the necessary libraries. Here, also the library `random`is imported to generate a random blinking pattern.

We have taken care when developing this programme that it is versatile and can be used in many ways, namely, that it easily possible to add more than only four LEDs. Today good programming practice embraces such flexibility. Using the example of the Raspberry Pi, such coded applications can not only be broadened by adding new GPIO ports, but it is also easy to rewrite them for other GPIO ports, should it be required by some hardware technicalities.

`LED = [4,18,23,24]` Again a list of GPIO port numbers for the LEDs is defined, so that these ports are recorded only once in the programmr.

```
for i in LED:
    GPIO.setup(i, GPIO.OUT, initial=0)
```

Unlike earlier programmes where the GPIO ports of the LEDs were individually initialized, we will run a `for` loop across the list of `LED` this time. The loop counter `i` accepts each value from the list in succession, in our example, the GPIO port numbers of LEDs, and is not simply incremented as in the previously used `for` loops.. Lists of any size can thus be processed. The length of the list may not even be known when the programme is being developed.

The four GPIO ports are defined as outputs for the LEDs and are set to `0` so that any LEDs that may still be lit as a result of previous experiments are disabled.

```
z = len(LED); w = 5; t = 0.2
```

For the programme to be universal and easily alterable, we will ow define three variables:

`z`	Number of LEDs	The number of LEDs is automatically adopted from the list `LED` by using the function`len()`.
`w`	Repetitions	Each pattern is repeated five times by default so that the pattern is better for better recognize it better detected. This number can be changed as desired, and is then applies to all patterns.
`t`	Time	This variable indicates how long a flashing LED is turned on. The pause, that follows will last just as long. The name `t` is commonly used in all programming languages for variables that store time intervals in programs.

The values defined as variables are introduced only here at this point within the program and can be easily changed. Following these definitions the actual program begins.

```
print (“Light effects selection"); print ("1 – cyclical chaser")
print ("2 – chase bidirectional"); print ("3 – ascending and descending")
print ("4 – all flashing simultaneously"); print ("5 – all flashing at random")
print (“Ctrl+C exits the program")
```

These lines will provide the user with an on-screen guidance on which numeric key will display which pattern.

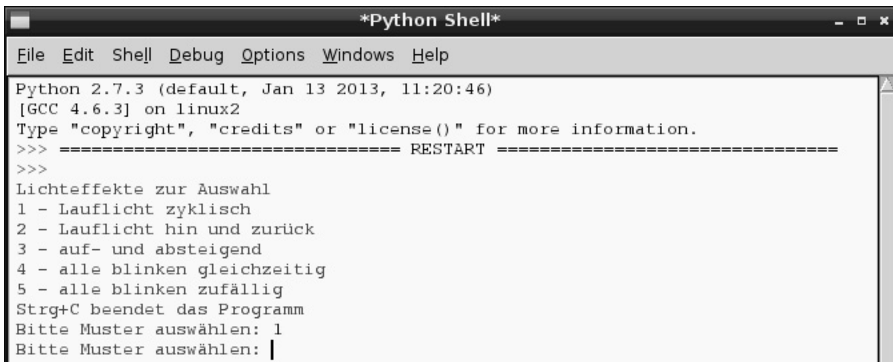


Fig. 5.3: The program on-screen.

After the selection has been indicated, the main loop of the program will start. We are also using here the infinite loop `while True:`, whicmeh is embedded in the statement `try...except`.

`e = raw_input (“Please select a pattern: ")` Right at the start of the loop, the program waits for the user's input which then is stored in the variable `e`. The function `raw_input()` adopts the input as plain text, without evaluating. In contrast, `input()`, which are mathematical operations or variable names, are evaluated on spot. That means that `raw_input()` is in most cases therefore the better choice, because you don't have to worry about the many events of possible inputs.

The program waits until the user has typed a character and has pressed the Enter key. Depending on the number the user has entered, the LEDs should now perform according to certain pattern. We will query this with the use of the construct if...elif...else.

Pattern 1

If the entry was 1, the indented part of the programme that comes after this line is executed.

if e == "1": Note that indents in Python are not used as optical effects. As we do with loops, we also start those queries with an indent.

> Equal is not equal equal
> Python uses two types of the equal sign. The simple = is used to assign a specific value to a variable. The double equal sign == is used in queries, and verifies whether two values are really the same.

If the user has entered 1 on the keyboard, a loop starts which produces a cyclical chaser light. These loops basically have for all LED patterns used the same structure.

for i in range(w): The outer loop repeats the pattern as many times as is indicated for the above-defined variable w. Within this loop lies another, which generates the respective pattern. This differs with each pattern.

```
        for j in range(z):
            GPIO.output(LED[j], True); time.sleep(t)
            GPIO.output(LED[j], False)
```

In the case of the simple cyclical chaser this loop runs one after the other through each LED in the list once. The number of the LEDs has been saved to the variable zat the beginning of the program. The LED with the number of the current state of the loop counter is turned on. The program waits now for the time frame that was initially stored in the variable t and then turns the LED off. The next loop run starts with the next LED. The outer loop repeats the entire inner loop five times.

Pattern 2

A similar loop is started when the user enters the input 2. The LEDs here are not only counted in one direction, instead at the end of the chase the counting order is reversed. The light alternates backward and forward.

elif e == "2": After the first query any queries that follow are using the query stringelif, , which means that they will only be executed if the previous query has returned as a Falseresult.

```
    for i in range(w):
        for j in range(z):
            GPIO.output(LED[j], True); time.sleep(t)
            GPIO.output(LED[j], False)
        for j in range(z-1, -1, -1):
            GPIO.output(LED[j], True); time.sleep(t)
            GPIO.output(LED[j], False)
```

Here too nested loops are used. After the first inner loop, which corresponds to the part of the program described above, that is, after the LED with the number 3 lights up, another loop starts for the chaser light now running in the opposite direction. Elements in the list start always with 0. So the fourth LED has the number 3.

For loop to run backwards, we are going to use the extended syntax `for...range()`. Here, instead of specifying only one final value, three parameters can be set: Start value, step value and end value. These are in our example:

Start value	z-1	The variable `z` contains the number of LEDs. Since numbering of list elements starts with `0`, the last LED will have the number `z-1`.
Step value	-1	When the increment is`-1` each loop run counts one number backwards.
End value	-1	The end value in a loop is always the first value, which is not reached. In the first forward counting loop, the loop counter starts at `0` and in our example, reaches the values `0`, `1`, `2`, `3` to address the LEDs. `4` is not processed within the four-time loop run. The backward counting loop should end with `0` and thus should not reach the value `-1` as the first value.

The second loop described permits the flashing of the four LEDs in sequence and in reverse direction. After that, the outer loop will restart the cycle, which is here twice as long as in the first part of the program because each LED flashes twice.

Pattern 3

A similar loop is started when the user enters the input `3`. Here the LEDs are also counted bidirectional but they will not turn off immediately after being switched on.

```
        elif e == "3":
            for i in range(w):
                for j in range(z):
                    GPIO.output(LED[j], True); time.sleep(t)
                time.sleep(2*t)
                for j in range(z-1, -1, -1):
                    GPIO.output(LED[j], False); time.sleep(t)
                time.sleep(2*t)
```

The first inner loop will turn on the LEDs one at a time with a time lag. At the end of the loop, indicated by the dedent of the line `time.sleep(2*t)` it waits for the double lagged time. That is when all LEDs are illuminated. Then another loop that counts backwards starts and one LED after the other turns off. Here, too, when all LEDs are off, there is a waiting for the doubled time delay at the end, before the outer loop will restart the whole cycle again.

Pattern 4

If the user has entered 4, a different flashing pattern starts, in which all LEDs are flashing simultaneously and are not called up in sequence.

```
        elif e == "4":
            for i in range(w):
                for j in range(z):
                    GPIO.output(LED[j], True)
                time.sleep(2*t)
                for j in range(z):
                    GPIO.output(LED[j], False)
                time.sleep(t)
```

Not all GPIO ports can be switched on or off at once with a single statement, that is why also here loops are used, however, without a time lag within the loop. The four LEDs are turned on instantly one after the other. For the human eye, it seems to happen simultaneously. At the end of the first inner loop the program waits for the lapse of the doubled delay time before all LEDs are turned off again.

The flashing lights generate different effects of light and dark through the use of different time intervals. Flashes are noticed more when the flash duration is longer than the dark period. Very short flashes with a relatively long dark period produce a photoflash effect.

Pattern 5

If the user has entered 5, the LEDs flash totally at random.

```
        elif e == "5":
            for i in range(w*z):
                j = random.randint(0,z-1)
                GPIO.output(LED[j], True); time.sleep(t)
                GPIO.output(LED[j], False)
```

No nested loops are used here, so we have the loop run through more often. An LED will blink roughly as often as in the first pattern, when the variables w and z are multiplied.

The function random.randint() writes a random number in the variable j. This random number is greater than or equal to the first parameter and less or equal to the second parameter. In our case it can have the values 0, 1, 2, 3.

The randomly selected LED is turned on and then after lapse of the delay time turns off again. After that, the loop restarts and a new LED is selected at random.

Invalid entry

Programs that require an user input must be able to intercept incorrect entries. If the user enters something that was not intended, the programme must respond.

```
        else:
             print (“Invalid entry")
```

If the user has entered something else, the statement specified in `else`is executed. This section of a query is always applied, if no other queries have delivered true results. In our case, the programme displays a message on-screen.

As in previous experiments, the programme is terminated via `KeyboardInterrupt`, that is, the user pressing the key combination `Ctrl`+`C`. The last line closes the GPIO ports used and thus switches off the LEDs.

6 Dimming the LED using pulse width modulation

LEDs are typical components of digital electronics to output signals. They can assume two different states, on and off, 0 and 1 or `True` und `False`. The same applies to those GPIO ports which are defined as digital outputs. Theoretically dimming a LED seems to be impossible.

A little trick makes it possible to control the brightness of an LED on a digital GPIO port. If you let a LED flash fast enough, then the human eye will not identify that as true flashing. The technique termed pulse width modulation generates a pulsing signal that turns on and off at very short intervals. The voltage of the signal remains always the same, only the ratio between level `False` (0 V) and level `True` (+ 3.3V) is changed. The duty cycle is the ratio of the length of the on-state to the total duration of a switching cycle.

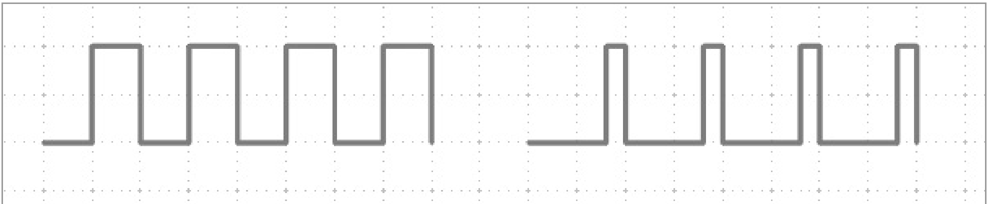


Fig. 6.1: Left: Duty cycle 50 % – right: Duty cycle 20 %.

The smaller the duty ratio, the shorter is the lighting time of the LED within a switching cycle. Thus, the LED seems darker than a permanently illuminated LED.

For the next experiment you will connect an LED via a series resistor to GPIO port 18.

