

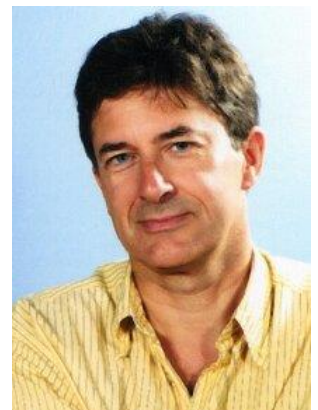
EV3 Python

This is a course on programming the Lego EV3 robot (either the home or education version) with EV3 Python. **This course does not teach the basics of Python, so you should be familiar with basic Python before taking this course.** In this course, 'EV3 Python' refers to the EV3dev Python3 library (version 2) and the EV3dev operating system. Bugs may exist in EV3 Python as in any other software - neither I nor the other members of the EV3dev team can be held responsible for any damage caused by the use of EV3dev Python. You use the software at your own risk. Microsoft is a trademark of Microsoft Corporation, LEGO, the LEGO logo, MINDSTORMS and the MINDSTORMS EV3 logo are trademarks and/ or copyrights of the LEGO Group. Lego Corporation does not endorse this course or any of my websites. This course is produced by Nigel Ward, teacher of physics and computer science.

This document is essentially the script of the videos that form the core of this course. I encourage you to watch the videos AND read through this document because video is not the ideal medium for presenting Python scripts. This course, including this document, is the result of months of work. This document is not to be distributed freely – it is only for those who have signed up for the Udemy course.

Should I take this course?

Hello, I'm Nigel Ward. I've taught computer science and physics in many of the world's top international schools but recently I've been focused on helping people learn how to program the world's most popular pedagogical robot, the Lego EV3, with the world's most taught textual programming language, Python.



So, should you take this course?

You should not take this course if you don't have access to a Lego EV3 robot, of course.

You should not take this course if you want to make lots of cool-looking Lego robots because this course is about programming, not about building many different robots. Using variations on a standard design, the official Lego Education Vehicle is also a deliberate choice to make it possible for this course to be used in schools, where teachers don't have the time or pedagogical justification to have kids making many different models.

You should not take this course if you don't already have some knowledge of the basics of the Python programming language. This course will NOT teach you the basics of Python, and learning the basics of Python by learning how to program robots would make no sense because the special commands used to control robots are not basic Python commands. There are many ways to learn the basics of Python, including some good courses on Udemy.com.

You should not take this course if you are looking for some version of EV3 Python other than the one based on the ev3dev operating system and Python version 3. In particular, this course is not about the EV3 Python variant known as EV3 MicroPython.

You **SHOULD** consider taking this course if you like working with the Lego EV3 robot but want to program it with a more powerful, more standard programming language than the standard Lego one, sometimes called EV3-G. In fact the standard Lego programming language isn't really a language at all – it's a set of icons that you configure and connect together – this is very different to the textual programming languages used by professional programmers. In this course you will learn how to program the EV3 with the Python programming language, the most taught textual programming language in the world. This course won't teach you the *basics* of Python, but it will give you plenty of *practice* in working with basic Python concepts, and at the same time you will be learning about robots, which is really important given that robots and artificial intelligence are going to have a huge impact on human society in the

coming decades. Billions of jobs are going to be partially or completely automated, and you should do all you can to prepare for the coming changes and to boost your chances of getting a job in one of the few fields that are likely to grow. Anything that deepens your knowledge of Python and robots increases your chances of getting a well-paid job. Python programmers are generally very well paid – **in the US for example the average salary of Python programmers is well over 100 thousand dollars per year**, equivalent to more than 4 million dollars over a whole career. With those kind of figures in mind, ask yourself whether it's reasonable to worry about the cost of taking this course, and don't forget also that Udemy has a money-back guarantee if you're not satisfied.

Working with robots, even a robot as modest as the Lego EV3, also increases your chances of getting a job working with robots, of course, and is one job area that is likely to grow quickly while others are automated out of existence in the coming years. Even if you don't work with robots, understanding how they work will make you feel more comfortable in a world where they will strongly impact our lives.

Wait a moment, did I just describe the EV3 robot as 'modest'? It's actually quite expensive compared to many other small robots out there. OK, so you're paying for Lego quality, reliability and modularity, but you're also paying a lot for something else: the possibility of programming your robot with many different programming languages. The EV3 can do that because at its heart is the so-called 'intelligent brick' a proper little Linux computer. When you buy an EV3 kit with more than 600 pieces there is one piece that costs more than all the other pieces combined – it's the intelligent brick. So if you spend all that money to have the possibility of programming in multiple languages and then only ever use the quirky Lego software then I'm afraid, my friend, that you may have wasted quite a lot of money.

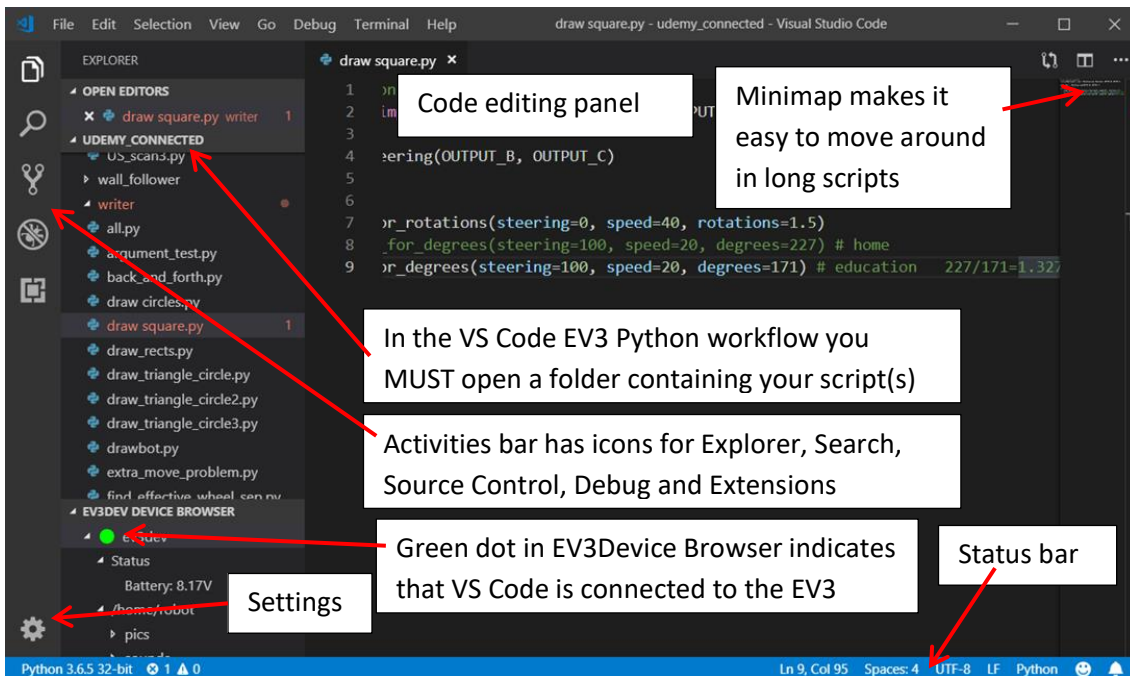
Speaking of buying an EV3 kit, there are two types of kit out there: the retail or home version, and the educational version. Although the kits are rather different, this course is designed to be compatible with both types of kit.

As I said, the EV3 intelligent brick can be programmed in many different languages, but the fact that you're watching this video indicates that you've already figured out that Python is the best choice among the many textual programming languages out there. It's become the most-taught textual programming languages in the world thanks to its combination of power, simple syntax and conciseness. In fact the well-known site Codecademy.com now considers Python to be the world's most popular programming language.

This course is cheap compared to the cost of an EV3 and compared to the boost that Python and robotic skills can give to your career prospects. Could you get the information in this course elsewhere? Some of it, no doubt, but this course is built around a new and very simple way of programming the EV3 with Python, so you won't find much out there to compete with this course. This course is based on programming with a free, open source, multiplatform code editor called **Microsoft Visual Studio Code**, or simply VS Code. VS Code is compatible with Windows, MacOS and Linux. Recently an extension was released for VS Code that makes it very easy to write and run Python scripts for the EV3. How easy? As easy as 1, 2, 3. Once everything is set up, all you need to do is:

- 1) Write your Python script
- 2) Save it
- 3) Run it by pressing the F5 key (or by running it directly on the EV3, of course)

This is what VS Code looks like:



The EV3 and Python extensions have been installed and the green light in the EV3 Device Browser indicates that VS Code is already connected to the EV3. My system is set up so that when I press the F5 key the script is downloaded to the EV3 and run there – it's as simple as that.

In order for all this to work, the EV3 needs to be running an operating system called **ev3dev** which you will install on a micro SD card and insert in the EV3. The great thing about that process is that it doesn't modify the EV3's firmware so any time you want to return to using the Lego software all you need to do is turn off the EV3 and take out the card. Like VS Code, the ev3dev software is completely free.

In addition to this trailer video, this course includes videos to show you how to set up your EV3, how to install and configure VS Code on your computer, and how to establish a connection between VS Code and the EV3. Then we get a sense of how VS Code works by writing and running a couple of *non*-EV3 Python scripts, since that's easier than writing and running EV3 Python scripts, then there are several videos that guide you through writing and running EV3 Python scripts that interact with the EV3 motors, sensors, screen, buttons and speaker. Finally there are some videos that present more complex scripts for a drawing robot and a writing robot (model building instructions are included for both the home set and the education set). There are about 5 hours of video in total.

There are about 20 videos altogether, with a total duration of about 5 hours. The essentials of EV3 Python programming are explained in parts 1 and 2 (two hours) and parts 3 and 4 mainly give practice and examples. As such, parts 3 and 4 are optional.

In addition to the videos, this course includes:

- A ZIP file with most of the Python scripts that are discussed during the course.
- A PDF document (about 100 pages) which is essentially the script of the various videos.
- Build instructions for the home version of the Education Vehicle and a link to build instructions for the education version of the Education Vehicle.
- Build instructions for a bumper attachment.
- Build instructions for a drawing and writing robot that is the subject of parts 3 and 4 of this course.

Table of Contents of the videos

A more detailed table of contents is provided as a separate document. Parts 1 and 2 (totalling 2 hours) teach you the basics of EV3 Python programming and parts 3 and 4 present examples to give you more practice. Therefore parts 3 and 4 can be considered optional.

Should I take this course?

- **(11 minutes) The trailer video for this course**

Introduction

- **(2 minutes) A brief recap of what was presented in the trailer video 'Should I take this course?'**

Part 1: Setting up

- **1A (9 minutes) Set up the EV3**
- **1B (5 minutes) Connect the EV3 to the computer**
- **1C (23 minutes) Install VS Code and the two needed extensions, configure, and connect to EV3**

Part 2: The Components

- **2A (24 minutes) Motors**
- **2B (30 minutes) The Intelligent Brick (display, buttons, LEDs and loudspeaker)**
- **2C (19 minutes) Sensors**

Part 3: Putting the pieces together (43 minutes)

- **3A (3 minutes) Collide, back up, turn and continue**
- **3B (2 minutes) Line follower**
- **3C (7 minutes) Wall follower**
- **3D (9 minutes) Steer with light**
- **3E (2 minutes) Follow an object**
- **3F (3 minutes) Follow a beacon (only for home model)**
- **3G (5 minutes) Program with colors**
- **3H (9 minutes) Self-parking**
- **3I (3 minutes) Beware of steep slopes (only for education model)**

Part 4: Make a drawbot and a writerbot

- **4A Drawbot part 1 (38 minutes, with the option of skipping the final 15 minutes which is a mathematical derivation)**
- **4B Drawbot part 2 (22 minutes)**
- **4C Writerbot part 1 (33 minutes) Write a script that can write characters that do not contain arcs.**
- **4D Writerbot part 2 (22 minutes) Modify the script so that it can also write characters that contain arcs**

If you're hesitating about taking this course, you can check out my websites ev3Python.com, mind-storms.com and technofiles.eu.

A last note: this course is NOT aimed at hackers or people with advanced Python skills. In particular, this course does NOT explain how to communicate with the EV3 from VS Code via a Secure **SH**ell (SSH) connection, even though that's quite easy to do from VS Code, and it does not explain how to connect the EV3 to the internet via your computer, since that is not needed for this course. Furthermore, this course does not cover the use of third party sensors or motors, only the ones that are included in the standard EV3 kits, and this course does not explain how to use EV3dev to work with Raspberry Pi or Beaglebone, even though ev3dev is also compatible with those platforms.

Introductory video (2 minutes)

This video summarises the trailer video and therefore the script for the introductory video is not included in this document.

Part 1: Setting up

Video 1A: Set up the EV3

In the introductory video I explained how easy it can be to program the LegoEV3 robot using Python – the trick is to use the free code editor **Microsoft Visual Studio Code** with the **EV3 extension** that was released just recently. This course continues with three videos showing how to set up your EV3 and computer for EV3 Python programming. In this video I'll show you how to prepare the EV3. In the next video I'll show you how to establish a connection between the EV3 and the computer. In the video after that I'll show you how to install and configure VS Code on your computer, and how to run a couple of *non*-EV3 Python scripts. Then there will be a number of videos showing you how to write and run Python scripts on your EV3 using VS Code with the EV3 extension.

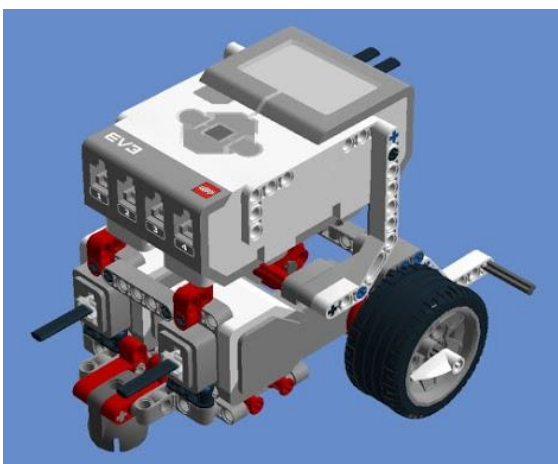
First, though, we must build the robot! Almost all the exercises in this course assume that you are using the '**Education Vehicle**' model (also called the **Robot Educator** or **Driving Base**). This is an official Lego model that is the basis for the most of the exercises included with the education version of the EV3 software. Instructions for building that model with the *education* version of the EV3 can be found here:

- education.lego.com/en-us/support/mindstorms-ev3/building-instructions#robot
- robotsquare.com/2013/10/01/education-ev3-45544-instruction/

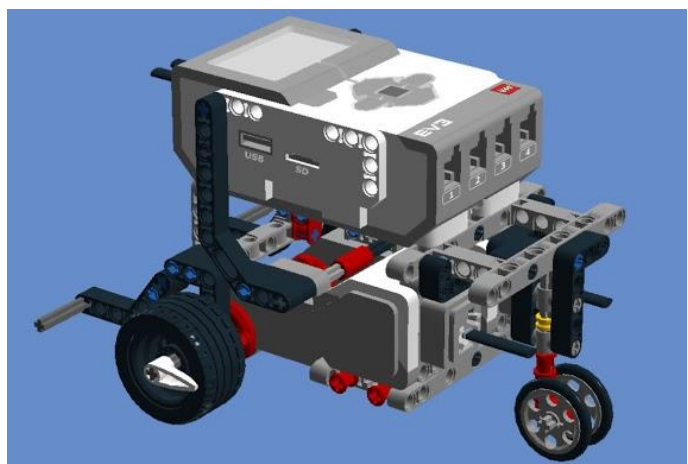
I've designed a similar model that can be built with the *home* version of the EV3. The biggest differences are in the wheels: the education kit has large wheels that are 30% larger in diameter than those in the home kit, and we will need to take this into account in our code. The home version does not include a caster ball wheel like the one in the education kit, so the rear wheel assembly is quite different. Also, when we start using sensors we will have to adjust the exercises to take into account that the education version of the EV3 includes ultrasonic and gyro sensors that are not included in the home version, and the home version includes an infrared sensor and remote control / beacon that is not included in the education version. The build instructions for making my home version of the education vehicle are included with the resources attached to this course and can also be found here:

sites.google.com/site/gask3t/lego-ev3/building-plans/educator-vehicle-retail-kit-version

Education kit version



Home (retail) kit version



Tip: the above right image of the home version robot shows rubber tires on the rear wheels. Rubber tires give increased grip which is exactly what you do NOT want the rear wheels to have so it's a good idea to *not* put the rubber tires on those wheels.

Now that you've built the robot, is what you need to do to prepare your EV3 robot for use with EV3 Python:

1. Obtain a suitable microSD memory card.
2. Download the latest **Linux Debian Stretch ev3dev image**.
3. Download and install **Etcher**, a free utility that will allow you to flash the ev3dev image to the microSD card.
4. Use Etcher to flash the image to the card.
5. Insert the card into the SD slot on the EV3, turn on the EV3 and explore the Brickman interface

Now in more detail:

Obtain a suitable microSD or microSDHC memory card

Obtain a microSD or microSDHC memory card with a capacity between 2GB and 32GB. MicroSDXC cards are not supported on the EV3. If you need to buy a card I suggest you buy an 8GB card from a well-known manufacturer such as Sandisk or Kensington.

Download the latest Linux Debian Stretch ev3dev image

Note that the VS Code extension is compatible only with *Stretch* and not with *Jessie* versions of ev3dev. Go straight to this address: www.ev3dev.org/downloads/ Be sure to download the latest *Stretch* version and not a *Jessie* version. **Do not download the 'Lego-approved version' since that uses MicroPython which, as the name suggests, is just a subset of normal Python – it does not include useful features like speech synthesis and the scripts in this course will not work with MicroPython.** Stretch versions are available for EV3, Raspberry Pi 1, Raspberry Pi 2 and Beaglebone, but only the EV3 is discussed in this course. The download is about 340MB and may take a couple of minutes. The downloaded file may be zipped (compressed) in which case unzip it before proceeding.

It would be a good idea to check every month or so whether a more recent image is available. If it is then using Etcher to flash the image to the SD card is easier and faster than trying to connect your EV3 to the internet and then updating the older operating system on the card – that's why this video does not explain how to connect your EV3 to the internet. **If you use Etcher to flash a newer image to the card then any Python scripts on the card will be deleted** but this should not be a problem because you have the same scripts on your computer and it's quick and easy to download them to the card again.

Download and install Etcher

Etcher is a free, open source, multiplatform (Windows, Mac OS, Linux) utility that will allow you to 'flash' the ev3dev image to the microSD card. Download it from etcher.io and install it on your computer.

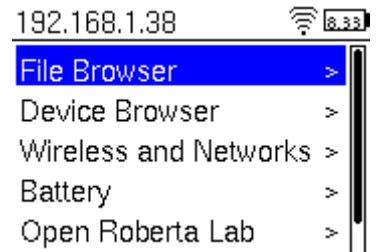
Flash the image to the card

Insert the SD card in your computer's card reader (in the unlikely event that your computer does not have a card reader you will need to buy one). If you get a message that you need to format the card you can dismiss that for Etcher will format the card for you. Start Etcher, select the ev3dev Stretch image you just downloaded, navigate to your microSD card (be sure to select the correct card since **everything on the card will be erased during the flashing process**) and click 'Flash!'. The flashing and validation process may take a couple of minutes.

Insert the card into the EV3

With the EV3 turned off, insert the microSD card into the slot on the side of the EV3 marked 'SD'. The card is fairly easy to insert but more difficult to remove – when you want to remove the card you may find it helpful to use tweezers.

Boot the EV3 by pressing the Enter (center) button for a few seconds. Remember that the ev3dev operating system runs entirely off the microSD card – the EV3 firmware is not modified at all, so if you wish to return to using the standard Lego EV3 software all you need to do is power down the EV3, remove the card and reboot. When the boot process has finished, which takes a couple of minutes, the EV3's LED lights will glow green and the ev3dev interface called **Brickman** will be displayed on the EV3 display. At the top of the Brickman interface is a header line where the EV3's IP address will be displayed, as well as indicators for WiFi or Bluetooth connections and an indication of the voltage of the EV3 battery. Under that are the main sections proposed by Brickman: File Browser, Device Browser, Wireless and Networks, Battery, Open Roberta Lab and About. Take a couple of minutes to explore the Brickman interface. Don't forget you can move back up through the menus by pressing the Back key on the EV3.



What we need to do next is connect the EV3 to the computer via USB, WiFi, Bluetooth or Ethernet, and that will be the topic of the next video.

Video 1B: Connect the EV3 to the Computer

In this video we'll see how to establish a connection between the EV3 and the computer, which can be via USB, Bluetooth, WiFi or Ethernet. Your EV3 should be turned on and you should be looking at the Brickman interface.

- 1. USB connection.** This is perhaps the easiest. Simply connect a USB cable from your computer to the 'PC' socket on the EV3. You've probably done that before when you used the standard Lego EV software. The computer should then be connected to the EV3 no matter whether Brickman indicates 'Online', 'Connected' or 'Disconnected'. Even if Brickman indicates 'disconnected' that only means that there is no IPv4 connection to the computer - an IPv6 connection should automatically have been established between the EV3 and the PC. Note that this video is intended for EV3 Python beginners so I won't be explaining how to get the EV3 'online' i.e. connected to the internet via the computer. You really shouldn't need to connect the EV3 to the internet to use the VS Code workflow described in this video, and neither should you need to establish a **Secure SHell** (SSH) connection between VS Code and the EV3. If you are a more advanced user and do want to use SSH then you should refer to ev3Python.com and ev3dev.org.
- 2. Bluetooth connection.** Bluetooth functionality is built in to the EV3 but if you are using a desktop computer you may need to purchase a Bluetooth dongle for the computer. For the setup procedure, refer to ev3dev.org/docs/tutorials/connecting-to-the-internet-via-bluetooth . Ignore the line at the bottom of that page that says you should connect via SSH – that shouldn't be necessary in our wonderful VS Code workflow. On my Windows PC I find it easier to establish a WiFi connection than a Bluetooth connection so I prefer to use a WiFi connection.
- 3. WiFi connection.** To make a WiFi connection you will need to buy an adaptor if you do not already own one. Unlike the standard Lego EV3 software which demands a very specific WiFi adaptor (see photo), the ev3dev software is compatible with a wide range of WiFi adaptors. Plug the adaptor into the EV3's USB socket, then, in the Brickman main menu, choose '**Wireless and Networks**'. Select '**Powered**' (press the Enter button) if it is not already selected. An indicator will come on in Brickman's header to indicate that the WiFi adaptor is powered. Then choose '**Start Scan**' and wait until



available WiFi hotspots have been found. Choose the hotspot you wish to connect to then choose '**Connect**'. Press the Enter key to see the on-screen keyboard then enter your WiFi password. You can press the Back button if you make a mistake. Choose '**OK**' when you have finished, then choose '**Accept**'. You should see 'Status: Connected' or 'Status: Online' – for our purposes it doesn't matter which you see. Now scroll down below the 'Forget' option and choose '**Network Connection**'. Turn on the option to '**Connect Automatically**'. You can move back up through the Brickman menu system by pressing the Back button.

4. **Ethernet connection.** If you want to use an Ethernet connection you must buy a USB to Ethernet adaptor such as this one. You will plug the USB plug into the socket on the EV3 marked USB, and the Ethernet cable into your network hub. The nice thing about the Ethernet connection is that it should 'just work', with no need for any configuration.



Once you've established a connection from the EV3 to the computer, you're ready to install VS Code and the needed extensions, configure VS Code, and establish a connection between VS Code and the EV3. That will be the topic of the next video, which will also invite you to run a couple of non-EV3 Python scripts to get a feel for VS Code.

Video 1C: Install and set up Visual Studio Code

In this video we'll see how to install VS Code and the two needed extensions, how to configure VS Code, and how to establish a connection between VS Code and the EV3. Here are the main steps:

1. Download and install **Microsoft Visual Studio Code** (VS Code). This is a free multiplatform code editor, compatible with Windows, Mac OS and Linux.
2. Start VS Code and install the two extensions that we need.
3. Configure VS Code.
4. Write and run some non-EV3 Python scripts.
5. Connect VS Code to your EV3.
6. Open the starter project folder in VS Code and run the starter script.

Here are the steps in detail:

Download and install Microsoft Visual Studio Code

This is a free multiplatform code editor, compatible with Windows, Mac OS and Linux and not to be confused with Microsoft Visual Studio. Download and install Visual Studio Code (VS Code) from <https://code.visualstudio.com/Download>

For more help with getting started with VS Code, along with some helpful videos, see

<https://code.visualstudio.com/docs>

VS Code is in English by default but is available in more than a dozen other languages. If your English is decent then I encourage you to keep the program in English because arguably English is the language of computing and because it will be easier for you to follow this course if you keep VS Code in English. If you absolutely want to have VS Code in a different language then you will need to install a language pack – see the instructions here:

<https://code.visualstudio.com/docs/getstarted/locales>

Start VS Code and install the two extensions that we need

We're going to use two extensions: the Microsoft Python extension and the EV3 Browser extension. In the Activities bar at the left we have the Explorer view, the Search view, the Source Control view, the Debug view and the Extensions view. Switch to 'Extensions' view – you may have 'Python' proposed here as a popular choice, otherwise

you can search for 'Python'. **Install the Microsoft Python extension and the most recent EV3 Browser extension.** The Python extension requires that VS Code be restarted so do that now.

Configure VS Code

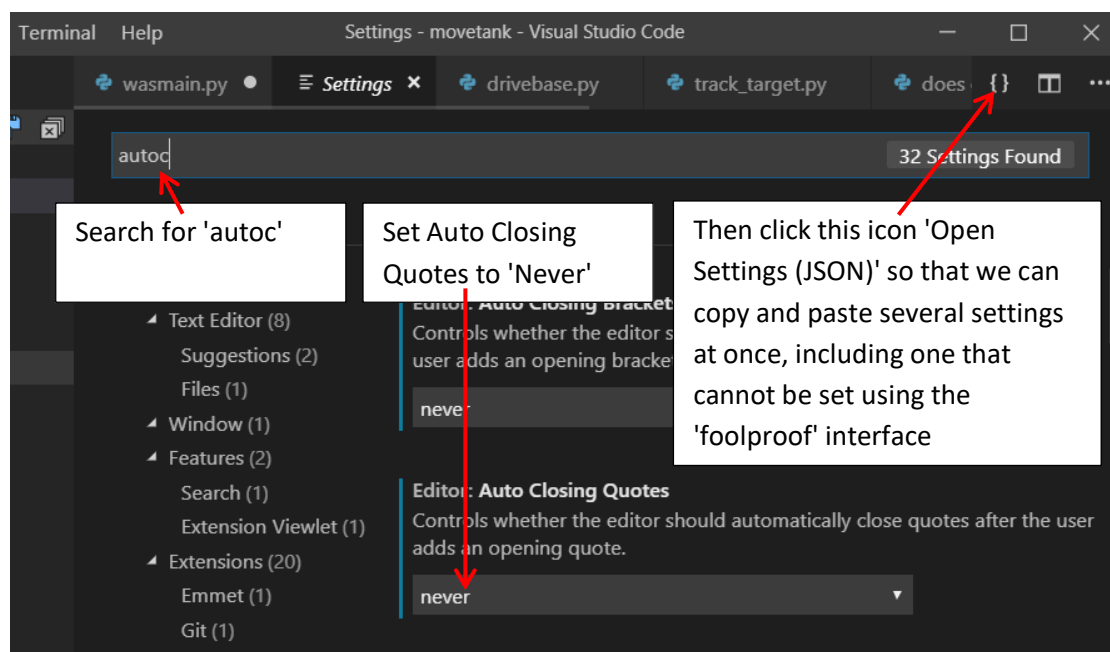
VS Code is highly customisable via several hundred settings that you can modify. Don't worry – we only need to adjust about nine of them. It's important to understand that settings can exist at three different levels: **workspace**, **user** and **project**.

- **Workspace settings** are the default settings that apply to all projects unless overridden by user or project settings that you have set. Workspace settings cannot be modified directly.
- **User settings** override the default workspace settings and apply to all projects unless overridden by project settings. User settings are held in a file called **settings.json** which is associated with the VS Code application.
- **Project settings** exist within a file called **settings.json** inside a folder called **.vscode** within specific projects. This file should not be confused with the file with the same name that contains the *user* settings and which is associated with the VS Code application rather than a specific project. If your project has project settings, which is not obligatory, then they will override the corresponding user settings and workspace settings. **In this course, to keep things as simple as possible, we will not use project settings.** It is possible that an unasked for .vscode folder containing settings and launch configurations will appear in your project folder – please delete any .vscode folder that appears or things will get very confusing!

This is important stuff, so I suggest you read the above text again, and be clear in your mind that **user settings override workspace (default) settings and project settings override user settings AND workspace settings.**

As previously stated, the user settings are held in a file called **settings.json** which is associated with the VS Code application. VS Code provides an interface which makes it easy and rather fool-proof to create or modify user settings by setting options rather than by typing code, so that it is not necessary to edit the settings.json file directly. Let's try using the standard settings interface to create a user setting.

Open the settings editor by choosing **File>Preferences>Settings** or by clicking the gear icon at the bottom left and choosing 'Settings'. We will add a user setting to override a so-called 'feature' of VS Code that most beginner coders find very annoying: the automatic closing of quotes. Search for 'editor.autoc' and remove the checkmark from the 'Auto Closing Quotes' option. This change should be saved automatically.



Unfortunately one of the settings that we need to modify cannot be modified via the user-friendly interface – it requires us to modify the settings.json file directly. Since we have to modify the settings.json file directly, we might as well set *all* the user settings simultaneously, by copying and pasting the code that we need.

Open the settings.json file as shown in the image above.

If the file contains any settings other than the 'auto-closing quotes:never' that we set previously then make a backup of the settings.json file by choosing 'Save as...' and then saving the copy with a name such as **settings_old.json**. Close the file **settings_old.json** that you have just created and re-open the file settings.json. **Then replace all the code in the settings.json file with new code that you can copy and paste from a file called settings.txt which is included in the ZIP file that accompanies this course. You can open the settings.txt file in VS Code if you want. Don't try to copy and paste the code below from this PDF document because it is likely that you will lose the indentation within the code, and indentation is a critical element of Python scripts. Once you have pasted in the code, save and close the file, then restart VS Code.**

```
// Place your settings in this file to overwrite the default settings
{
  "launch": {
    "version": "0.2.0",
    "configurations": [{
      "name": "Download and Run",
      "type": "ev3devBrowser",
      "request": "launch",
      "program": "/home/robot/${workspaceRootFolderName}/${relativeFile}"
    },
    {
      "name": "Python: Current File (Integrated Terminal)",
      "type": "Python",
      "request": "launch",
      "program": "${file}",
      "console": "integratedTerminal"
    }
  ]
},
"git.ignoreMissingGitWarning": true,
"ev3devBrowser.confirmDelete": false,
"editor.autoClosingBrackets": "never",
"editor.autoClosingQuotes": "never",
"files.eol": "\n",
"ev3devBrowser.download.exclude": "{**/*.*,LICENSE,README.md}",
"editor.fontSize": 15,
"debug.openDebug": "neverOpen",
}
```

The most important thing to notice in the code above is that we have created two '**configurations**', one called '**Download and Run**' and the other called '**Python: Current File (Integrated Terminal)**'. We will use the first configuration whenever we want to run a Python file on the EV3 from VS Code, and the other whenever we want to run Python files locally on the PC.

The code `${relativeFile}` refers to the name of the active script. We are setting up the 'Download and Run' configuration so that when you press the F5 key the active script will be run, whichever open scripts that happens to be.

Write and run some non-EV3 Python scripts

Even though our main aim is to write and run EV3 Python scripts, we'll start with *non*-EV3 scripts because that is easier to do and also a good way to get a feel for VS Code. Non-EV3 Python scripts will run locally within VS Code. For

non-EV3 Python scripts, you can either open a folder containing your script(s), or you can simply open a 'loose' Python script without opening the folder that contains it. However, since working with EV3 Python scripts *requires* that you open a folder containing your script or scripts it would be a good idea to have that habit even when you are working with non-EV3 scripts.

Do **File> Open Folder**, navigate to a convenient location for a folder that will contain your non-EV3 Python scripts and click '**New Folder**'. Name the new folder '**nonev3 python scripts**' and click the '**Select Folder**' button. Assuming you are in Explorer view, the folder will open in the side bar. The folder that holds your script or scripts is called the 'project folder'.

Do **File>New File** and before you even type any code save the file with the name **helloworld.py**. The reason to do that before typing the code is that as soon as VS Code recognises the extension for Python scripts it begins using helpful color coding. Type 'pr' and notice how the Intellisense feature of VS Code proposes auto-completion to save you some typing. Choose the word you want and press Enter or Tab to confirm. After 'print', type an opening parenthesis and notice how Intellisense helps you in another way by giving you tips about the print() function. Finish typing the line so that it says **print('Hello, world')** and save your code again. Python is case-sensitive so, for example, Print('Hello, world') will not work. Be aware also that in Python 3 the print() function requires parentheses, which is not the case in Python 2. Recall that you MUST use Python 3 and not Python 2 with the VS Code EV3 Python programming workflow.