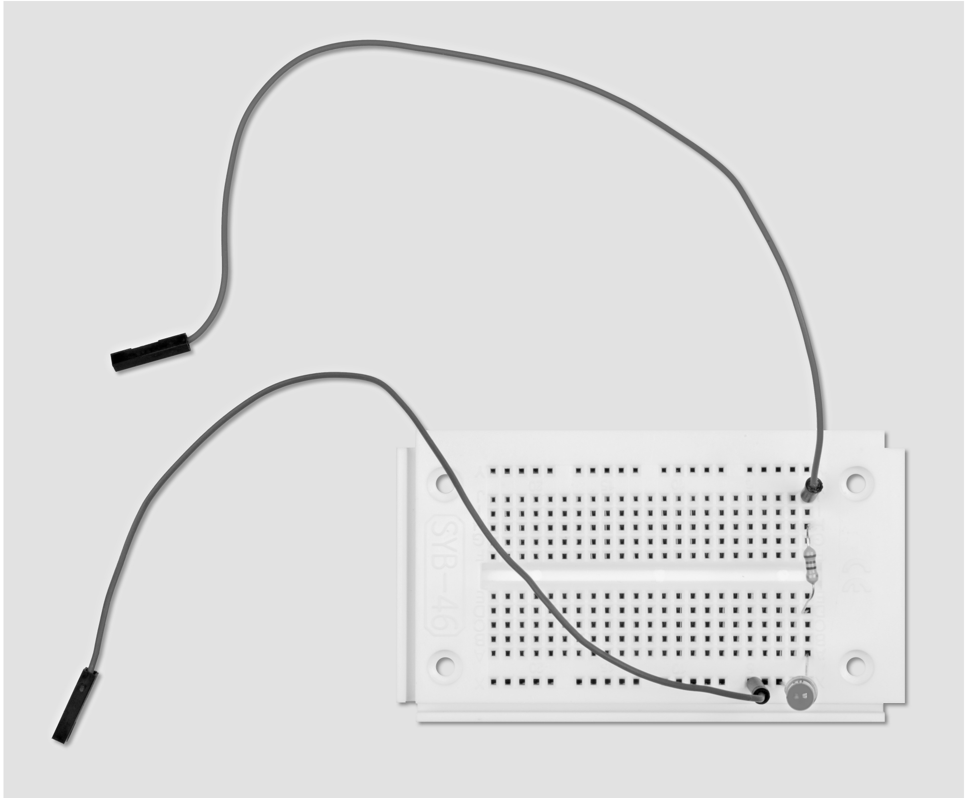


Fig. 6.2: Breadboard assembly with LED.

Components required:
1x breadboard
1x LED yellow
1x 220-ohm series resistor
2x connecting wire

Fig. 6.3: An LED at GPIO port 18.

The programme `leddimmen01.py` dims the LED using cycles of light and dark phases and utilizes the PWM function from the GPIO library. The PWM signal is generated as a separate thread. This allows the use of a dimmed LED (almost) like any normally lit LED in a programmr.

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM); LED = 18
GPIO.setup(i, GPIO.OUT, initial=0)
print (“Ctrl+C exits the program")
p = GPIO.PWM(LED, 50); p.start(0)
try:
    while True:
        for c in range(0, 101, 2):
            p.ChangeDutyCycle(c); time.sleep(0.1)
        for c in range(100, -1, -2):
            p.ChangeDutyCycle(c); time.sleep(0.1)
except KeyboardInterrupt:
    p.stop(); GPIO.cleanup()
```

6.1.1 How does it work?

Parts of this programme may seem familiar, some elements, however not at all. At this point we will delve into object-oriented programming. As already known, we will first import libraries. This time, we define only one variable, `LED`, for GPIO port 18, this is initialized as output.

`print("Ctrl+C exits the program")` Since this programme also runs with a `try ... except` construct and has to be stopped by the user, a certain information will be displayed.

`p=GPIO.PWM (LED, 50)` The function `GPIO.PWM ()` from the GPIO library is crucial for the output of PWM signals. This function requires two parameters, the GPIO port and the frequency of the PWM signal. In our case, the variable `LED` will define the GPIO port, the frequency is 50 hertz (oscillations per second).

50 hertz is the ideal frequency for PWM

The human eye does not detect light changes which are faster than 20 Hertz. The frequency of the oscillations of alternating current in the European electric power grid is 50 hertz. Many flashing lighting bodies using this frequency are not detected by the eye. If an LED would flash in more than 20 Hertz, but less than 50 Hertz, it could cause interferences with other sources of light, whereby the dimming light effect would seem no longer consistent.

`GPIO.PWM()` generates a so-called object which will be stored in the variable `p`. Such objects are much more than just simple variables. Objects can have different properties and can be influenced by so-called methods. Methods are separated by a period, and are expressed directly after the object name.

`p.start(0)` The method `start()` starts the generation of the PWM signal. A duty cycle needs to be specified too. In our case, the duty cycle is `0`, the LED is therefore always off. The infinity loop will start now. It has two loops embedded directly in succession that will alternate the LED light and dark levels.

`for c in range(0, 101, 2):` The loop counts in steps from`2` to `0` until`100`. At the end of a`for` loop, the value which is not processed, in our case this is `101`, is always given.

`p.ChangeDutyCycle(c)` In each loop run the method `ChangeDutyCycle` sets the duty cycle of the PWM object to the value of the loop counter, thus each time increasing by 2% until in the last cycle it stands at 100%, and the LED shines in full brightness.

`time.sleep(0.1)` Each loop run has a latency of 0.1 seconds, before the next iteration increases the duty cycle by 2%.

```
        for c in range(100, -1, -2):
            p.ChangeDutyCycle(c); time.sleep(0.1)
```

After the LED has achieved its full brightness, a second loop regulates the intensity in the same way but reverse. This loop counts down from `100` in increments of `-2`. This cycle iterated until a user stops it with the key combination `Crtl`+`C`.

```
except KeyboardInterrupt:
    p.stop(); GPIO.cleanup()
```

The `KeyboardInterrupt` triggers now also the method `stop()`of the PWM object. This method stops the generation of a PWM signal. Then the GPIO ports are reset as already known from past programmes.

6.1.2 Dimming two LEDs independently of one another

A programme time in the Python script is not required for programming the PWM signal, thus several LEDs can be dimmed independently from one another, as the next experiment shows. Connect an additional LED via a series resistor to GPIO port 25.

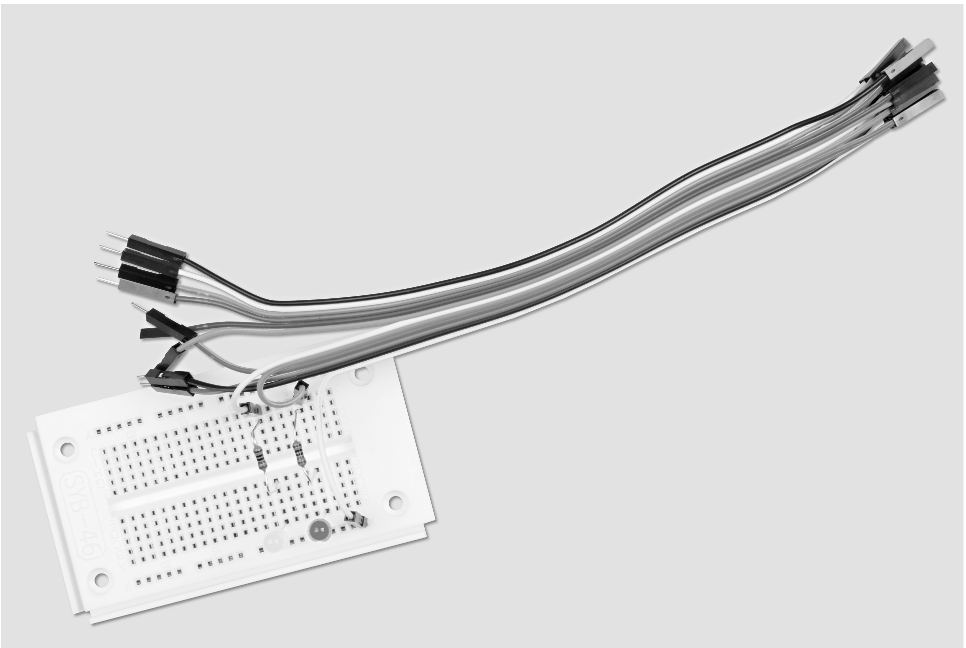


Fig. 6.4: Breadboard assembly to dim two LEDs

Components required:
1x breadboard
1x LED yellow
1x LED red
2x 220-ohm series resistor
3x connecting wire

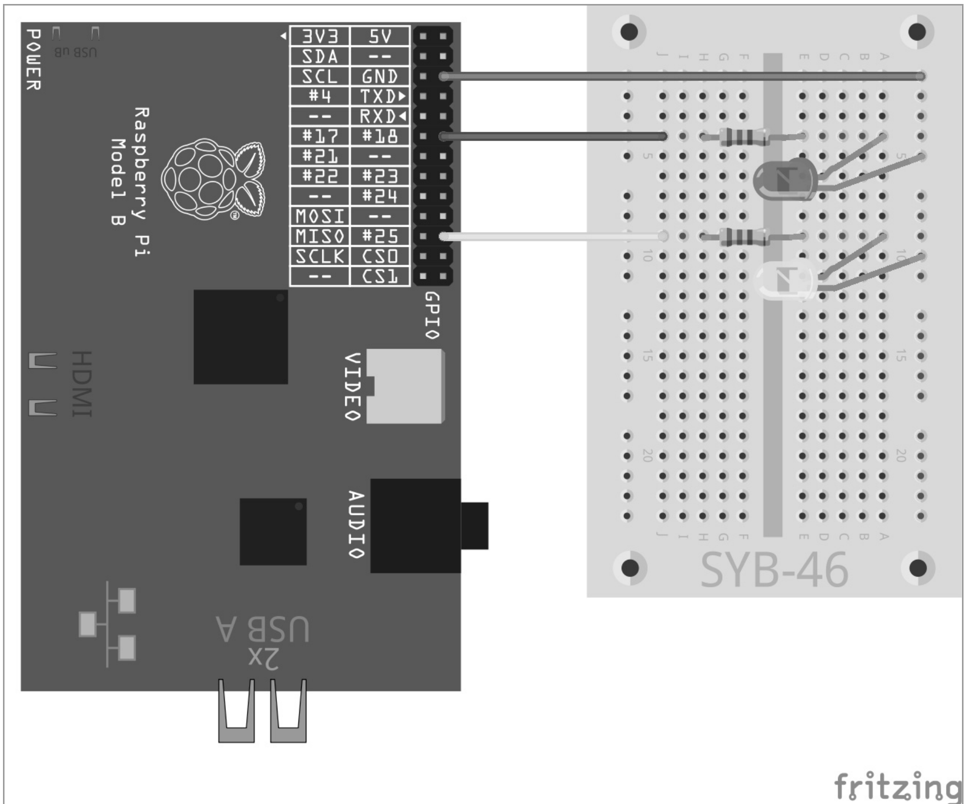


Fig. 6.5: A second LED at GPIO port 25.

The programme `leddimmen02.py` dims one of the LED using cycles of light and dark phases, while the other LED together with the first LED light gets brighter, but in another cycle it does not get darker, instead its brightness intensifies starting from 0 and it is also flickers quickly.

```
import RPi.GPIO as GPIO
import time
```

```
GPIO.setmode(GPIO.BCM); LED = [18,25]
GPIO.setup(LED[0], GPIO.OUT); GPIO.setup(LED[1], GPIO.OUT)

print ("Ctrl+C exits the program")

p = GPIO.PWM(LED[0], 50); q = GPIO.PWM(LED[1], 50)
p.start(0)
p.start(0)

try:
    while True:
        for c in range(0, 101, 2):
            p.ChangeDutyCycle(c); q.ChangeDutyCycle(c)
            time.sleep(0.1)
        q.ChangeFrequency(10)
        for c in range(0, 101, 2):
            p.ChangeDutyCycle(100-c); q.ChangeDutyCycle(c)
            time.sleep(0.1)
        q.ChangeFrequency(50)

except KeyboardInterrupt:
    p.stop(); GPIO.cleanup()
```

6.1.3 How does it work?

The basic structure of the program corresponds to the previous experiment with a few small extensions.

```
LED = [18,25]; GPIO.setup(LED[0], GPIO.OUT); GPIO.setup(LED[1], GPIO.OUT)
```

Instead of a variable for the GPIO port we will define a list of two variables is now defined, and by that two GPIO ports, `18` and `25`, are initialized as the outputs for LEDs.

```
p = GPIO.PWM(LED[0], 50); q = GPIO.PWM(LED[1], 50); p.start(0); q.start(0)
```

Then the two objects `p` and `q` are created which generate the PWM signals for the two LEDs, each with 50 Hertz.

```
        for c in range(0, 101, 2):
            p.ChangeDutyCycle(c); q.ChangeDutyCycle(c)
            time.sleep(0.1)
```

The duty cycles of both PWM objects are simultaneously increased step by step in the first loop. The two LEDs have the same behavior in this phase.

`q.Change Frequency(10)` At the end of this loop, when both LEDs have achieved full brightness, the frequency of the PWM signal of the second LED is reduced to 10 Hertz by the method `Change Frequency()`. This frequency is still detected by the human eye as blinking.

```
        for c in range(0, 101, 2):
            p.ChangeDutyCycle(100-c); q.ChangeDutyCycle(c)

            time.sleep(0.1)
```

The second loop is starting now, for reasons of clarity it will count in an ascending order this time. The corresponding values for the duty cycle of each iteration are calculated for the first LED from the PWM object which in this cycle is dimmed step by step. The duty cycle for the second LED is simply incremented again by the PWM object `q`. The blinking effect is caused by the modified frequency.

`q.ChangeFrequency(50)` At the end of the second loop, the frequency of this LED is reset to 50 Hertz, so that in the next cycle it slowly intensifies in brightness just like the first LED.

7 Indicator with LEDs for free space on the memory card

Memory cards as hard drives are always full far too quickly. What is needed is a simple optical status indicator, so that you can always see at a glance when space on the memory is running low. This can be perfectly implemented on the Raspberry Pi using three LEDs. Functions of the operating system are used here, which are queried via Python.

Fig. 7.1: Of course, it is also possible to show the free memory space directly in the File Manager on the Raspberry Pi.

We will take the three LEDs which we used for the traffic lights circuit which can feature different color combinations for indicating the free space on the memory card.

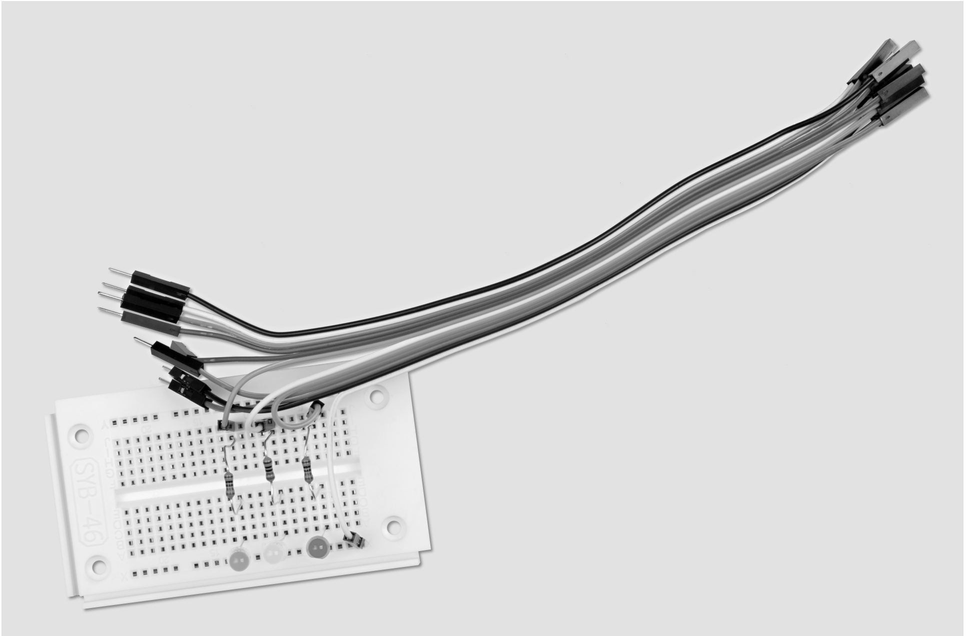


Fig. 7.2: Breadboard assembly for indicator of the free space on a memory card.

Components required:
1x breadboard
1x LED red
1x LED yellow
1x LED green
3x 220-ohm series resistor
4x connecting wire

Fig. 7.3: Three LEDs will show the free space on the memory card.

The programme `speicheranzeige.py` provides different LED indications depending on the free space on the memory card:

Free space	LED display
< 1 MB	Red
1 MB to 10 MB	Red-yellow
10 MB to 100 MB	Yellow
100 MB to 500 MB	Yellow-green
> 500 MB	Green

Tab. 7.1: This is how the memory status is displayed.

```
import RPi.GPIO as GPIO
import time
import os

g1 = 1; g2 = 10; g3 = 100; g4 = 500

GPIO.setmode(GPIO.BCM)
LED = [4,18,23]
for i in range(3):
    GPIO.setup(LED[i], GPIO.OUT, initial=False)

print (“Ctrl+C exits the programme")

try:
  while True:
    s = os.statvfs('/')
    f = s.f_bsize * s.f_bavail / 1000000

    if f < g1:
      x = "100"
    elif f < g2:
      x = "110"
    elif f < g3:
      x = "010"
    elif f < g4:
      x = "011"
    else:
      x = "001"

    for i in range(3):
      GPIO.output(LED[i], int(x[i]))
    time.sleep(1.0)

except KeyboardInterrupt:
  GPIO.cleanup()
```

When you run the programme the three LEDs will constantly show the free space on the memory card. Try it out by copying large files over the network to the memory card and delete them. The indicator updates automatically.

7.1.1 How does it work?

The programme uses the Python module `os` to calculate the free space, which provides the basic functions of the operating system.

`import os` The module `os` as any other module needs to be imported at the beginning of the programme.

`g1 = 1; g2 = 10; g3 = 100; g4 = 500` Those command lines define the limits of the free space areas, where the indication is supposed to switch. The programme uses megabytes and not bytes, as such figures are better comprehensible, to keep it simple, so to say. You can modify the limits at any time; however, the four values must be arranged in ascending order according to size.

```
GPIO.setmode(GPIO.BCM)
LED = [4,18,23]
for i in range(3):
    GPIO.setup(LED[i], GPIO.OUT, initial=False)
```

A list defines the GPIO port numbers of the three LEDs. The loop initializes the three GPIO ports as outputs and resets all LEDs to its off-state.

Also in this experiment, we use the construct `try...except`and an infinity loop to automatically run the programme over and over until it stopped by the user with `Ctrl`+`C`]. Now the really interesting functions follow which will access the operating system and check for the free space.

`s = os.statvfs('/')` The statistics module `os.statvfs()` from the library `os` provides various statistical information about the file system, which are rewritten as an object in the variable `s` at each loop run within the infinity loop.

`f = s.f_bsize * s.f_bavail / 1048576` Now method `s.f_bsize` delivers the size of the memory space block in byte. `s.f_bavail` shows the number of free blocks. The product of the two values is therefore the number of free bytes, divided here by 1,048,576 to get the number of free megabytes. The result is stored in the variable `f`.

```
    if f < g1:
      x = "100"
```

The free space is smaller than the first limit value (1 MB), the string sequence `x`, which indicates the pattern of turned on LEDs, is set to `"100"`. The first, the red LED, will light up. The pattern is a simple string containing the digits 0 and 1.

```
    elif f < g2:
      x = "110"
    elif f < g3:
      x = "010"
    elif f < g4:
      x = "011"
```

By means of the `elif` queries the other limits are queried and the LED pattern is set accordingly, if the first query is not valid, that is, there is more than 1 MB of free disk space.

```
    else:

      x = "001"
```

If none of the queries are true, that is, there is more free disk space than the highest limit, the LED pattern is set to `"001"`. The last, the green LED should light up.

```
    for i in range(3):

      GPIO.output(LED[i], int(x[i]))
```

A loop defines the GPIO output values for the three LEDs. The numerical value of the digit in question from the string`0` or `1`is assigned to all LEDs in succession. The values `0` und `1`can be used like `False` or `True` to

activate or deactivate GPIO outputs. The function `int()` calculates a character's numeric value. The character is read out from a certain position of the pattern string via the loop counter `i`.

`time.sleep (1.0)` The program waits 1 second until the next iteration. To save performance, you can also prolong the latency until the calculation of free space is repeated.

At this point, the `while...True` loop restarts. If the user in the meantime however, has pressed the shortkey [Ctrl]+[C] the `KeyboardInterrupt` is triggered and the loop exits. After that, the GPIO ports are closed and shutting the LEDs off.

8 Graphical dice

An addicting game needs graphics and not just text output as was the case in the times of the very first DOS computer. The library PyGamehas predefined functions and objects to display graphics and programming for games. Thus no need to invent everything from scratch.

Many games require a dice, but often there is none at hand. The next programming sample shows how easy it is to use the Raspberry Pi with the help of Python and PyGame as a dice:

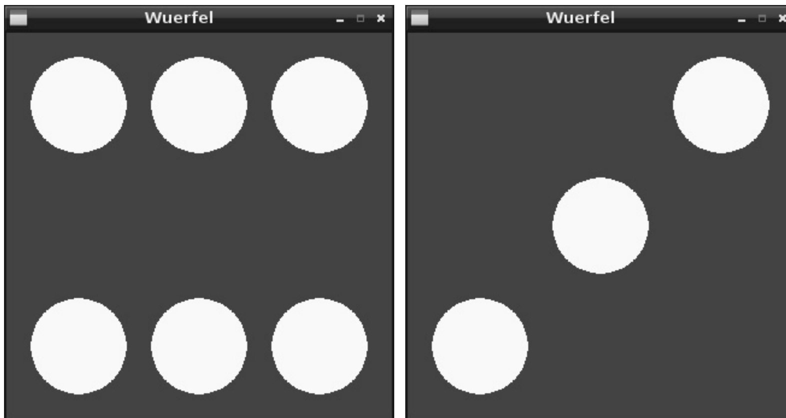


Fig. 8.1: The Raspberry Pi as dice.

The dice should be as simple as possible and be operated by using only one key while the randomly diced result should be graphically represented as a "real" dice. The following program `wuerfel.py` simulate such a dice on the screen.

```
# -*- coding: utf-8 -*-
import pygame, sys, random
from pygame.locals import *
pygame.init()

FELD = pygame.display.set_mode((320, 320))
```

```
pygame.display.set_caption(“Dicel")

BlUE = (0, 0, 255); WHITE = (255, 255, 255)
P1 = ((160, 160)); P2 = ((60, 60)); P3 = ((160, 60));
P4 = ((260, 60))
P5 = ((60, 260)); P6 = ((160, 260)); P7 = ((260, 260))
mainloop = True

print “Press any key to roll the dice, [Esc]
exits the game”

while mainloop:
    for event in pygame.event.get():
        if event.type == QUIT or (event.type == KEYUP
and event.key == K_ESCAPE):
            mainloop = False
        if event.type == KEYDOWN:
            FELD.fill(BUE)
            NUMBER = random.randrange (1, 7); print NUMBER
            if NUMBER == 1:
                pygame.draw.circle(FIELD, WHITE, P1, 40)
            if NUMBER == 2:
                pygame.draw.circle(FIELD, WHITE, P2, 40)
                pygame.draw.circle(FIELD, WHITE, P7, 40)
            if NUMBER == 3:
                pygame.draw.circle(FIELD, WHITE, P1, 40)
                pygame.draw.circle(FIELD, WHITE, P4, 40)
                pygame.draw.circle(FIELD, WHITE, P5, 40)
            if NUMBER == 4:
                pygame.draw.circle(FIELD, WHITE, P2, 40)
                pygame.draw.circle(FIELD, WHITE, P4, 40)
                pygame.draw.circle(FIELD, WHITE, P5, 40)
                pygame.draw.circle(FIELD, WHITE, P7, 40)
            if NUMBER == 5:
                pygame.draw.circle(FIELD, WHITE, P1, 40)
                pygame.draw.circle(FIELD, WHITE, P2, 40)
                pygame.draw.circle(FIELD, WHITE, P4, 40)
                pygame.draw.circle(FIELD, WHITE, P5, 40)
                pygame.draw.circle(FIELD, WHITE, P7, 40)
            if NUMBER == 6:
                pygame.draw.circle(FIELD, WHITE, P2, 40)
                pygame.draw.circle(FIELD, WHITE, P3, 40)
                pygame.draw.circle(FIELD, WHITE, P4, 40)
                pygame.draw.circle(FIELD, WHITE, P5, 40)
                pygame.draw.circle(FIELD, WHITE, P6, 40)
                pygame.draw.circle(FIELD, WHITE, P7, 40)
    pygame.display.update()
pygame.quit()
```

Runs without sudo

This programme does not require any GPIO ports, and also works without superuser privileges. You can simply start the Python IDE using the desktop icon *IDLE*.

8.1.1 How does it work?

This programmr displays many new features, especially for graphics output by the PyGame library which of course is not only used for games only for games but also for any other graphics on the screen.