Run your script by right-clicking the script and choosing 'Run Python File in Terminal'. The text will be printed to the terminal panel.

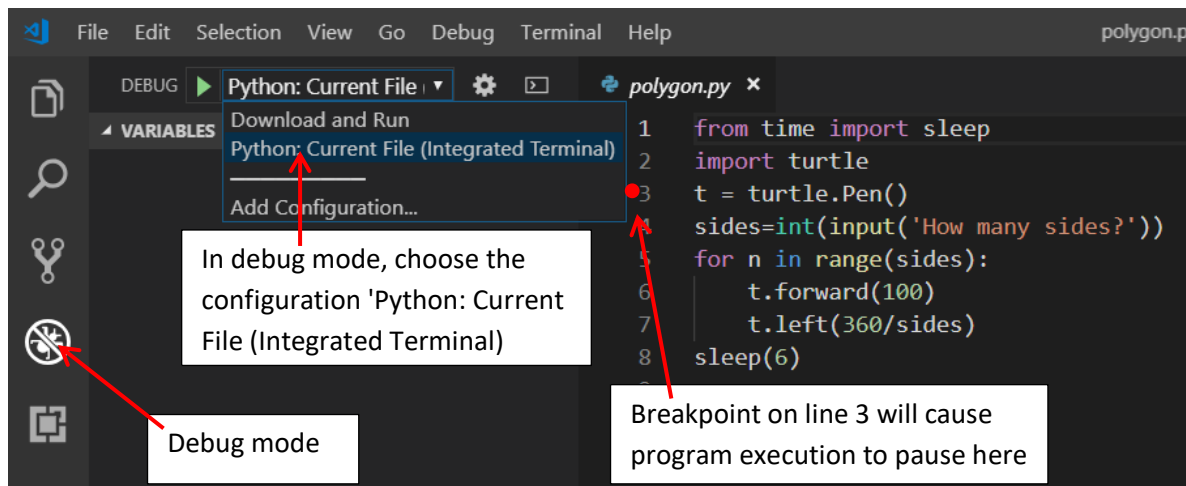
Let's make another non-EV3 Python script, this time one that uses the turtle module that hopefully is installed with your Python interpreter (this should be the case if you downloaded your copy of Python from python.org). This script will ask the user to input a number and then draw a polygon with that number of sides.

Do **File>New File**, save the file with the name **polygon.py** then type or copy/paste this script. **If you copy/paste then be aware that copying text from a PDF document does not necessarily preserve any indentation. Correct indentation is critical in Python scripts so be sure to restore the indentation in the pasted script. By strong convention, each level of indentation in Python scripts should be four spaces.**

```
from time import sleep
import turtle
t = turtle.Pen()
sides=int(input('How many sides?'))
for n in range(sides):
    t.forward(100)
    t.left(360/sides)
sleep(6)
```

You can run your script by right-clicking the script and choosing '**Run Python File in Terminal**'. You may find that when you type the number into the terminal the graphics window disappears behind the VS Code window and that you cannot make it visible again. In that case make the VS Code window small and you should be able to see the polygon being drawn.

You can also run the script while in the Debug view. Before doing that it would be helpful to put a breakpoint on line 3 by clicking to the left of the line number such that a red dot appears. **In debug mode, make sure that the 'Integrated Terminal' configuration is selected** so that you can run the program by pressing F5 or choosing Debug>Start Debugging.



If you then choose **Debug>Start Debugging** (or press the F5 key) then script execution will pause at line 3 and you can then repeatedly click the 'Step Over' icon in the control bar to step through the script until it terminates.

Note that we have put two scripts in the same folder. That's fine for beginners and it makes it easy to copy and paste code between scripts, but advanced coders working on complex projects would not put unrelated files in the same project folder.

Open the 'ev3 python scripts' folder in VS Code

In order to be able to run your EV3 Python scripts on the EV3 from VS Code you **MUST** have open in VS Code a **folder that contains your script**. The folder that holds your script or scripts is called the '**project folder**'. As far as we are concerned as VS Code beginners, the project folder can also be called your '**workspace**'. Don't mix EV3 Python scripts with non-EV3 Python scripts in the same folder because they need to be launched in different ways.

In VS Code, do **File > Open Folder** and open the folder called **ev3 python scripts** which is included in the resources that accompany this course. This folder contains a file, **starter.py**, which is the following Python script.

```
#!/usr/bin/env python3

from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)

steer_pair.on_for_degrees(steering=100, speed=20, degrees=930)
```

This script simply makes the robot turn on the spot. Don't try running it yet, though, for we are not ready.

Connect VS Code to your EV3

In a previous step you connected your EV3 to your computer, and now it's time to connect VS Code's ev3dev browser extension to the EV3. Make sure the Explorer view is chosen in the activities bar at left and you should see an 'EV3DEV DEVICE BROWSER' section at the bottom of the Explorer panel. Click where it says 'Click here to connect a device' (you may need to click the header of the section in order to see those words). You should then see at the top of the window the words 'I don't see my device' and above that you should see a name that represents your EV3. If there is such a name then click it and after a couple of seconds the connection to the EV3 should be established – a green dot will appear under the header of the EV3 Device Browser section. A red dot indicates a lost connection and an amber dot indicates that an attempt is being made to establish a connection.

If you do not see an EV3 name above the words 'I don't see my device' or if you are not able to connect to the named device then you have no choice but to click where it says 'I don't see my device'. You will then be asked to give a name to the connection you are trying to create – a name such as 'EV3 via USB' would be helpful. You will then

be asked for the IP address of the EV3 which should be displayed at the top of the Brickman interface. When you enter information in this way your 'User settings' are modified. If you have tried to create a connection by inputting a name and IP address then you will see this has been incorporated into a User setting.

You can voluntarily disconnect VS Code from the EV3 by right-clicking the green dot and choosing 'Disconnect'. That right-click menu also gives you options of taking a screenshot, getting system info or opening an SSH (Secure SHell) terminal.

Note that VS Code will disconnect from the EV3 if you close the project folder or open a different folder.

If you are connected to the EV3 by a wireless connection (WiFi or Bluetooth) you may sometimes experience an involuntary disconnect. VS Code will not automatically try to reconnect if the connection is lost.

If you are disconnected from the EV3 and therefore see a red dot, then you can right-click the red dot and attempt to reconnect. You also have the option 'Connect to a different device' which is rather misleading because this is also the option you would choose if you want to connect to the *same* device but via a different connection type (e.g. switching from USB to WiFi). Of course, connecting or reconnecting VS Code to the EV3 only works if there is a connection between the *computer* and the EV3.

Run the starter script

Before we run the **starter.py** script there is one more thing we need to do. Remember how we set up two configurations, one for running non-EV3 Python scripts locally on the PC and one for running scripts remotely, on the EV3? The last script that we ran was the polygon script – that ran locally using the configuration '**Python: Current File (Integrated Terminal)**'. We need to switch to the other configuration '**Download and Run**' before we can run the script **starter.py** remotely on the EV3. Switch to Debug mode and use the pull-down list at the top of the debug panel to select the configuration '**Download and Run**'. Then switch back to Explorer view.

Everything is now ready for us to run the script **starter.py**:

- we have assembled the 'education vehicle' model, with large motors attached to ports B and C
- the script is open and active
- the correct configuration (Download and Run) has been selected
- we have a connection between the computer and the EV3
- we have a connection between VS Code and the EV3, as indicated by the green dot under the header of the EV3 Device Browser section.

To run the script, simply press the F5 key – this should cause the script to be downloaded to the EV3, the Python interpreter to be started up on the EV3 and the script to run. Does that work for you after all that effort setting up? If so, then **congratulations – you've just run a Python script on your EV3 for the first time ever!** If you are having problems, look back over the instructions and also check out ev3Python.com and ev3dev.org. There may also be a troubleshooting resource included in this course.

When you press the F5 key, notice how the status bar at the bottom of the VS Code window tells you that your files are being downloaded and also gives you an indication of which configuration setting is being used. By the way, in the default color theme called 'Dark+' the color of the bar has meaning as follows:

- Blue: normal color
- Orange: a script is running /being debugged LOCALLY (not on EV3)
- Purple: no folder is open

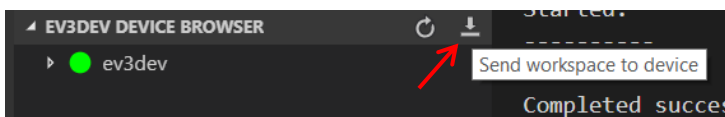
You can change the color theme if you want by clicking the gear icon at bottom left, then choosing Color Theme.

When you press F5 not only is the active script downloaded to the EV3 but in fact ALL the scripts in the project folder are downloaded, so it's a bad idea to have more than about twenty scripts in the project folder since that would add a few seconds to the time taken to download and run your script every time you do so. **Note that even if there is only one short script in the project folder there will always be a delay of several seconds between the F5 key being pressed and the script actually running because it takes several seconds for the Python interpreter on the EV3 to start up and this must happen every time you run a script.** This may make the EV3 Python workflow described in this course less than ideal for competitions such as First Lego League (FLL) although in some competitions you are allowed to start a program in advance such that the program pauses until you pressing a button.

Did you notice that while the script was running a **control bar** appeared at the top of the VS Code window? It includes a stop button that you could use to force the script to stop running – especially useful for scripts that do not stop running by themselves. You can also force a script to stop running by pressing the Back button on the EV3.



What if you want to launch your script from the Brickman interface on the EV3 rather than from VS Code? In that case you can download the contents of the project folder *without* running the active script by clicking the '**Send workspace to device**' icon in the header of the EV3 Device Browser panel, and then, in Brickman, choose **File Browser > ev3 python scripts > starter.py**.



In the next section we will take a detailed look at EV3 Python code, including the script we have just run.

General notes about EV3 Python

Let's take a look at the code of the starter script in a bit more detail.

Before you can get into the nitty gritty of your Python script, you have to include three 'boring bits' at the beginning of each script.

Shebang

The first 'boring bit' is a special one, always the same, that tells the EV3 to use the Python3 interpreter to interpret the program, and tells the EV3 where to find the interpreter in its file hierarchy. That special line, called a 'shebang', must be exactly this:

```
#!/usr/bin/env python3
```

It must appear, exactly as above, as the very first line of your script. If you don't include this exact shebang you will probably get this error message when you try to run the script:

```
Starting remote process failed: Failed to execute child process
"/home/robot/ev3 python scripts/starter.py" (Permission denied)
-----
Exited with error code 1.
```

Import Classes and Functions

Next you must import the classes and functions that your script will use. In EV3 Python3 version 1 it was often sufficient to do this with just one line of code but with EV3 Python3 v2 you will normally need several lines of import code, and the lines will be longer. It's conceivable that in a complex script you might need to import most of this, or more (most but not all of what you see here is probably meaningful to you already):


```
from ev3dev2.motor import LargeMotor, MediumMotor, OUTPUT_A, OUTPUT_B,
OUTPUT_C, OUTPUT_D, SpeedDPS, MoveSteering, MoveTank
from ev3dev2.sensor import INPUT_1, INPUT_2, INPUT_3, INPUT_4
from ev3dev2.sensor.lego import TouchSensor, GyroSensor, InfraredSensor,
ColorSensor, UltrasonicSensor
from ev3dev2.led import Leds
from ev3dev2.button import Button
from ev3dev2.sound import Sound
from ev3dev2.display import Display
import ev3dev2.fonts as fonts
```

Usually you will need to import much less than that. Why not keep the old version 1 format which is so much shorter? The main reason is to **import no more than is needed**, because that should make the importing a little faster. Including all the imports above might add about 10 seconds to the time it takes for your script to start running, relative to importing just one or two items.

Make objects (instances of classes)

The import statements import 'classes', such as the **LargeMotor** class. Classes are like templates or blueprints or descriptions of something – they are not actual 'objects'. An analogy in everyday life would be that the blueprint for making an iPad is not an actual iPad. Therefore the next step is to make actual '**objects**' based on the classes that we have imported. When we make an object based on a class we can also say that we are making an '**instance**' of that class, which can also be called '**instantiation**'. Of course we can make several instances of the same class if necessary, just as several iPads can be made from the iPad blueprints. The object-making lines corresponding to the imports above might look like this (again, most but not all of this is probably meaningful to you at this stage):

```
large_motor = LargeMotor(OUTPUT_B)
medium_motor = MediumMotor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)
ts = TouchSensor()
ir = InfraredSensor()
cl = ColorSensor()
gyro = GyroSensor()
us = UltrasonicSensor()
leds = Leds()
btn = Button()
screen = Display()
sound = Sound()
```

You don't have to use the above names for the objects. You could, for example, name the LargeMotor object 'lm' instead of 'large_motor'. Nevertheless, the names I propose are probably the best because they are very clear. You can't use spaces in the names, and don't forget that Python is case-sensitive, so if you make an object called 'large_motor' and then later refer to that object as 'Large_Motor' then you will get an error.

Part 2: The Components

Video 2A: Motors

I'm sure the number one priority for you as you start working with EV3 Python is to get the motors going, so let's start with that.

Large Motor

Usually you will want the large motor to do one of the following:

- turn through a specific angle at a specific speed
- turn on for a specific time at a specific speed
- simply turn on at a specific speed, knowing that code later in the script will turn it off again.

Turn through a specific angle

To turn the motor through a specific angle you can express the angle in *degrees* or in *rotations* (one rotation = 360 degrees, of course)

To turn through an angle in degrees use

```
on_for_degrees(speed, degrees, brake=True, block=True)
```

For example, to turn on an EV3 large motor called 'large_motor' for 180 degrees at a speed of 50, use

```
large_motor.on_for_degrees(speed=50, degrees=180)
```

You're not obliged to mention the parameter names here so you might be tempted to write

```
large_motor.on_for_degrees(50, 180)
```

and that would indeed work and would make the code shorter but it would make your code less clear so it's preferable to include these parameter names. Coding often involves compromises between clarity and conciseness.

Note that a negative angle or speed will cause the motor to turn in the opposite direction which I will call 'backwards', but if *both* the angle and the speed are negative then the motor will turn forwards, which makes sense mathematically of course. For the EV3 large motors, it doesn't really make much sense to talk about the motor turning 'clockwise' or 'counter-clockwise' because that depends which side of the motor you're looking at (the convention that gives meaning to those words is to imagine holding the large motor horizontally such that its red shaft is bottom right).

The 'speed' parameter is a percentage of the rated maximum speed of the motor. It corresponds to the 'power' parameter in EV3-G. It is an integer in the range -100 to +100. **For the EV3 large motor, the rated maximum speed is 1050 degrees per second.** For example, if the speed is set to +50 then the motor will turn in the 'forward' direction at 50% of its rated maximum speed i.e. $50\% \times 1050 = 525$ degrees per second. If the speed is set to -20 then the motor will turn in the 'backwards' direction at 20% of its rated maximum speed i.e. $0.2 \times 1050 = 210$ degrees per second. Note that neither the EV3 large motor nor the EV3 medium motor are actually capable of reaching their theoretical rated maximum speed and in practice you should **avoid using speed values greater than about 90.**

It's rather awkward that the rated maximum speed of the EV3 large motor is 1050 degrees per second and not simply 1000 degrees per second since that makes it rather difficult to convert degrees per second into speed values. Fortunately, a conversion function is available in EV3 Python to do the conversion for you. It's called **SpeedDPS()** and it converts Degrees Per Second into the corresponding speed value. For example, if you run this line

```
large_motor.on_for_degrees(speed= SpeedDPS(500), degrees=180)
```

then the motor will turn in the 'forward' direction at 500 degrees per second. There are several other similar conversion functions such as **SpeedRPS()** which converts **rotations per second** into a speed value, and **SpeedRPM()** which converts **rotations per minute** into a speed value.

Earlier on I mentioned that **on_for_degrees()** has two other parameters in addition to speed and degrees. These are **'brake'** and **'block'**. These are 'keyword parameters' with default values.

The **block** parameter determines whether program execution is blocked (paused) until the motor movement is complete. If **block = True** then the program waits for the motor motion to complete before executing the next instruction in the script, otherwise the next instruction executes immediately. You will almost always want **block** to

be True. This is the default value so you don't need to mention 'block' in your motor command unless for some reason you want to turn off blocking.

The **brake** parameter determines whether the brake will be applied once the motor has completed its motion (as opposed to the motor gradually 'coasting' to a standstill, slowed only by friction). The motor doesn't have a physical 'brake' of course – the brake effect is achieved by feeding power to the motor if necessary in order to stop the motor rotating. Like for the block parameter, the default value 'True' is almost always what you want so you don't need to mention 'brake' in your motor command unless for some reason you want to turn off braking.

You can also specify the angle in rotations with

```
on_for_rotations(speed, rotations, brake=True, block=True)
```

so you could replace the command

```
large_motor.on_for_degrees(speed=50, degrees=180)
```

with the equivalent command

```
large_motor.on_for_rotations (speed=50, rotations=0.5)
```

We're going to make our motor do several movements and we would like to make the program pause between each movement so that we can better observe what is happening. You should not be learning EV3 Python unless you have already a decent command of standard Python, in which case you should already be familiar with the **sleep()** function which makes program execution pause for a given number of seconds. In order to be able to use the **sleep()** function we will need to be able to import it from the time module so we will include this code in our import block as shown below:

```
from time import sleep
```

We're ready to write and run our first EV3 Python script using VS Code! In VS Code, open the file `large_motor.py` in the `ev3 python scripts` folder.

Our script begins as always with the shebang, followed by the import and instantiation lines, followed by the 'nitty gritty'. There are also some comments to help you. Comments can be made by placing a hash character (#) before the comment. Comments are ignored by the Python interpreter and are only there to help you understand the script.

```
#!/usr/bin/env python3
from ev3dev2.motor import LargeMotor, OUTPUT_B
from ev3dev2.motor import SpeedDPS, SpeedRPS, SpeedRPM
from time import sleep

large_motor = LargeMotor(OUTPUT_B)

# We'll make the motor turn 180 degrees
# at speed 50 (525 dps for the large motor)
large_motor.on_for_degrees(speed=50, degrees=180)
# then wait one second
sleep(1)
# then - 180 degrees at 500 dps
large_motor.on_for_degrees(speed=SpeedDPS(500), degrees=-180)
sleep(1)
# then 0.5 rotations at speed 50 (525 dps)
large_motor.on_for_rotations (speed=50, rotations=0.5)
sleep(1)
# then - 0.5 rotations at 1 rotation per second (360 dps)
large_motor.on_for_rotations (speed=SpeedRPS(1), rotations=-0.5)
```

Connect an EV3 large motor to motor port B of your EV3 and attach a rod and a pointer piece to the motor so that you can more easily see what angles the motor turns through. Then press the F5 key ... and cross your fingers ... !

If the script runs for you then... congratulations! It may be a very modest script that won't impress anyone except perhaps yourself, but know that it's very easy for errors to creep into scripts and usually even a tiny error like a comma in the wrong place can cause your script to fail. That's why I tell my students that in coding 99% correct is a failing grade! Your script has to be 100% valid!

Assuming the script runs for you without problems, you should be able to hear that in the second motion the motor is turning a little more slowly (500dps as opposed to 525 dps).

Turn on for a specific time at a specific speed

If you want to run your motor for a specific time then use

```
on_for_seconds(speed, seconds, brake=True, block=True)
```

For example, to turn on our motor for 5 seconds at 60 rotations per minute (1 rotation per second) use

```
large_motor.on_for_seconds(speed=SpeedRPM(60), seconds=5)
```

Add the above line to the existing script and run it again (precede it with a one second sleep).

Turn on at a specific speed 'forever'

If you want to turn on your motor 'forever' (until some other code turns it off) then use

```
on(speed, brake=True, block=False)
```

For example, if you want to turn on your motor 'forever' at 40% of the rated maximum speed (i.e. speed=40) then use

```
large_motor.on(speed=40)
```

Note that this version of 'on' is the only version for which block=False by default, which makes sense because this motion will never complete by itself so program execution *must* be allowed to continue so that at some point the motor will get the order to stop or do something else. Also note that for this function there is a brake parameter even though it could be argued that it doesn't make much sense to have a brake parameter for a function that instructs the motor to turn forever!

As you would have guessed, there is an **off(brake=True)** function to stop the motor that has been instructed to turn 'forever'.

Add these four lines to your script and run it again:

```
sleep(1)
large_motor.on(speed=60)
sleep(2)
large_motor.off()
```

Of course, these last three lines are equivalent to **large_motor.on_for_seconds(speed=60, seconds=2)**. We don't know enough code yet to do anything other than wait between turning the motor on and turning it off.

Medium Motor

The code for the medium motor is essentially the same as for the large motor, but the rated maximum speed of the medium motor is different: 1560 degrees per second. That means that if you run a large motor and a medium motor both with speed=50 then the large motor will run at 525 dps and the medium motor will run at 780 dps. But of course if you run them both with a conversion function such as **SpeedDPS(500)** then they will run at the same speed (500 dps).

Connect a medium motor to motor port A and try running the following script. It's the same as the last one, just modified to suit the medium motor. Changes are highlighted.

```
#!/usr/bin/env python3
from ev3dev2.motor import MediumMotor, OUTPUT_A
from ev3dev2.motor import SpeedDPS, SpeedRPS, SpeedRPM
from time import sleep
medium_motor = MediumMotor(OUTPUT_A)
# We'll make the motor turn 180 degrees
# at speed 50 (780 dps for the medium motor)
medium_motor.on_for_degrees(speed=50, degrees=180)
# then wait one second
sleep(1)
# then - 180 degrees at 500 dps
medium_motor.on_for_degrees(speed= SpeedDPS(500), degrees=-180)
sleep(1)
# then 0.5 rotations at speed 50 (780 dps)
medium_motor.on_for_rotations (speed=50, rotations=0.5)
sleep(1)
# then - 0.5 rotations at 1 rotation per second (360 dps)
medium_motor.on_for_rotations (speed=SpeedRPS(1), rotations=-0.5)
sleep(1)
medium_motor.on_for_seconds(speed=SpeedRPM(60), seconds=5)
sleep(1)
medium_motor.on(speed=60)
sleep(2)
medium_motor.off()
```

Motor pair classes

Next we come to some very exciting new functions that were not available in version 1 of EV3 Python: the motor pair functions. We have two new classes: **MoveTank** and **MoveSteering**, which mirror the functionality of the Move Tank and Move Steering blocks in EV3-G. In order to be able to use these new classes they must first be imported and then objects must be created based on them, for example like this:

```
from ev3dev2.motor import MoveTank, MoveSteering, OUTPUT_B, OUTPUT_C
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)
steer_pair = MoveSteering (OUTPUT_B, OUTPUT_C)
```

You don't have to use these exact names for the motor pair objects but I recommend you do because they make for clear, legible, unambiguous code. There is also a strong and useful convention that for mobile robots the two driving motors should be connected to motor ports B and C, as indicated above. The assumption here is that the driving motors are EV3 *large* motors since only one *medium* motor is included in the EV3 kit, but if you happen to have two medium motors it's also possible to use them with the driving wheels provided you make a modification to the code that is given later. It's NOT possible to use a pair of motors that are of different types such as a medium motor and larger motor to make up the motor pair.

The functions that work with the new classes are similar to the functions that you have already used to control single motors. Many of the parameters are the same (block, brake, speed, degrees etc) but there are a few differences.

MoveTank

As you must know from your work with EV3-G, MoveTank controls a pair of motors by specifying the speed of each motor in the parameters of a single function. For the MoveTank functions, the meaning of each parameter should not need much explaining.

Move Tank for a given angle

```
tank_pair.on_for_rotations(left_speed, right_speed, rotations, brake=True, block=True)
```

The rotations parameter refers to the motor with the faster speed. Let's look at an example:

```
tank_pair.on_for_rotations(left_speed=50, right_speed=75, rotations=3)
```

This instruction will cause the robot to advance with the left wheel turning at 50% of its rated maximum speed and the right motor turning at 75% of its rated maximum speed. Therefore the robot will turn to the left as it advances. In this case the right wheel is the faster one so **rotations=3** will apply to that wheel. The left wheel will turn through an angle that the function will calculate based on the speed ratio and the rotations value – in this case it will be $(50/75) \times 3 = 2$ rotations. The calculation is such that the two motors *should* stop rotating at the same time but the real times may differ very slightly because EV3 Python does not yet have 'motor synchronisation' which IS a feature of EV3-G.

The above command could also have been written like this:

```
tank_pair.on_for_rotations(50, 75, 3)
```

but of course that's much less clear so it's better to use the longer version.

As you must have guessed already, you can also use **on_for_degrees()** to set the angle. The following command is equivalent to the one we just saw, since $3 \times 360 = 1080$:

```
tank_pair.on_for_degrees(left_speed=50, right_speed=75, degrees=1080)
```

If **left_speed** and **right_speed** have the same value then the robot will move in a straight line (forwards if the speeds are positive and backwards if the speeds are negative). If the speeds are equal and opposite then the robot will turn on the spot. For example this would make the robot turn left on the spot:

```
tank_pair.on_for_rotations(left_speed=-50, right_speed=50, rotations=4)
```

Move Tank for a given time

```
on_for_seconds(left_speed, right_speed, seconds, brake=True, block=True)
```

This will rotate the motors at **left_speed** and **right_speed** for **seconds**.

Move Tank 'forever'

```
on(left_speed, right_speed)
```

This will start rotating the motors according to **left_speed** and **right_speed** forever.

Stop the motors

```
off(brake=True)
```

Stop both motors immediately.

Putting it all together, here's a script for you to try. Open the script **movetank.py** in the **ev3 python scripts** folder. Be sure to connect large motors to ports B and C. Note that you can have multiple statements on the same line if you separate them with semicolons.

```
#!/usr/bin/env python3
```



```

from ev3dev2.motor import MoveTank, OUTPUT_B, OUTPUT_C
from time import sleep

tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)

# gentle turn left
tank_pair.on_for_rotations(left_speed=50, right_speed=75, rotations=3);sleep(1)
# same as above but using degrees instead of rotations
tank_pair.on_for_degrees(left_speed=50, right_speed=75, degrees=1080);sleep(1)
# turn right on the spot
tank_pair.on_for_rotations(left_speed=-50, right_speed=50, rotations=4);sleep(1)
# go straight forwards for four seconds
tank_pair.on_for_seconds(left_speed=40, right_speed=40, seconds=4);sleep(1)
# equivalent to above
tank_pair.on(left_speed=40, right_speed=40)
sleep(4)
tank_pair.off()

```

Before we move on from **MoveTank** to **MoveSteering**, let me mention what you have to do in the unlikely event that you want to use a pair of EV3 medium motors to propel your robot. You would need to include lines like these in your code (but you don't need to use ports A and D of course):

```

from ev3dev2.motor import MediumMotor, MoveTank, OUTPUT_A, OUTPUT_D
tank_pair = MoveTank(OUTPUT_A, OUTPUT_D, motor_class=MediumMotor)

```

Note the necessity of importing **MediumMotor** and specifying **motor_class**, because when you use large motors it is NOT necessary to import **LargeMotor** and it is NOT necessary to specify the **motor_class**.

Move Steering

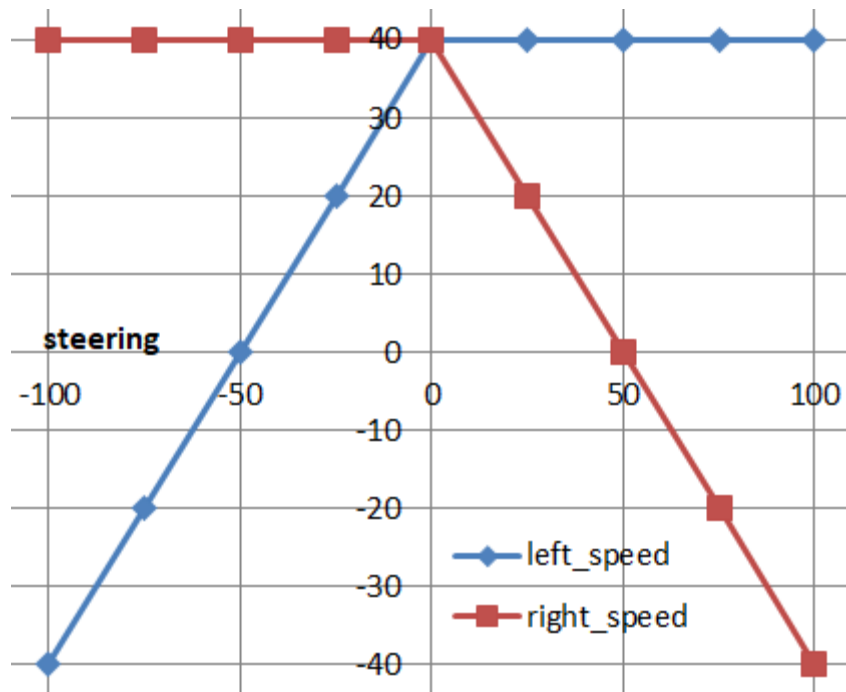
EV3 Python now has a **MoveSteering** class that corresponds to the Move Steering block in EV3-G. The **MoveSteering** functions have a **speed** parameter and a **steering** parameter that correspond to the **power** and **steering** parameters in EV3-G. The **steering** parameter (along with the diameter and separation of the wheels) determines the path that will be followed by your robot, and the **speed** parameter of course determines how fast the robot will move along that path. You probably already know that a **steering** value of:

- -100 means turn left on the spot
- -50 means turn left with the left wheel not turning
- 0 means go straight
- +50 means turn right with the right wheel not turning
- +100 means turn right on the spot.

But do you really know how the **steering** parameter is related to the **speed** parameter, the **left_speed** and the **right_speed**? Let's take a moment to get clear on that.

The **speed** value refers to the speed of the faster wheel (sometimes the wheels may be equally fast, such as when **steering** = 0). The speed of the slower wheel is automatically calculated based on the **steering** and **speed** values.

This chart may help you understand the relationship between **MoveTank** and **MoveSteering**. It is intended to show what values of **left_speed** and **right_speed** (the **MoveTank** parameters) correspond to all possible values of **steering** and a **speed** value of 40 (the **MoveSteering** parameters).



- For all values of **steering** (-100 to +100) there is always at least one motor with a speed equal to the **speed** value, 40.
- When **steering**=0 we note that **left_speed** and **right_speed** are both equal to **speed**, which is 40.
- When **steering**=25 we note that **left_speed**=40 and **right_speed**=20.
- When **steering**=50 we note that **left_speed**=40 and **right_speed**=0.
- When **steering**=100 we note that **left_speed**=40 and **right_speed**=-40.

Now that we've got a good understanding of exactly what the 'steering' parameter represents, you should have no trouble working with the **MoveSteering** functions:

Move Steering for a given angle

Of course we are referring to a given angle of rotation of the faster motor and NOT to the angle through which the robot itself may turn. As usual, the angle can be specified in degrees or rotations.

```
on_for_degrees(steering, speed, rotations, brake=True, block=True)
```

Let's look at an example:

```
steer_pair.on_for_rotations(steering=-25, speed=75, rotations=2)
```

This instruction will cause the robot to gently turn left as it moves forward with the faster motor (the right motor) turning at 75% of its rated maximum speed. Since the right wheel is the faster one **rotations=3** will apply to that wheel. The left wheel will turn through a smaller angle.

As for the equivalent MoveTank function, you can also use **on_for_degrees()** to set the angle. The following command is equivalent to the one we just saw, since $2 \times 360 = 720$:

```
steer_pair.on_for_degrees(steering=-25, speed=75, degrees=720)
```

Don't forget that the new functions never ignore the sign of the speed or of the angle. If the steering is set to 0 and *either* the speed *or* the angle are negative, but not both, then the robot will move *backwards*. If *both* the speed *and* the angle are negative then the robot will move *forwards*.