```
import pygame, sys, random
from pygame.locals import *
pygame.init()
```

These three lines of code stand at the beginning of almost every programme that uses PyGame. Besides the aforementioned module `random` to generate random numbers, the module `pygame` itself and the module `sys` are loaded, as it contains important, by PyGame required system functions such as the opening and closing of windows. All functions of the PyGame library are imported, and then the actual PyGame module is initialized.

```
FIELD = pygame.display.set_mode((320, 320))
```

This is an important function in any programme that uses a graphical output, it defines a drawing area, a so-called surface, which in our example has the size 320 x 320 pixels and is named `FIELD`. Note the notation in double brackets; these are basically used for graphical screen coordinates. Such a surface is displayed in a new window on-screen.

```
pygame.display.set_caption(“Dice")
```

This line will enter the window name.

```
BlUE = (0, 0, 255); WHITE = (255, 255, 255)
```

These commando lines define the two colors used, blue and white. You could also directly give the color values each time in the programme, but this is not very helpful in terms of clarity.

Representation of colours on-screen.
Colours in Python and in most other programming languages are defined by three numbers between 0 and 255 that define the three colour components, red, green and blue Monitors use additive colour mixing, where all three colour combined together in full saturation result in White.

```
P1 = ((160, 160)); P2 = ((60, 60)); P3 = ((160, 60)); P4 = ((260, 60)); P5 = ((60,
260)); P6 = ((160, 260)); P7 = ((260, 260))
```

These lines define the midpoints of the dice. On the 320 x 320 pixels large character field, the three axes of the dots have the coordinates `60`, `160` and `260` respectively.

The coordinate system for computer graphics
Each dot in a window or on a surface object is denoted by an x and y coordinate. The zero point of the coordinate system is on the top left and not as we have learnt in school, bottom left. Just as reading a text from top left to bottom right, also the x-axis extends from left to right, the y-axis from top to bottom.

The seven points P1 to P7mark in the graphic the midpoints of the dice. Each dot of a dice has a radius of 40 pixels. At 80 pixels center distance, 20 pixels remain between the dots of the dice and 20 pixels to the window edges.

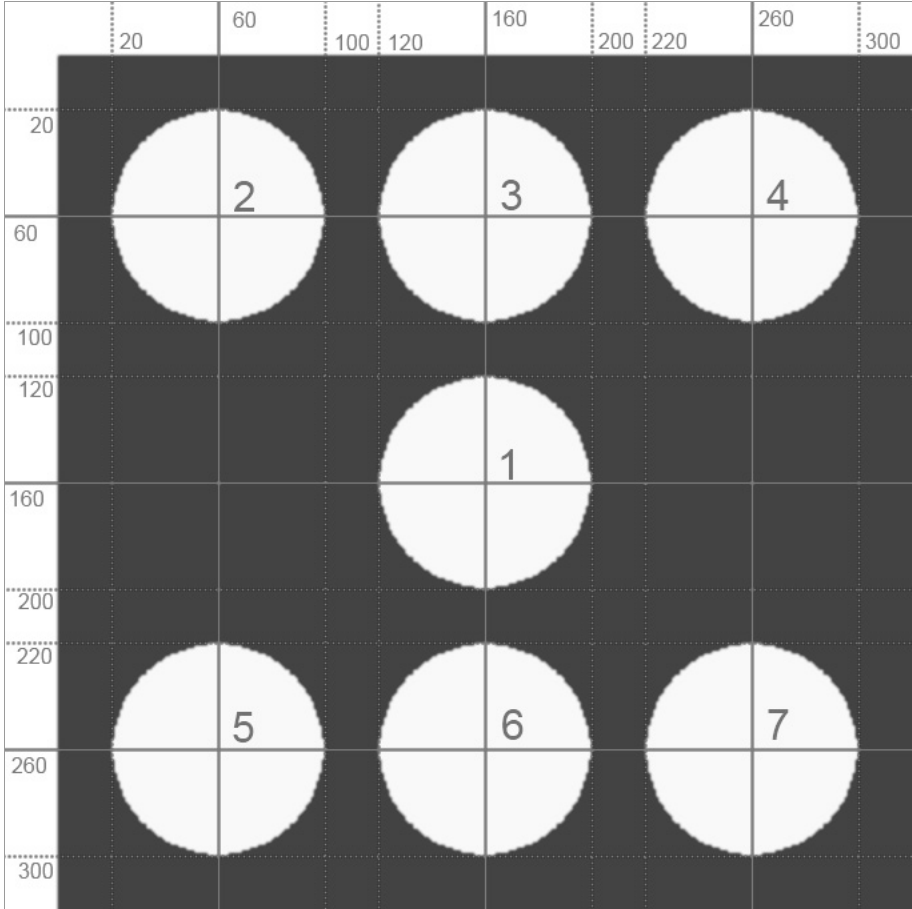


Fig. 8.2: The dots of the dice and their coordinates.

At this point, a variable that goes by the name mainloop is set to True along with the other variables which we will need for the main loop of the game at a later stage.

mainloop = True We have created the basics, now we can start with the real game.

```
print “Press any key to roll the dice, [Esc]
exits the game”
```

This line tells the user, what to do. Each time you press any key on the keyboard rolls the dice. `print` always writes into the Python Shell window, not in the new graphical window.

`while mainloop`: Now the main loop of the game starts. In many games an infinite loop is used which is repeated over and over again and constantly queries any user activity. Somewhere in the loop an abort statement must be defined, which ensures that the game can exit.

We use the variable `mainloop` here which only adopts the two Boolean values `True` und `False` (true and false, on and off). It starts with `True` and is queried with each loop run. If it adopts the value `False` during the loop, the loop is aborted before the next run.

`for event in pygame.event.get()`: This line reads the most recent user activity and saves it as `event`. The game has only two types of game-relevant user activities: The user presses a key and thus rolls the dice, or the user wants to end the game.

```
        if event.type == QUIT or (event.type == KEYUP
and event.key == K_ESCAPE):
            mainloop = False
```

There are two ways to exit the game: Either you can click the X icon in the upper right corner on the window or you can press the Esc key. By clicking on the x icon the operation system delivers the `event.type == QUIT`. If you press and release a key that is the `event.type == KEYUP`. In this case, the pressed key is also stored in `event.key`.

The described `if` statement verifies whether the user wants to close the window, or (`or`) has pressed and released a key and (`and`) whether this is the key with the internally used name `K_ESCAPE`. If this is the case, the variable `mainloop` is set to `False` that will exit the main loop of the game prior the next run.

`if event.type == KEYDOWN`: The second type of user activity that reoccurs more than once again during the game, is the user presses a key. It is not important which key, yet with the exception of the key Esc. As soon as a key is pressed `KEYDOWN`), an important part of the program is actuated, which generates and displays the dice result.

`FIELD.fill(BLUE)` First the surface-object, the actually window of the application named `FIELD` is filled with the color defined as `BLUE` to paint over the dice result scored.

```
            NUMBER = random.randrange (1, 7)
```

The random function `random` now generates a random number between `1` and `6` and saves it in the variable `NUMBER`.

`print ZAHL` This line writes the score of the dice roll into the Python Shell window. You may also omit this line if you rather want to do without the text-based output.

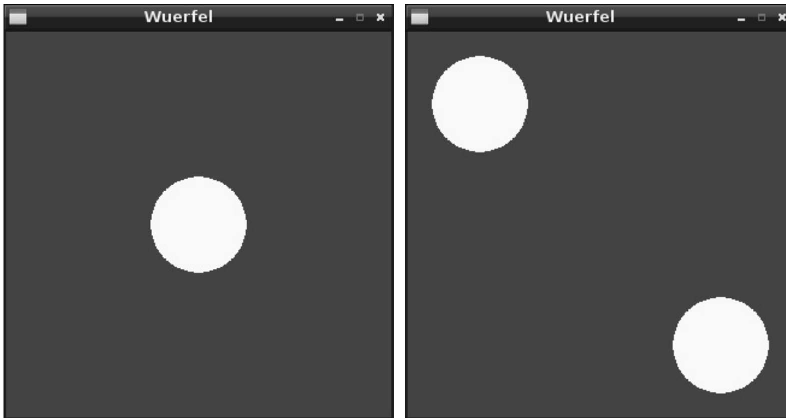
```
            if NUMBER == 1:
                pygame.draw.circle(FIELD, WHITE, P1, 40)
```

This is followed by six queries all performed in the same way. If the random number thrown has a certain value, one to six dice spot will be marked accordingly. The function employed is `pygame.draw.circle()` and requires four to five parameters:

- *Surface* specifies the drawing canvas, which is `FIELD` in the example.
- *Colour* specifies the colour of the circle, in the example, the previously defined colour `WHITE`.
- *Midpopint* specifies the centre of the circle.
- *Radius* specifies the radius of the circle.
- *Thickness* is the line width of the circle line. If this parameter is omitted or set to `0`, the circle is filled.

If one of the conditions is met, the dice spots are first saved to the virtual canvas.

`pygame.display.update()` This is the line at the end of the loop that updates the graphic on the screen. Now you can in fact see the dots on the dice.



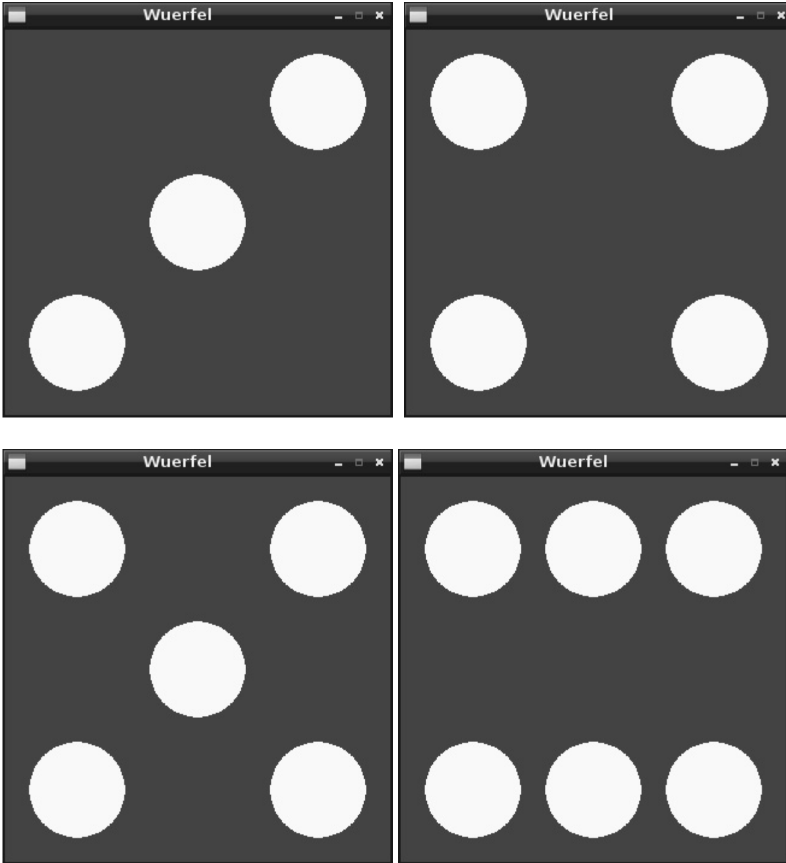


Fig. 8.3: Six possible dice scores.

The loop restarts immediately and waits for the user's keystroke. If, however, `mainloop` is set to `False` during the loop because the user wants to quit the game, the loop will not rerun and the following line is executed:

`pygame.quit()` This line terminates the PyGame module and closes the graphical window and then the whole programme.

9 Analogue clock on-screen

The digital display of time on computers, that we are accustomed to today has come into fashion only in the 70's. For centuries time was displayed analogously with pointers on a dial. The boom in digital watches has declined somewhat in recent years. We have come to realize that analogue clocks are better readable and can be seen and read under poor weather conditions or over long distances, such as in railway stations, much faster and clearer. The human eye detects a graphic faster than it detects numbers or letters. The image of an analogue clock brands into short-term memory, so that even if the image is incomplete or blurred, the brain can translate it properly. On the other hand, if a digital clock is not seen clearly, we cannot reliably gather the displayed time.

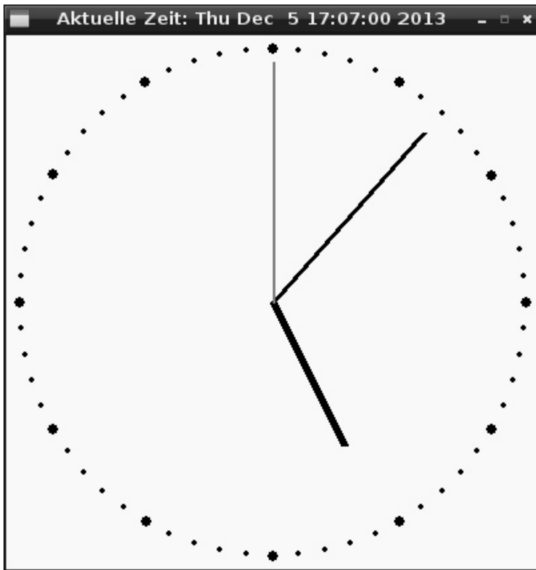


Fig. 9.1: Analogue clock, programmed with PyGame.

This programme is designed to show how to programme a clock, while at the same time it also illustrates the fundamental principles for the representation of analogue displays, which are not only used for clocks, but also to represent various measured values or statistical data.

Three clock hands circle round the midpoint and indicate the hour, minute, and second. Up in the window title a digital time display is also counting.

The programme `uhr01.py` shows the depicted analogue clock on the screen:

```
import pygame, time
from pygame.locals import *
from math import sin, cos, radians
pygame.init()
RED = (255, 0, 0); WHITE = (255, 255, 255); BLACK = (0, 0, 0)
```

```
FIELD = pygame.display.set_mode((400, 400))
FIELD.fill(WHITE)
MX = 200; MY = 200; MP = ((MX, MY))
def point(A, W):
    w1 = radians(W * 6 - 90); x1 = int(MX + A * cos(w1))
    y1 = int(MY + A * sin(w1)); return((x1, y1))
for i in range(60):
    pygame.draw.circle(FIELD, BLACK, point(190, i), 2)
for i in range(12):
    pygame.draw.circle(FIELD,BLACK, point(190, i * 5), 4)
mainloop = True; s1 = 0
while mainloop:
    time = time.localtime()
    s = time.tm_sec; m = time.tm_min; h = time.tm_hour
    if h > 12:
        h = h - 12
    hm = (h + m / 60.0) * 5
    if s1 <> s:
        pygame.draw.circle(FIELD, WHITE, MP, 182)
        pygame.draw.line(FIELD, BLACK, MP, point(120, hm), 6)
        pygame.draw.line(FIELD, BLACK, MP, point(170, m), 4)
        pygame.draw.line(FIELD, RED, MP, point(180, s), 2)
        s1 = s
        pygame.display.set_caption("Current time: " +
time.asctime())
        pygame.display.update()
        for event in pygame.event.get():
            if event.type == QUIT or (event.type ==
KEYUP and event.key == K_ESCAPE):
                mainloop = False
pygame.quit()
```

9.1.1 How does it work?

This program shows other features of the PyGame library and the `time` library and simple trigonometric functions that are used for the presentation of analogue displays.

```
import pygame, time
from pygame.locals import *
from math import sin, cos, radians
pygame.init()
```

As in the last programme, the PyGame library is first initialized. In addition, the `time` library is imported to establish the time and also imported are three functions from the very extensive `math` library.

```
RED = (255, 0, 0); WHITE = (255, 255, 255); BLACK = (0, 0, 0)
```

The three colours used in the graphic are saved to three variables.

```
FIELD = pygame.display.set_mode((400, 400)); FELD.fill(WHITE)
```

A window of the size 400 x 400-pixel is opened and filled with white.

```
MX = 200; MY = 200; MP = ((MX, MY))
```

Three variables determine the coordinates of the midpoint; all other graphical elements, the dial and the hands are in alignment with the midpoint. The variables MX and MY contain the x- and y-coordinates of the midpoint, the variable MP the midpoint point as dot, as used for graphics functions.

Next comes the definition of an important function that calculates dots based on the distance to the centre and angle points in the coordinate system. This function is called up several times by the programme to represent both, the dial and the clock hands.

```
def point(A, W):
    w1 = radians(W * 6 - 90); x1 = int(MX + A * cos(w1))
    y1 = int(MY + A * sin(w1)); return((x1, y1))
```

The function uses two parameters: A is the distance of the desired dot from the centre, W is the angle relative to the centre. To keep in the case the representation of the clock simple, we will calculate the clockwise angle relative to the vertical 12-clock direction. The angle is expressed in minutes and not in degree; 1/60 of a full circle is passed on to the function. Such assumptions save a lot of intermediate calculations

Python measures, just like most programming languages, angle units in radians and not in degrees. The function radian() from the library math converts degrees to radians. For this, the angle indication used in the program is multiplied by 6 minutes to arrive at the degree, and then 90 degrees will be subtracted, so that the 0-direction is pointing vertically upward, as the 0 minute at every hour. This angle converted to radians is stored for further calculations within the function in the variable w1.

Displaying a clock relies on the trigonometric functions sine and cosine. Thus from the angle of a point in radians relative to the midpoint w1, the coordinates in the rectangular coordinate system x1 and y1) will be determined. The coordinates of the midpoint are taken from the variables MX and MY which are defined outside the function and are globally applied. The distance of the dot from the midpoint will be transferred to the parameter A of the function. The function int() calculates from the result the integer value (integer), because pixel coordinates are only specified as an integer.

The return value of the function is a geometric point with the calculated coordinates x1 and y1, which is embraced by double brackets like all points are.

Following the definition of this function, the dial is drawn.

```
for i in range(60):
    pygame.draw.circle(FIELD, BLACK, point(190, i), 2)
```

A loop draws now the 60 minute dots on a circle, one after the other. All points are identified by the function point(). They have the same distance from the centre which is 190 pixels and is within the four quadrants exactly 10 pixels away from the window border. The points have a radius of 2pixels.

```
for i in range(12):
    pygame.draw.circle(FIELD,BLACK, point(190, i * 5), 4)
```

A second loop draws 12 larger circles that will mark the hours on the dial. These have a radius of 4 pixels, are drawn simply on the existing circles and overlap them completely. The hour symbols follow in

succession at an angular distance of five minute units; that is achieved by multiplying the angle data by 5 which is passed on to the function.

mainloop = True; s1 = 0 Before the main loop of the program starts, two auxiliary variables are defined, which are needed in the following process. As in the last program sample, mainloop indicates whether the loop should continue running or if the user wants to exit the programme. s1 saves the last second that was displayed.

```
while mainloop:
    time = time.localtime()
```

Now he main loop of the program starts, which in each run, regardless of how long it lasts, writes the actual time to the object time. The function time.localtime() of the time library is used for this purpose. The result is a data structure that is composed of several individual values.

```
    s = time.tm_sec; m = time.tm_min; h = time.tm_hour
```

The three values, seconds, minutes and hours relevant for the clock are adopted from the structure and written in three variables s, m and h.

```
    if h > 12:
        h = h - 12
```

Analogue clocks display only twelve hours. The function time.localtime() provides all time specifications in 24-hour format. For past midday times 12 hours are simply subtracted.

Time representation for analogue clocks
Depending on the mechanism used, two different displays for analogue clocks are available. In real analogue running clocks the minute hand performs a uniform circular motion; digitally controlled clocks such as station clocks, the minute jumps from full minute to the next full minute. The latter method has the advantage that the time is read accurately read by the minute easily at a glance. Fractions of minutes are usually not important in everyday life. We also use for our clock application the same method. However, the hour hand must perform a uniform circular motion. It would be very odd and confusing, if the hour hand would jump an hour forward at every hour.

hm = (h + m / 60.0) * 5 The variable hm stores the angle of the hour hand in minute units, as applied throughout the programme. Therefore the minutes value is added to the current hour 1/60. Each minute, the hour hand advances 1/60 of an hour. The calculated value is multiplied by 5, because the hour hand advances on the dial five minute units in one hour.

if s1 <> s: The duration of an iteration in the programme is unknown. This means for the analogue clock, that the graphic does not need to update at each loop run; it needs to update however, if the current second is different from the last one drawn. Thus the drawn second will be stored in the variable s1 later on in the programme; the current second is always expressed in the variable s.

If the second has changed in respect to the last one drawn changed, the graphics of the clock is updated by the following statement. If nothing has changed, the graphics is not updated, and the loop restarts with yet another query of the current system time.

```
        pygame.draw.circle(FIELD, WHITE, MP, 182)
```

A white circular area is drawn first, which covers the hand completely. The radius with 182 pixels is slightly larger than the longest pointer, to ensure that no remnants remain. It is by far easier to draw an over-all surface of a circle than to cover the last drawn pointer accurately to the pixel with paint.

```
      pygame.draw.line(FIELD, BLACK, MP, point(120, hm), 6)
```

This line draws the hour hand as a line with a width of `6` pixels starting from the midpoint, `120` pixels long at an anglethat is specified by the variable `hm`. We have not used so far the function `pygame.draw.line()`. The command requires five parameters:

- *Surface* specifies the drawing canvas, which is `FIELD` in the example.
- *Color* specifies the colour of the circle, in the example, the previously defined color `BLACK`.
- *Startpoint* indicates the starting point of the line, in this example it is the midpoint of the clock.
- *Endpoint* indicates the end point of the line, in this example it is the function `point()` calculated from the angle of the hour hand.
- *Thickness* is the line width.

This same function also draws the other two hands of the clock.

```
       pygame.draw.line(FIELD, BLACK, MP, point(170, m), 4)
```

This line draws the hour hand as a line with a width of `4` pixels starting from the midpoint, `170` pixels long at an angle that is specified by the minute value.

```
        pygame.draw.line(FIELD, RED, MP, point(180, s), 2)
```

This line draws the minute hand as a red line with a width of `2` pixels starting from the midpoint, `180` pixels long at an angle, which is specified by the second value.

`s1 = s` the second which is now displayed is stored in the variable `s1` to compare this value with the current second in the next loop run.

```
        pygame.display.set_caption("Current time: " +
time.asctime())
```

This line writes the current time in digital form into the window caption. The function `time.asctime()` of the `time` library is used here, which delivers the time as ready formatted string.

`pygame.display.update()` So far, all the graphic elements have been drawn only virtually. But this line restructures the graphic display genuinely. It is updated simultaneously. Therefore, you can't see any flickering when the individual pointers are drawn in sequence.