Move Steering for a given time

```
on_for_seconds(steering, speed, seconds, brake=True, block=True)
```

Example: this command will make the robot turn left on the spot for 3 seconds. Both wheels will turn equally fast but the left wheel will turn backwards while the right wheel turns forwards.

```
steer_pair.on_for_seconds(steering=-100, speed=40, seconds=3)
```

Move Steering 'forever'

```
on(steering, speed)
```

Example: As we saw when discussing the meaning of the steering parameter, a **steering** value of plus or minus 50 will cause the robot to turn with only one wheel turning. In the example below, **steering=50** which means the right wheel will not turn. The speed is *negative* so the left wheel will turn *backwards*.

```
on(steering=50, speed=-30)
```

Putting it all together, here's a script for you to try. Open the script **movesteering.py** in the **ev3 python scripts** folder. Be sure to connect large motors to ports B and C.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)

# gentle turn left
steer_pair.on_for_rotations(steering=-25, speed=75, rotations=2); sleep(1)
# same as above but using degrees instead of rotations
steer_pair.on_for_degrees(steering=-25, speed=75, degrees=720); sleep(1)
# turn right on the spot for 3 seconds
steer_pair.on_for_seconds(steering=100, speed=40, seconds=3); sleep(1)
# equivalent to above
steer_pair.on(steering=100, speed=40)
sleep(3)
steer_pair.off()
```

Video 2B: The Intelligent Brick

In this section we'll learn how to use EV3 Python to control the brick's LEDs, screen, sound and buttons.

LEDs

The EV3 brick has two pairs of LEDs underneath the buttons, sometimes called 'status lights'. On each side, left and right, there is both a red LED and a green LED. By making the green and red LEDs both glow with different brightnesses it's possible to obtain other colors like yellow, orange and amber. The complete list of allowable colors is 'RED', 'GREEN', 'YELLOW', 'ORANGE', 'AMBER', 'BLACK'. Choosing 'BLACK' is of course the same as turning off both LEDs.

EV3 Python is also supposed to have the possibility of making LEDs flash but as of October 2018 that feature is not yet working.

Before setting the LEDs to a certain color you need to turn them both off because that will turn off the flashing that normally occurs when a program is running.

The command to set the color of an LED pair is **set_color(group, color, pct=1)** but for now the **pct** parameter is not working so you can ignore that. Here, then is a complete working script (**leds.py** in the **ev3 python scripts** folder) that will make the left pair of LEDs glow steady amber for 4 seconds. Note the need to import the **Leds** class and to create an instance of that class.

```
#!/usr/bin/env python3
from ev3dev2.led import Leds
from time import sleep

leds = Leds()

leds.all_off() # Turn all LEDs off. This also turns off the flashing.
sleep(1)

leds.set_color('LEFT', 'AMBER')
sleep(4)
```

Challenge: Make a script based on this one but that sets BOTH pairs of LEDs to glow yellow for 5 seconds.

Screen

The screen is a rather complex topic since there are many dozens of functions that can be used to draw to the screen (rectangles, arcs etc). The EV3 has a 178 x 128 pixels monochrome (grayscale) LCD screen. It's not backlit so it's usually quite hard to see. The coordinates of the top-left pixel are (0, 0) and the coordinates of the bottom-right pixel are (177, 127).

Print()

Let's start with something rather easy: the **print()** command. What you need to know about this command is that:

- It's only for text, not graphics.
- But it can print text in the form of strings, integers, tuples etc., even in the same print() command as long as you do not try to concatenate (join) different types together. For example, this is ok: `print('Age: ', 5)` and this is ok: `print('Age: ' + str(5))` but this is not ok: `print('Age: ' + 5)` because it tries to concatenate a string and an integer. This course never uses 'argument specifiers' (placeholders) as in `print("%s %d" % ('Age:', 5))` because I find this to be too difficult for Python beginners.
- By default, it will print in a tiny font that is very hard to read (but it's possible to use larger fonts)
- Text will wrap to the next line when it hits the right edge of the screen, but individual characters wrap rather than whole words.
- Text will scroll up the screen once text hits the bottom of the screen, and text that has scrolled out of view cannot be retrieved.

This (**print.py** in the **ev3 python scripts** folder) is probably the simplest possible script you could write to print 'Hello, world!' to the EV3 screen and make it remain visible for 5 seconds:

```
#!/usr/bin/env python3
from time import sleep

print('Hello, world!')
sleep(5)
```

If you run that you will see that it really does print small! This is a screenshot of what you will see:



Let's take a break for a moment to see how I made that screenshot. It's really easy. Just right-click the green dot in the VS Code EV3 device browser panel and choose 'Take Screenshot'. That will display the screenshot in a new window, but it won't save it automatically. Choose 'Save As' so that you can save it in a convenient location with a sensible name. Be sure to end the name with the .png extension, otherwise it may save in the wrong format. Don't choose 'Save' otherwise the image will probably be saved in a location that is not convenient.

Sometimes having such tiny text is helpful, such as when you have to display a lot of text on the screen. Let's imagine you have to display 100 integers, perhaps readings from one of your sensors. We haven't yet seen how to work with sensors so I'll just use loops to generate a sequence of integers. This script (**print_integers.py** in the **ev3 python scripts** folder) will print 56 integers, but of course most of them will be lost as they scroll off the top of the screen, leaving us with what you see in the screenshot underneath the script.

```
#!/usr/bin/env python3
from time import sleep

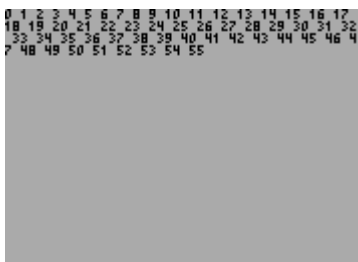
for i in range (56):
    print(i)

sleep(8)
```



Obviously, in the above script a new line is started after each integer is printed – we need to stop that. You can do that once you understand that the **print()** command has a parameter called '**end**' which determines what will be printed after the given text has been printed. By default the argument of the end parameter is the new line character '\n' (officially called the line feed (LF) character), which explains why a new line is always started in the script above, but a different argument can be supplied, such as a space. Check out the script below (**print_integers2.py** in the **ev3 python scripts** folder) and the screenshot underneath.

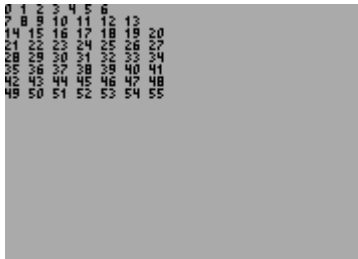
```
#!/usr/bin/env python3
from time import sleep
for i in range (56):
    print(i, end=' ')
print('') # start a new line
sleep(8)
```



We get our 56 integers, but it's not pretty. The final **print('')** command is needed because otherwise the line that we printed earlier is never 'finished' and does not print. Try running the script with the **print('')** command commented out if you don't believe me.

It would be nice if we could display the integers in 8 rows each containing 7 integers. How can we do that? Check out the script below (**print_integers3.py** in the **ev3 python scripts** folder) and its accompanying screenshot:

```
#!/usr/bin/env python3
from time import sleep
for i in range (8):
    for j in range (7):
        print(i*7+j, end=' ')
    print('') # start a new line
sleep(8)
```

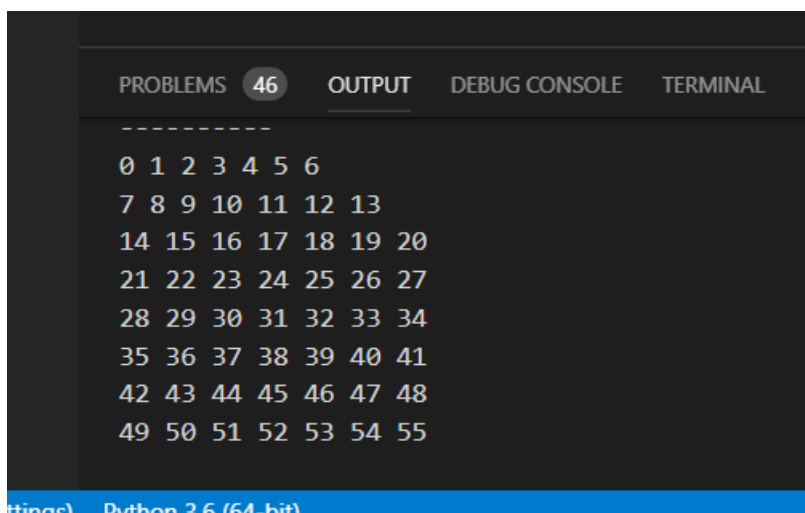


Good enough?

Actually, there's another great way to print these numbers, but it will only work if you run the script from VS Code and not from the Brickman interface. By some kind of magic, we can actually get this script which is running on the EV3 to print... to VS Code! You could argue that since we're no longer printing to the EV3 screen this explanation doesn't belong in this section about the brick but here we go anyway:

You can put **file=stderr** ('standard error') as a parameter of the print statement and it will then print to the Output panel of VS Code (as long as the script was launched for VS Code, of course). In order for this to work stderr has to be imported as shown below (**print_to_vscode.py** in the **ev3 python scripts** folder):

```
#!/usr/bin/env python3
from sys import stderr
from time import sleep
for i in range (8):
    for j in range (7):
        print(i*7+j, end=' ', file=stderr)
    print('', file=stderr)
sleep(8)
```



Printing to the VS Code output panel has many advantages over printing to the EV3 screen:

- The text is larger and easy to read

- There's not really any limit to how much you print for you can always scroll up to see text that has disappeared due to scrolling.
- You can copy and paste the text if you need to.
- You don't need a `sleep()` command to keep the text visible.

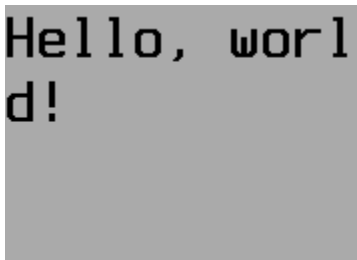
Of course, your script can print to both the EV3 screen AND VS Code if you want it to.

We've seen that by default the **`print()`** command prints very small text on the EV3 screen. Let's learn how to use bigger fonts. Many fonts are available, some bold, some italic, currently up to size 32 (32 pixels tall). Available fonts include, for example, one called 'Lat15-TerminusBold32x16'. Here is our 'Hello, world!' script, modified to use that larger font, along with a corresponding screenshot (**`print_large.py`** in the **ev3 python scripts** folder):

```
#!/usr/bin/env python3
from time import sleep
from os import system

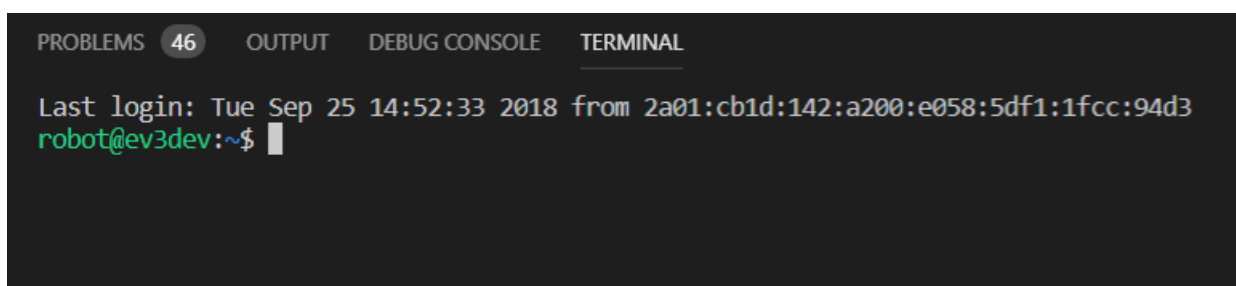
system('setfont Lat15-TerminusBold32x16')

print('Hello, world!')
sleep(5)
```



This font is MUCH bigger and more legible than the default font. In fact what we have done is create a new default font, so from now on the **`print()`** command will always use this new font until either it is changed with a new 'setfont' or else the EV3 is restarted. That means you can now run the original 'Hello, world!' script and it will use the new large default font.

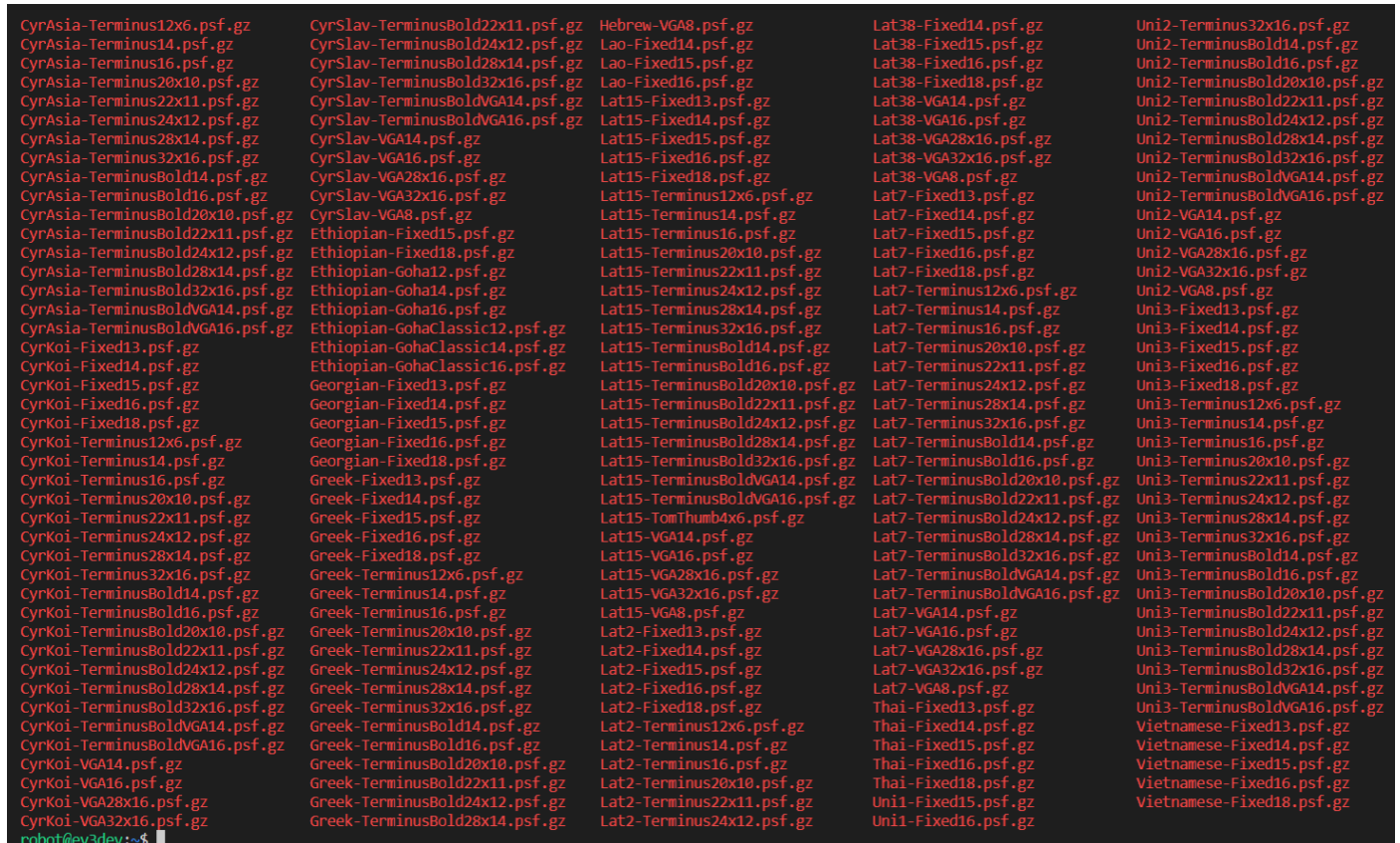
We'd probably prefer to use a slightly smaller font to avoid that text wrapping – what other fonts are available? That question gives me an excuse to introduce a very powerful feature of our VS Code workflow – the ability to begin a **'Secure SHell' (SSH)** session with just a single click. **SSH makes it possible to access a remote computer in a secure fashion.** You can establish an SSH connection between VS Code and the EV3 simply by right-clicking the green 'connected' dot in the EV3 device browser and choosing 'Open SSH Terminal'. You may need to enter your user name **'robot'** and the password **'maker'**. **Also, due to a bug, you may need to press the enter key as many as a dozen times before you can see a prompt or type anything in the terminal.** You should then see something like this:



You are now interacting with the Linux operating system on the EV3, so if you needed to use the SSH connection often you would have to know a lot about Linux. One of the great strengths of the VS Code is that you should NEVER need to use SSH, and even now we are only using it to get a feel for it and to get a list of fonts that can also be found online if you know where to look. Our interaction with Linux will be limited to typing one command:

```
ls /usr/share/consolefonts
```

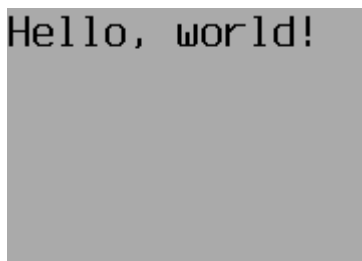
When you run that command you should see something like this:



```
CyrAsia-Terminus12x6.psf.gz  CyrSlav-TerminusBold22x11.psf.gz  Hebrew-VGA8.psf.gz  Lat38-Fixed14.psf.gz  Uni2-Terminus32x16.psf.gz
CyrAsia-Terminus14.psf.gz  CyrSlav-TerminusBold24x12.psf.gz  Lao-Fixed14.psf.gz  Lat38-Fixed15.psf.gz  Uni2-TerminusBold14.psf.gz
CyrAsia-Terminus16.psf.gz  CyrSlav-TerminusBold28x14.psf.gz  Lao-Fixed15.psf.gz  Lat38-Fixed16.psf.gz  Uni2-TerminusBold16.psf.gz
CyrAsia-Terminus20x10.psf.gz  CyrSlav-TerminusBold32x16.psf.gz  Lao-Fixed16.psf.gz  Lat38-Fixed18.psf.gz  Uni2-TerminusBold20x10.psf.gz
CyrAsia-Terminus22x11.psf.gz  CyrSlav-TerminusBoldVGA14.psf.gz  Lat15-Fixed13.psf.gz  Lat38-VGA14.psf.gz  Uni2-TerminusBold22x11.psf.gz
CyrAsia-Terminus24x12.psf.gz  CyrSlav-TerminusBoldVGA16.psf.gz  Lat15-Fixed14.psf.gz  Lat38-VGA16.psf.gz  Uni2-TerminusBold24x12.psf.gz
CyrAsia-Terminus28x14.psf.gz  CyrSlav-VGA14.psf.gz  Lat15-Fixed15.psf.gz  Lat38-VGA28x16.psf.gz  Uni2-TerminusBold28x14.psf.gz
CyrAsia-Terminus32x16.psf.gz  CyrSlav-VGA16.psf.gz  Lat15-Fixed16.psf.gz  Lat38-VGA32x16.psf.gz  Uni2-TerminusBold32x16.psf.gz
CyrAsia-TerminusBold14.psf.gz  CyrSlav-VGA28x16.psf.gz  Lat15-Fixed18.psf.gz  Lat38-VGA8.psf.gz  Uni2-TerminusBoldVGA14.psf.gz
CyrAsia-TerminusBold16.psf.gz  CyrSlav-VGA32x16.psf.gz  Lat15-Terminus12x6.psf.gz  Lat7-Fixed13.psf.gz  Uni2-TerminusBoldVGA16.psf.gz
CyrAsia-TerminusBold20x10.psf.gz  CyrSlav-VGA8.psf.gz  Lat15-Terminus14.psf.gz  Lat7-Fixed14.psf.gz  Uni2-VGA14.psf.gz
CyrAsia-TerminusBold22x11.psf.gz  Ethiopian-Fixed15.psf.gz  Lat15-Terminus16.psf.gz  Lat7-Fixed15.psf.gz  Uni2-VGA16.psf.gz
CyrAsia-TerminusBold24x12.psf.gz  Ethiopian-Fixed18.psf.gz  Lat15-Terminus20x10.psf.gz  Lat7-Fixed16.psf.gz  Uni2-VGA28x16.psf.gz
CyrAsia-TerminusBold28x14.psf.gz  Ethiopian-Goha12.psf.gz  Lat15-Terminus22x11.psf.gz  Lat7-Fixed18.psf.gz  Uni2-VGA32x16.psf.gz
CyrAsia-TerminusBold32x16.psf.gz  Ethiopian-Goha14.psf.gz  Lat15-Terminus24x12.psf.gz  Lat7-Terminus12x6.psf.gz  Uni2-VGA8.psf.gz
CyrAsia-TerminusBoldVGA14.psf.gz  Ethiopian-Goha16.psf.gz  Lat15-Terminus28x14.psf.gz  Lat7-Terminus14.psf.gz  Uni3-Fixed13.psf.gz
CyrKoi-Fixed13.psf.gz  Ethiopian-GohaClassic12.psf.gz  Lat15-Terminus32x16.psf.gz  Lat7-Terminus16.psf.gz  Uni3-Fixed14.psf.gz
CyrKoi-Fixed14.psf.gz  Ethiopian-GohaClassic14.psf.gz  Lat15-TerminusBold14.psf.gz  Lat7-Terminus20x10.psf.gz  Uni3-Fixed15.psf.gz
CyrKoi-Fixed15.psf.gz  Ethiopian-GohaClassic16.psf.gz  Lat15-TerminusBold16.psf.gz  Lat7-Terminus22x11.psf.gz  Uni3-Fixed16.psf.gz
CyrKoi-Fixed16.psf.gz  Georgian-Fixed13.psf.gz  Lat15-TerminusBold20x10.psf.gz  Lat7-Terminus24x12.psf.gz  Uni3-Fixed18.psf.gz
CyrKoi-Fixed18.psf.gz  Georgian-Fixed14.psf.gz  Lat15-TerminusBold22x11.psf.gz  Lat7-Terminus28x14.psf.gz  Uni3-Fixed20x10.psf.gz
CyrKoi-Terminus12x6.psf.gz  Georgian-Fixed15.psf.gz  Lat15-TerminusBold24x12.psf.gz  Lat7-Terminus32x16.psf.gz  Uni3-Terminus14.psf.gz
CyrKoi-Terminus14.psf.gz  Georgian-Fixed16.psf.gz  Lat15-TerminusBold28x14.psf.gz  Lat7-TerminusBold14.psf.gz  Uni3-Terminus16.psf.gz
CyrKoi-Terminus16.psf.gz  Georgian-Fixed18.psf.gz  Lat15-TerminusBold32x16.psf.gz  Lat7-TerminusBold16.psf.gz  Uni3-Terminus20x10.psf.gz
CyrKoi-Terminus20x10.psf.gz  Greek-Fixed13.psf.gz  Lat15-TerminusBoldVGA14.psf.gz  Lat7-TerminusBold20x10.psf.gz  Uni3-Terminus22x11.psf.gz
CyrKoi-Terminus22x11.psf.gz  Greek-Fixed14.psf.gz  Lat15-TerminusBoldVGA16.psf.gz  Lat7-TerminusBold22x11.psf.gz  Uni3-Terminus24x12.psf.gz
CyrKoi-Terminus24x12.psf.gz  Greek-Fixed15.psf.gz  Lat15-TomThumb4x6.psf.gz  Lat7-TerminusBold24x12.psf.gz  Uni3-Terminus28x14.psf.gz
CyrKoi-Terminus28x14.psf.gz  Greek-Fixed16.psf.gz  Lat15-VGA14.psf.gz  Lat7-TerminusBold28x14.psf.gz  Uni3-Terminus32x16.psf.gz
CyrKoi-Terminus32x16.psf.gz  Greek-Fixed18.psf.gz  Lat15-VGA28x16.psf.gz  Lat7-TerminusBold32x16.psf.gz  Uni3-TerminusBold14.psf.gz
CyrKoi-TerminusBold14.psf.gz  Greek-Terminus12x6.psf.gz  Lat15-VGA32x16.psf.gz  Lat7-TerminusBoldVGA14.psf.gz  Uni3-TerminusBold16.psf.gz
CyrKoi-TerminusBold16.psf.gz  Greek-Terminus14.psf.gz  Lat15-VGA8.psf.gz  Lat7-TerminusBoldVGA16.psf.gz  Uni3-TerminusBold20x10.psf.gz
CyrKoi-TerminusBold20x10.psf.gz  Greek-Terminus16.psf.gz  Lat2-Fixed13.psf.gz  Lat7-VGA14.psf.gz  Uni3-TerminusBold22x11.psf.gz
CyrKoi-TerminusBold22x11.psf.gz  Greek-Terminus20x10.psf.gz  Lat2-Fixed14.psf.gz  Lat7-VGA16.psf.gz  Uni3-TerminusBold24x12.psf.gz
CyrKoi-TerminusBold24x12.psf.gz  Greek-Terminus22x11.psf.gz  Lat2-Fixed15.psf.gz  Lat7-VGA28x16.psf.gz  Uni3-TerminusBold28x14.psf.gz
CyrKoi-TerminusBold28x14.psf.gz  Greek-Terminus24x12.psf.gz  Lat2-Fixed16.psf.gz  Lat7-VGA32x16.psf.gz  Uni3-TerminusBold32x16.psf.gz
CyrKoi-TerminusBold32x16.psf.gz  Greek-Terminus28x14.psf.gz  Lat2-Fixed18.psf.gz  Lat7-VGA8.psf.gz  Uni3-TerminusBoldVGA14.psf.gz
CyrKoi-TerminusBoldVGA14.psf.gz  Greek-Terminus32x16.psf.gz  Lat2-Terminus12x6.psf.gz  Thai-Fixed13.psf.gz  Uni3-TerminusBoldVGA16.psf.gz
CyrKoi-TerminusBoldVGA16.psf.gz  Greek-TerminusBold14.psf.gz  Lat2-Terminus14.psf.gz  Thai-Fixed14.psf.gz  Vietnamese-Fixed13.psf.gz
CyrKoi-VGA14.psf.gz  Greek-TerminusBold16.psf.gz  Lat2-Terminus16.psf.gz  Thai-Fixed15.psf.gz  Vietnamese-Fixed14.psf.gz
CyrKoi-VGA16.psf.gz  Greek-TerminusBold20x10.psf.gz  Lat2-Terminus18.psf.gz  Thai-Fixed16.psf.gz  Vietnamese-Fixed15.psf.gz
CyrKoi-VGA28x16.psf.gz  Greek-TerminusBold22x11.psf.gz  Lat2-Terminus20x10.psf.gz  Thai-Fixed18.psf.gz  Vietnamese-Fixed16.psf.gz
CyrKoi-VGA32x16.psf.gz  Greek-TerminusBold24x12.psf.gz  Lat2-Terminus22x11.psf.gz  Thai-Fixed20x10.psf.gz  Vietnamese-Fixed18.psf.gz
Greek-TerminusBold28x14.psf.gz  Lat2-Terminus24x12.psf.gz  Uni1-Fixed15.psf.gz  Uni1-Fixed16.psf.gz
```

As I said before, there are a LOT of fonts available, though most users will only be interested in the ones beginning 'Lat'. If you want to use one of these fonts, drop the .psf.gz from the end of the font name. The ones I usually use myself are:

- **Lat15-TerminusBold32x16** (which we already tried)
- **Lat15-TerminusBold28x14**
- **Lat15-TerminusBold24x12** which gives the result below when used with the 'Hello, world!' script:



```
Hello, world!
```

We've finished with the SSH terminal, so you can close it by typing **exit** and pressing the Enter key.

Display text with `text_pixels()` or `text_grid()`

Displaying text with these functions is more difficult than with **print()** but gives you more control, as you can specify where the text is to be placed . You can also display different fonts simultaneously, which you cannot do with **print()**. The text that you display with **text_pixels()** or **text_grid()** will not wrap at the right edge of the screen and will not scroll. Also, you need to **update()** the screen after using these functions otherwise nothing will be displayed on the screen.

```
text_pixels(text, clear_screen=True, x=0, y=0, text_color='black', font=None)
```

displays text such that the top-left corner of the text is in the pixel location given by x, y. Recall that the EV3 screen is 178 x 128 pixels and that the top-left pixel has coordinates (0,0).

```
text_grid(text, clear_screen=True, x=0, y=0, text_color='black', font=None)
```

displays 'text' starting at grid (x, y) where the grid is made of cells 8 pixels wide and 10 pixels tall such that the EV3 display is 22 columns wide and 12 rows tall. The size of the grid cells does not depend on the size of the chosen font. There is no fixed-width font available which has dimensions that match the grid cell size and therefore **I recommend that you avoid using this function and that you stick to using `text_pixels()` instead.** Even if a font were available that matches the grid cell dimensions it would be a very small font (size 10, meaning 10 pixels tall) and would be very hard to read.

Both these functions have a **clear_screen** parameter which is True by default, meaning that if your script writes a line of text followed by a second line then if you forget to set **clear_screen** to False for the second line then the first line will not print.

You must **update()** the screen in order for pending changes to be written to the screen. If you forget to update the screen then nothing at all will print!

text_pixels() and **text_grid()** do not use the same set of fonts as the **print()** function. There are many fonts available for use with **text_pixels()** and **text_grid()** and you can find the full list at <https://python-ev3dev.readthedocs.io/en/latest/display.html#ev3dev2.display.Display>. All the fonts are available in the following sizes (heights in pixels): 8, 10, 12, 14, 18, 24.

For maximum legibility and attractiveness, I recommend that your choice of *variable width* font should be the '**helvB**' set (that's 'B' for 'bold'). *Fixed width* fonts are less attractive but easier to work with. For *fixed width* fonts I recommend '**courB**' and '**lutBS**'. You can see what these fonts look like lower down the page. I recommend that you leave at least one pixel of vertical space between one line of text and the next. For example, if you choose to use a font with size 24 (24 pixels tall) then I recommend allowing 25 pixels of space for each line. In fact size 24 text allows for 5 lines of text to fit neatly on the EV3 screen if you print to these y coordinates: 0, 25, 50, 75, 100.

The fixed width fonts **courB24** and **lutBS24** have character widths of 15 pixels and 14 pixels respectively, meaning that 12 characters fit horizontally quite well on the EV3 screen, as you can see in the screenshots below.

Note that you can include the new line character '\n' in the strings that you ask these functions to print but I recommend against that because it gives a line spacing that I find too large.

Here is a script (**print_fonts.py** in the **ev3 python scripts** folder) that first prints a chosen font set in sizes 10, 14, 18 and 24 and then, 6 seconds later, prints a whole screenful of text at size 24. Use different values for 'style' to try different font sets. Underneath the script is the result of running the script with style set to **helvB**, **courB** and **lutBS** respectively.

```
#!/usr/bin/env python3
from ev3dev2.display import Display
from ev3dev2.sound import Sound
from time import sleep

lcd = Display()
sound = Sound()

def show_for(seconds):
    lcd.update()
    sound.beep()
    sleep(seconds)
```

```

lcd.clear()

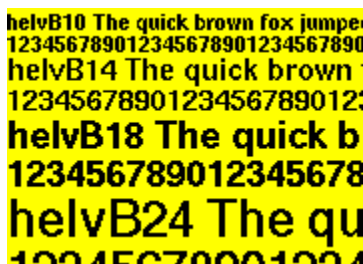
# Try each of these different sets:
style = 'helvB'
#style = 'courB'
#style = 'lutBS'

y_value = 0
str1 = ' The quick brown fox jumped'
str2 = '123456789012345678901234567890'
for height in [10, 14, 18, 24]:
    text = style+str(height)+str1
    lcd.text_pixels(text, False, 0, y_value, font=style+str(height))
    y_value += height+1 # short for y_value = y_value+height+1
    lcd.text_pixels(str2, False, 0, y_value, font=style+str(height))
    y_value += height+1
show_for(6)

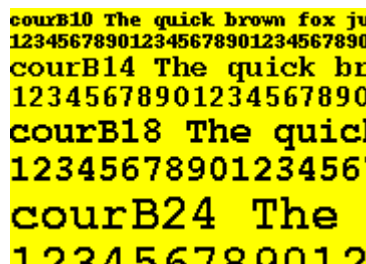
strings = [] # create an empty list
# Screen width can accommodate 12 fixed
# width characters with widths 14 or 15
#      123456789012
strings.append(style+'24 The')
strings.append('quick brown ')
strings.append('fox jumps   ')
strings.append('over the dog')
strings.append('123456789012')

for i in range(len(strings)):
    lcd.text_pixels(strings[i], False, 0, 25*i, font=style+'24')
show_for(6)

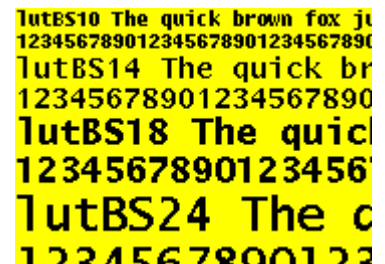
```



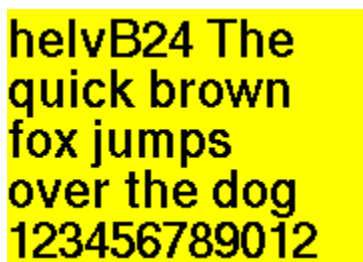
helvB10 The quick brown fox jumped
123456789012345678901234567890
helvB14 The quick brown
123456789012345678901234567890
helvB18 The quick b
123456789012345678
helvB24 The qu
123456789012345678901234567890



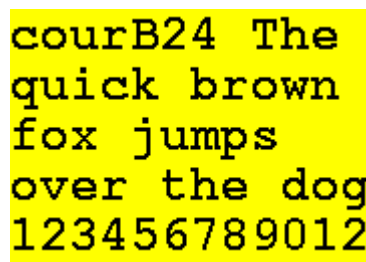
courB10 The quick brown fox ju
123456789012345678901234567890
courB14 The quick br
12345678901234567890
courB18 The quic
1234567890123456
courB24 The
123456789012345678901234567890



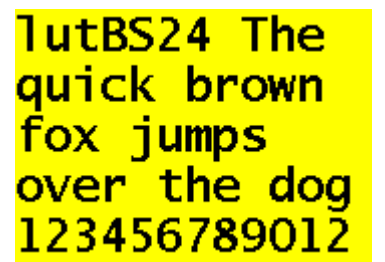
lutBS10 The quick brown fox ju
123456789012345678901234567890
lutBS14 The quick br
12345678901234567890
lutBS18 The quic
1234567890123456
lutBS24 The c
123456789012345678901234567890



helvB24 The
quick brown
fox jumps
over the dog
123456789012



courB24 The
quick brown
fox jumps
over the dog
123456789012



lutBS24 The
quick brown
fox jumps
over the dog
123456789012

Notes on the above script:

I usually include the parameter names (keywords) to make the code clearer but in some of the lines above I have omitted certain names because the lines would otherwise be too long. For example:

```

lcd.text_pixels(text, False, 0, y_value, font=style+str(height))

```

The arguments in bold are positional arguments whose meaning is known by the function according to their positions in the argument list, so the inclusion of parameter names is optional. However, I *had to* include the name 'font' - it could not be processed as a positional argument because it's in 'the wrong place'. In the function definition we saw

```
text_pixels(text, clear_screen=True, x=0, y=0, text_color='black', font=None)
```

so in order to be valid as a positional argument the **font** argument has to be the *sixth* argument but in my code it's the *fifth* argument since I have not specified a **text_color** value (I'm happy to use the default value, 'black'). Since the argument is not a valid positional argument when placed fifth, the name must be given, making it into a keyword argument, or 'kwarg'.

The script above demonstrates one way of building a list (called an 'array' in EV3-G).

Center the text

It's quite easy to *horizontally* center a line of text in a fixed-width font: for each character in the string, deduct half a character width from 89 (the horizontal midpoint of the EV3 screen). For example, if you want to center the string 'Hello' written in the font lutBS24 which has a character width of 14 then deduct $5 \times 7 = 35$ pixels from 89 to give $89 - 35 = 54$ so print to x coordinate 54.

To *vertically* center a *single* line of text the procedure would be similar: deduct half the font height from 63 (the vertical midpoint of the EV3 screen). For example, if you want to vertically center the string 'Hello' written in the font lutBS24 which has a height of 24 pixels then deduct 12 pixels from 63 to give 51, so print to y coordinate 51.

The following script is capable of calculating the y coordinates for printing *multiple* lines, one above the other. That's quite a bit more complicated - I'll let you figure it out for yourself. The script leaves one empty horizontal line of pixels between each line of text.

The script makes use of a standard Python function **wrap(text, width)** which wraps the **text** string so every line is at most **width** characters long. It returns a *list* of output lines i.e. a list of strings. For example

```
wrap('The quick brown fox jumps over the lazy dog', width=12)
```

returns ['The quick', 'brown fox', 'jumps over', 'the lazy dog']. In order to be able to use the **wrap()** function you must first import it with

```
from textwrap import wrap
```

Here's a script (**center_text.py** in the **ev3 python scripts** folder) that horizontally and vertically centers some text printed in several different fixed-width fonts. I know it's a rather complex script that I haven't explained it in detail - it might be worth a few minutes of your attention. Be sure to note the difference between **string** which is a *single* string and **strings** which a *list* of strings. The text centering could be very useful in future scripts so it's been made into a user-defined function that we can use elsewhere. The result of running the script is below it.

```
#!/usr/bin/env python3
from ev3dev2.display import Display
from textwrap import wrap
from time import sleep

lcd = Display()

def show_text(string, font_name='courB24', font_width=15, font_height=24):
    lcd.clear()
    strings = wrap(string, width=int(180/font_width))
    for i in range(len(strings)):
```

```

    x_val = 89-font_width/2*len(strings[i])
    y_val = 63-(font_height+1)*(len(strings)/2-i)
    lcd.text_pixels(strings[i], False, x_val, y_val, font=font_name)
lcd.update()

```

```
my_text = 'The quick brown fox jumps over the lazy dog'
```

```
show_text(my_text, 'courB14', 9, 14)
sleep(5)
```

```
show_text(my_text, 'lutBS14', 9, 14)
sleep(5)
```

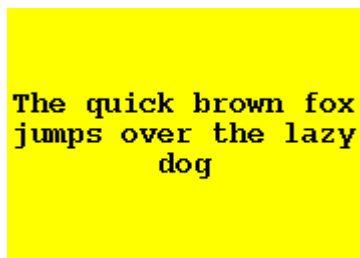
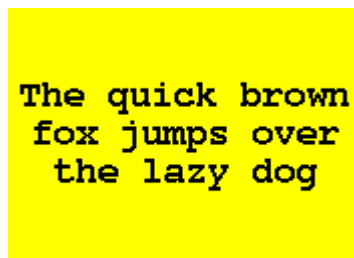
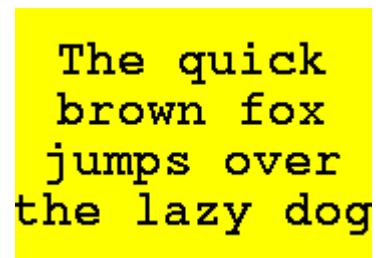
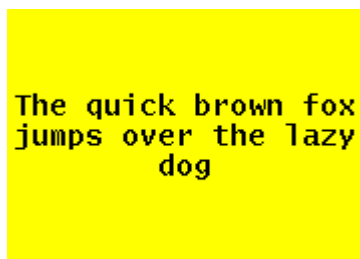
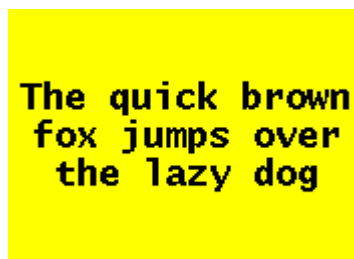
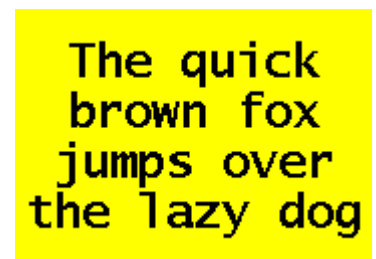
```
show_text(my_text, 'courB18', 11, 18)
sleep(5)
```

```
show_text(my_text, 'lutBS18', 11, 18)
sleep(5)
```

```
show_text(my_text, 'courB24', 15, 24)
# or simply show_text(my_text) since default values
# have been set for courB24
sleep(5)
```

```
show_text(my_text, 'lutBS24', 14, 24)
sleep(5)
```

Default values have been included in the function definition so that if you want to use the **courB24** font all you have to do is: `show_text('This is my text')`

Graphics

EV3 Python makes available a few easy-to-use commands for drawing simple shapes:

```
line(clear_screen=True, x1=10, y1=10, x2=50, y2=50, line_color='black', width=1)
```

Draw a line from (x1, y1) to (x2, y2)

```
circle(clear_screen=True, x=50, y=50, radius=40, fill_color='black', outline_color='black')
```

Draw a circle of radius 'radius' centered at (x, y)

```
rectangle(clear_screen=True, x1=10, y1=10, x2=80, y2=40, fill_color='black', outline_color='black')
```

Draw a rectangle where the top left corner is at (x1, y1) and the bottom right corner is at (x2, y2)


```
point(clear_screen=True, x=10, y=10, point_color='black')
```

Draw a single pixel at (x, y)

The following script (**smiley.py** in the **ev3 python scripts** folder) should be easy to understand. The result is underneath. With regard to the colors, know that **the EV3 LCD screen can only display four shades of grey** which correspond to these color names: black, grey, lightgrey, white. Stick to these color names because otherwise you will have an inconsistency between the LCD display (4 shades of grey) and screenshots (many more shades of grey).

```
#!/usr/bin/env python3
from ev3dev2.display import Display
from time import sleep

lcd = Display()

# Draw a circle of 'radius' centered at (x, y)
lcd.circle(clear_screen=False, x=89, y=64, radius=61, fill_color='lightgrey')
lcd.circle(False, x=65, y=45, radius=10, fill_color='black')
lcd.circle(False, x=113, y=45, radius=10, fill_color='black')

# Draw a line from (x1, y1) to (x2, y2)
lcd.line(False, x1=89, y1=50, x2=89, y2=80, line_color='grey', width=4)

# Draw a rectangle where the top left corner is at (x1, y1)
# and the bottom right corner is at (x2, y2).
lcd.rectangle(False, x1=69, y1=90, x2=109, y2=105, fill_color='grey')

# Draw a single pixel at (x, y)
lcd.point(False, x=65, y=45, point_color='white')
lcd.point(False, x=113, y=45, point_color='white')

lcd.update()
sleep(10)
```



Note that the instruction to draw the large circle must come before the others otherwise the large circle will obliterate whatever had been drawn previously.

There are many other **draw** commands that we could look at, such as those that draw arcs, ellipses or polygons, but this is meant to be a robotics course and robotics is more about motors and sensors than graphics so we won't spend any more time on graphics functions here. Anyway, there are many graphics functions that are not specific to EV3 Python so you may well be familiar with them already. To learn more about them, see my site ev3Python.com.

Display an image file

In addition to the draw commands there is one more graphics command we should learn about – the one that lets us display image files. EV3 Python can't display images in the RGF format used by EV3-G but in fact the EV3-G software includes BMP files as well as RGF files and EV3 Python *can* display those BMP files. You have the BMP files in multiple folders on your PC but to save you having to dig for them I've put them all together into a single folder called 'pics' which I've zipped and made available to you via a link at the bottom of this page:

sites.google.com/site/ev3devPython/learn_ev3_Python/screen/bmp-image-collection

You can see thumbnail images of all the available images on the same page.

To get the BMP images onto the EV3, unzip the file, open the 'pics' folder containing the images in VS Code and then click the 'send project to device' icon in the header of the EV3 device browser. Many scripts in this course assume that you have indeed got the BMP image collection in a folder called 'pics' inside your 'robot' folder. To display the BMP image file on the EV3 you need to import the **Image** module, open and name the image, and then paste the image to the screen. Here is a sample script (**display_image_file.py** in the **ev3 python scripts** folder) and the result:

```
#!/usr/bin/env python3
from ev3dev2.display import Display
from time import sleep
from PIL import Image

lcd = Display()

logo = Image.open('/home/robot/pics/Bomb.bmp')
lcd.image.paste(logo, (0,0))
lcd.update()
sleep(5)
```



Sound

The sound capabilities of EV3 Python are a lot more powerful than those of EV3-G. You can make the EV3 play a tune, using frequencies or musical notes, and you can make the EV3 speak any English phrase you choose. You can also play any of the standard Lego sounds as long as you download them in WAV format to a folder on the EV3. You can also download other WAV files that you find on the internet, and use them too.

In order to be able to use any of the sound functions, you must first import the **Sound** class and make an instance of it. All the sound functions have a **play_type** parameter which is 0 by default, meaning the playing of the sound will block (pause) the program until the sound completes. You can specify **play_type=1** if you don't want the program to block like that, and **play_type=2** if you want the sound to *loop*. **play_type=2** blocks the program but never completes so the only way to stop the loop is to kill the program, which means this option is not very useful. For all the sound functions that have a **duration** parameter, the duration is in seconds. There is one exception: the old **tone()** function uses durations in milliseconds.

Play a beep

```
beep(play_type=0)
```

Example:

```
#!/usr/bin/env python3
from ev3dev2.sound import Sound

sound = Sound()

sound.beep()
```

Play a SINGLE tone

To play a SINGLE tone of given frequency (Hz) and duration (seconds), use

```
play_tone(frequency, duration, play_type=0)
```

Example: To play a single tone of frequency 1500 Hz for 2 seconds:

```
sound.play_tone(1500, 2)
```

Play a SEQUENCE of tones

Currently (February 2019) the new **play_tone()** function cannot play a tone sequence but the old **tone()** function can. Therefore, to play a SEQUENCE of tones, use **tone(tone_sequence, play_type=0)**

The **tone_sequence** parameter is a *list of tuples*, where each tuple contains up to three numbers. You should know by now that a tuple is like a list except that a list can be modified and a tuple cannot. The first number is frequency in Hz, the second is duration in milliseconds, and the third is delay in milliseconds between this and the next tone in the sequence. Example:

```
#!/usr/bin/env python3
from ev3dev2.sound import Sound

sound = Sound()

# Play a 200Hz tone for 2 seconds and then wait 0.4 seconds
# before playing the next tone, 800Hz for 1 second
# followed by a 3 second delay
sound.tone([(200, 2000, 400), (800, 1000, 3000)])
```

Play a WAV sound file

To play a WAV sound file, use **play_file(wav_file, play_type=0)**. You cannot use mp3 files with the **play_file()** command, nor can you use Lego sound files in the RSF or RSO format. But you can download from the bottom of this page sites.google.com/site/ev3devPython/learn_ev3_Python/sound a zip file containing all of the official EV3 sounds in WAV format. To get the sound files into the correct place on the EV3, first unzip the file to a convenient location on your PC (all the sounds are in the same folder, not separated into different folders as in EV3-G). Then open the folder containing the sound files in VS Code, then click the 'Send Project to Device' icon in the header of the EV3 Device Browser at the bottom left of the VS Code window. That will download the 'sounds' folder and its contents into your 'robot' folder on the EV3. Here is a list of the available sounds, not including the .wav extensions in the file names:

Activate	Boing	Dog sniff	Game over	Laser	One	Slide load	T-rex roar
Air release	Boo	Dog whine	General alert	Laughing 1	Ouch	Smack	Ten
Airbrake	Bravo	Down	Go	Laughing 2	Overpower	Snake hiss	Thank you
Analyze	Brown	Download	Good job	Left	Power down	Snake rattle	Three
Arm 1	Cat purr	Drop load	Good	Lift load	Ratchet	Snap	Tick tack
Arm 2	Cheering	EV3	Goodbye	MINDSTORMS	Ready	Sneezing	Touch
Arm 3	Click	Eight	Green	Magic wand	Red	Snoring	Turn
Arm 4	Color	Elephant call	Hello	Morning	Right	Sonar	Two
Backing alert	Confirm	Error alarm	Hi	Motor idle	Searching	Sorry	Uh-oh
Backwards	Connect	Error	Horn 1	Motor start	Servo 1	Speed down	Up
Black	Crunching	Fanfare	Horn 2	Motor stop	Servo 2	Speed idle	Walk
Blip 1	Crying	Fantastic	Insect buzz 1	Nine	Servo 3	Speed up	White
Blip 2	Detected	Five	Insect buzz 2	No	Servo 4	Speeding	Yellow
Blip 3	Dog bark 1	Flashing	Insect chirp	Object	Seven	Start up	Yes
Blip 4	Dog bark 2	Forward	Kung fu	Okay	Shouting	Start	Zero
Blue	Dog growl	Four	LEGO	Okey-dokey	Six	Stop	

If you want to download additional WAV files to use with the EV3 Python then check out wavsource.com, a good source for free WAV sound files.

Once you have inside your 'robot' folder a 'sounds' folder containing the EV3 sound files in WAV format you can try running the following script called **play_wav_file.py** which is in the **ev3 python scripts** folder. A beep sound is included to prove to you that the program will pause (block) until the sound has finished playing before beeping:

```
#!/usr/bin/env python3
from ev3dev2.sound import Sound
from time import sleep

sound = Sound()

sound.play_file('/home/robot/sounds/MINDSTORMS.wav')
sound.beep()
```

Text to Speech

Just use **speak(text, play_type=0)**. Don't use a string that ends in a period otherwise the robot may say 'dot'. Either omit the period or put a space after it.

Example Script

Try running this script (**sound.py** in the **ev3 python scripts** folder) which demonstrates all of the sound functions described above.

```
#!/usr/bin/env python3
from ev3dev2.sound import Sound
from time import sleep

sound = Sound()

#play a standard beep
sound.beep()
sleep(2) # pause for 2 seconds
# Play a SINGLE 2000 Hz tone for 1.5 seconds
sound.play_tone(2000, 1.5)
sleep(2)
# Play a SEQUENCE of tones
sound.tone([(200, 2000, 400), (800, 1800, 2000)])
sleep(2)
# Play a 500 Hz tone for 1 second and then wait 0.4 seconds
# before playing the next tone
# Play the tone three times
# [(500, 1000, 400)] * 3 is the same as
# [(500, 1000, 400), (500, 1000, 400), (500, 1000, 400)]
sound.tone([(500, 1000, 400)] * 3)
sleep(2)
#text to speech
sound.speak('Hello, my name is E V 3!')
```

Other sound functions

There are other sound functions available, such as **play_note()** and **play_song()**. These use musical notes rather than frequencies, and I won't describe them here, but you can read about them in the official documentation:

Python-ev3dev.readthedocs.io/en/ev3dev-stretch/sound.html

Buttons

There are various ways of getting scripts to interact with the pressing of the buttons on the EV3 intelligent brick. A commonly used method it to use the **process()** function to detect changes in the pattern of which buttons are

pressed. If the function detects any changes then it calls the corresponding 'button event handlers', as demonstrated in the script below (**buttons.py** in the **ev3 pythons scripts** folder). For example, if it detects that the left button has just been pressed then it triggers a 'left button state change' event and a 'button change' event. It also assigns a value of True to the parameter '**state**' if the button is pressed and a value of False if the button is released. Event handlers respond to the events. For example, if the left button is pressed then the 'left button state change' event will trigger the **on_left** event handler which will call the function 'left' (the function does not have to have this name). Note the need to import the Button class and to create an instance of that class.

```
#!/usr/bin/env python3
from ev3dev2.button import Button
from time import sleep

btn = Button()

# Do something when state of any button changes:
def left(state):
    if state:
        print('Left button pressed')
    else:
        print('Left button released')

def right(state): # neater use of 'if' follows:
    print('Right button pressed' if state else 'Right button released')

def up(state):
    print('Up button pressed' if state else 'Up button released')

def down(state):
    print('Down button pressed' if state else 'Down button released')

def enter(state):
    print('Enter button pressed' if state else 'Enter button released')

def backspace(state):
    print('Backspace button pressed' if state else 'Backspace button released')

btn.on_left = left
btn.on_right = right
btn.on_up = up
btn.on_down = down
btn.on_enter = enter
btn.on_backspace = backspace

# This loop checks buttons state continuously (every 0.01s).
# If the new state differs from the old state then the appropriate
# button event handlers are called.
while True:
    btn.process() # Check for currently pressed buttons.
    sleep(0.01)

# If running this script from VS Code, press the stop button to quit
# If running from Brickman, long-press backspace button to quit
```

Video 2C: Sensors

This section explains how to use all the sensors that come with either the home or education version of the EV3 kit but don't forget that the ultrasonic and gyro sensors are included only with the education version and the infrared sensor (and the infrared remote control/beacon) is included only with the home kit. These sensors are available for

purchase as optional extras. As far as possible, I propose similar scripts for the ultrasonic and infrared sensors, both of which are capable of measuring the proximity of objects (the ultrasonic sensor is capable of far greater accuracy).

Not that EV3 Python is compatible not only with all the EV3 sensors but also with the older NXT sensors except that the NXT light sensor cannot be used to detect colors. EV3 Python is also compatible with many non-Lego sensors but that is not discussed here for this course only covers what can be achieved with standard EV3 kit.

EV3 and/or NXT sensors can be attached to any of the EV3's four sensor ports but I recommend that you adhere to the useful convention that they should be attached as follows:

port 1 = touch	port 2 = gyro	port 3 = color	port 4 = infrared or ultrasonic
----------------	---------------	----------------	---------------------------------

This convention is useful because it means you don't have to reconnect the sensors as you move from one EV3 tutorial to another.

If you don't attach more than one sensor of a particular type, then it will not matter which port a sensor is plugged in to. You don't need to include any reference in your code to the sensor port number - your program will just work. However, if you connect more than one sensor of a given type then you need to specify the port number with INPUT_1, INPUT_2 etc.

If you have previously used the older sensor commands of version 1 of EV3 Python, you'll be happy to learn that in the new EV3 Python v2 library it is no longer necessary to set sensor 'modes'. In version 2 this happens automatically according to what function you are using. For example, if you use the property **ambient_light_intensity** with the EV3 color sensor it will automatically be put into ambient light measuring mode. Other changes include the introduction of a useful **wait_for_bump()** function for the touch sensor and a **wait_until_angle_changed_by(delta)** function for the gyro sensor.

If you try to create an instance of a sensor class but the corresponding sensor is not actually attached to the brick then you will receive an error informing you of the problem. Therefore you need not (in fact you MUST not) use the '**connected**' property that was used with version 1 of EV3 Python.

Touch sensor

The touch sensor has a property **is_pressed** which indicates whether the current touch sensor is being pressed. It's a Boolean property, meaning it can be either True or False. The touch sensor also has a **wait_for_bump()** function which waits for the touch sensor to be 'bumped' (pressed and then released). Here's a little script to demonstrate both the property and the function. This **touch_sensor.py** script can be found in the **ev3 python scripts** folder. It waits for the touch sensor to be bumped, then beeps, then prints the value of the **is_pressed** property every half second to both the EV3 screen and VS Code (if launched from VS Code). Stop the program by clicking the stop button in VS Code or pressing the Backspace button on the EV3.

```
#!/usr/bin/env python3
from ev3dev2.sensor.lego import TouchSensor
from ev3dev2.sound import Sound
from sys import stderr
from time import sleep

ts = TouchSensor()
sound = Sound()

ts.wait_for_bump()
sound.beep()

while True:
    print(ts.is_pressed)    # Print to EV3 screen
    print(ts.is_pressed, file=stderr) # Print to VS Code terminal
```

```
sleep(0.5)
```

In addition to **wait_for_bump()**, the touch sensor also has **wait_for_pressed()** and **wait_for_released()** functions, just like the brick buttons.

Color Sensor

The EV3 'color sensor' should not be confused with the NXT 'light sensor'. The color sensor has four basic properties (as usual, I will only be presenting a selection of properties and functions that are of interest to beginners – see the official documentation at [Python-ev3dev.readthedocs.io/en/ev3dev-stretch/sensors.html](https://python-ev3dev.readthedocs.io/en/ev3dev-stretch/sensors.html) or [ev3Python.com](https://ev3python.com) for more.)

reflected_light_intensity is a measure of the reflected light intensity expressed as an integer percentage (0-100). This mode of operation is indicated by a red light on the sensor.

ambient_light_intensity returns an integer in the range 0-100 that represents the ambient light intensity. The light on the sensor is dimly lit blue.

color represents the color detected by the sensor, represented by an integer:

0: No color 1: Black 2: Blue 3: Green 4: Yellow 5: Red 6: White 7: Brown

Color detection works best if the colors are close to the official Lego colors (the colors of the Lego bricks) and if the object is about 8mm away from the sensor.

color_name returns a string: 'NoColor', 'Black', 'Blue', etc.

Here's an example script that plays tones whose frequencies depend on the ambient light intensity. The (rather clever) formula $110 \cdot 2^{(cl.ambient_light_intensity/12)}$ generates frequencies that are part of the standard '440Hz=A' musical scale. The double asterisk (**) is used in Python to represent powers.

```
#!/usr/bin/env python3
from ev3dev2.sensor.lego import ColorSensor
from ev3dev2.sound import Sound
from time import sleep

cl = ColorSensor()
sound = Sound()

while True:
    freq = 110*2**(cl.ambient_light_intensity/12)
    sound.play_tone(frequency=freq, duration=0.5)
    sleep(0.1)
```

We'll see examples of the other three properties in use in part 3, where we start connecting the pieces that we have learnt about in part 2.

Gyro Sensor

The gyro sensor is included in the education version of the EV3 kit but not the home version. It can be bought separately.

Whenever you use the gyro sensor, make sure that the sensor is absolutely still when the brick is booting up or when the gyro is plugged in otherwise its readings may wander away from the correct values.

Here are the three properties of most interest to beginners:

angle is the number of degrees that the sensor has been rotated since the gyro was powered up (i.e. since the brick was powered up or since the gyro was plugged in). Clockwise rotation is considered positive, so if the angle is initially

zero and then the sensor is rotated two rotations counterclockwise then angle will equal -720 degrees. There is currently no way to reset the gyro to zero degrees other than to disconnect and reconnect the sensor.

rate is the rate at which the sensor is rotating, in degrees/second.

angle_and_rate You can probably guess this one! This script, called **gyro_angle_and_rate.py**, can be found in the **ev3 python scripts** folder. This script makes the robot turn to the right and then, while the robot is turning, prints the **angle** and **rate** readings to the VS Code terminal panel. Note that **block** is set to False when the motors are set in motion so that the program can continue running code. If **block** were not set to False then the **angle** and **rate** readings would only be printed after the robot has stopped turning.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import GyroSensor
from sys import stderr
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
gyro = GyroSensor()

steer_pair.on_for_seconds(steering=100, speed=20, seconds=6, block=False)

for i in range(15):
    print('Angle and Rate: ' + str(gyro.angle_and_rate), file=stderr)
    sleep(0.5)
```

Here are my results from running the above script. Note that this property returns values in a tuple.

Angle and Rate: (365, 1)
Angle and Rate: (390, 76)
Angle and Rate: (430, 83)
Angle and Rate: (476, 82)
Angle and Rate: (519, 86)
Angle and Rate: (562, 86)
Angle and Rate: (607, 89)
Angle and Rate: (647, 92)
Angle and Rate: (691, 92)
Angle and Rate: (736, 81)
Angle and Rate: (782, 85)
Angle and Rate: (828, 83)
Angle and Rate: (871, 15)
Angle and Rate: (869, 0)
Angle and Rate: (869, 0)

As you can see, the angle increases steadily and the rate of turn is fairly steady (between 81 and 92 degrees per second) except when the robot is starting to turn or stopping (in bold).

There is also a function, **wait_until_angle_changed_by(degrees)**, which makes the program wait until the angle has changed by a specified amount. This function doesn't care in which direction the sensor is turned, therefore degrees should always be positive. Here's a script, **gyro.py**, that makes the robot rotate until it has turned though 45°, then moves forward two wheel rotations.

```
#!/usr/bin/env python3
from ev3dev2.motor import OUTPUT_B, OUTPUT_C, MoveTank
from ev3dev2.sensor.lego import GyroSensor
from time import sleep
```

```

gyro = GyroSensor()
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)

# Start the left motor with speed 30% to initiate a medium turn right.
tank_pair.on(left_speed=30, right_speed=0)

# Wait until the gyro sensor detects that the robot has turned
# (at least) 45 deg in either direction
gyro.wait_until_angle_changed_by(45)

# Robot moves straight ahead with speed 50% until the wheels
# have turned through two rotations
tank_pair.on_for_rotations(left_speed=50, right_speed=50, rotations=2)

```

Ultrasonic Sensor

As you know, the ultrasonic sensor works by using the principle of SONAR (SOund NAVigation Ranging) which is that by timing how long it takes for an ultrasonic pulse to travel to an object and back it is possible to calculate the distance to the object with reasonable precision. Ultrasound is used rather than audible sound not just because we can't hear it (and be bothered by it) but also because ultrasound can more easily be formed into a beam than normal sound.

distance_centimeters is a measure of the distance detected by the sensor in centimeters, up to a maximum of 255.0 cm, with one decimal place of precision.

distance_inches is a measure of the distance detected by the sensor in inches, up to a maximum of 100.0 inches, with one decimal place of precision.

Here is a script (**us_sensor.py** in the **ev3 python scripts** folder) that makes both LED pairs go steady red when the object is closer than 40 cm and steady green when the object is further away. It also prints the measured distance to the VS Code terminal. The line **leds.all_off()** is needed because without it the program will initially run with *pulsing* green LEDs if the object is further than 40cm away.

```

#!/usr/bin/env python3
from ev3dev2.sensor.lego import UltrasonicSensor
from ev3dev2.led import Leds
from sys import stderr
from time import sleep

us = UltrasonicSensor()
leds = Leds()

leds.all_off() # Needed, to stop the LEDs flashing
while True:
    if us.distance_centimeters < 40: # to detect objects closer than 40cm
        # In the above line you can also use inches: us.distance_inches < 16
        leds.set_color('LEFT', 'RED')
        leds.set_color('RIGHT', 'RED')
    else:
        leds.set_color('LEFT', 'GREEN')
        leds.set_color('RIGHT', 'GREEN')
    print(us.distance_centimeters, file=stderr)
    sleep (0.5)

```

Ultrasound, unlike normal sound, can rather easily be made into a *beam*. In fact it can even be used to make images, as you know from having seen ultrasound images of babies in the womb. To get a feel for how narrow the ultrasound beam of the EV3 sensor is, I moved the



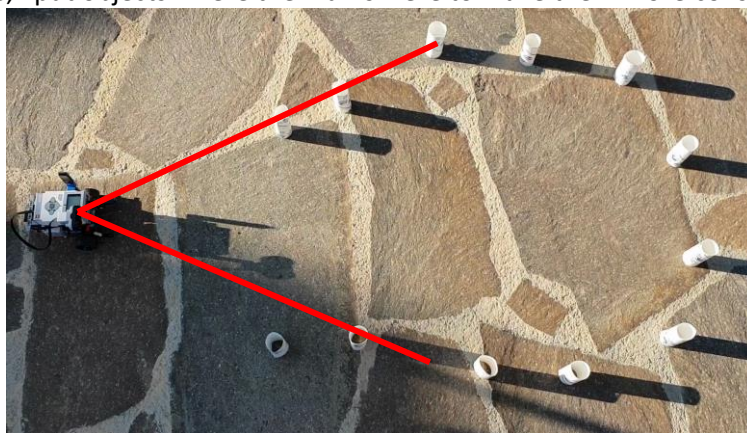
tube towards the sensor at various angles and marked the floor where the sensor was first able to pick up a reflection from the tube. The script I used is very simple – it just beeps when it detects a distance of less than 255cm. If an object is closer than 255cm but the sensor returns 255cm then obviously that means that it has not detected the reflected ultrasound pulse.

```
#!/usr/bin/env python3
from ev3dev2.sensor.lego import UltrasonicSensor
from ev3dev2.sound import Sound
from time import sleep

us = UltrasonicSensor()
sound = Sound()

while True:
    if us.distance_centimeters < 255:
        sound.beep()
        sleep (0.5)
```

Then, after trying all angles, I put objects where the marks were to make them more conspicuous. Here is the result:



It's nice to be able to visualize the sensitivity of the ultrasonic sensor in this way – it's clear that the beam is not narrow like a laser beam – we can say that in my test it spans about 45 degrees and we should keep this in mind in future projects.

Infrared Sensor (including Remote Control functions)

The **proximity** measure returned by the IR sensor can be compared to the **distance** measure of the ultrasonic sensor, except that **proximity** cannot be used to obtain a distance value that is at all *accurate*. That's because **proximity** is calculated based on the intensity of the reflected IR radiation - the stronger the reflection the smaller the proximity value. Therefore the proximity value will be much bigger for a small black object (weak reflection) than for a large white object in the same location (strong reflection). For largish light-colored objects the **proximity** value can be VERY roughly converted to a distance in cm by multiplying by 0.7 and the distance in cm can be VERY roughly converted to a **proximity** value by multiplying by 1.4, not forgetting that the proximity value is always in the range 0-100. You can experiment to get more accurate conversion factors for the particular object and distance ranges that you are using.

Note that in EV3-G 'proximity' can have two different meanings: it can refer to the proximity of an *object* (in 'measure>proximity' mode) or it can refer to the proximity of the *IR beacon* (in 'measure>beacon>proximity' mode). In EV3 Python **proximity** refers always to *object* proximity and the closeness of a *beacon* is called **distance** (see below).