```
        for event in pygame.event.get():
            if event.type == QUIT or (event.type ==
KEYUP and event.key == K_ESCAPE):
                mainloop = False
```

We are still within the `if` query, where once per second, the relatively performance-hungry query attempts to detect any system events to verify whether the user wanted to close the window of the clock within the last second or has pressed the `Esc` key. If that is the case, the variable `mainloop` is set to `False` and the loop will not be restarted.

`pygame.quit()` This last line terminates the PyGame module first, so that the graphical window closes and then the entire application.

10 Graphical dialog fields for program control

There is no modern program that requires some interaction with the user and is running in pure text mode. Graphical user interfaces, where you click on buttons, instead of having to enter data via the keyboard are abundant.

Python itself has no graphical user interface for programmes, but we have got several external modules, similar to the PyGame described earlier which are particularly meant for creating graphical interfaces. One such modules is the famous *Tkinter*, which produces the graphical interface *Tk* for Python, and is also of use for various other programming languages.

The structures of the graphical toolkit Tk are slightly different from Python and may look a bit unfamiliar at first. So let's start with a very simple example: An LED is switched on and off by buttons in a dialog box.

Components required:
1x breadboard
1x LED red
1x 220-ohm series resistor
2x connecting wire

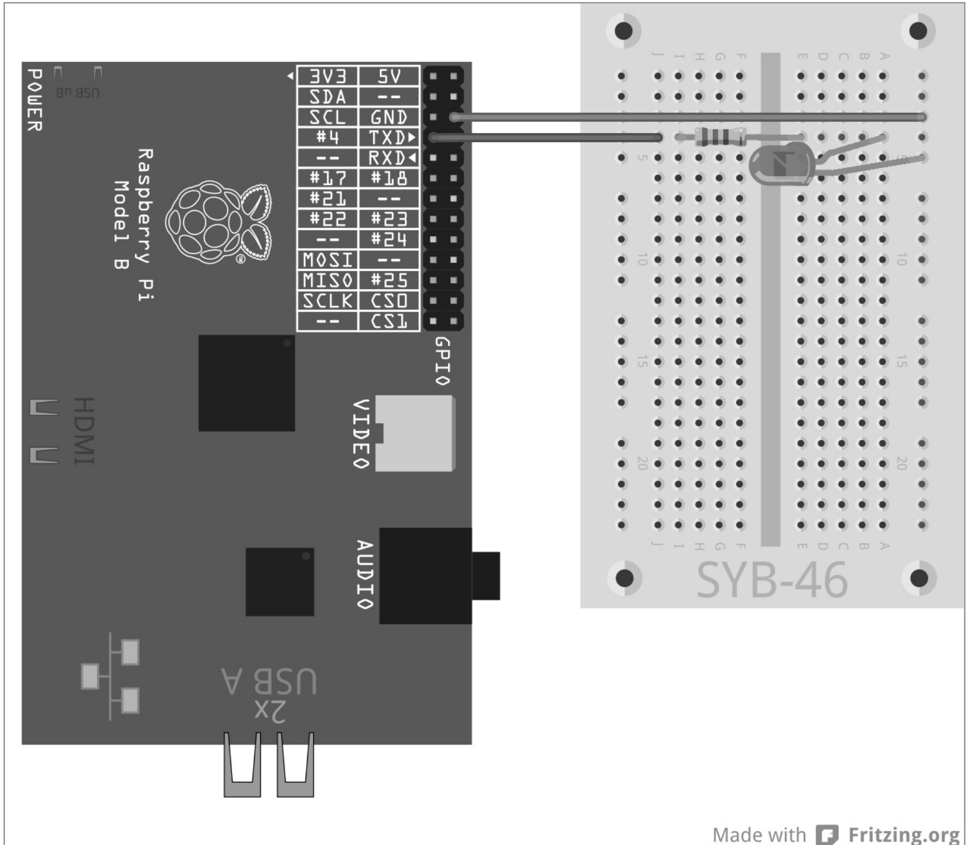


Fig. 10.1: A single LED at GPIO port 4.

Connect an LED via a series resistor to the GPIO port 4. The programme `ledtk01.py` will turn the lights on.

```
import RPi.GPIO as GPIO
from Tkinter import *
LED = 4; GPIO.setmode(GPIO.BCM); GPIO.setup(LED,GPIO.OUT)
def LedOn():
    GPIO.output(LED,True)

def LedOff():
    GPIO.output(LED,True)

root = Tk(); root.title("LED")
Label(root,
      text=”Please click a button to turn the LED on
or off").pack()
Button(root, text=”On", command=LedOn).pack(side=LEFT)
```

```
Button(root, text=”Off", command=LedOff).pack(side=LEFT)
root.mainloop()
GPIO.cleanup()
```

Fig. 10.2: The final dialog box will look like this.

10.1.1 How does it work?

This programme shows the basic functions of the Tkinter library for the building graphical dialogs. The size of the dialog boxes and controls in Tkinter is acquired automatically according to the size that is desired and can also be modified manually later on, if necessary unlike the graphics library PyGame where graphics are constructed accurately pixel by pixel.

```
import RPi.GPIO as GPIO
from Tkinter import *
```

After having imported the GPIO library, we also import the elements of the Tkinter library.

```
LED = 4
GPIO.setmode(GPIO.BCM)
GPIO.setup(LED,GPIO.OUT)
```

The lines are familiar. The GPIO port `4` is defined as output port for an LED and is denoted by the LED variables `LED`.

```
def LedOn():
    GPIO.output(LED,True)
```

The function `LedOn()` which turns the LED on, is now defined.

```
def LedOff():
    GPIO.output(LED,False)
```

A similar function, `LedAus()`, turns the LED off. These two functions are called by the two buttons in the dialog box at a later stage.

Up to here we had plain Python, now we continue with Tk and its peculiarities.

`root = Tk()` Tkinter works with so-called widgets. These are independent screen elemente, in most cases these are to dialog boxes, which in turn contain various elements. Each programme needs a `root` widget, that calls up all the other objects. This `root` widget is always named `Tk()`, it automatically generates a window and also initializes the Tkinter library.

`root.title("LED")` Objects in Tkinter make different methods for different purposes available. The method `title()` in a widget sets the window caption, and in our case, it writes the word `LED` into the title bar of the new window.

Each widget can contain multiple objects that are defined separately. Tkinter knows different types of objects, each of them features different parameters to describe the object's properties. The parameters separated by commas, are embedded in a bracket after the object type. Since this list may become very long, each parameter is usually written on a separate command line, so that all parameters are aligned below each other. Contrary to the indents of loops and queries in Python, these indents of the Tkinter objects are not mandatory.

```
Label (root, text = "Please click a button to turn the LED on
or off").pack()
```

Objects of the type `Label` are plain text in a widget. These can be modified by the program, but do not offer any interaction with the user. The first parameter in any Tkinter object is the name of the parent widget, often of the object's window. In our case, the only window in the program, the `root` widget.

The parameter `text` contains the text that should be shown on the label. At the end of the so-called object definition the extension `.pack()` as the method is added to the so-called packer. This packer structures the object into the dialog box and generates the geometry of the widget.

```
Button(root,
       text=”On",
       command=LedOn).pack(side=LEFT)
```

Objects of the type `Button` are buttons that the user clicks to trigger a specific action. Here too, the parameter `text` contains the text that will be displayed on the button.

The parameter `command` contains a function that is called by the button when clicked. Here no parameters can be passed on, and the function name is specified without brackets. This button calls the function `LedOn()` which turns on the LED.

The method `.pack()` may also contain parameters that determine how an object is arranged within the dialog box. `side=LEFT` means that the button is left-aligned and not centred.

```
Button(root, text=”Off", command=LedOff).pack(side=LEFT)
```

Following the same pattern, a second button is still created, which turns off the LED using the function L`LedOff()`.

Now that all functions and objects are defined, the actual program can begin.

`root.mainloop()` The main program has only this single line. It starts the main loop `mainloop()`, a method of the `root` widget. This programme loop waits for the user to activate a widget which will thus trigger an action.

There is no need in Tkinter to specifically define the x-icon at the top right that closes the window. If the user closes the main window `root`, the main loop `mainloop ()`is terminated automatically.

`GPIO.cleanup()` The programme continues running until it comes to the last line and with it closes the open GPIO ports.

After starting the programme a dialog box appears on-screen. Click on the button *On* to turn on the LED, then on *Off*to turn it off.

10.2 Controlling chaser by a graphical user interface

The Python library Tkinter provides far more control elements than just these simple buttons. With radio buttons you can build menus, where the user can select one of several options.

> What are radio buttons?
> The name "Radio Button" is actually derived from those old radios that featured radio station buttons for pre-set radio stations. Whenever you hit one of those keys, the last one pressed automatically jumped out using a refined mechanism. Radio buttons feature the same behaviour. If the user selects an option, the others will automatically be turned off.

The next experiment shows different LED patterns similar to those of the experiment "Colorful LED patterns and chaser lights". However, unlike there, the user is not required to enter numbers into the screen of the text input, instead chooses comfortably from a simple list of desired patterns.

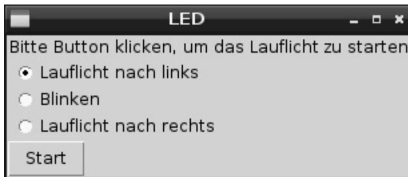


Fig. 10.3: The dialog box offers three LED patterns to choose from.

The assembly of the circuit is the same as in the experiment "Colourful LED patterns and chaser lights".

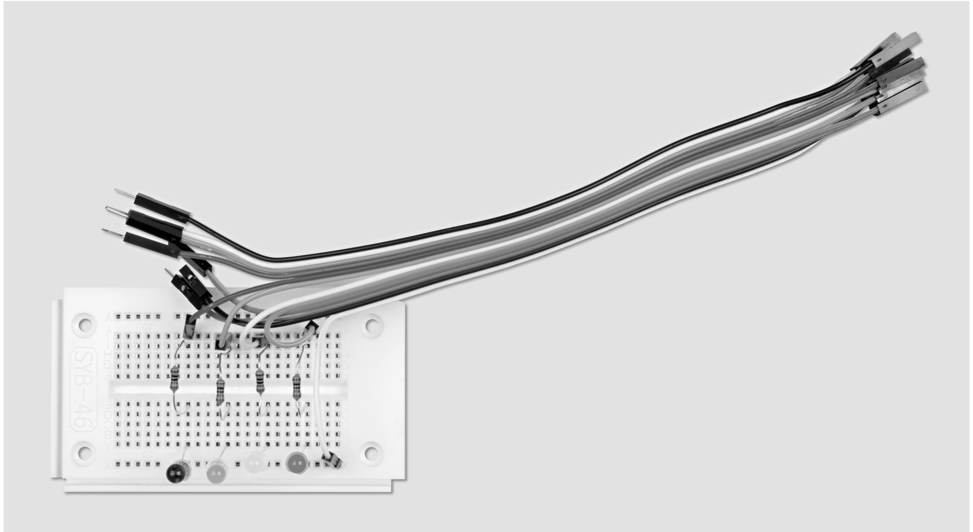


Fig. 10.4: Breadboard assembly for experiment 10.2

Components required:
1x breadboard
1x LED red
1x LED yellow
1x LED green
1x LED blue
4x 220-ohm series resistor
5x connecting wire

Fig. 10.5: Four LEDs are flashing using different patterns.

The programme `ledtk02.py` is built on the previous programme, but has been expanded to include the radio buttons and the functions of the LED chasing lights and flashing patterns.

```
import RPi.GPIO as GPIO
import time
from Tkinter import *
GPIO.setmode(GPIO.BCM)
LED = [4,18,23,24]
for i in LED:
    GPIO.setup(i, GPIO.OUT, initial=0)
w = 5; t = 0.2
pattern = [
    (“Chasing to the left",1),
    (“Flashing",2),
    (“Chasing to the right”,3),
```

```
    ]
root = Tk(); root.title("LED"); v = IntVar(); v.set(1)
def LedOn():
    e = v.get()
    if e == 1:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[j], True)
                time.sleep(t)
                GPIO.output(LED[j], False)
    elif e == 2:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[j], True)
            time.sleep(t)
            for j in range(4):
                GPIO.output(LED[j], False)
            time.sleep(t)
    else:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[3-j], True); time.sleep(t)
                GPIO.output(LED[3-j], False)
Label(root,
      text="Please click a button to start the chaser
").pack()
for txt, m in pattern:
    Radiobutton(root, text = txt,
                variable = v, value = m).pack(anchor=W)
Button(root, text="Start", command=LedOn).pack(side=LEFT)
root.mainloop()
GPIO.cleanup()
```

10.2.1 How does it work?

To start with, we will first import libraries. In addition to the last program, the `time` library is also required and needed for the waiting times for the LED flashing effects.

```
GPIO.setmode(GPIO.BCM); LED = [4,18,23,24]
for i in LED:
    GPIO.setup(i, GPIO.OUT, initial=0)
```

A list for the four LEDs is then defined. The corresponding GPIO ports are defined as outputs and set to 0, so that all LEDs are turned initially off.

```
w = 5; t = 0.2
```

Two variables define two values in the programme: the number of repetitions `w` of a pattern w and the flashing time `t`. Both values could also be entered as they happen to occur in the programme. However, as they are defined in only one place, they are much more easily adjusted that way.

```
pattern = [
    ("Chasing to the left",1), ("Flashing",2), ("Chasing to the right",3)
    ]
```

The texts of the three patterns of choice are defined in a separate list form. Each of the three list elements consists of a pair of values, each consisting of the text displayed and a numeric value, which is returned later when the respective radio button is selected.

```
root = Tk(); root.title("LED")
```

The initialization of the `root` widget corresponds to the previous program, however, the contents of the dialog box differ.

```
root = Tk(); root.title("LED"); v = IntVar(); v.set(1)
```

Variables that are used in Tk dialog boxes must be declared prior to first use and that is therefore contrary to the normal Python variables. These two lines declare a variable `v` as an integer and the initial value is set to `1`.