Here is a script that mirrors the behavior of the previous script except that this one uses the IR sensor rather than the US sensor. The line **leds.all_off()** is needed because without it the program will initially run with *pulsing* green LEDs if the object is further than 40cm away.

```
#!/usr/bin/env python3
```

```

from ev3dev2.sensor.lego import InfraredSensor
from ev3dev2.led import Leds
from sys import stderr
from time import sleep

ir = InfraredSensor()
leds = Leds()

leds.all_off() # Needed, to stop the LEDs flashing
while True:
    if ir.proximity < 40*1.4: # To convert cm to proximity, multiply by 1.4
        leds.set_color('LEFT', 'RED')
        leds.set_color('RIGHT', 'RED')
    else:
        leds.set_color('LEFT', 'GREEN')
        leds.set_color('RIGHT', 'GREEN')
    print(str(int(ir.proximity*0.7)) + ' cm approx', file=stderr)
    # To convert proximity to cm, multiply by 0.7.
    # Distance values are VERY approximate.
    sleep (0.5)

```

Beacon and remote control functions

distance(channel=1) returns distance (as an integer, 0 to 100) to the beacon on the given channel. It returns None when the beacon is not found.

heading(channel=1) returns the approximate heading in degrees (-25 to +25) to the beacon on the given channel.

heading_and_distance(channel=1) returns heading and distance to the beacon on the given channel as a tuple.

top_left/bottom_left/top_right/bottom_right(channel=1) checks if the corresponding button on the remote control is pressed and returns True or False.

beacon(channel=1) checks if the beacon button on the remote control is pressed and returns True or False.

buttons_pressed(channel=1) returns a list of currently pressed buttons. It will return (as a list of strings) a maximum of two buttons pressed (if you press more than two buttons then it returns an empty list). It correctly returns 'beacon' if ONLY the beacon button is pressed but if the beacon button is pressed at the same time as any other button then it returns an empty list.

process() checks for currently pressed buttons. If the new state differs from the old state, it calls the appropriate button event handlers.

Here below is a script (**remote_control.py** in the **ev3 python scripts** folder) that uses **process()** and button event handlers to remote control the robot. The **process()** function checks for currently pressed buttons and then, if the new state differs from the old state, calls the appropriate button event handler. The event handlers themselves must be associated with functions you have defined, as in the script below. The **process()** function can only deal with presses of one button at a time but that should be sufficient for you as a beginner. If you want to also be able to use the simultaneous pressing of *pairs* of buttons then please check my site ev3Python.com.

The following script allows the presses of the buttons on the remote control to control a robot as follows:

- top left: go forwards
- bottom left: go backwards
- top right: turn right on the spot
- bottom right: turn left on the spot

The **process()** function in the script below does not specify a channel number and therefore the default channel, channel 1, will be used. Therefore you must make sure the remote control is set to channel 1.

```
#!/usr/bin/env python3
from ev3dev2.motor import OUTPUT_B, OUTPUT_C, MoveSteering
from ev3dev2.sensor.lego import InfraredSensor
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
ir = InfraredSensor() # Set the remote to channel 1

def top_left_channel_1_action(state):
    if state: # if state == True (pressed)
        steer_pair.on(steering=0, speed=40)
    else:
        steer_pair.off()

def bottom_left_channel_1_action(state):
    if state:
        steer_pair.on(steering=0, speed=-40)
    else:
        steer_pair.off()

def top_right_channel_1_action(state):
    if state:
        steer_pair.on(steering=100, speed=30)
    else:
        steer_pair.off()

def bottom_right_channel_1_action(state):
    if state:
        steer_pair.on(steering=-100, speed=30)
    else:
        steer_pair.off()

# Associate the event handlers with the functions defined above
ir.on_channell_top_left = top_left_channel_1_action
ir.on_channell_bottom_left = bottom_left_channel_1_action
ir.on_channell_top_right = top_right_channel_1_action
ir.on_channell_bottom_right = bottom_right_channel_1_action

while True:
    ir.process()
    sleep(0.01)
```

We haven't seen any scripts yet showing how to work with the remote control/beacon in *beacon* mode rather than *remote control* mode but we will see a couple of examples in in part 3.

That brings us to the end of part 2 where we looked at how EV3 Python can interact with the different types of hardware that come with the EV3 education kit and/or the EV3 home kit. In part 2 we worked mainly with only one type of hardware at a time. In part 3 we'll start putting the pieces together to make more interesting scripts, and in the final part, part 4, we'll look at some even more complex scripts.

Remember that this course is essentially for beginners and doesn't by any means cover all the functions and properties that are available in EV3 Python3 version 2. To learn about functions not discussed here, see the official documentation at [Python-ev3dev.readthedocs.io/en/ev3dev-stretch/](https://python-ev3dev.readthedocs.io/en/ev3dev-stretch/) and my site ev3Python.com.

PART 3: Putting the pieces together

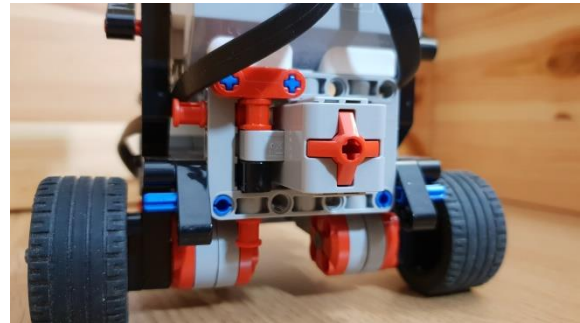
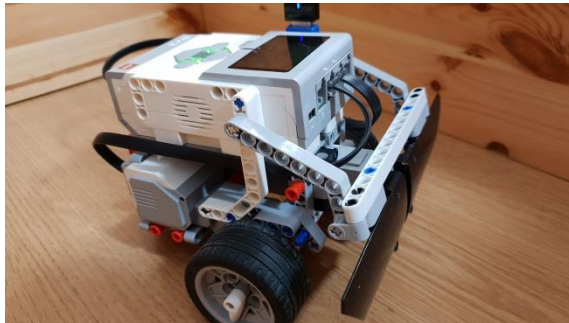
Video 3: Putting the pieces together (there is only one video in this part)

In this section I will assume that you have carefully worked through part 2 and no longer need much guidance as to how the various EV3 Python functions work. We'll be connecting the various components to make scripts that are more interesting and more complex than those seen in part 2. The projects are presented roughly in order of difficulty. Buckle up!

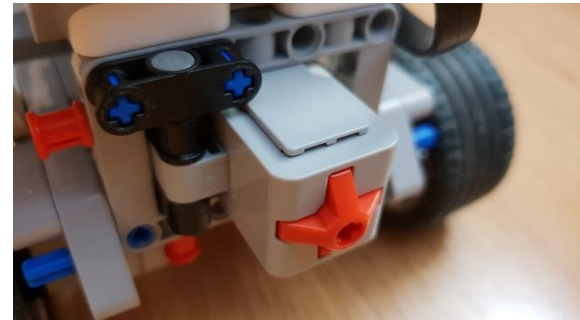
Collide, back up, turn and continue

In this little project we will use a single touch sensor to detect when our robot has collided with a wall or barrier. Then the robot will back up, turn a little, and continue on its way. In fact there's no reason why the robot should not turn *while it backs up*, so let's do that. We'll play a sound 'Backing Alert' as the robot starts to back up, and we'll override the default value 0 of **play_type** by setting it to 1 so that the script does not wait until the sound file has finished playing before starting to back up. Our robot needs to be given a bumper attachment of some kind, and the touch sensor needs to be placed so that a push on the bumper becomes a push on the touch sensor. **You can find instructions for making the bumper in the PDF files associated with this lesson.** The bumper attachment, like the other attachments described in this course, is designed to be attached to the 'educator vehicle' model that you have assembled, either with the education kit or the home kit. The photographs show how to attach the bumper and touch sensor to the model.

Home version:



Education version:



Once you've installed the bumper attachment correctly, try the following script called **collide_reverse_turn.py** which you can find in the **part3** folder. Note that in this case I am proposing the same script for both versions so the home version will move more slowly due to its smaller wheels. Feel free to adjust the arguments for the backing up move.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import TouchSensor
from ev3dev2.sound import Sound
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
ts = TouchSensor()
sound = Sound()

while True: # forever
```



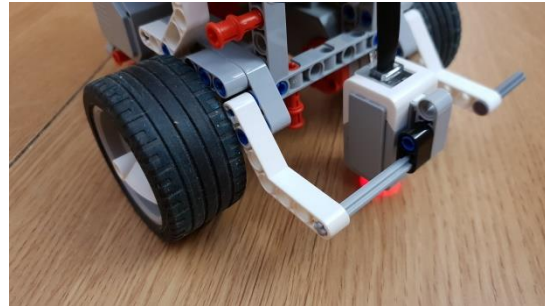
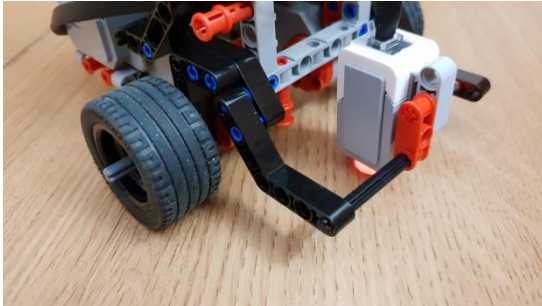
```

steer_pair.on(steering=0, speed=30)
ts.wait_for_pressed()
# play_type=1 means 'Play the sound without blocking the program'
sound.play_file('/home/robot/sounds/Backing alert.wav',play_type=1)
# Back up along a curved path
steer_pair.on_for_rotations(steering=-20, speed=-25, rotations=3)

```

Line Follower

Following a line is a classic robot application. We'll use a single color sensor to try to make our robot follow a thick black line on the floor. Here's a possible way to attach the sensor (home version at left, education version at right):



You've probably seen programs like this before, so you'll know that there's a bit more to it than just 'following the line'. The problem is that if the color sensor detects that it has moved off the line (because the light reflected from the floor has become brighter) that is not enough to tell us whether the robot has to turn left or right to get back on the line because the robot could have wandered off either way. Therefore the trick to line following with a single sensor is to *follow one edge of the line* – it doesn't matter which one. Let's say you decide to follow the right edge of the line. Then if the sensor detects that it is over a white surface then the robot needs to curve left and if the sensor detects a dark surface then the robot should curve right to approach the edge again. Once we understand all that then it's not too hard to write the code. Be sure that the sensor is around 4mm above the surface for the best results. The black line should ideally be at least 2cm wide – you might want to try using black electrical insulating tape. Don't make the curves in the line too sharp for that makes it difficult to get good results. The sharper the curves in the line, the sharper the turns that the EV3 needs to make, of course. Here's some code to get you started. This script, called **line_follower.py**, can be found in the **part3** folder. If you are using the education version of the EV3 then don't forget to change the value of the wheel factor, wf, to 0.77 to compensate for its larger wheels.

```

#!/usr/bin/env python3
# This script will follow the RIGHT EDGE of a black line.
from ev3dev2.motor import MoveTank, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import ColorSensor
from time import sleep

# Connect an EV3 color sensor to any sensor port.
cl = ColorSensor()
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)

wf = 1 # Use wheel factor = 1 for home version and 0.77 for edu version.
while True:    # forever
    if cl.reflected_light_intensity<30:    # weak reflection so over black line
        # medium turn right
        tank_pair.on(left_speed=50*wf, right_speed=15)
    else:    # strong reflection (>=30) so over white surface
        # medium turn left
        tank_pair.on(left_speed=15, right_speed=50*wf)
    sleep(0.02) # wait for 0.02 seconds

```

The above script is nice and short but doesn't do a brilliant job – do some research and you can learn how to make line follower scripts that work much better than this at the expense of more complex code.

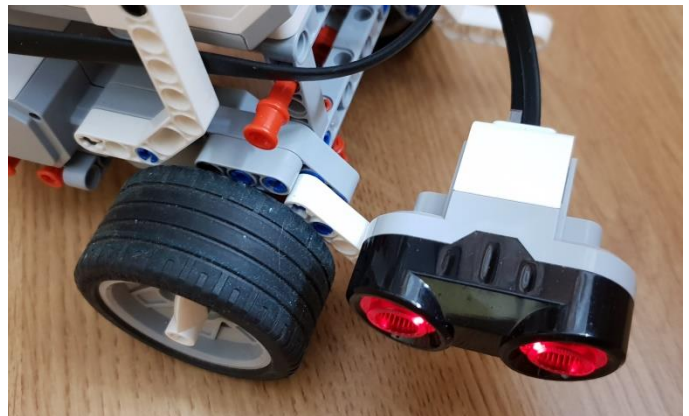
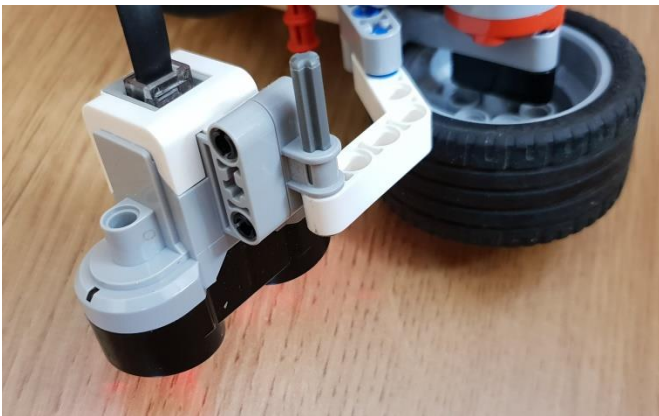
Wall follower

A wall follower script is designed to make the robot move close to and parallel to a wall. We'll assume the wall is on the robot's *right* side. To make a robot that runs parallel to a straight wall the ultrasonic sensor is ideal. It should be possible to substitute an infrared sensor for that, but the result will be less satisfactory. The ultrasonic or infrared sensor needs to be mounted on the robot pointing sideways at the wall. These images show how the sensor could be attached:

Home version:



Education version:



The idea will be that we will have a target range of separation between the sensor and the robot of, say, 13 to 15cm, and then the robot will curve its motion towards the wall if the detected separation is too big and vice versa. Therefore the robot will advance along a wavy line – you can see there are strong similarities between a wall follower script and the line follower script that we looked at earlier. These scripts, called **wall_follower_IR.py** and **wall_follower_US.py**, can be found in the **part3** folder.

Ultrasonic version

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveTank, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import UltrasonicSensor
from time import sleep

us = UltrasonicSensor()
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)

while True:
    distance = us.distance_centimeters
    if distance > 15:
        # medium turn right
        tank_pair.on(left_speed=30, right_speed=25)
    elif distance < 13:
        # medium turn left
        tank_pair.on(left_speed=25, right_speed=30)
    else:
        # go straight
        tank_pair.on(left_speed=30, right_speed=30)
    sleep(0.01) # wait for 0.01 seconds
```

Infrared version

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveTank, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import InfraredSensor
from time import sleep

ir = InfraredSensor()
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)

while True:
    # for a light-colored wall, multiply
    # proximity by 0.7 to obtain APPROXIMATE distance in cm
    distance = ir.proximity * 0.7
    if distance > 15:
        # medium turn right
        tank_pair.on(left_speed=40, right_speed=35)
    elif distance < 13:
        # medium turn left
        tank_pair.on(left_speed=35, right_speed=40)
    else:
        # go straight
        tank_pair.on(left_speed=40, right_speed=40)
    sleep(0.01) # wait for 0.01 seconds
```

There's nothing particularly 'wrong' with these scripts, but it's surprisingly hard to get a really good result. One problem is that as the robot twists back and forth it is often not parallel to the wall and when it is not parallel to the wall the distance measured by the sensor is bigger than it should be. That's especially problematic when the robot is too close to the wall because if the sensor reports that it's too far away then it will turn even more away from parallel thus approaching the wall even more quickly and making the problem even worse, leading to a collision with the wall. That problem is reduced by putting the sensor at the front of the robot rather than at the center or the back. To keep the robot more or less parallel to the wall, it's helpful to instruct the robot to make only *gentle* turns, as in the script above, but that means the robot will do a very bad job if it has to turn sharp right when it reaches a corner in the wall. Maybe the solution would be to have the robot do gentle turns if it is close to the target range and sharper turns only if it is far from the target range. That means the sharpness of the turn would be *proportional* to the error in the distance, and that takes us neatly to the next project.

Proportional wall follower

We've just discussed why it might be helpful to write a script such that the sharpness of the turn is *proportional* to the deviation away from the target distance. That deviation or error is easy to calculate: it's just the measured distance minus the target distance. The bigger the error, the sharper the turn that we want the robot to make – the sharpness of the turn is basically what the steering parameter of the MoveSteering class represents so will use that class rather than the MoveTank class for this project. If we set the steering parameter to be equal to the error in centimeters then the steering parameter will have quite small values even when the robot is far from the target distance, so we probably need to multiply the error in centimeters by a '*proportionality constant*' to get more reasonable values. I've used a proportionality constant of 3 in the following script but you should try other values to see what works best for you. Multiplying the error by a proportionality constant could conceivably give a steering value outside the allowed range of -100 to +100 so we need to include code to make sure that that cannot happen. We can do that with the functions **max()** and **min()** which are standard built-in functions in Python. The function **max()** returns the largest value among the supplied arguments and the function **min()** gives the smallest value. So we can make sure that the steering value 'steer' never exceeds 100 with **min(steer, 100)** and we can make sure that 'steer' is never less than -100 with **max(steer, -100)**. Combining these into one line of code we get **steer = min(max(steer,-100),100)** as seen in the following script. It works! Try it! These scripts, called **wall_follower_prop_IR.py** and **wall_follower_prop_US.py**, can be found in the **part3** folder.

Ultrasonic version

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import UltrasonicSensor
from time import sleep
```

```

us = UltrasonicSensor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)

def limit(value):
    # returns a value within the range -100 to +100
    return min(max(value,-100),100)

while True:
    error = us.distance_centimeters -15
    # Try changing the proportionality constant in the next line
    steer_pair.on(steering=limit(error*3), speed=30)
    sleep(0.01) # wait for 0.01 seconds

```

Infrared version

```

#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import InfraredSensor
from time import sleep

ir = InfraredSensor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)

def limit(value):
    # returns a value within the range -100 to +100
    return min(max(value,-100),100)

while True:
    # for a light-colored wall, multiply
    # proximity by 0.7 to obtain APPROXIMATE distance in cm
    distance = ir.proximity * 0.7
    error = distance - 15
    # Try changing the proportionality constant in the next line
    steer_pair.on(steering=limit(error*3), speed=30)
    sleep(0.01) # wait for 0.01 seconds

```

Note how neat and simple the main block of code is in these scripts. And there's a good chance that the script will work quite well when there is a corner in the wall such that the robot has to make a sharp turn to the *right*. That's because once the robot has 'run out of wall' the sensor will return a very large value which will cause the robot to make a sharp turn. The videos demonstrating this script show that the home version can make the right turn without difficulty but that the education version starts rotating on the spot because the steer value has become +100. A separate video shows that if the steering value is limited to +25 by using this code `return min(max(value, -100), 25)` then the robot is indeed capable making the right turn. Of course, a corner in the wall will normally mean that our robot has to turn *left*, not right, and you may have success with left turns with the IR sensor by pointing it forwards as well as sideways like this:



This arrangement is unlikely to work with the US sensor since when ultrasound hits a surface at a 45 degree angle insufficient wave energy is reflected back towards the source for it to be detected. If you want to make a maze-solving robot you will need to devise a model that is capable of turning either right or left as needed.

PID wall following

Our proportional wall follower does a decent job of following the wall but you must have noticed that the path does not get less wavy as time goes by, as we would wish. There is a way to make that happen, but it's quite a lot more complex so I will just mention it here and invite you to do more research if you're interested. It's called a **proportional–integral–derivative controller (PID controller)**. The terms 'integral' and 'derivative' should be meaningful to you if you have studied calculus in math class. Quoting from en.wikipedia.org/wiki/PID_controller, 'A proportional–integral–derivative controller (PID controller) is a control loop feedback mechanism widely used in industrial control systems ... A PID controller continuously calculates an error value as the difference between desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively).'

Remember how we adjusted the 'proportionality constant' when making the proportional follower? PID controllers have *three* constants to optimize and they interact with one another, so it's very difficult to find the best values. If this doesn't put you off and you want to know more then check out these links:

www.inpharmix.com/jps/PID_Controller_For_Lego_Mindstorms_Robots.html
thetechnicgear.com/2014/03/howto-create-line-following-robot-using-mindstorms/
projects.raspberrypi.org/en/projects/robotPID
youtu.be/UR0hOmjaHp0

Maze solver

The step from a wall follower to a maze solver is actually surprising small, for you can solve a normal maze by simply advancing with one hand touching the wall, which is similar to what our wall follower does. We already have a script to make the robot follow a wall on the robot's right side and it can make the robot go around right-angle corners as long as that involves a turn to the *right*. What we need to add is the capability of detecting and negotiating a right-angle corner in the wall that requires the robot to turn *left*. Our ultrasonic or infrared sensor is already busy detecting the wall to the right so the most obvious way to detect a wall straight ahead is to use the touch sensor on the front of the robot, with or without a bumper. I'll leave the development of the maze solver robot as a challenge for you!

Steer with light

In this exercise we'll use a bright flashlight to steer the robot, perhaps around an obstacle course. This is not a conventional way to steer a vehicle – if the concept is not clear to you then I encourage you to watch the corresponding video. That video shows me steering the robot in a 'figure of eight' circuit around two posts – this is a good challenge.

How can we indicate with the brightness of the flashlight which way we want the robot to turn? One way would be to have the robot turn left when there is minimal light, go straight when there is 'medium' light and turn right when the light is 'bright' because you are pointing the flashlight straight at the color sensor. The sensor's **ambient_light_intensity** property can have values 0-100 so if we want to use the 'on' function from the MoveSteering class and if we want to be able to use the full range of 'steering' from -100 to +100 then we need to come up with a formula to convert the light intensity to that range. How about **(ambient_light_intensity*2)-100** ? Put a light intensity value of 0 into that formula and it will give us -100, as wanted, and putting a value of 100 in will give a result of +100, which is also correct. We'll keep the 'speed' set to a constant, say, 30. Check out the very short script below. How well does it work? Let's find out. Fix the color sensor so that it's facing vertically upwards, grab your flashlight, then run the script (**steer_with_light.py** in the part3 folder):

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import ColorSensor
from time import sleep

cl = ColorSensor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
```



```
while True:
    steer_pair.on(steering=(cl.ambient_light_intensity * 2)-100, speed=30)
    sleep(0.1) # wait for 0.1 seconds
```

You're probably going to be disappointed with the result, but at least you should be able to see that you can *influence* the motion of the robot with your flashlight. The problem is that the light intensity is only ever going to be zero if the room is rather dark, which is unlikely, and probably also the brightness detected when you shine your flashlight directly at the sensor will not be anywhere close to 100, so we can't get the steering range of -100 to +100 that we were hoping for. In the next exercise we'll modify the code to get better results. This adaptation to real world conditions is part of what makes programming robots so interesting.

Steer with *Calibrated* Light

The last script was really short and you probably had some trouble adjusting the formula to get good results, so let's do a bit more work on it. It would be nice to be able to *calibrate* the light intensity in some way so that the robot knows what is the dimmest and brightest light that it should expect. I propose this calibration procedure:

1. The robot tells us to press the Enter button when the color sensor is in dim light.
2. We press the Enter button as instructed.
3. The robot records that dim light intensity in a variable called 'dim'.
4. The robot beeps and then instructs us to press the Enter button again when the color sensor is in the bright light of the flashlight.
5. We point the flashlight at the sensor and press the button as instructed.
6. The robot records that bright light intensity in a variable called 'bright'.
7. The robot beeps, says '3, 2, 1, go!' and then begins its motion.

What would that look like in code? The trickiest part is the calculation of the steering value based on the values of dim, bright and the light intensity. This is not a math course and I don't have time to explain this formula properly to you, but check this out:

```
steer = (200*(intensity-dim)/(bright-dim))-100
```

Ask yourself what would happen if the sensor, as it trundles along, is exposed to light with an intensity equal to 'dim'. Then in the formula above, 'intensity-dim' will equal zero, so we will have **200*(0/(bright-dim))-100** which is -100, as we want. When the sensor gives an intensity value equal to 'bright' then the part of the formula **(intensity-dim)/(bright-dim)** becomes **(bright-dim)/(bright-dim)** which is just 1. The whole formula then gives **200*1-100** which is +100 which is what we want (recall that multiplication is done before subtraction unless parentheses indicate otherwise). That's not a proper mathematical justification of the formula but it's all we have time for right now – at least it should convince you that the formula should work.

Or should it? What if we fix the 'bright' value with the flashlight 40cm from the sensor and then, while the robot is moving, we bring the flashlight 30cm from the sensor? Then the intensity will exceed the 'bright' value we recorded and the clever formula we looked at will give a value greater than 100 which will generate an error if we try to use it as the 'steering' value for the motor command. We'll need to use the **min(max(steer,-100),100)** expression that we developed earlier. That expression is rather neat and our code is starting to get a bit long and complex – two good reasons to consider making a **user-defined function**. Making a user-defined function has multiple advantages:

- The user-defined function can be easily recycled in other scripts
- It allows us to make the structure of the main part of the program simpler and easier to understand, with fewer distractions.

Let's see how that function could be made into a user-defined function called **limit()**. This script is **steer_with_cal_light2.py** in the **part3** folder.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import ColorSensor
from ev3dev2.button import Button
from ev3dev2.sound import Sound
from time import sleep

cl = ColorSensor()
btn = Button()
sound = Sound()
```

```

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)

def limit(value):
    # returns a value within the range -100 to +100
    return min(max(value,-100),+100)

sound.speak('Press the Enter key when the sensor is in dim light')
btn.wait_for_bump('enter')
dim = cl.ambient_light_intensity
sound.beep()

sound.speak('Press the Enter key when the sensor is in bright light')
btn.wait_for_bump('enter')
bright = cl.ambient_light_intensity
sound.beep()
sound.speak('3, 2, 1, go!')

while not btn.any(): # Press any key to exit
    intensity = cl.ambient_light_intensity
    steer = (200*(intensity-dim)/(bright-dim))-100
    steer = limit(steer)
    steer_pair.on(steering=steer, speed=30)
    sleep(0.1) # wait for 0.1 seconds

```

That should work well enough and it's kind of fun to be able to control the motion of the robot like this. By all means set up a little obstacle course if you want to test your skills in controlling the robot in this novel way. As previously stated, placing two posts and then trying to move in a figure of eight around them is a good, simple challenge because it forces you to alternate between turning left and turning right.

If you happen to have an NXT sound sensor you can modify the above script so as to be able to control the robot's motion with your voice – stay silent and the robot will turn left, make medium noise to make the robot go straight and make a real racket if you want the robot to turn right. Don't blame me if you get complaints from the neighbors! It's actually quite unlikely that you do have an NXT sound sensor so I don't give you instructions for using it in this course but you can find instructions in the official documentation here:

<https://Python-ev3dev.readthedocs.io/en/ev3dev-stretch/sensors.html#sound-sensor>

Follow an object

Now let's get the robot to follow an object at a respectful distance, say 10 cm, without getting too close. Also, if the object gets too far ahead of the robot then the robot will give up the chase and stop. The 'object' could be, for example, your hand, or a sheet of white paper. In this script we'll keep it simple: our robot will only move in a straight line. First we'll write a version of the program that uses the ultrasonic sensor found in the education version of the EV3 kit, and then we'll modify that for use with the infrared sensor found in the home version of the kit. Whichever sensor you use, you'll need to mount it on the robot facing forwards, preferably not too close to the floor so as to avoid picking up reflections from the floor. The code below has the neat feature that if the object approaches closer than the target separation the robot will actually back away.

Ultrasonic version (object_follower_us.py in the part3 folder)

```

#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import UltrasonicSensor
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
us = UltrasonicSensor()

while True:    # forever
    distance = us.distance_centimeters
    error = distance - 10

```

```

if distance < 50:
    steer_pair.on(steering=0, speed=error*2)
else:
    steer_pair.off()
sleep(0.2)

```

Infrared version (**object_follower_ir.py** in the **part3** folder)

Whereas the ultrasonic sensor returns a rather accurate distance value in centimeters or inches, the infrared sensor just returns a **proximity** value which is based on the amount of infrared light reflected by the object. Yes, that depends on how close it is, but it also depends on its color and how big it is, so the proximity value cannot be converted into an accurate distance value. However, we can say that for a largish object that is light colored:

- The proximity value can be *very roughly* converted to distance in centimeters by multiplying by 0.7.
- A distance in centimeters can be *very roughly* converted to a proximity value by multiplying by 1.4 (not forgetting that proximity can only be in the range 0-100).

Differences with the US version are in bold.

```

#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import InfraredSensor
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
ir = InfraredSensor()

while True:    # forever
    # for a largish light-colored object,
    # multiply proximity by 0.7 to obtain distance in cm
    distance = ir.proximity * 0.7
    error = distance - 10
    if distance < 50:
        steer_pair.on(steering=0, speed=error*2)
    else:
        steer_pair.off()
    sleep(0.2)

```

Follow a beacon

The infrared remote control / beacon and the infrared sensor are included with the *home* version of the E3 but not with the *education* version, though they can also be purchased separately. The infrared sensor can detect not only the **distance** to the beacon (based on the detected brightness) but also the **heading** (-25 to +25) of the beacon. Note that **distance()** and **heading()** are *functions* (note the parentheses) while **proximity** is a *property* (no parentheses). You always need to be alert as to the difference between properties and functions!

We have to remember that if the sensor cannot detect the beacon then the **distance** is returned as **None** – we have to make sure that this will not make our script fail. There is no such problem with the **heading** value since this simply becomes 0 if no beacon is detected. The script below is protected against **distance** values of **None** by the line

```

if distance:
    which is equivalent to
if distance = True:

```

We are making use of the fact that in Python **None** is a '*falsy*', i.e. it evaluates to False when evaluated as a Boolean value. Other examples of falsies in Python are the integer 0, empty lists and the empty string ''. Examples of '*truthies*' are non-zero integers, non-empty lists and non-empty strings. So if the sensor returns the value None then this is treated like False in the **if** statement, causing the following line not to run.

Make sure the beacon is set to channel 1 since this is the default value and the script below does not specify a different value. In the script below I multiply the heading value by 2 to make the robot turn more strongly to follow the beacon. I calculate the error by subtracting 10 from the distance value for multiple reasons:

- This means that the robot will stop if it gets to a distance of 10 from the beacon (very roughly 10cm) – we don't the robot to get 'too close' or to risk bumping into the beacon

- This means that if we bring the beacon very close to the robot then the robot will actually reverse away from the beacon, which is nice to see.

Try the code below (**beacon_follower.py** in the **part3** folder). Note that I'm using the **min()** function to make sure the speed value does not exceed 90.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import InfraredSensor
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
ir = InfraredSensor()

while True:    # forever
    distance = ir.distance()
    if distance:    # If distance is not None
        error = distance - 10
        steer_pair.on(steering=2*ir.heading(), speed=min(90, 3*error))
    else:
        steer_pair.off()
        sleep(0.1)
```

Program with colors

It's nice to be able to give the robot instructions after the program has been launched, and one way of doing that is to show a sequence of colors to the color sensor. We'll associate each color with a certain movement that we want the robot to execute. We'll store the colors in a list (called an 'array' in EV3-G) so this exercise will give us practice working with lists.

More specifically, the procedure will be:

- The robot beeps to indicate that it is ready.
- The user bumps the touch sensor button to indicate that (s)he is also ready.
- The program waits until the user presents an object with a valid color (blue, green, red, yellow or white) to the color sensor.
- When the color sensor recognizes blue, green, red or yellow it stores the corresponding color string (such as 'Blue') in a list. A Python 'list' is equivalent to an EV3-G 'array'.
- The program also speaks the recognized color.
- The above sequence is repeated until the sensor is shown white.
- The robot then plays the sound 'Horn 2' at 100% volume, waiting for the sound to finish playing before continuing.
- Then the program reads the contents of the list, carrying out the corresponding movement for each color string.
- The program stops when all the color strings have been read and the corresponding movements carried out.

Here are the intended movements that correspond to each color:

- 'Blue': turn 90° left.
- 'Yellow': turn 90° right.
- 'Green': go straight forward for two wheel rotations.
- 'Red': go straight back for two wheel rotations.

If you have the education version you should be able to make a color ribbon like this using a pair of 10 unit cross axles:



But the home edition doesn't have any suitable green or yellow beams so you will probably need to find non-Lego objects with the same colors.

Since we're going to store the colors in a list, let's review what we know about lists. The order of the elements in a list is important. Each element in the list has an index number, and the first element has index number zero. Here is an example of a list in Python: [5, 2.7, 3.1]. This array has three elements - we say the array has a 'length' of three. The elements have index numbers 0, 1 and 2, so the third element has index number 2, not 3.

This script (**program_with_colors.py** in the **part3** folder) assumes that you have downloaded the EV3 sounds in WAV format (including the file 'Horn 2.wav') to a folder called 'sounds' in your 'robot' folder as described previously.

Can you make the script trace out the shape of the letter 'T'?

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveTank, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import ColorSensor, TouchSensor
from ev3dev2.sound import Sound
from time import sleep

cl = ColorSensor()
ts = TouchSensor()
tank_pair = MoveTank(OUTPUT_B, OUTPUT_C)
sound = Sound()

def get_color():
    while True:    # Wait for a valid color to be detected
        color = cl.color_name
        if color in ('Blue', 'Green', 'Yellow', 'Red', 'White'):
            sound.speak(color)
            return color
        break      # exit the loop
    sleep(0.1)

color_list = [] # create empty list
while True:
    sound.beep()
    ts.wait_for_bump()
    my_color = get_color() # get a valid color
    if my_color == 'White':
        break # break out of loop
    else:
        color_list.append(my_color)

sound.play_file('/home/robot/sounds/Horn 2.wav')

for col in color_list:
    if col=='Blue':    # turn the robot 90 degrees left
        # try 171 with the edu version, 222 with home version
        tank_pair.on_for_degrees(-50, 50, degrees=171)
    elif col=='Yellow': # turn the robot 90 degrees right
        tank_pair.on_for_degrees(50, -50, degrees=171)
    elif col=='Green': # go straight forward 2 wheel rotations
        tank_pair.on_for_rotations(50, 50, rotations=2)
    elif col=='Red':   # go straight backwards 2 wheel rotations
        tank_pair.on_for_rotations(-50, -50, rotations=2)
```

Notes:

- To make the lines shorter, I have omitted the parameter names **left_speed** and **right_speed**.
- The sensor sometimes sees 'Black' when it should see some other color so for best results you should avoid using that color in the code.