```
def LedOn():
    e = v.get()
```

Another function is defined, which is called `LedOn()` like in the last example. However, this time it does not only turns on an LED, but also starts an LED pattern. The second function `LedOff()` of the previous example is not needed here. The first line of the new function reads the user input from the Tk variables `v` and writes the value to the Python variable `e`. We will find out below in the explanation about radio buttons how exactly the value gets into the variable `v`.

Depending on the user's selection, three different programme loops are started:

```
    if e == 1:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[j], True); time.sleep(t)
                GPIO.output(LED[j], False)
```

In the first case, a loop will run through five times in succession, and will switch on each of the four LEDs for 0.2 seconds and off again. The five repetitions and the 0.2 seconds flashing time are defined by the variables `w` und `t` at the beginning of the programme.

```
    elif e == 2:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[j], True)
            time.sleep(t)
            for j in range(4):
                GPIO.output(LED[j], False)
            time.sleep(t)
```

In the second case, all four LEDs are turned on simultaneously five times in succession and, after being lit for 0.2 seconds turn off again simultaneously.

```
    else:
        for i in range(w):
            for j in range(4):
```

```
                GPIO.output(LED[3-j], True); time.sleep(t)
                GPIO.output(LED[3-j], False)
```

The third case is similar to the first, with the difference that the LEDs are counted down backwards and thus reversing the direction of the chaser's run.

When this function is defined, the elements of the graphical user interface are created.

```
Label(root,
      text=”Please click a button to start the chaser
").pack()
```

The text in the dialog is again defined as `Label` object. New is the definition of the three radio buttons.

```
for txt, m in pattern:
    Radiobutton(root, text = txt, variable = v, value = m).pack(anchor=W)
```

The radio buttons are defined using a special `for` loop form. Instead of the loop counter, two variables are specified which are counted in parallel. The two counters run through the elements of the list `pattern` one by one. The first count variable `txt` adopts the first value of the pair of values: text to be displayed next to the radio button. The second count variable `m` adopts the number from the second value of each pair of values of a pattern.

The loop creates three radio buttons that way, that have as the first parameter always `root` and the widget where the radio buttons are located. Tarameter `text` of a radio button shows the text that should be displayed, which in this case is read from the variable `txt`. The parameter `variable` specifies a previously declared Tk variable into which the value of the selected radio button is entered after the user has made a selection.

The parameter `value` specifies for each radio button a numerical value, which in this case is read from the variable `m`. When a user clicks on this radio button, the value of the parameter `value` is written to the under `variable` entered variable. Each of the three radio buttons will be built into the dialog box according to their definition using the method `.pack()`. he parameter `anchor=W` ensures that the radio buttons are left-aligned below each other.

```
Button(root, text="Start", command=LedOn).pack(side=LEFT)
```

The button is as defined as in the previous example.

```
root.mainloop(); GPIO.cleanup()
```

The main loop and the end of the programme correspond to the last example.

Start the program and select a a flashing pattern using the radio button. Via the variable `v` the first selection is preselected. If you use radio buttons in a dialog box, always aim to set a reasonable initial selection so that undefined result are avoided, if the user has not selected anything. A click on *Start* launches the selected pattern which is run five times. Select a different pattern after that.

10.3 Setting the flashing rate

In a third step, the dialog box is expanded again. The user can now adjust with a slider, the blinking speed.

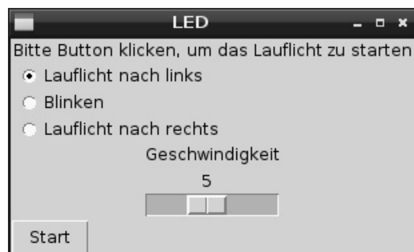


Fig. 10.6: Three LED patterns to choose from and the customizable flashing rate.

Use of the sliders

Slide controls provide a very intuitive way to enter numerical values within a defined range. We are spared the plausibility query, which verifies whether the user has entered a value that the programme is able to implement, because slider don't allow values that are lying outside the specified range. Configure the slider always in a way, that the values are comprehensible for the user. It makes no sense to set up values in the millions. If the absolute numerical value itself is not an issue, give the user an easy a scale ranging from 1 to 10 or 100 and convert the value in the programme accordingly. The values should increase from left to right, if it were the other way around, most users would consider that odd. Also always give a meaningful value, which wil be applied in case the user does not modify the slider.

The programme `ledtk03.py` is almost similar to the previous example, only the regulation of speed is added.

```
import RPi.GPIO as GPIO
import time
from Tkinter import *

GPIO.setmode(GPIO.BCM); LED = [4,18,23,24]
for i in LED:
    GPIO.setup(i, GPIO.OUT, initial=0)

w = 5
pattern = [
    ("Chasing to the left",1), ("Flashing",2), ("Chasing to the right”,3)
    ]

root = Tk(); root.title("LED"); v = IntVar(); v.set(1); g = IntVar(); g.set(5)

def LedOn():
    e = v.get()
    t = 1.0/g.get()
    if e == 1:
        for i in range(w):
            for j in range(4):
```

```
                GPIO.output(LED[j], True); time.sleep(t)
                GPIO.output(LED[j], False)
    elif e == 2:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[j], True)
            time.sleep(t)
            for j in range(4):
                GPIO.output(LED[j], False)
            time.sleep(t)
    else:
        for i in range(w):
            for j in range(4):
                GPIO.output(LED[3-j], True); time.sleep(t)
                GPIO.output(LED[3-j], False)

Label(root,
      text="Please click a button to start the chaser
").pack()

for txt, m in pattern:
    Radiobutton(root, text = txt, variable = v, value = m).pack(anchor=W)

Label(root, text="Speed").pack()

Scale(root, orient=HORIZONTAL, from_ = 1, to = 10, variable = g).pack()

Button(root, text="Start", command=LedOn).pack(side=LEFT)

root.mainloop()
GPIO.cleanup()
```

10.3.1 How does it work?

The initialization of the libraries and GPIO ports and the definition of the list for the three flashing patterns correspond to the previous programme. The determination of the variable `t` for the flashing time is dropped, because the slider will read it later.

`g = IntVar(); g.set(5)` In addition to the Tk variables `v`, *i* where the selected flashing pattern is stored, another integer variable `g` is defined for speed. The given start value of `5` corresponds to the mean value of the slider.

```
def LedOn():
    e = v.get(); t = 1.0/g.get()
```

The function that lets the LEDs flashing also corresponds to the previous example, yet there is one difference. The variable `t` for the duration of flashing is calculated from the value of the slider `g`.

Because a user intuitively links faster flashing to higher speed, the sliders will deliver higher values to the right. However, in the programme a shorter waiting time is needed for higher speed, therefore a smaller value is required. This is achieved by a reciprocal calculation that calculates based on the slider's values `1` to `10` the values `1.0` to `0.1` for the variable `t`. `1.0` must stand in the formula and not `1`, so that the result will be a floating point number and not an integer.

Converting integers to floating-point values
The result of a calculation is automatically saved as floating point number if at least one of the values in the formula is a floating point number. If all values in the formula are integers (Integer), the result will also be reduced to an integer.

The definition of the label and the radio buttons in the dialog box are taken from the previous example.

```
Label(root,
      Label(root, text="Speed").pack()
```

Another label is written in the dialog box for the slider declaration. Since this does not contain any parameters in the `pack()`method, it is centered horizontally below the radio buttons.

```
Scale(root, orient=HORIZONTAL, from_ = 1, to = 10, variable = g).pack()
```

The slider is an object of type `Scale`, which shall contain `root` as a first parameter like all objects in the dialog box. The parameters `orient=HORIZONTAL` indicates that the slider is horizontal. Without this parameter, it would be in a vertical position. The parameters `from_` and `to` specify the slider's start and end values. Note the notation `from_` because `from` without underscore is a term reserved in Python to import libraries. The parameter `variable`specifies a previously declared Tk variable, where the currently set value of the slider is entered. The start value is taken from the value which was defined for the variable declaration, which in this case is `5`.

The slider is centered horizontally in the dialog box by means of the `pack()` method.

The other program components - the *Start* button, the main loop and the end of the program - are taken from the previous example without any alteration.

Start the program, select a flashing pattern and set the speed. Higher settings let the pattern blink faster. When you click on the *Start* button function `LedOn()`reads the selected flashing pattern from the radio buttons and also the speed from the position of the slider.

11 PiDance and LEDs

In the late 70's, before any real computer games, there was an electronic game with four colored lamps, which in 1979 was awarded in the very first selection list the Game of the Year. The game was known in Germany under the name *Senso*. Atari released a replica under the name of *Touch Me* which had the size of a pocket calculator. Another replica appeared as *Einstein,* and on the English speaking market Senso was marketed as *Simon*.

Raspbian provides a graphical version of this game with the *Python Games* under the name *Simulate*.

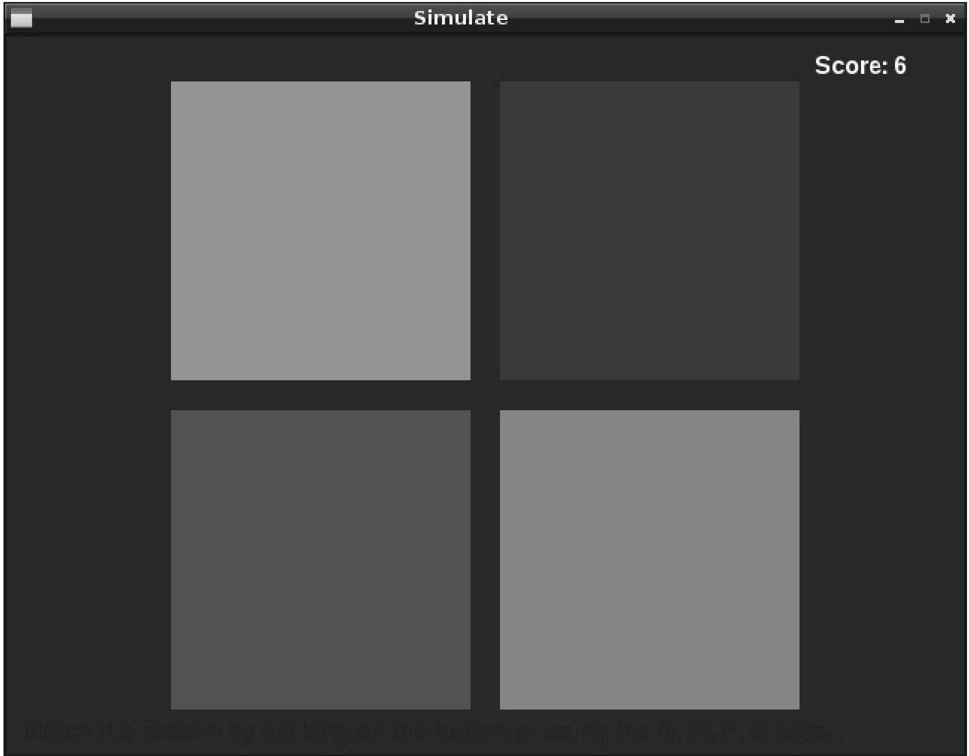


Fig. 11.1: The game Simulate of the Python Games

Our game PiDance is also based on the principle of this game. LEDs are flashing in random order. The user has to press the buttons in the same order. With each round, another LED lights up, so that it becomes more difficult to remember the order. One mistake, and the game is over.

The game is built on two breadboards, so that the buttons are located on the edge and can be operated comfortably without accidentally pulling cables from the boards. For better stability, the breadboards can be plugged together on the long sides.

Besides the connecting cables already known also four short jumpers are required. Use sharp pliers or a wire cutter to cut about 2.5 centimeters long pieces from the supplied jumper wire and remove the 7 mm of the insulation at both ends with a sharp knife. Form these pieces of wire into a U shape. Thus you are able to link two rows on a breadboard.

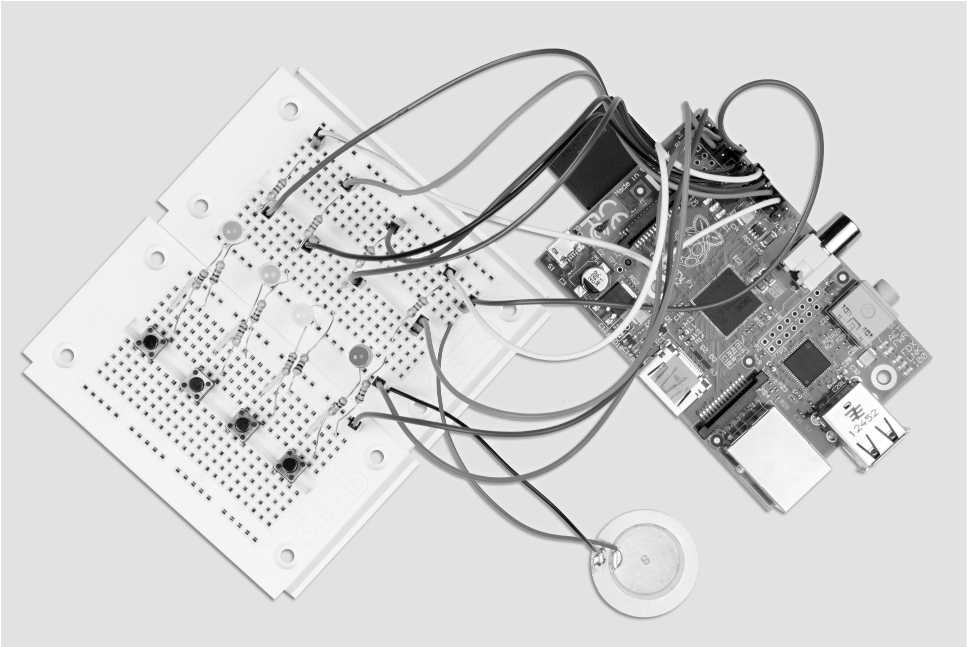


Fig. 11.2: Breadboard assembly for experiment 11

Components required:
2x breadboard
1x LED red
1x LED yellow
1x LED green
1x LED blue
4x 220-ohm series resistor
4x 1-kOhm resistor
4x 10-kOhm resistor
4x button
10x connecting wire
4x short jumper bridge

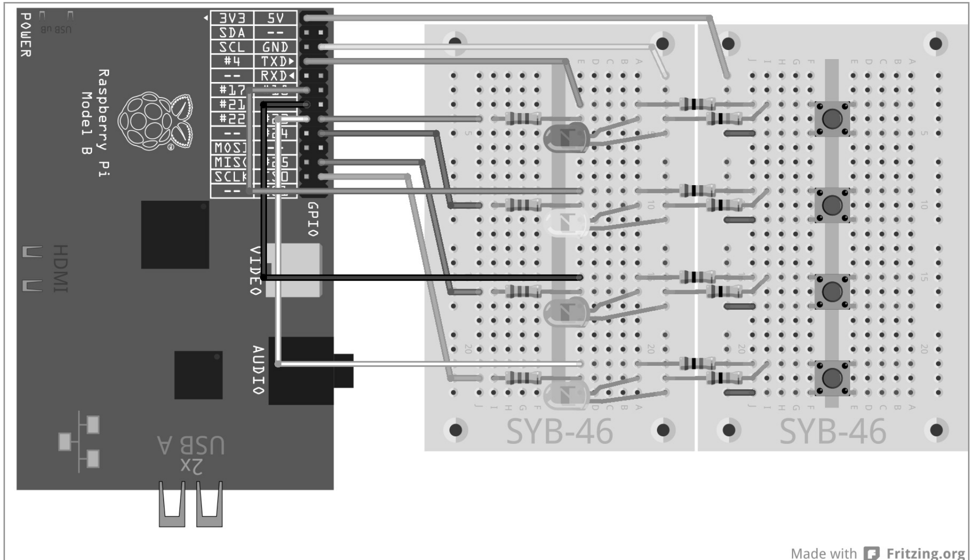


Fig. 11.3: PiDance with LEDs and buttons using two breadboards

The buttons are built opposite to the associated LEDs. The two middle longitudinal rows of the breadboard on both sides of the joint serve as the circuit's 0 V and +3.3 V line.

The programme `pidance01.py` contains the game.

```
# -*- coding: utf-8 -*-
import time, random
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
rnumber = 10; color = []
for i in range(w):
    color.append(random.randrange(4))
LED = [4,18,23,24]
for i in LED:
    GPIO.setup(LED[i], GPIO.OUT, initial=False)
BUT = [4,17,21,22]
for i in TAST:
    GPIO.setup(i, GPIO.IN)
def LEDOn(n, z):
    GPIO.output(LED[n], True); time.sleep(z)
    GPIO.output(LED[j], False)
def Press():
    while True:
        if(GPIO.input(TAST[0])):
            return 0
        if(GPIO.input(TAST[1])):
            return 1
```

```
        if(GPIO.input(TAST[2])):
            return 2
        if(GPIO.input(TAST[3])):
            return 3
ok = True
for round in range(1, rnumber +1):
    print "Round", round
    for i in TAST:
        LEDOn(color[i], 1)
    for i in range(round):
        button = press()
        LEDOn(button, 0.2)
        if(button != color[i]):
            print "Lost!"
            print "You have made it to round", round - 1, "success"
            for j in range(4):
                GPIO.output(LED[j], True)
            for j in range(4):
                time.sleep(0.5)
                GPIO.output(LED[j], False)
            ok = False
            break
    if(ok == False):
        break
    time.sleep(0.5)
if(ok == True):
    print "Well done!"
    for i in range(5):
        for j in range(4):
            GPIO.output(LED[j], True)
        time.sleep(0.05)
        for j in range(4):
            GPIO.output(LED[j], False)
        time.sleep(0.05)
GPIO.cleanup()
```

11.1.1 How does it work?

There is much new here in the programme, however, the basics of the GPIO control are known.

`rnumberl = 10` Ater the import of the modules `time`, `random` and `RPi.GPIO` a variable `rnumber` is created, which determines the number of rounds that will be played. You can also play more than ten rounds of course- the more rounds, the more difficult it is to remember the flashing sequence.

```
colour = []
for i in range(rnumber):
    color.append(random.randrange(4))
```

The list `colour` uses the loop to fill in as many random numbers between 0 to 3 as rounds are played. The method `append()` is used, which is available in any list. It adds the element that is transferred as parameter to the list.

```
LED = [23,24,25,8]
for i in LED:
    GPIO.setup(i, GPIO.OUT, initial=False)
```

The GPIO ports for LEDs are set up as outputs in a list `LED` following the familiar procedure and all LEDs are set to off.

```
TAST = [4,17,21,22]
for i in TAST:
    GPIO.setup(i, GPIO.IN)
```

Now following the same principle, the GPIO ports for the four buttons are set up in a list `TAST` as inputs.

We have created the foundation, and will now define two more functions that are needed several times in the programme.

```
def LEDOn(n, z):
    GPIO.output(LED[n], True); time.sleep(z)
    GPIO.output(LED[n], False); time.sleep(0.15)
```

The function `LEDOn()` turns on an LED for a certain period of time. The function uses two parameters: The first parameter, `n` specifies the number of the LED between 0 and 3, the second parameter, `z`, specifies the time during which the LED will glow. After the LED has been turned off again, the function waits for 0.15 seconds until it is terminates in order to facilitate short breaks between the illumination of the LEDs upon multiple calls. This is particularly important if a LED is repeatedly illuminated. Otherwise it would not get noticed.

```
def Push():
    while True:
        if(GPIO.input(TAST[0])):
            return 0
        if(GPIO.input(TAST[1])):
            return 1
        if(GPIO.input(TAST[2])):
            return 2
        if(GPIO.input(TAST[3])):
            return 3
```

The function `Druecken()` consists of a continuous loop, which waits for the user to push one of the buttons. Then the number of the button is returned to the main programme.

`ok = True` Following the definition of the functions the actual main programme starts and as a first task it sets a variable `ok` to `True`. As soon as the player makes a mistake, `ok` is set to `False`. If the variable after the designated number of rounds is still `True`, the player has won.

```
for round in range(1, rnumberl +1):
```

The game runs defined by the variable `rnumber` over a fixed number of rounds. The round counter is shifted by 1 upwards so the game starts with Round 1 and not with Round 0.

```
    print “Round", round
```

The actual round is displayed in the Python Shell window

```
    for i in range(round):
        LEDOn(color[i], 1)
```

Now the program performs the pattern that the player has to remember. Depending on the actual number of rounds, LEDs light up one after the other in the random colours according to the list `colour` which was set up in the program previously. Since the counter `round` starts with 1, one LED is already lit in the first round. To illuminate the LED, the function `LEDOn()` is used that uses as the first parameter the colour from the corresponding list position, the second parameter lets each LED light up for one second.

`for i in range(round)`: When the colour pattern has been played, another loop starts and here the player has to reproduce the memorized pattern using the buttons.

`button = push()` This calls the function `Push()` with a waiting period until the player has pressed a button. The number of the button pressed is stored in the variable `button`

`LEDOn(button, 0.2)` When a button is pressed the associated LED is lit for 0.2 seconds.

`if(button != color[i])`: If the last button pressed does not match the colour at the corresponding position in the list, the player has lost. The operator `!=` stands for not equal. Here also <> can be used.

```
            print “Lost!"
            print “You have made it to round", round - 1, “success"
```

The program displays on-screen that the player has lost and the number of rounds managed to play. The number of rounds passed is one less than the current round counter.

```
            for j in range(4):
                GPIO.output(LED[j], True)
```

As an optically visible sign all LEDs are switched on ...

```
            for j in range(4):
                time.sleep(0.5); GPIO.output(LED[j], False)
```

... Then one after another at intervals of 0.5 seconds they are turned off again. This results in a significant reduction effect.

`ok = False` The variable `ok` that identifies whether the player is still playing is set to `False` ...

`break` ... and the loop is aborted. The player can no longer press a button. The first mistakes ends the game.

```
    if(ok == False):
        break
```

If `ok` is on `False` the outer loop is aborted and no other rounds follow.

`time.sleep(0.5)` If the entry of the sequence was correct, the programme waits for 0.5 second and starts the next round.

`if(ok == True):` The programme reaches this point when either the loop has fully run though, that is, the player has entered the correct sequences, or the previous loop has been discontinued due to a mistake made by the player. If `ok` is still on `True` the awarding ceremony takes place. Otherwise this block is skipped, and the game only executes the last command line of the programme.

```
    print “Well done!"
    for i in range(5):
        for j in range(4):
            GPIO.output(LED[j], True)
        time.sleep(0.05)
        for j in range(4):
            GPIO.output(LED[j], False)
        time.sleep(0.05)
```

If the game is won, a message is displayed in the Python shell window. All LEDs are then flashing briefly five times in succession.

`GPIO.cleanup()` the last command line is always executed. It closes here the GPIO ports that are used.

Impressum
© 2014 Franzis Verlag GmbH, Richard-Reitzner-Allee 2, 85540 Haar near Munich
www.elo-web.de
Author: Christian Immler

ISBN 978-3-645-10145-5

All rights reserved, including the rights of photomechanical reproduction and storage on electronic media. Creating and distributing copies on paper, on disk or on the Internet, in particular as PDF, is permitted only with the explicit permission of the publisher and is failing to do will result in criminal proceedings.
Most of the product designation of hardware and software, as well as company names and logos mentioned in this work, are also generally registered trademarks and should be treated as such. In terms of product designation the publisher follows primarily the notations of the manufacturers.
All circuits and programmes presented in this book have been developed, tested and verified with utmost care. Nevertheless, errors contained in book and software cannot be completely ruled out. Publisher and author are liable in cases of intent or gross negligence in accordance with statutory provisions. Publisher and author are otherwise liable only in accordance with the product liability law for injury to life, limb or health or for culpable violation of essential contractual obligations. The claim for damages for breaching of contractual obligations is limited to the typical contractual relevant, foreseeable damage if no case of strict liability is provided under the product liability law.

Electrical and electronic devices may not be disposed of with household garbage!
Dispose of the product at the end of its service life in accordance with the applicable legal regulations. Collection points have been set up, where you return electrical appliances free of charge. Your local authority will inform you where to find such a collection facility.

The product conforms to the relevant CE directives to the extent as you are using it in accordance with enclosed instructions. The description is part of the product and must be included when passing it on.

Caution: Eye protection and LEDs:

Do not look directly at close range into an LED, because directly looking into it may cause retinal damage! This is particularly true for bright LEDs in transparent enclosing and here in particular power LEDs. The apparent brightness of white, blue, violet and ultraviolet LEDs, create an improper impression of the actual risk to your eyes. Be especially careful when using converging lenses. Do not operate the LEDs with higher currents other than provided by the instructions.