- For my robot (education version), 171° of *wheel* rotation (by both wheels, moving in opposite directions) is roughly correct to make the *robot* turn 90°. You may need to use a different value, depending on the type of wheels you have and on their spacing. For the home version, try using 222 degrees instead of 171 degrees.

Challenge:

Add another color (I suggest 'Grey') and another movement to the script.

Self-Parking

We hear a lot about self-driving robotic cars and no doubt they will outnumber human-driven cars within a couple of decades. The photo shows a prototype of Waymo's self-driving car, navigating public streets in Mountain View, California in 2017. (By Grendelkhan - Own work, CC BY-SA 4.0, commons.wikimedia.org/w/index.php?curid=56611386)



We can expect that as soon as it becomes clear that self-driving cars are much safer than human-driven cars it will be made illegal for humans to drive any more on public roads – I wonder how many people understand that (see www.bbc.com/news/technology-48334449). Anyway, to be comfortable with technology we need to be familiar with it and understand it, so let's see whether we can make a simple self-parking vehicle. We'll set our vehicle moving in a straight line parallel to a 'line of cars' (actually just a straight wall) and our script will try to detect a break in the line of cars that is long enough and wide enough for our vehicle to be able to park, and then our vehicle will reverse neatly into the space, or so we hope. Actually it's not obvious that we should be *reversing* into the space with our 'educator vehicle' – we'll have to see whether it's better to reverse in or go in forwards...

We'll use a sideways-pointing ultrasound sensor to detect spaces in the 'line of cars'. If you don't have an ultrasound sensor we can try with the infrared sensor but it is much less precise than the ultrasonic sensor – much less directional and affected by the size and color of the object being detected, so we can't expect that to work well.

It would be tempting to use our wall-following skills to follow the line of cars but our wall-following means the vehicle moving in a slightly wavy path that would make it difficult to evaluate spaces in the line, and also a wall follower will have a tendency to drive forwards into the first space, whether or not it is big enough to accommodate the vehicle. So we'll just set the vehicle to move in a straight line and hope that it sticks to a path parallel to the line of cars long enough for our vehicle to correctly identify and reverse into a suitable space.

Prior to making the self-parking script we need to know what are the minimum dimensions of a 'suitable space'.

The following script (**self_park1.py** in the **part3** folder) is intended to allow you to adjust the turns so that *your* robot reverses into a space as desired. Adjust the steering and rotations values so that the vehicle moves completely out of the traffic lane. Then use barriers of some kind (maybe cartons or half-open hardback books) to determine how much space is needed for the manoeuvre.

Remember that we don't know yet whether the robot should reverse into the space or drive in forwards. The robot reverses so slowly that it should be possible to use a marker to trace the path followed by the right edge of the robot as it reverses (assuming the robot is on a surface that you can use a marker on, such as a large sheet of paper). To understand these scripts, I strongly recommend that you watch the corresponding video demonstrations.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)

wf = 1 # Set wf to 1 for home version and 0.77 for edu version
# adjust steer and rots until the robot moves neatly out of the traffic lane
steer = 28
rots = -1.8
# move forward
steer_pair.on_for_rotations(steering=0, speed=25, rotations=3.7*wf)
# reverse through two curved motions
steer_pair.on_for_rotations(steering=steer, speed=15, rotations=rots*wf)
steer_pair.on_for_rotations(steering=-steer, speed=15, rotations=rots*wf)
# move forward to the center of the parking space
steer_pair.on_for_rotations(steering=0, speed=25, rotations=0.7*wf)
```

Once the minimum dimensions of the parking space have been determined using the above script, place objects such as cartons or books to mark out the limits of the parking space, adding a margin of error. The margin of error probably doesn't need to be more than a couple of centimeters for the ultrasonic sensor in the education kit, but the infrared sensor in the home kit measures distances ('proximity') so inaccurately that the margin of error needs to be much greater - see the video demonstrations.

In the final self-parking script, we'll use the **position** property of the left motor, motor B, to detect when the parking space is big enough to attempt the parking maneuver. For the EV3 motors, the **position** property is simply the angle in degrees through which the motor has rotated since the position property was last reset to zero. In the script below, the position property is continually reset to zero until the IR sensor detects that there is a parking space that is sufficiently deep, then the position property (the angle through which the motor has turned) is allowed to increase. The loop stops running when the position property corresponds to a distance of 37cm, the minimum length needed for our parking maneuver. Then the parking code is run.

For the home version wheels, the position property can be converted to the distance moved in mm by multiplying by 0.377. That's because the 'home' wheel has diameter 43.2mm so its circumference (diameter x pi) is 135.7mm. Thus when the wheel turns 360 degrees, it advances 135.7mm, which means for every one degree of turn the robot advances $135.7/360 = 0.377\text{mm}$.

If the distance moved reaches 370mm, the minimum length for a successful maneuver as measured previously, then we run the same maneuver code as before, except that the first forward move is just a small movement to put the robot into the correct position to begin reversing.

A print command is included so that useful information is printed to the VS Code output panel roughly every 0.1 second, assuming the script is launched from VS Code.

Here is the home (infrared) version of the finished script (**self_park_ir.py** in the **part3** folder). Differences relative to the previous script are in bold. **If you have the education version then modifying the home script below to match the education version which uses the ultrasonic sensor will be a great exercise...**

```
#!/usr/bin/env python3
from ev3dev2.motor import LargeMotor, MoveSteering, OUTPUT_B, OUTPUT_C, SpeedDPS
from ev3dev2.sensor.lego import InfraredSensor
from sys import stderr
from time import sleep

motorB = LargeMotor(OUTPUT_B)
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
ir = InfraredSensor()

wf = 1
motorB.position = 0
steer_pair.on(steering=0, speed=SpeedDPS(265))
```

```

while motorB.position*0.377 < 370: # target length is 37cm
    if ir.proximity * 0.7 < 16.5: # target width is 16.5cm
        motorB.position = 0
    # Print measured parking space length and width in cm.
    print(motorB.position*0.377, ir.proximity * 0.7, file = stderr)
    sleep(0.1)

steer = 28
rots = -1.8
steer_pair.on_for_rotations(steering=0, speed=25, rotations=0.5*wf)
steer_pair.on_for_rotations(steering=steer, speed=15, rotations=rots*wf)
steer_pair.on_for_rotations(steering=-steer, speed=15, rotations=rots*wf)
steer_pair.on_for_rotations(steering=0, speed=25, rotations=0.7*wf)

```

Notes on above script

The speed value is set to **SpeedDPS(265)** so that the motors turn 265 degrees per second. We have already seen that (for the home version) when the motors turn 360° the robot advances 135.7mm. That means that to advance 1mm the motors must turn $360/135.7 = 2.653$ degrees. Therefore if the motors are made to turn 265 degrees per second then the robot will advance at 100mm per second, or 10cm per second. Since the while loop repeats about once every 0.1 second (a little more than this because not only is there a 0.1s 'sleep' but the other code in the loop also takes a little time to run) the robot will advance about 1cm for each cycle of the loop and thus 1cm between each activation of the print statement. Knowing this makes the printout more meaningful.

Although the above script is set to look for a minimum width of 16.5cm and a minimum length of 37cm you will need to include a significant margin of error in the placing of the barriers due to the inaccuracy of the IR sensor when measuring distance (proximity). In the accompanying video demonstration I needed to increase the space to 40cm x 23cm so that the IR sensor before the above script would work. The ultrasonic sensor in the education kit should work much better and need less margin of error.

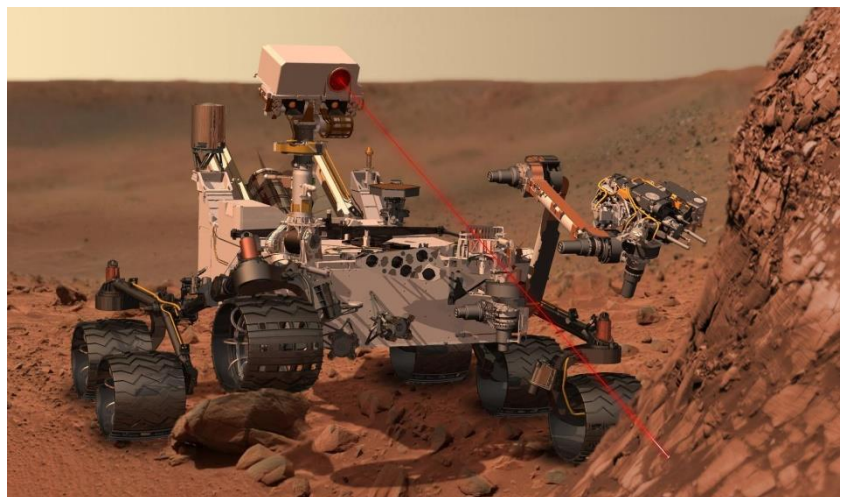
Don't forget that it should be possible to use much smaller parking spaces if you drive forwards into the space rather than reverse in. That is to say you need to first drive past the space to detect it, then back up to the start of the space, then drive in forwards.

The above exercise gives a very crude introduction to self-parking – it goes without saying that a commercial self-parking vehicle would use multiple sensors and a more complex script.

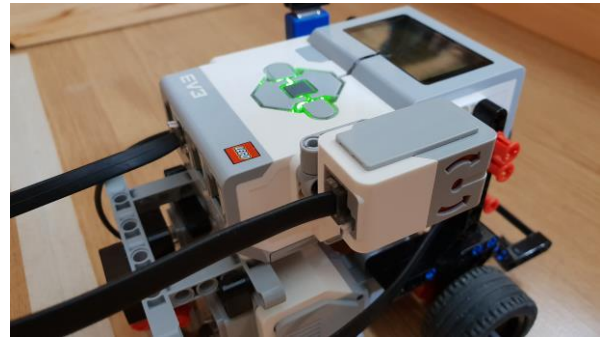
Beware of steep slopes

This exercise will use the gyro sensor which you probably don't own if you bought the home edition of the EV3, in which case you can skip this exercise.

You're familiar, of course, with the Curiosity rover, shown at right, that is currently exploring the surface of planet Mars. At a cost of 2.5 billion dollars, this must be the most expensive robot ever made! It takes about 30 minutes for a round trip radio signal from the robot to earth and back so there's no way the robot could be controlled by real-time remote control. If the operator were to see video from the rover showing that is approaching a steep downward slope then the signal to the rover to stop and back up would arrive much too late to avoid a catastrophic fall. In fact by the time the operator received the video, the rover would probably already be a mangled mess at the bottom of the slope! So the rover drives mainly autonomously, controlled by a very carefully written software program as well as many sensors including gyro sensors like the EV3 gyro sensor. The fact that Curiosity has managed to move around since 2012 without an accident is a tribute to the skill of the NASA programmers. To learn more about Curiosity check out <https://mars.jpl.nasa.gov/msl/>

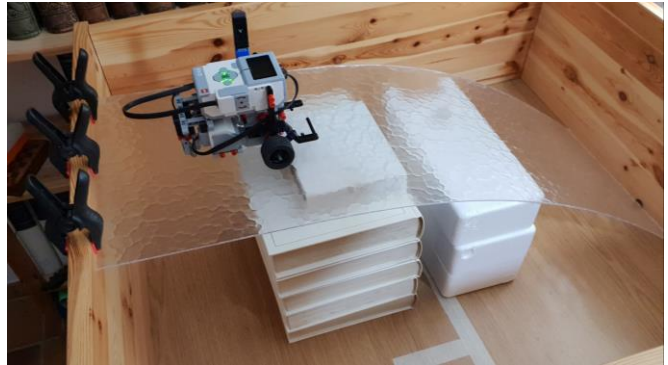


Let's make a script that monitors the steepness of the terrain that the robot is moving over and makes the robot stop and back up if the terrain gets too steep, say if the angle exceeds 15 degrees. For this to work you will have to attach the gyro sensor to the robot so that the side with the red arrows on it is vertical. You could attach the sensor like this:



Whenever you use the gyro sensor always make sure that the robot is very still when the brick is turned on or the gyro sensor is plugged in otherwise its readings will wander even when the robot is still.

You could set up a variable-gradient ramp like this (if you bend a board then be careful not to break it):



Here's a script for you to try (**beware_steep_slopes.py** in the **part3** folder). As usual it assumes that you have a folder called 'sounds' inside your 'robot' folder and that the 'sounds' folder contains the standard EV3 sounds in WAV format.

```
#!/usr/bin/env python3
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C
from ev3dev2.sensor.lego import GyroSensor
from ev3dev2.sound import Sound
from time import sleep

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
gyro = GyroSensor()
sound = Sound()

steer_pair.on(steering=0, speed=20)
gyro.wait_until_angle_changed_by(15)
steer_pair.off()
# play_type=1 does NOT block the program while the sound is playing
sound.play_file('/home/robot/sounds/Backing alert.wav', play_type=1)
steer_pair.on_for_rotations(steering=0, speed=-25, rotations=3)
```

The function **wait_until_angle_changed_by()** does not care whether the change is positive or negative, so the same script can detect the robot moving onto a steep downward slope OR a steep upward slope.

Part 4: Make a Drawbot / Writerbot

Videos 4A and 4B: Make a drawing robot or 'drawbot'.

Drawing

It would be neat to be able to make drawings with the EV3! Making drawings gives a tangible product, gives lots of potential for writing different programs and lots of scope for creativity. Good results have already been achieved by making the drawing with an arm that is attached to a fixed base since that stops the robot from wandering away from its intended location. But that severely limits the drawing to be a small size (see the video on the home page of my site ev3Python.com). Making a drawing with a robot that is free to move removes the size limit on drawings but also make it difficult to get a satisfactory result due to the robot's tendency to wander away from its intended position.

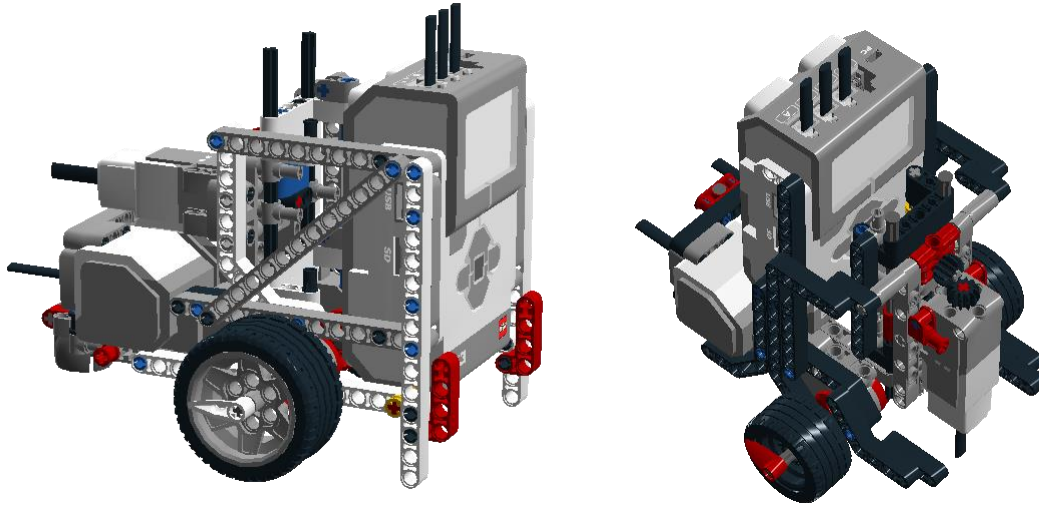
A drawing can consist of many strokes, so the wandering errors accumulate and can become a real problem. See this official Lego video and you will note that even Lego acknowledges this problem: www.lego.com/en-us/mindstorms/videos/fan-robot-banner-print3r-c827d39be8e94e15b18cb957c1d59c24 Using sensors may help to keep the robot in its intended position but they would need to be quite precise since even an error of a centimeter or a few degrees can spoil a drawing. The most obvious sensor to use would be the *gyro* sensor since this might allow us to get more accurate turns, but most EV3 owners have the *home* kit which does not include the gyro sensor so I will avoid using that sensor. The *ultrasonic* sensor is capable of pretty good accuracy but is also included only in the educational version of the EV3. The *infrared* sensor is included only in the home set and in any case it's certainly not accurate enough to be helpful here. So the best solution may be just to design and code the robot very carefully and not use any sensors.

I have spent several days designing a drawing robot, based on the previous and following considerations:

- For accurate drawing and simple coding it is highly desirable that **the pen should be located exactly at the mid-point between the two driving wheels**. This is the only way that sharp corners such as those in a square can be drawn satisfactorily. I tried modifying the standard Lego EV3 'education vehicle' model by moving the brick back to make space for a vertical pen to be located between the driving wheels but this puts too much weight over the rear wheel or castor ball, causing too much friction there. Also, taking weight off the driving wheels makes their movements less accurate since they have less grip. I concluded that in this case it would be necessary to design a completely new model.
- The mechanism that holds the pen should be able to raise and lower the pen using the medium motor in a reliable and reproducible fashion.
- Minimal weight should be on the rear wheel to reduce friction there. With very little weight on the rear wheel there is however a risk that the robot will tip forwards if the robot suddenly stops moving quickly forwards.
- We want maximum weight on the driving wheels for increased grip and accuracy.
- The mechanism that holds the pen should be compatible with standard markers with thick points.
- The motor ports should be easily accessible.
- The sensor ports need not necessarily be accessible since we probably will not use sensors with this model for the reasons already discussed.
- The PC socket should be easily accessible so that the robot can be easily programmed via a USB Cable.
- The USB socket should be easily accessible so that the robot can be connected to the PC via WiFi if desired.
- For the home model, it should be easy to remove the batteries for replacement or recharging.
- For the education model the recharging socket should be easily accessible.
- The screen should be visible and the buttons reasonably accessible.
- To stop the pen from drying out, it should be possible to put the cap on the pen either while it is in position on the robot or by easily removing the pen from the robot.
- The model should include 'crick' pieces to lift the driving wheels for stationary testing which is sometimes useful e.g. when debugging.

- The driving wheels should be fairly far apart since this makes turns more accurate because more wheel rotations are needed in order to make the robot rotate.

As you can see, there are a large number of factors to be taken into account when designing an EV3 drawbot – no wonder it took me several days to come up with satisfactory models. As you noticed in the above list, the requirements are slightly different for the home version and the education version, so my finished designs are also quite different for the two versions (education version at left, home version right):



My home version (right) makes it very easy to remove the batteries. You can see that the brick is placed vertically between the axis of the driving wheels and the rear wheel. In that location, there is enough weight over the rear wheel that the robot does not tip forwards even if the robot suddenly stops a fast forward motion. The buttons can be accessed, though it's a little awkward.

For the education vehicle (left) the charging socket must be accessible so the design used for the home version cannot be used. Instead, I put the brick vertical in front of the driving wheels. This makes the charging socket and the buttons very accessible but has the disadvantage that there is now very little weight over the rear wheel – so little that the robot would tend to tip forwards when stopping suddenly after a fast forward motion. To stop the robot from tipping forward there are two pieces at the front of the robot that almost touch the ground. This is fine for a drawbot because a drawbot will always be used a very flat surface – normally a flat sheet of paper though you may also choose to save paper by doing most of your drawings with a dry-erase marker on a smooth surface that can be wiped clean. I won't be held responsible if you use a permanent marker to draw on your expensive mahogany desk!

You can download the instructions for making the drawing robot in the files attached to this lesson. If you're hesitating about whether it's worth the trouble to assemble this model, know that we will use the same model for the next and final project, which is to make a *writing* robot.

To make the drawing robot you will need, in addition to your EV3 set, a couple of small elastic bands and a marker pen. The elastic bands I'm using have a 'diameter' of 2.4cm which works well, but you can use longer bands if you wrap them twice around the pegs to increase the tension. As for the marker pen, it should have a 'bullet tip' as opposed to a 'chisel tip':



Bullet tip: ideal



Chisel tip: not so good

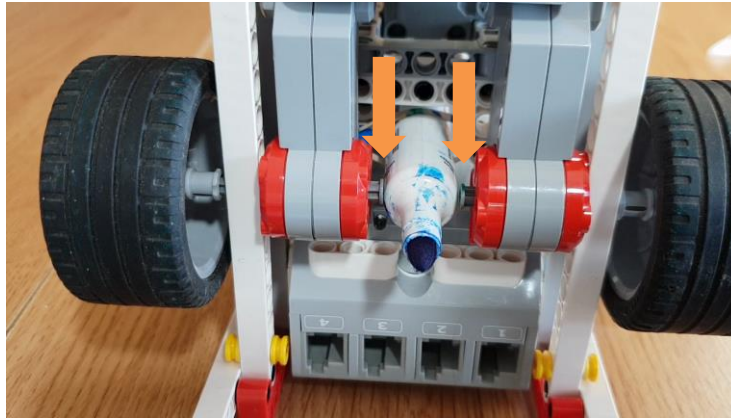


The one I'm using in the videos is a Bic whiteboard marker (shown above) but be sure to plug the cap into the end of the marker if you want to use that because otherwise it won't be long enough. With the cap plugged into the end of the marker the Bic has length 14.4cm and diameter 1.8 cm. Don't try to use any marker with a diameter bigger than this because it won't fit. I've also had success with a 'Schneider 290 whiteboard + flipchart marker' which is 13.6cm long and 1.4 cm wide with the cap off.



Don't bother trying to use a ball-point pen or a pencil – they won't have enough weight and the line that they draw will be too thin. Regarding weight: if you think your pen would benefit from a little extra downward force then you can perhaps attach some weight such as a few big metal washers to the top of the pen or the mechanism that holds the pen, but the markers I'm using work fine without any extra weight. Whiteboard markers dry out quickly (and smell bad) so don't forget to put the cap back on the marker when you're not using it.

When you are ready to try making a drawing with your drawbot, remove the cap from the pen before inserting it into the pen-holding mechanism. Slide the pen into the elastic band loops from above and be sure to center it carefully. Twisting the pen back and forth as you insert it or remove it is helpful. The rubber band loop must not be too loose otherwise the pen will move out of position – you may need to make a double loop with the elastic band to increase the tension. You can also push in the wheel axes slightly until they are *almost* touching the base of the pen as shown below (education version). That will stop the pen moving sideways.



The pen is raised and lowered by a cam that is attached to the medium motor. When the program is started the cam needs to be in the upward position and when it is in this position the pen should be just a couple of millimeters above the paper or other drawing surface. Making sure the pen is placed correctly in every dimension is really critical to making successful drawings. For convenience, there is a separate script (`adjust_cam.py` in the **part4** folder) to adjust the position of the cam to make sure that it is in the upward position when the drawing script is started.

We will limit the drawing options initially to turning on the spot, drawing straight lines and drawing arcs. Drawing arcs with a given radius of curvature is much more challenging than you might think.

Make the robot go straight

Of course, we've discussed this already, but here is a reminder with extra emphasis on precision. Drawing straight lines is pretty easy and accurate. The basic idea is that **when a wheel makes one revolution the whole circumference of the tire rolls along the ground**. Knowing this, it's straightforward to calculate how many degrees of turn are needed to make the robot advance, say, 1 cm. The circumference of the tire is given by $2 \cdot \pi \cdot r$ or simply $\pi \cdot d$ where d is the diameter. The 'official' diameter of the tire is marked on the wall of the tire: 5.6 cm for the large tire in the education kit and 4.32 cm for the large tire in the home kit. But these figures are for tires that have not been distorted by the weight of a robot pushing down on them. In actual use, with the weight of the robot distorting the rubber tires, their effective diameter will be a little less, depending on the weight of the robot and the weight distribution between the front and rear tires. **Based on my experiments, the effective diameter of the tire in the context of the drawbot is 5.53 cm for the education vehicle and 4.27 cm for the home vehicle.** That's only just over 1% smaller than the official diameter but the point is that if we do not take this into account and every move and turn is off by more than 1% then the errors will soon accumulate and become problematic.

Therefore the *effective circumference* of the **education** tire is $\pi \cdot 5.53 = 17.37$ cm. So when the wheel turns through 360° the robot should advance by 17.37 cm and **in order to make the robot advance by 1 cm the wheel must turn $360/17.37 = 20.7$ degrees.**

The *effective* circumference of the **home** tire is $\pi \cdot 4.27 = 13.41$ cm. So when the wheel turns through 360° the robot should advance by 13.41 cm and **in order to make the robot advance by 1 cm the wheel must turn $360/13.41 = 26.84$ degrees.**

It's worth noting that the diameter of the wheels of the education version are almost exactly 30% bigger than the diameter of the wheels of the home version so it's always true that **the wheels of the home version must turn 30% more than the wheels of the education version in order to achieve the same forward movement in a straight line.**

Make the robot turn on the spot

To make the robot turn on the spot, the wheels must turn through equal and opposite angles, but how many degrees of *wheel* rotation are needed in order for the *robot* to turn through one rotation? The answer depends on the separation of the driving wheels, so we need to determine that now. We could try to simply measure the 'separation' but the tires are quite wide – should we measure from the inner edge of one tire to the inner edge of the other, or between the outer edges, or between the centers, or is the *effective* separation equal to some other value? This is a very important question since errors here will affect every turn made by the robot. It's not possible to know how to make the measurement without doing some experiments and I will spare you the effort of having to do that by telling you that **the effective separation of the wheels of both the education version and the home version is close to 10.5 cm**. Note that this does NOT correspond to the separation of the centers of the tires and you might expect (the centers of the tires will be 11.9 cm apart if you carefully assembled the education version of the drawing robot according to my instructions and 11.1 cm if you have carefully assembled the home version).

When the robot rotates on the spot, each wheel moves in a circle whose diameter is equal to the effective separation of the tires which we have just discussed. (If it's not obvious to you that this is the case then think about it before proceeding!) Since both the education version and home version have an effective wheel separation of about 10.5 cm, the tires in both cases therefore move in a circle with circumference = $\pi * d = \pi * 10.5 = 33$ cm.

For the **education** version, we have previously calculated that in order to move forward 1 cm the tire must turn 20.7 degrees so in order to move forwards 33 cm each tire must turn $33 * 20.7 = 683$ degrees. **Therefore for each one degree of turn of the robot the wheels must each turn through $683/360 = 1.9$ degrees.**

For the **home** version, we have previously calculated that in order to move forward 1 cm the tire must turn 26.84 degrees so in order to move forwards 33 cm each tire must turn $33 * 26.84 = 886$ degrees. **Therefore for each one degree of turn of the robot the wheels must each turn through $886/360 = 2.46$ degrees.**

We're talking about real robots here, not some on-screen character, so these figures may not work perfectly for your robot, but they should be pretty close.

Make the robot trace an arc

This is the tricky part, making the robot draw arcs with a given radius of curvature and a given angle. We could just try to determine this by trial and error but it's a great mathematical exercise to do the job properly with a calculation so let's go for it. Of course our calculated solution may need to be tweaked slightly based on our experimental observations. In all the following calculations, be sure to apply the correct order of operations: do division and multiplication before addition and subtraction. If you don't want to work through the math then you can just skip to the end but what a shame that would be! If you want to skip the mathematical derivation then skip ahead to the section called 'Summary for Education Vehicle'.

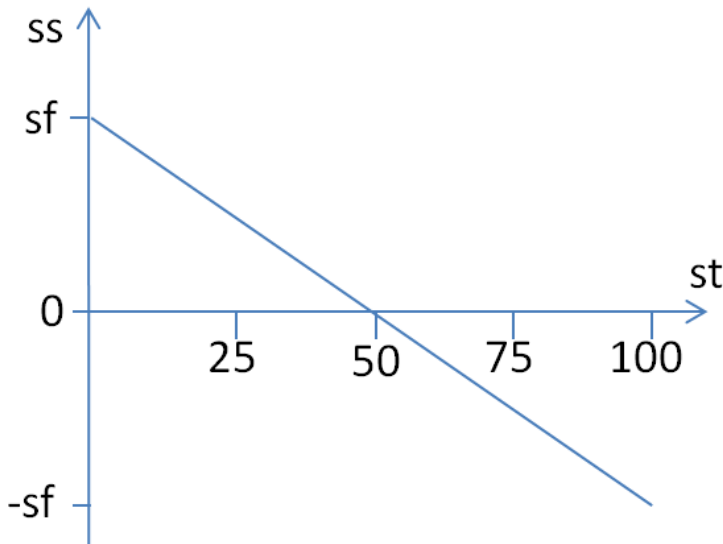
First let's find the relation between the steering value, the fast wheel speed and the slow wheel speed.

Let the speed of the slower wheel be **ss**.

Let the speed of the faster wheel be **sf**.

Let **st** be the *absolute value* of the steering parameter.

Here is a graph similar to one we saw before. It plots the speed of the slower wheel, **ss**, against the *absolute value* of the steering parameter, **st**. Note that it does not show negative values for **st** because we just defined **st** to be the *absolute value* of the steering parameter, so **st** is never negative.



Checking that the graph makes sense, we see that:

- when **st** (the absolute value of the steering parameter) is equal to zero then the speed of the slower wheel (**ss**) is equal to the speed of the faster wheel (**sf**).
- When **st** is equal to 50 then the slower wheel does not turn, which is correct
- when **st** = 100 the speed of the slower wheel (**ss**) is equal and opposite to the speed of the faster wheel, causing the robot to rotate on the spot.

The equation relating **st** to **ss** can be easily found (remember $y = mx + b$):

$$ss = (-2 * sf / 100) * st + sf = -sf * st / 50 + sf = sf * (1 - st / 50) \quad \dots \text{equation (1)}$$

Again, let's check whether that equation makes sense. Let's try with **st** = 25:

$$ss = sf * (1 - 25 / 50) = sf * (1 - 0.5) = 0.5 * sf, \text{ which agrees with the graph.}$$

Equation (1) can be rearranged to give **st**:

$$ss / sf = 1 - st / 50 \quad st / 50 = 1 - ss / sf \quad st = 50 * (1 - ss / sf) \quad \dots \text{equation (2)}$$

Sanity check with **ss** = **sf**:

$st = 50 * (1 - ss / sf) = 50 * (1 - 1) = 0$, which is correct. If the fast and slow wheels have the same speed then the steering value must be zero. The equation is not insane!

Don't forget that this highlighted equation above gives only the *absolute value* of the steering parameter – for the time being you will have to use common sense to decide where the steering parameter should be negative or positive but this should not be a problem. Later we will address this issue.

That's all well and good, but what we really need for our drawing program is to be able to find the *steering values* and *wheel turn angle values* that correspond to the pen drawing an arc with a given angle and radius of curvature. The diagram below is intended to represent the motion of the driving wheels and the pen as the robot follows a curved path. Each wheel and the pen move in an arc and each of these arcs has a different radius of curvature but each arc is part of a circle that has its center at the same point, as shown.

Let **ss**, **sf** and **st** have the same meanings as before.

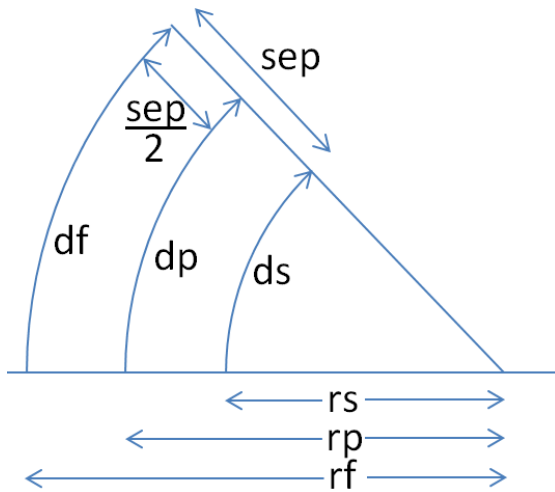
Let the radius of curvature of the arc traced by the faster wheel be **rf**.

Let the radius of curvature of the arc traced by the pen be **rp**.

Let the radius of curvature of the arc traced by the slower wheel be **rs**.

Let the distances moved by the faster wheel, the pen and the slower wheel be **df**, **dp** and **ds** respectively.

Let the speed of the pen be **sp**.



The above diagram shows the paths of the wheels and pen that have moved 'forwards' along their respective arcs. You may realize that the pen can also be made to draw an arc by having the wheels moving in opposite directions, but it turns out that the calculations below are valid also in such cases.

The wheels and the pen all move through the same arc angles therefore the ratio of the speed to the radius is the same for each wheel and for the pen:

$$\mathbf{ss/rs = sp/rp = sf/rf} \quad \text{from which} \quad \mathbf{ss/sf = rs/rf}$$

The separation of the wheels (**sep**) equals **rf – rs**.

$$\mathbf{ss/sf = rs/rf = rs/(rs+sep)}$$

Previously, in equation (2), we found that

$$\mathbf{st = 50*(1 - (ss/sf))}$$

Earlier we got **ss/rs = sf/rf** which can be rearranged to give

$$\mathbf{ss/sf = rs/rf} \quad \text{so} \quad \mathbf{st = 50*(1 - (rs/rf))}$$

But **rf-rs = sep** so **rs = rf-sep** so, substituting for **rs**:

$$\mathbf{st = 50*(1-(rf-sep)/rf) = 50*(rf-rf+sep)/rf = 50*sep/rf}$$

But we're trying to get the relationship between **st** and **rp**, not **rf**, ...

Recalling that **rp** is halfway between **rs** and **rf**, you can see from the diagram that

$$\mathbf{rf = rp + sep/2} \quad \text{or} \quad \mathbf{2*rf = 2*rp + sep}$$

Substituting for **rf** in the equation we had before:

$$\mathbf{st = 50*sep/rf = 50*sep/(rp + sep/2) = 100*sep/(2*rp + sep)}$$

$$\mathbf{st = 100/(2*rp/sep + 1)} \quad \text{..... equation (3)}$$

Since the effective wheel separation in our case is 10.5 cm, **st = 100/(0.19*rp + 1)**

That expression works correctly for positive values of rp – they will give positive values for st and thus the forward-moving robot will draw a *clockwise* arc (we will assume that arcs will always be drawn with a forward movement of the robot for simplicity). We would like negative values of rp to cause the robot to draw an arc *counterclockwise* but the above expression instead would give an error since st would get a positive value that is greater than 100, which is not valid. The following expression can be used to cause negative values of rp to make the robot draw arcs counterclockwise (negative values for st): **$st = \text{copysign}(100/(0.19*\text{abs}(rp) + 1), rp)$** equation (4)

In the part **$100/(0.19*\text{abs}(rp) + 1)$** we have taken the absolute value of the radius so this part gives the same (valid) steering value for a negative value of rp as we would get with the corresponding positive value.

Then the **$\text{copysign}()$** function is used to make the above expression negative if rp is negative, so the steering value becomes negative and the robot arcs counterclockwise rather than clockwise.

In Python, the **$\text{copysign}(x,y)$** function returns x with the sign of y . It's quite often used with $x=1$, in which case it works like the **$\text{sign}(x)$** function found in many other languages, returning +1.0 if y is positive or zero and -1.0 if y is negative. For example:

```
>>> math.copysign(1, -4)
-1.0
>>> math.copysign(1, 3)
1.0
```

In equation 4 we have used **$\text{copysign}()$** to return the expression **$100/(0.19*\text{abs}(rp) + 1)$** with the sign of rp .

Don't forget to include `from math import copysign` before using the **$\text{copysign}()$** function.

Oof! That was hard work!

Sanity check: To get **either** version of the drawbot to draw a *clockwise* arc with radius of curvature +3 cm, knowing that the effective separation of the wheels is 10.5 cm:

$st = \text{copysign}(100/(0.19*\text{abs}(3) + 1), 3) = 100/(0.19*3 + 1) = 100/(0.57 + 1) = 100/1.57 = 63.7$

To get **either** version of the drawbot to draw a *counterclockwise* arc with radius of curvature -3 cm, knowing that the effective separation of the wheels is 10.5 cm:

$st = \text{copysign}(100/(0.19*\text{abs}(-3) + 1), -3) = -100/(0.19*3 + 1) = -100/(0.57 + 1) = -100/1.57 = -63.7$

The minus sign highlighted in yellow comes from the **$\text{copysign}()$** function which sees that rp is negative and therefore makes the expression negative.

Trying this on my education and home drawbots I get reasonably accurate results – I'll invite you to experiment with your own robot shortly...

Now **we just need to know how many degrees to make the faster wheel turn to get a given angle of arc** (recall that the turn angle in the movesteering commands refers to the faster motor). Initially, let's assume that the arc will be drawn clockwise.

For a full circle (an arc of 360°), the arc length 'd' (the circumference) = $2*\pi*\text{radius}$

For an arc in general, the arc length is therefore equal to **$(\text{arcangle}/360)*2*\pi*\text{radius}$** = $\text{arcangle}*\pi*\text{radius}/180$

In our case, the radius is the radius of curvature of the arc traced by the *faster wheel*, rf

$d = \text{arcangle}*\pi*\text{rf}/180$

but we don't know rf , only rp , so we'll have to substitute for rf

$$d = \text{arcangle} * \pi * (\text{rp} + (\text{sep}/2)) / 180$$

We have been assuming that the arc will be drawn clockwise. In fact, it doesn't make any difference whether the arc is drawn clockwise or counterclockwise because either way the motors need to turn through the same angles, so we should ignore any minus sign in the **rp** value by taking its absolute value:

$$d = \text{arcangle} * \pi * (\text{abs}(\text{rp}) + (\text{sep}/2)) / 180$$

We learnt previously that, for the **education** version of the drawbot, in order to make the robot advance by 1 cm the wheel must turn $360/17.6 = 20.7$ degrees, so to convert the arc length in cm into a wheel turn in degrees we must multiply by 20.7:

$$\begin{aligned} &\text{Wheel turn angle in degrees to make the robot trace an arc with arc angle 'arcangle'} \\ &= 20.7 * (\text{arcangle} * \pi * (\text{abs}(\text{rp}) + (\text{sep}/2)) / 180) = (20.7 * \pi / 180) * (\text{arcangle} * (\text{abs}(\text{rp}) + (\text{sep}/2))) \\ &= 0.361 * \text{arcangle} * (\text{abs}(\text{rp}) + (\text{sep}/2)) \end{aligned}$$

Since the effective wheel separation (sep) in our case is 10.5 cm the wheel angle = $0.361 * \text{arcangle} * (\text{abs}(\text{rp}) + 5.25)$

For example, to make the **education** robot's pen draw a *clockwise* quarter circle (90° arc) with radius +3 cm we must set the faster wheel to turn this number of degrees:

$$0.361 * 90 * (\text{abs}(3) + 5.25) = 0.361 * 90 * 8.25 = 268$$

For a *counterclockwise* quarter circle with radius -3 cm the angle would still be 268 degrees because the absolute value of the radius is being used.

For the **home** version of the drawbot, in order to make the robot advance by 1 cm the wheel must turn 26.84 degrees:

$$\begin{aligned} &\text{Wheel turn angle in degrees to make the robot trace an arc with arc angle 'arcangle'} \\ &= 26.84 * (\text{arcangle} * \pi * (\text{abs}(\text{rp}) + (\text{sep}/2)) / 180) = (26.84 * \pi / 180) * (\text{arcangle} * (\text{abs}(\text{rp}) + (\text{sep}/2))) \\ &= 0.468 * \text{arcangle} * (\text{abs}(\text{rp}) + (\text{sep}/2)) \end{aligned}$$

Since the effective wheel separation (sep) in our case is 10.5 cm the wheel angle = $0.468 * \text{arcangle} * (\text{abs}(\text{rp}) + 5.25)$

For example, to make the **home** robot's pen draw a *clockwise* quarter circle (90° arc) with radius +3 cm we must set the faster wheel to turn this number of degrees:

$$0.468 * 90 * (\text{abs}(3) + 5.25) = 0.468 * 90 * 8.25 = 348$$

For a *counterclockwise* quarter circle with radius -3 cm the angle would still be 348 degrees because the absolute value of the radius is being used.

Summary for Education version

To make the robot **move forward** in a straight line through a given distance, **multiply the distance in cm by 20.7 to obtain the turn angle in degrees for the wheels** (and the motors).

To make the robot **turn on the spot** (steering = +100 or -100), **multiply the desired angle of turn of the robot by 1.9 to find the corresponding turn angle for the wheels** (which will move through equal angles but in opposite directions).

To make the pen **draw an arc** with radius of curvature 'rp' **set the steering value to equal $\text{copysign}(100 / (0.19 * \text{abs}(\text{rp}) + 1), \text{rp})$** . The wheel rotation angle in degrees needed to obtain a given arc angle 'arcangle' for a given radius of curvature can be found with $0.361 * \text{arcangle} * (\text{abs}(\text{rp}) + 5.25)$ (assuming that the effective wheel separation is 10.5 cm).

Summary for Home version

To make the robot **move forward** in a straight line through a given distance, **multiply the distance in cm by 26.84 to obtain the turn angle in degrees for the wheels** (and the motors).

To make the robot **turn on the spot** (steering = +100 or -100), **multiply the desired angle of turn of the robot by 2.46 to find the corresponding turn angle for the wheels** (which will move through equal angles but in opposite directions).

To make the pen **draw an arc** with radius of curvature 'rp' **set the steering value to equal $\text{copysign}(100/(0.19*\text{abs}(\text{rp}) + 1), \text{rp})$** . The wheel rotation angle in degrees needed to obtain a given arc angle 'arcangle' for a given radius of curvature can be found with **$0.468*\text{arcangle}*(\text{abs}(\text{rp}) + 5.25)$** (assuming that the effective wheel separation is 10.5 cm).

Until now we used the labels **rs**, **rf** and **rp** to refer the radius of curvature of the arcs traced by the slower wheel, the faster wheel and the pen, respectively, but from now on we are only interested in the arc traced by the *pen* so we will call its radius of curvature simply **r** rather than **rp**.

We're about ready to attempt our first drawing, but you'll probably want to try making several different drawings and you want the coding to be as easy as possible as you move from one drawing to the next. It would great if we could just write the program using simple commands like these (these commands were used by the Logo programming language which was used to control a 'turtle ' robot):

- **fd(distance)** = go forward
- **bk(distance)** = go backwards
- **lt(angle)** = left turn
- **rt(angle)** = right turn
- **arc(angle, radius)** = draw arc
- **pd()** = pen down
- **pu()** = pen up

Of course our script will also need user-defined subroutines to interpret these commands.

Let's now see whether our formulas work for *your* drawbot (they should work quite well if you followed the build instructions correctly and if you take care to ensure that the pen is carefully centered in its holder and held firmly by the elastic bands).

Let's make a script that simply draws an equilateral triangle with sides 15 cm long, and then draws a circle (an arc with angle 360°) with diameter 12 cm.



We'll assume the robot is initially positioned and pointing as shown by the arrow. We first need to turn the robot 30° clockwise before we start drawing the triangle. (An equilateral triangle is one whose sides all have the same length and whose internal angles are all equal to 60°.) Then we'll draw the triangle clockwise, finishing with a leftwards movement to draw the base of the triangle. Then we'll move into position to draw the circle, starting at the bottom of the circle and drawing clockwise.

The main part of our script would then look like this:

```
rt(30)
pd()
for i in range(3):
```

```

    fd(15)
    if i !=2:
        rt(120)
pu()
bk(20)
pd()
arc(360, 6)
pu()

```

Try to work through the instructions above and make sense of them. There are two special things to note

- We draw the triangle by moving forward and then turning 120° and then repeating that pair of commands twice more *except that we do not want the robot to turn 120° after drawing the base of the triangle*, which explains the line `if i !=2:`. That line says that the right turn should only be done if the loop counter 'i' is not equal to 2. The counter *will* equal 2 as the loop executes for the third and final time.
- You probably expected that the robot would turn 180° to face right after drawing the base of the triangle before heading into position to draw the circle, but that would be a big mistake. The larger the turn angle the bigger the turn error is likely to be, and in fact the robot does not need to turn at all before making a straight line move into the correct position to start drawing the circle. The robot simply needs to move backwards! **In general, you should try to avoid making the robot turn more than 90° on the spot because that would mean that a turn of less than 90 degrees could be used followed by a reversal in the robot's subsequent motion.** Note that I didn't follow that advice when drawing the triangle since it would have made the drawing more complicated.

Our drawbot script assumes that the medium motor's cam piece is initially in the up position – here's a little script (**adjust_cam.py** in the **part4** folder) that you can use to adjust the orientation of the cam if necessary before running the drawbot script:

```

#!/usr/bin/env python3
# Before running the drawbot script, run this script and
# use the left and right buttons
# to make the medium motor cam point upwards.

from ev3dev2.motor import MediumMotor
from ev3dev2.button import Button
from time import sleep

medium_motor = MediumMotor()
btn = Button()

# Press left button to turn medium motor left (counterclockwise)
def left(state):
    if state: # if state = True
        medium_motor.on(speed=-10)
    else:
        medium_motor.stop()

# Press right button to turn medium motor right (clockwise)
def right(state):
    if state:
        medium_motor.on(speed=10)
    else:
        medium_motor.stop()

btn.on_left = left
btn.on_right = right

while True: # This loop checks buttons state continuously,
            # calls appropriate event handlers
    btn.process() # Check for currently pressed buttons.
    # If the new state differs from the old state,

```

```
# call the appropriate button event handlers.
sleep(0.01) # buttons state will be checked every 0.01 second
```

Now that you have ensured that the cam is the up position you can try this full working drawbot script below (**drawbot.py** in the part4 folder). There are notes underneath. Don't forget that it assumes that you have used the script **adjust_cam.py** to ensure that the cam attached to the medium motor is initially in the up position such that the marker is held a couple of mm above the surface to be drawn on.

Note how the user-defined functions make use of all the formulas that we arduously derived earlier. We previously noted that the wheels of the education vehicle are 30% bigger in diameter than the wheels of the home version and that it is therefore often necessary to make the wheels of the home version turn 30% more than the wheels of the education version. Put another way, the wheels of the education version robot need to turn 0.77 times as much as the wheels of the home version robot, and that factor is incorporated into the script as the wheel factor, 'wf'. **Be sure to set the wheel factor correctly for your robot: 1 for the home version and 0.77 for the education version.**

```
#!/usr/bin/env python3
# Use the separate script to ensure the cam is pointing up before running this script
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C, MediumMotor
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
medium_motor = MediumMotor()
sp = 25 # speed, try other values if you like
wf = 1 # wheel factor. Use 1 for home version and 0.77 for edu version
degs_per_cm = 26.84 * wf # degrees of wheel turn per cm advanced
degs_per_robot_deg = 2.46 * wf # angle wheel must turn through
# such that the robot turns one degree
def fd(distance):
    steer_pair.on_for_degrees(steering=0, speed=sp, degrees= distance*degs_per_cm)
def bk(distance):
    steer_pair.on_for_degrees(steering=0, speed=sp, degrees=-distance*degs_per_cm)
def lt(angle):
    steer_pair.on_for_degrees(steering=-100, speed=sp, degrees=angle*degs_per_robot_deg)
def rt(angle):
    steer_pair.on_for_degrees(steering= 100, speed=sp, degrees=angle*degs_per_robot_deg)
def arc(angle, r): # r = radius. Negative radius means draw the arc counterclockwise.
    st = copysign(100/(0.19*abs(r) + 1),r)
    steer_pair.on_for_degrees(steering = st, speed = sp, angle*42.16*wf*(abs(r)+5.25)/90)
def pu():
    medium_motor.on_for_degrees(speed=sp, degrees=180)
def pd():
    medium_motor.on_for_degrees(speed=sp, degrees=180)

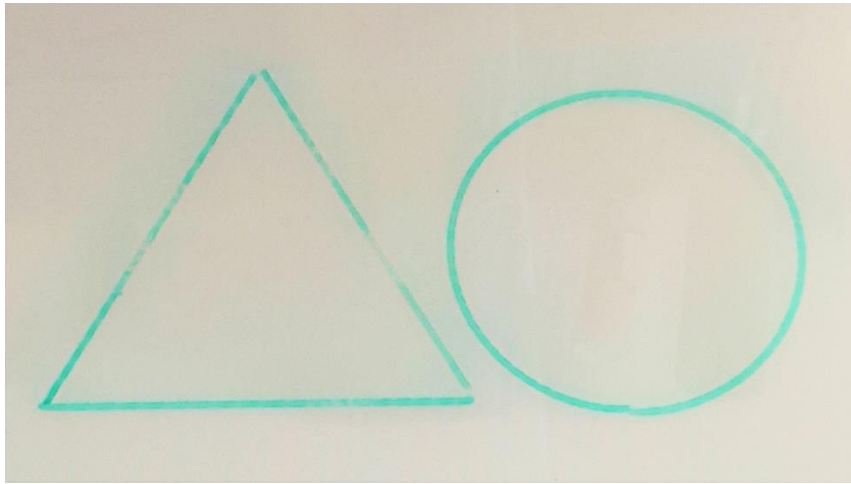
rt(30)
pd()
for i in range(3):
    fd(15)
    if i !=2:
        rt(120)

pu()
bk(20)
pd()
arc(360, 6) # radius of about 6 is not ideal because it causes slower wheel to move TOO
# slowly and therefore it completes its motion slightly after the faster wheel
pu()
```

Notes:

- The functions **pu()** and **pd()** have exactly the same code which makes the medium motor turn 180°. The assumption is that **pu()** will be run when the pen is *down* in order to raise it and that **pd()** will be run when the pen is *up* in order to lower it. The reason to have two separate commands even though they have the same code is for consistency with the Logo language commands.
- In the script above the functions **lt()** is not used – it is included for completeness.

Here is my drawing made with the above script:



When you run the above script my guess is that your result will not be as good as mine, at least initially, and that you will be disappointed by the result! Whereas a similar drawing exercise on screen (with Scratch, for example) can give a perfect result, a drawing made by a robot like this is likely to be far from perfect. Welcome to the real world! I've stressed since the beginning that robots operate in 'messy' and unpredictable environments and that their behavior is often going to suffer from some random variation. In the case of the drawbot there are so many factors that can cause errors: varying smoothness of the surface being drawn on, variations in the effective wheel separation, 'play' in the mechanism that holds the pen, incorrect placement of the pen, the type of tip the pen has (bullet tip is best, chisel tip not so good), elastic bands too loose, etc, etc. Believe me, the script above is the result of many hours of experimentation!

I'd like to point out one particular source of error that you might not have thought of. You've probably noticed that when the EV3 motors are set to move at very slow speeds (less than about 15, for example) they tend to move jerkily. In EV3 Python there is a secondary problem that when set to move slowly, the motors move a little more slowly than they should, and compensate for that by turning for slightly longer than expected, meaning that the slower motor may continue to turn for a moment after the faster motor has stopped. You might be thinking that shouldn't be a problem since we have never set 'speed' to be less than 15, but don't forget that 'speed' refers to the faster motor, not the slower one that I'm referring to. The faster the two wheels move the less of a problem this will be but you wouldn't expect to get accurate drawings if the robot is racing around fast either, so what's the optimum speed setting and should some steer values be avoided? I think you should aim for 'speed' values of between 20 and 30 (for the faster wheel). Steer values of -100, 0 and +100 are safe since in all those cases the motors rotate at the same speeds (though not necessarily in the same direction). So straight line moves and turns on the spot should not suffer from this problem, but the drawing of arcs may. In the drawing you just made, probably the triangle worked better than the circle? It's more difficult to draw arcs accurately than draw straight lines or turn on the spot and I've just given you one reason why this is so. We know that the slower wheel does not move at all with steering = 50 or -50, which means the jerkiness error is also not a problem in that specific case, but *for steer values near 50 or -50 the slow wheel will try to turn slowly and then we'll get our jerkiness problem*. But we're not setting the steering value in our scripts – we're setting the radius of curvature of the arcs. **It turns out that radii in the range 3.5 to 6 cm are the most likely to experience this source of error, so I recommend you try to avoid using radii in that range.**

The circle we just drew had a radius of 6cm which was at the limit of the acceptable range. You may get a better result if you modify the script to draw a circle with radius 9 cm instead of 6 cm. Try it!

If you're disappointed by the drawing you just made you might be tempted to tweak the values but I advise against that for a tweak that helps one drawing may not necessarily help another so you may just waste your time.

Now that we have that script we can easily try to make any number of drawings – just try to follow these guidelines:

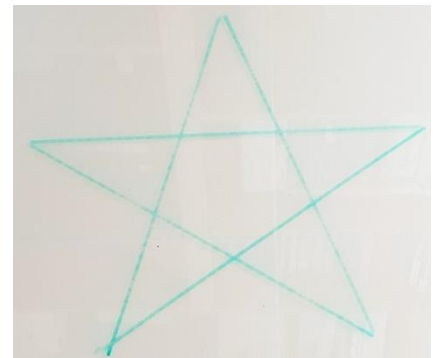
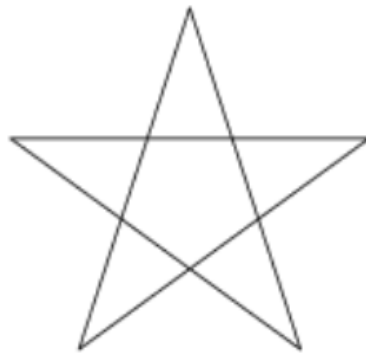
- The more complex the drawing, the more errors will accumulate so don't make it too complex.
- Very small drawings probably won't work well.
- Set the 'speed' value to some value between 20 and 30
- Avoid drawing arcs with radii between 3.5 and 6 cm if possible.

- Make sure the marker is firmly held by elastic bands that are not too loose and make sure the pen is correctly centered in its holder.
- Instead of making the robot turn on the spot more than 90° before making a forward movement, consider making the robot move less than 90° followed by a backwards motion. The bigger the turn angle the less accurate it is likely to be.

So, what would you like to draw? How about a 5-point star? 5-point stars or 'pentagrams' used to represent the "sacred feminine" but in modern American pop culture it more commonly represents devil worship. Perfect! What are the angles in a pentagram? I made you work hard on the math of the drawbot functions so I'll give you a break on this one by telling you that the internal (acute) angle in the point of the pentagram has an angle of 36° (but feel free to work out why that is true). The supplementary angle to 36° is $180 - 36 = 144^\circ$ so you could draw the pentagram by turning 144° after each straight line before drawing the next one but do you remember my advice? Try to avoid turns of more than 90° because they can usually be replaced by a turn of less than 90° followed by a reversal of direction. So instead of turning 144° you should get a better result by just turning 36° and then reversing the direction of the robot for the next straight line. This code could do the trick – it will try to draw a pentagon with sides of 30 cm – about the same as the pentagon drawn in blood in The Da Vinci Code!



```
pd()
for i in range(5):
    if i%2:
        bk(30)
    else:
        fd(30)
    lt(36)
pu()
```



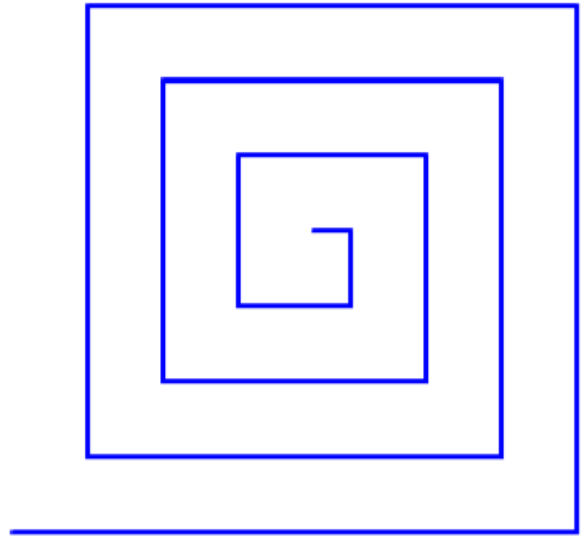
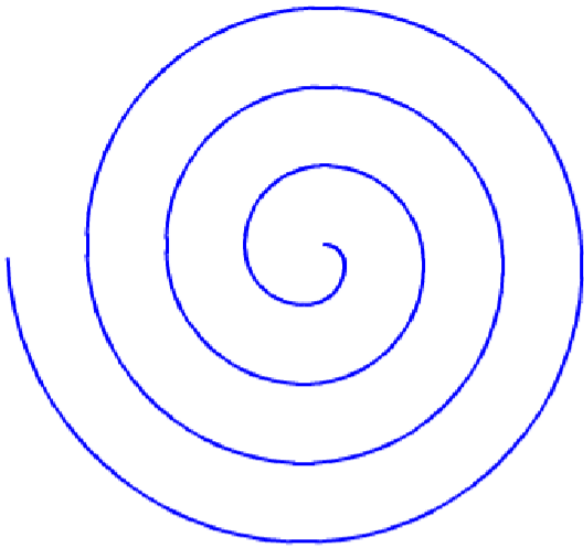
Just substitute this code for the last few lines of our previous drawbot script and you're ready to go! The green pentagram above was drawn by my drawbot running that script.

You are assumed to have some familiarity with Python before you started this course so you probably understand the expression `i%2` and why I used it. Firstly, `if i%2:` is equivalent to `if i%2==True:` Secondly, the '%' sign is used in Python to represent the *modulo* operator, and the modulo operator finds the remainder after division of one number by another. The first time the loop code is run `i` will equal 0 and `0%2` (the remainder when 0 is divided by 2) will also equal 0. In Python, 0 is a 'falsy' meaning it is considered equivalent to False. So the first time the loop code is run the expression will evaluate to False and the code `fd(20)` will run. The next time the loop is run `i` will equal 1 and `1%2` equals 1, a 'truthy' in Python, considered equivalent to True. Therefore the code `bk(20)` will run and we have the alternation between forward motion and backward motion that we wanted.

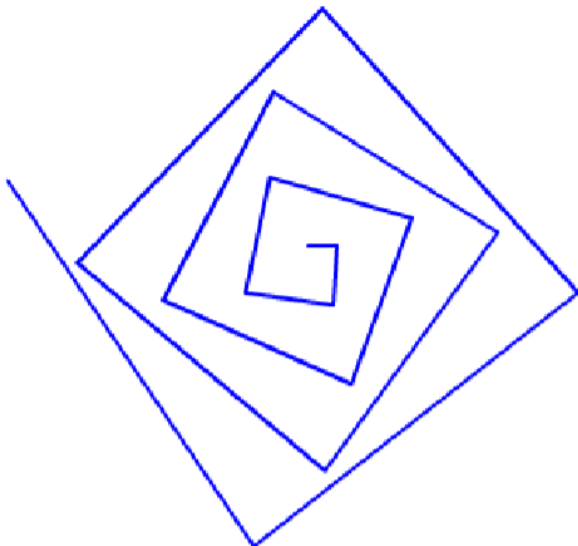
Challenge

Answers to some of the challenge exercises, including these, can be found at the end of this document.

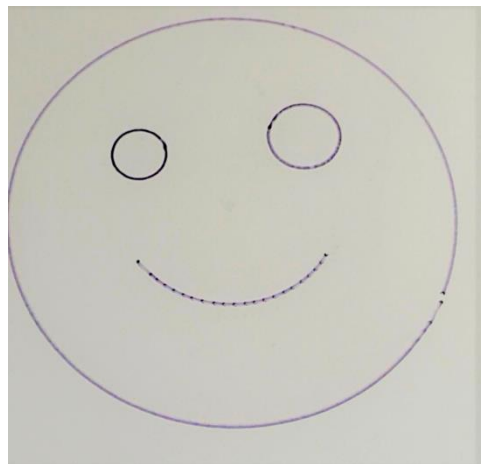
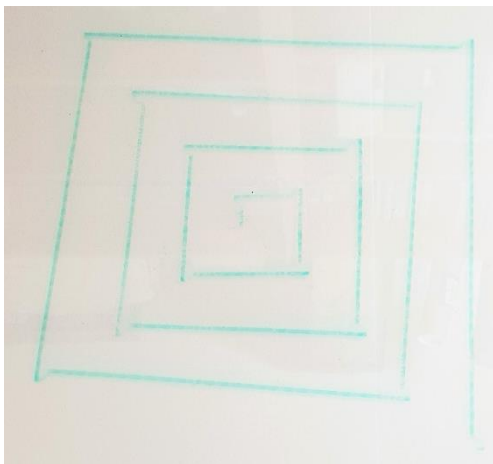
Can you make a spiral and a square spiral ('squiral'?) something like these:



How about these?



Here are the 'squiral' and the smiley drawn by my drawbot:



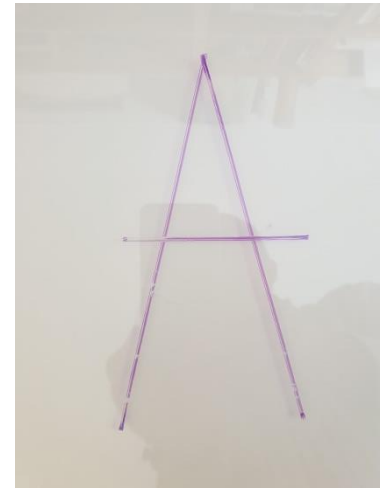


Finally, try this. This last drawing could also be called 'writing' and it is supposed to whet your appetite for the next project, a 'writerbot'. You might want to make each 'unit' of the diagram correspond to 4cm, for example. I know this 'A' look a little odd – you'll understand why later. To reproduce this drawing accurately you will need to think about trigonometry and Pythagoras' theorem. Hint: maybe your code will include this expression:

`degrees (atan (0.25))` ? Recall that Python does trigonometry in radians not degrees but you can convert radians to degrees with the **degrees()** function. If you

want to use the above expression then don't forget the line
`from math import degrees, atan`

Here at right is my attempt at drawing the 'A'. You can see this and the other shapes being drawn in the accompanying videos.



Of course there's no limit to the number of other drawings that can be attempted – have fun!

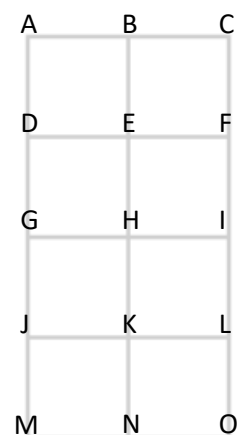
Video 4C: Make a writing robot or 'writerbot' that can write characters that consist only of straight lines.

Having made a drawbot, it seems logical to want to next make a writing robot, or 'writerbot'. Even if we limit ourselves to capital letters and digits (0-9) only, that's 36 characters, every one of which will have to be designed like the 'A' above so that the robot can draw it fairly easily. So this is quite an ambitious project! I could also lead to a very long script – my guess is that in order to make the 'A' above you probably had to do about a dozen moves (turns and straight line movements) so for 36 characters we might be looking at $36 \times 12 = 432$ movements and thus 432 lines of code, not counting another 50 or so lines of code to turn those patterns into actual robot moves. We could be looking at more than 500 lines of code!

So we need to simplify and compress the instructions as much as possible – we can do that by requiring, as we did for the 'A' above, that all the 'corners' of each character should correspond to nodes (intersections) on a grid that is 2 units wide and 4 units tall. We can associate a letter with each node, for example like in this diagram. And we can use the *case* of the letters to indicate whether the pen should be up or down when the robot moves to the node – let's use uppercase to indicate that the pen should be up and lowercase to indicate that the pen should be down.

Already we have a very neat way of specifying the sequence of moves that are needed to make letters that *consist only of straight lines*. For example, to make the letter 'A' that we have already seen (not to be confused with the 'node' A in the top left of the grid) we *could* write (assuming that we start at node M in the bottom left corner): **GiMbo** (there are many other possible solutions).

That means 'first go to node G with the pen up, then to node I with the pen down, then go to node M with the pen up, then to node B with the pen down, then to node O with the pen down.



Now you try one: what sequence would you use to make this letter 'E', assuming that the robot starts in position 'M' (as it always will)? There are many ways to do it, and some are equally good. So that you can compare your answer with mine, I ask you that you draw the lines in this order

1. The short (middle) horizontal line, left to right
2. The top horizontal line, right to left
3. The vertical line, top to bottom
4. The bottom horizontal line, left to right.

Done that? Compare your answer with mine and if they are not the same then GO AND STAND IN THE CORNER!

ghCamo

Don't worry – I 'm not going to ask you propose a sequence for every character like this, but I *will* challenge you to propose *some* of them.

As you can see (hopefully) it's really easy to specify in this way how a letter should be drawn, as long as it is made only of straight lines that conform to our grid. But there are many letters and digits that really need to be made with arcs as well as straight lines, for they don't look good if made with straight lines only.



Take the capital letter 'C', for example. Made from straight lines that snap to our grid it would look like this image at left, but if we allow arcs of radius one unit to be included then we can get this version on the right. Which do you prefer? I thought so!

Actually, drawing curves with a radius of curvature of one unit allows us to design a decent alphabet of characters and digits as shown below.





This project is definitely the most ambitious project we have attempted, in terms of complexity. So that we do not have to wait too long before having a runnable script, we will first make a script that can write only the characters made up only of straight lines, then we will extend the script so that it can also draw the characters that include arcs.

Make a script that can write characters made up only of straight lines

We're almost ready to start coding, but it's not a good idea to start complex coding until we have a well thought-out plan as to how we want the code to work. We would like to be able to specify a string of characters and then have the robot draw each character one by one. In this first stage we are limiting ourselves to writing characters that do not contain arcs, and are also avoiding the characters O-Z that you will later have to design the sequences for yourself (homework!). If we want to make a 'word' then that also means 'no digits', so we are left with these letters: AEFHIKLMN. A five letter word would be good. We know errors will accumulate as the characters are being drawn, so probably it would be unreasonable to ever attempt to write a string of more than 5 characters. So what 5 letter word would you like to write using the letters AEFHIKLMN? I'm going to use the word 'ALIKE' but you are free to make your own word, just remembering to use letters only from AEFHIKLMN and not exceeding 5 letters.

So here is our first line of code, at last:

```
my_string = 'ALIKE'.upper()
```

This is a line of code that we will need to modify later if we want to write different strings. The method `.upper()` is used to make sure `my_string` does not contain any lowercase characters for we have not designed any drawings for lower case characters.

We want to step through the characters one by one and we can do that with
`for char in my_string:`

Now that we have a character from **my_string** stored in a variable called 'char' we can work through the sequence of instructions that will draw that character one movement at a time. The best way to store the sequences is probably in a *'dictionary'* so that each 'key' in the dictionary corresponds to a corresponding 'value' which will be the sequence of grid nodes that must be visited in order to write the letter. See

www.w3schools.com/Python/Python_dictionaries.asp if you need a refresher on dictionaries. We can start making our dictionary of letters that consist only of straight lines like this:

```
sequence={'A':'GiMbo', 'E':'b1e2l1RDr', 'F':'b1e2l1RD2r'}
```

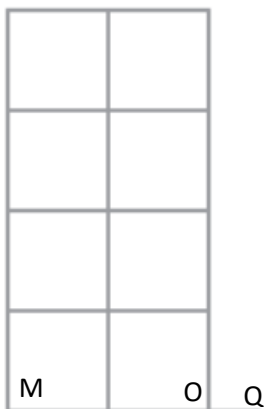
Note that dictionaries are written with curly brackets (also called 'curly braces' or just 'braces'), as above. That's about the limit for a legible line of code, but it only contains the sequences for the letters A, E and F. We can add other items to the dictionary like this:

```
sequence.update({'H':'aGiCo', 'I':'AcBnMo', 'K':'aGCgo', 'L':'Amo'})
```

I'm just planning a coding strategy here, I'm not suggesting that these lines will appear in this order or that other lines won't come first, such as all the import lines.

With a dictionary called 'sequence' set up as above we can refer to the value (the sequence) that corresponds to a certain letter key with (sequence[char]). For example, sequence['H'] would give us **'aGiCo'**. So we can step through the sequence that corresponds to a certain letter to be drawn with

```
for item in (sequence[char]):
```



We have said that we will always be in the bottom left corner (node M) when we start to draw a character, but how can we make sure we are in fact in that position after drawing a previous letter? Also, how do we make sure that there is a suitable space between each character and the next? My solution to both these problems is to add an extra node to the grid as shown. I will call the new node 'Q' and not 'P' since 'P' might confusingly be assumed to refer to the pen in some way. We will include code in our script that will make the robot move to node Q after visiting the last node specified in the drawing sequence – that way it will be in the right place for starting the next letter. My experiments suggest that if node Q is 0.6 units away from node O that gives a normal-looking space between the characters.

We want the robot to move to node Q after drawing a letter so we can add Q to the sequence of nodes to be visited like this:

```
for item in (sequence[char]+'Q'):
```

To actually make the robot move to the intended destination node we'll code a user-defined function called **move_straight()** which will take 'item' as its argument like this:

```
move_straight(item)
```

So far we've got few lines of code for our script that look like this:

```
my_string = 'ALIKE'.upper()
```

```
sequence={'A':'GiMbo', 'E':'b1e2l1RDr', 'F':'b1e2l1RD2r'}
```

```
sequence.update({'H':'aGiCo', 'I':'AcBnMo', 'K':'aGCgo', 'L':'Amo'})
```

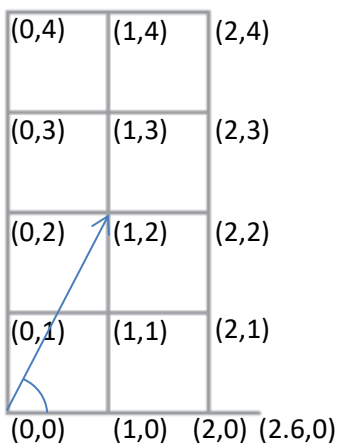
```
for char in my_string:
```

```
    for item in (sequence[char]+'Q'):
```

```
        move_straight(item)
```

Now the tricky part, the writing of the **move_straight ()** function. It's tricky because, unlike for the drawbot, all we have for each movement is a *destination* as opposed to a turn or straight line motion that would be easy to code. The movements needed in order to reach the destination depend on the location and the orientation or 'heading' of

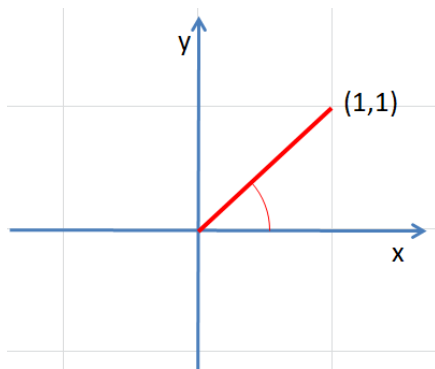
the robot before it makes the move. Since we'll have to do some maths to calculate the distances and angles for our movement, let's start using mathematical coordinates for the nodes, like this:



Now that we have these numerical coordinates it shouldn't be too hard to figure out how far we need to move and in what direction to get from one node to another. For example, if we want to get from the starting node at bottom left (0,0) to the central node, as shown, then we are moving along the hypotenuse of a right triangle with arm lengths 1 and 2, so Pythagoras gives us the length of the hypotenuse as $\text{sqrt}(2**2 + 1**2) = \text{sqrt}(5) = 2.236$.

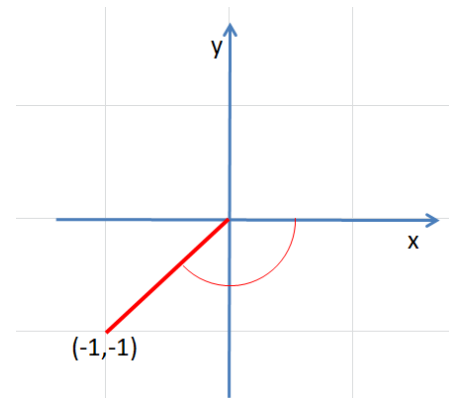
How about the angle that we need to point at – how do we measure angles? Well, we are using the *Cartesian coordinate system* and the conventional way to measure angles in that context is measure *counterclockwise from the x-axis*, so that's what we'll do for the time being, but be aware that for *navigation* purposes the convention is to measure angles (headings) clockwise from 'north' so we will have to convert from the 'Cartesian' way of measuring angles to the 'navigation' way later on. The angle we need then, shown in the diagram as

measured counterclockwise from the x-axis, can be best found with the Python function **atan2(y,x)**. You may be familiar with the Python function **atan(x)** which returns the arctangent of x, *in radians*, but are you familiar with the more powerful function **atan2(y,x)**? Because this function knows the sign of *both x and y* it can compute the correct *quadrant* for the angle, something that **atan()** cannot do.



Consider the angle in this diagram at left. **atan(1)** and **atan2(1,1)** will both give the same answer, 0.785 radians (45°).

For the diagram at right, **atan(-1/-1) = atan(1) = 0.785 radians (45°)**, which is the same result as for the angle in the left diagram, even though the lines are in different quadrants. But **atan2(-1,-1) = -2.356 radians (-135°)**, which tells us that in this case the angle is in the bottom-left quadrant (a positive angle goes



counterclockwise from the x-axis so a negative angle goes clockwise). For more about the **atan2()** function, visit www.tutorialspoint.com/Python/number_atan2.htm.

In the above paragraphs, I converted for you the angles in radians returned by the functions into degrees. **Python's trig functions always use radians** but we can easily convert from radians to degrees with the **degrees()** function. We can also convert degrees to radians with the **radians()** function, of course. Here's a screenshot from Visual Studio Code's terminal panel on my computer. You should be able to open the Python interpreter in the terminal panel by typing 'Python' (and not 'Python3'). If this does not work for you in Windows, it probably means that the path to the Python interpreter

```
>>> from math import atan, atan2, degrees
>>> print(atan(1))
0.7853981633974483
>>> print(degrees(atan(1)))
45.0
>>> print(atan2(-1,-1))
-2.356194490192345
>>> print(degrees(atan2(-1,-1)))
-135.0
>>>
```

(Python.exe) is not included in your Windows path variable. Go to this address to learn how to add Python to your Windows path: geek-university.com/Python/add-Python-to-the-windows-path/ You will need to know the location of your Python interpreter in order to follow the instructions. Note that in recent version of Windows the dialog for modifying the Windows path has been improved to make the procedure more fool-proof. You may have multiple Python interpreters on your computer – note that you can choose which one you want VS Code to use by clicking the name of the interpreter at the left of the status bar at the bottom of the VS Code window.

We can convert an angle in degrees measured the 'Cartesian' way we have just seen into a 'heading' measured clockwise from north with this formula: **(90-angle)%360**. In this screenshot you can see the formula being used to

convert the -135° that we had in the above diagram into a 'heading'. **>>> print((90--135)%360)**
225 The answer, 225°.

is correct for if you turn 180° clockwise from north and then an additional 45° then you will be pointing in the correct direction. The '%360' in the formula uses the '%' or 'modulo' operator which gives the remainder when one number is divided by another. We are using it here to make sure the heading is always between 0° and 360° (no negative angles) since that is the convention.

As an example, here is an aerial photo of one end of a runway at my local airport in Nice. To land at this end of this runway you must have a heading of 220° - it is not marked '-14' even though a heading of -140° would be the same as a heading of 220°. (By the way, what number would you expect to see at the other end of the same runway?)



So now we know not only how to determine the distance from one node to another (using Pythagoras) but also the heading, using the formula

$$(90 - (\text{degrees}(\text{atan2}(\text{dy}, \text{dx})))) \% 360$$

where **dy** and **dx** are the differences in the y coordinates and the x coordinates of the source and destination nodes of our desired move. For example, let's try the move represented by the blue arrow in this diagram. We want to go from node L at (2,1) to node A at (0,4) so the distance will be (from Pythagoras)

$$\text{sqrt}(2^2 + 4^2) = \text{sqrt}(4 + 16) = \text{sqrt}(20) = 4.472$$

and the heading angle shown will be (using **dx** = 0-2=-2 and **dy** = 4-1=3):

$$(90 - (\text{degrees}(\text{atan2}(3, -2)))) \% 360 = 326.3^\circ$$

Not convinced? Here's the screenshot:

```
>>> print((90-(degrees(atan2(3,-2))))%360)
326.30993247402023
```

If you find the above formula too difficult to follow then break it down into two steps. First find the angle in degrees measured from the x-axis (counterclockwise being positive):

$$\text{degrees}(\text{atan2}(3, -2)) = 123.7^\circ$$

then convert that angle into a heading measured from 'north' with clockwise being positive:

$$(90 - 123.7) \% 360 = 326.3$$

Would you be able to work out the distance and heading that correspond to the red arrow in the diagram? Obviously you will need a scientific calculator or the Python interpreter. Once you've had a go yourself, compare your answers with these:

$$\text{Distance} = \text{sqrt}(2^2 + 1^2) = \text{sqrt}(4 + 1) = \text{sqrt}(5) = 2.236$$

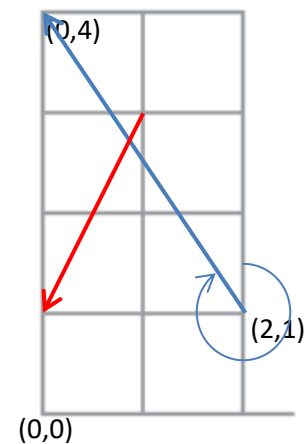
$$\text{Heading (with dx = -1 and dy = -2)} = (90 - (\text{degrees}(\text{atan2}(-2, -1)))) \% 360 = 206.6^\circ$$

Still with me? Congratulations! This is pretty serious maths we're doing! But since you already have considerable experience coding you must have figured out long ago that there's lots of maths in coding, right?

When the robot is at a certain node and is about to head on to the next one, the angle that the robot must turn through before advancing is usually NOT the same as the heading of the next node. That would only be true if the robot happened to be pointing north, which will usually not be the case. We have to keep track of the heading of the robot at all times and then calculate the turn angle as the *difference* between the robot's current heading and the heading it must get in order to head towards the next node. If you think about it, it makes no sense for the robot ever to turn more than 180°. We can make sure that the robot's turn angle is always in the range -180° to +180° with this line (I'll let you figure out why this is true...):

```
turn_angle = ((turn_angle + 180) % 360) - 180
```

Let's call the robot's current heading '**current_heading**' and the heading that the robot has to get before moving on '**new_heading**'. We've decided to have the robot pointing to the right (east) when the script is launched so we will set **current_heading** to equal 90 initially (90° clockwise from north). Let's also call the robot's current location (as node coordinates rather than a node letter) '**current_node**'. Each time the robot is about to write a letter we will reset **current_node** to (0,0) i.e. bottom-left before starting to write the letter.



We've spent so long working on this project already so you must be itching to see and try a working script! The script below is only intended to work with characters that *do not contain arcs*. Notice how coordinates have been associated with the node letters within a *dictionary*, just as we already discussed associating the letters to be drawn with a sequence of nodes in a dictionary. Notice also that I have not given sequences of nodes for the letters O to Z since your homework will be to design those sequences yourself!

Before you try to run this script (**writer1.py** in the **part4** folder), **be sure to set the wheel factor 'wf' to match your EV3 version**. The wheels of the home version have a diameter that is 77% of the diameter of the wheels of the education version and to compensate for that difference it is necessary to make the wheels of the education version turn only 77% as much to get the same forward motion. Also **make sure that the pen cam is in the upward position before running the script** – you can do that by running the same adjustment script that we used for the drawbot.

```
#!/usr/bin/env python3
# Ensure (using the separate script) that the pen is raised before launching this script
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C, MediumMotor
from time import sleep
from math import sqrt, pi, atan2, degrees
from ev3dev2.sound import Sound
from time import sleep

medium_motor = MediumMotor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
sound = Sound()

my_string = 'ALIKE'.upper() # Use only characters AEFHIKLMN. upper() converts lowercase to upper
wf = 1 # wheel factor. Use 1 for home version and 0.77 for edu version.
scl = 3 # Scale. scl=3 gives 3cm per grid unit. Use scl values between 3 and 5.
sp = 20 # speed of steer_pair. Use values between 15 and 30.
sep = 10.5 # effective wheel separation in centimeters.
degs_per_cm = 26.84 * wf # degrees of wheel turn per cm advanced.
degs_per_robot_deg = 2.46 * wf # angle wheels turn when robot turns one degree on the spot
medium_motor.position = 0 # needed to protect against a bug.
# 'direction' is the direction from the current node to the next node, measured cw from 'north'.
heading = 90 # 'heading' is the direction the robot is facing, measured clockwise from 'north'.

sequence={'A':'GiMbo','E':'GhCamo','F':'GhCam','H':'aGiCo','I':'AcBnMo'}
sequence.update({'K':'aCgo','L':'Amo','M':'ahco','N':'aoc'})

node={'a':(0,4),'b':(1,4),'c':(2,4),'d':(0,3),'e':(1,3),'f':(2,3),'g':(0,2)}
node.update({'h':(1,2),'i':(2,2),'j':(0,1),'k':(1,1)})
node.update({'l':(2,1),'m':(0,0),'n':(1,0),'o':(2,0),'q':(2.6,0)}) # Note the value of 'q'

def pu():
    if abs(medium_motor.position) > 10: # if pen is not already up
        medium_motor.on_to_position(speed=50, position=0)

def pd():
    if abs(medium_motor.position-180) > 10: # if pen is not already down
        medium_motor.on_to_position(speed=50, position=180)

def get_dist_and_dir(letter):
    dx = node[letter][0] - current_node[0]
    dy = node[letter][1] - current_node[1]
    distance = sqrt(dx**2+dy**2) # Pythagoras!
    angle = atan2(dy,dx) # 'Cartesian' angle in radians ccw from east
    # convert to angle in degrees cw from north, in range 0-360
    direction = (90-degrees(angle))%360
    return distance, direction # as a TUPLE

def move_straight(letter):
    global current_node, heading
    if letter.islower(): # lower the pen if necessary
        pd()
    else: # raise the pen if necessary
        pu()
    letter = letter.lower() # set the letter to lower case
    distance, direction = get_dist_and_dir(letter)
    turn_angle = direction - heading
    turn_angle = ((turn_angle+180)%360)-180 # get turn angle into range -180° to 180°
    turn(turn_angle) # make the turn!
    advance(distance) # make the robot advance!
```

```

heading = heading + turn_angle # modify heading since the robot has just turned
current_node = [node[letter][0], node[letter][1]] # update current_node

def turn(turn_angle):
    steer_pair.on_for_degrees(steering=100,speed=sp,degrees=turn_angle*deg_per_robot_deg)
def advance(distance):
    degs = distance*scl*deg_per_cm
    steer_pair.on_for_degrees(steering=0,speed=sp,degrees=degs)

for char in my_string:
    sound.speak(char)
    # set current_node to equal [0,0] (node M) each time we start writing a character
    current_node=[0,0]
    for letter in (sequence[char]+'Q'): # add Q to put space between the characters
        move_straight(letter)

```

Notes on the above script:

- The speed value used by the `steer_pair` functions is held in a variable called '**sp**' so that you can change it easily, though I recommend that you don't change it much – speeds between 15 and 30 might be okay.
- The script includes a scale factor '**scl**' so that you can choose how big you want the letters to be. The script sets **scl** to equal 3 so that each 'grid unit' will correspond to 3 cm and therefore the letters should be about 12 cm tall. Again, you won't want to change this much. I recommend you use values between 3 and 5.
- This script uses the **global** keyword (highlighted). As you probably know, use of the global keyword is frowned upon. An alternative is to pass the variables to the functions as arguments, and this will be discussed further later.
- The neatest way to raise and lower the pen is to use the **on_to_position()** function, as above, but as of January 2019 this function has a slight bug in that if the motor is disconnected and reconnected before the script is run then the **position** value is not initialized to zero correctly as the script launches. The workaround for this bug is to include the line `medium_motor.position = 0` as above.
- The same function **on_to_position()** also has a bug that may cause the program to freeze if the motor is instructed to turn through an angle of less than 10 degrees, so lines are included in the **pu()** and **pd()** definitions to prevent that from happening.
- The script includes lines to make the EV3 speak the letter it is about to draw. The program is blocked (paused) while it does this and that pause is helpful since it makes it clear which movements belong to which letter.
- %360 appears several times in the code as a way to ensure that the value is always in the range 0° to 360°, which is the conventional way to express headings.
- Unlike 'heading', which should by convention be in the range 0° to 360°, the turn angle should be in the range -180° to +180°. It would be stupid to ever make the robot turn 359°, for example!

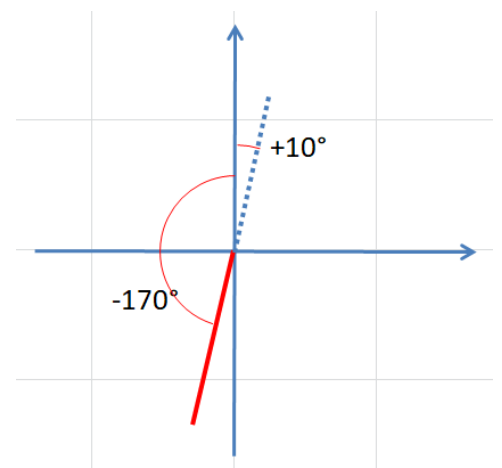
You were probably a bit disappointed by the accuracy of the writing, just as you were by the accuracy of the drawings done by the drawbot. You have to remember that the writerbot is making a lot of movements and that the errors that accompany each movement tend to accumulate. Instead of being disappointed, perhaps you should be thankful for being reminded about this important distinction between programming robots and programming for the computer screen: robots are badly behaved! Or rather they are subject to the messiness of the real world.

But the inaccuracy of the robot's writing is partly due to the robot doing something very stupid as it writes – it sometimes turns around through a large angle, or even 180°, when it only needs to turn through a small one or not at all. Sometimes, for example, the robot travels north with the pen up and then turns around 180° before drawing a line towards the south. Instead of turning, why not simply reverse? We discussed this in the context of the drawbot, when we said that ideally the robot should never turn more than 90°, but this time we have to put that extra intelligence into the code.

We can test whether **turn_angle** is greater than 90° with

```
if turn_angle > 90:
```

If it is, then we want to find the corresponding angle in the opposite quadrant expressed so that it is between -90° and +90°. For example, we would want -170° to become +10° as shown. But we can't just add 180° because that wouldn't work for positive angles such as 170° which would become 350° when it *should* become -10°. We have two choices: use a



more complex 'if' structure or use a clever mathematical expression. I've gone for the latter and derived this expression which does the job:

```
turn_angle = ((turn_angle+270)%360)-90
```

Can you figure that out? Try putting in a value like -170° . Does the expression give you $+10^\circ$? It should! When you put in 170° do you get -10° ? You should!

When the **turn_angle** is 'inverted' in this way, does that automatically mean that the robot should toggle between driving forwards and driving backwards? In the above example, *if* the robot had been moving forwards before changing its heading then it *does* need to reverse in order to move in the direction -170° (from now on you mustn't confuse the 'heading' of the robot, which is the direction in which it is *facing*, and the direction that it is actually *moving*, which might be the opposite of its heading. Isn't it amazing how complex this project is getting, just because of the interaction of a half dozen variables? You have to admire the coders who work on very long scripts with many more variables!

This section could get very long if I address the 'toggle direction?' question in detail, so instead I'll present you with two rules and invite you to contemplate them yourself:

- If **abs(turn_angle)** is less than 90 then the robot should move *forwards*.
- If **abs(turn_angle)** is greater than 90 before **turn_angle** is forced into the range -90 to +90 then the robot should move *backwards*.

We'll use a Boolean variable **go_forwards** to keep track of whether the robot is in 'go forward' mode or 'go backwards' mode. If **go_forwards** is True that means the robot is in 'go forward' mode. The full 'if' structure that will invert the angle and toggle the **go_forwards** variable if necessary is now:

```
if abs(turn_angle)>90:
    turn_angle = ((turn_angle+270)%360)-90 # get into range -90° to +90°
    go_forwards = False
else:
    go_forwards = True
```

We'll need to initialize **go_forwards** to True near the beginning of the script since the robot will start writing with a forward motion. Lastly, we need to modify the **advance()** function to take into account the value of the **go_forwards** variable. If **go_forwards** is False then the motor's turn angle needs to be made negative i.e. multiplied by -1. In a mathematical expression, the value True is treated as +1 and the value False is treated as zero, but the following expression will be equal to +1 if **go_forwards** is True and -1 if **go_forwards** is False: $(-1+2*\text{go_forwards})$. That expression $(-1+2*x)$ to convert x from True/False to +1/-1 is so useful you will see it being used several times in future scripts.

Here's the modified script (**writer2.py** in the **part4** folder), ready for you to try. Don't forget to set the wheel factor (wf) value to match your version of the EV3 kit. You may also want to write a different string to the one given, using the permitted characters and not exceeding five characters. Some comments that were in the last script have been omitted for brevity and modified code has been highlighted.

```
#!/usr/bin/env python3
# Ensure (using the separate script) that the pen is raised before launching this script
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C, MediumMotor
from sys import stderr
from math import sqrt, pi, atan2, degrees
from ev3dev2.sound import Sound
from time import sleep

medium_motor = MediumMotor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
sound = Sound()

my_string = 'ALIKE'.upper() # Use only characters AEFHIKLMN. upper() converts lowercase to upper
wf = 1 # wheel factor. Use 1 for home version and 0.77 for edu version.
scl = 3 # Scale. scl=4 gives 4cm per grid unit. Use scl values between 3 and 6.
sp = 20 # speed of steer_pair. Use values between 15 and 25.
sep = 10.5 # effective wheel separation in centimeters.
deg_per_cm = 26.84 * wf # degrees of wheel turn per cm advanced.
deg_per_robot_deg = 2.46 * wf # angle wheels turn when robot turns one degree on the spot
```

```

medium_motor.position = 0 # needed to protect against a bug in on_to_position()
# 'direction' is the direction from the current node to the next node, measured cw from 'north'.
heading = 90 # 'heading' is the direction the robot is facing, measured clockwise from 'north'.
go_forwards = True # initially the robot will move forwards (for positive distance values)

sequence={'A':'GiMbo','E':'GhCamo','F':'GhCam','H':'aGiCo','I':'AcBnMo'}
sequence.update({'K':'aCgo','L':'Amo','M':'ahco','N':'aoc'})

node={'a':(0,4),'b':(1,4),'c':(2,4),'d':(0,3),'e':(1,3),'f':(2,3),'g':(0,2)}
node.update({'h':(1,2),'i':(2,2),'j':(0,1),'k':(1,1)})
node.update({'l':(2,1),'m':(0,0),'n':(1,0),'o':(2,0),'q':(2.6,0)})

def pu():
    if abs(medium_motor.position) > 10: # if pen is not already up
        medium_motor.on_to_position(speed=50, position=0)

def pd():
    if abs(medium_motor.position-180) > 10: # if pen is not already down
        medium_motor.on_to_position(speed=50, position=180)

def get_dist_and_dir(letter):
    dx = node[letter][0] - current_node[0]
    dy = node[letter][1] - current_node[1]
    distance = sqrt(dx**2+dy**2) # Pythagoras!
    angle = atan2(dy,dx) # 'Cartesian' angle in radians ccw from east
    # convert to angle in degrees cw from north, in range 0-360
    direction = (90-degrees(angle))%360
    print('distance:'+str(round(distance,1))+ ' direction:'+str(int(direction)),file=stderr)
    return distance, direction # as a TUPLE

def move_straight(letter):
    global current_node, heading
    print('node letter:'+letter+' robot initial heading:'+str(int(heading)),file=stderr)
    if letter.islower(): # lower the pen if necessary
        pd()
    else: # raise the pen if necessary
        pu()
    letter = letter.lower() # set the letter to lower case
    distance, direction = get_dist_and_dir(letter)
    turn_angle = direction - heading
    turn_angle = ((turn_angle+180)%360)-180 # get turn angle into range -180° to 180°
    if abs(turn_angle)>90:
        turn_angle = ((turn_angle+270)%360)-90 # get into range -90° to 90°
        go_forwards = False
    else:
        go_forwards = True
    print('turn_angle:'+str(int(turn_angle))+ ' go_forwards:'+str(go_forwards),file=stderr)
    print('medium_motor.position (0=up, 180=down): '+str(int(medium_motor.position))+'\n',file=stderr)
    turn(turn_angle) # make the turn!
    advance(distance, go_forwards) # make the robot advance!
    heading = heading + turn_angle # modify heading since the robot has just turned
    current_node = [node[letter][0], node[letter][1]] # update current_node

def turn(turn_angle):
    steer_pair.on_for_degrees(steering=100,speed=sp,degrees=turn_angle*degs_per_robot_deg)

def advance(distance, go_forwards):
    degrees = (-1+2*go_forwards)*distance*scl*degs_per_cm
    steer_pair.on_for_degrees(steering=0,speed=sp,degrees=degrees)

for char in my_string:
    sound.speak(char)
    print('Letter to write: '+char+'\n', file=stderr)
    # set current_node to equal [0,0] (node M) each time we start writing a character
    current_node=[0,0]
    for letter in (sequence[char]+'Q'): # add Q to put space between the characters
        move_straight(letter)

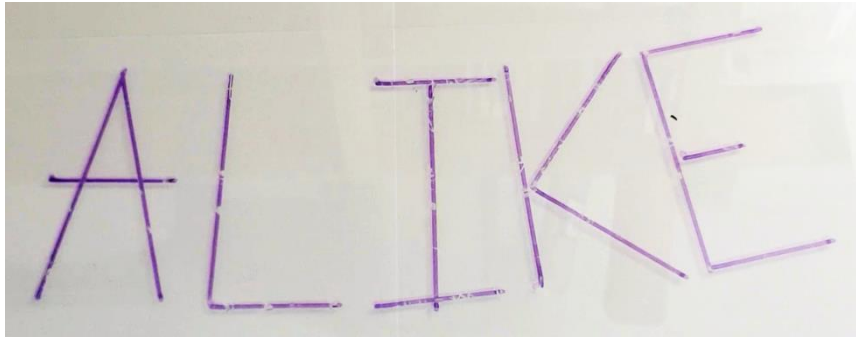
```

So, does the modified script write more accurately than the previous one? Did you notice the robot turning less and sometimes moving backwards?

Notes on the above script

- Two lines cause certain information to be printed to the VS Code's output panel (assuming you launch the script from VS Code and not from Brickman). This information helps us to understand what the script is doing and is can be very helpful in debugging. Many of the powerful debugging features that VS Code makes available to debug locally run programs cannot be used for programs run remotely as we are doing, so this trick of printing key information to the VS Code output panel is extremely helpful during script development.
- `\n` represents the new line character called Line Feed (LF).

Here is my result from running the above script with the education version. Note how errors have begun to accumulate towards the end, as is perhaps inevitable for this type of free-roaming writerbot which has to complete about 50 moves (straight line moves and turns) to write this word:



Challenge: update the dictionary called 'sequence' so that it includes node sequences for all the other letters that do not contain arcs (add **TVWXYZ**).

Now for the grand finale: modifying the script so that it can also draw characters with arcs...

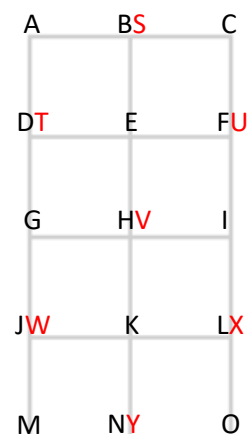
Video 4D: Extend the writerbot code to enable it to also write characters that contain arcs.

As we did with the drawing of straight lines, we will start by drawing arcs always with the robot moving forwards to simplify the coding, and then we'll think about making the robot move backwards sometimes to draw arcs if that will allow for smaller turn angles and thus greater precision.

Looking through the alphabet of characters that we saw previously, you may notice something that is very good news for our coding: all the arcs begin and end on one of the following nodes: BDFHJLN

But how will we tell our program to go to node 'D', for example, by drawing an arc instead of a straight line? It's not enough to tell the program to draw an arc, we also have to tell it whether the arc should curve be drawn with a clockwise or counterclockwise motion. We have already labeled the grid nodes with the letters A to O to indicate so that we can indicate which nodes the robot should move to *with straight line movements* – are there enough letters left to be able to give information about how *arcs* should be drawn? Yes, as shown in the diagram at right. You might want to copy that diagram onto a scrap of paper – you'll find that useful as we proceed.

We'll use the letters S through Y to indicate that the robot should draw an arc with radius one unit. If we include an 'S' in our sequence, that could mean 'go to node BS by drawing an arc with a clockwise motion'. ('Node B', 'node S' and 'node BS' all refer to the same location.) Furthermore, we can use the *case* to differentiate between clockwise arcs and counterclockwise arcs. So an 's' in our sequence would mean 'go to node BS with a *counterclockwise* motion'. **Let's decide to use *uppercase* to mean *clockwise* motions and *lowercase* to mean *counterclockwise* motions.**



But didn't we decide previously that we would use case to indicate whether the pen should be *up* or *down* for each movement? Yes, we did, for *straight line* movements. But nobody in their right mind would choose to have the robot move along an arc to a new location with the pen up since the move could be accomplished more accurately with a straight line move and the movement is not going to leave a trace anyway since the pen is up! Since we can be sure that arcs will always be drawn with the pen down we have the possibility of using the case for a different purpose. To summarize:

- For straight lines we will use case to indicate whether the pen should be *up* or *down*.
- For drawing arcs, the pen will *always be down*, therefore we can use case to indicate whether the arc should be drawn with a *clockwise* or *counterclockwise* motion.

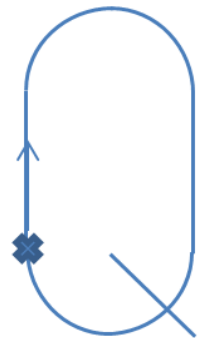
As an example of a sequence to draw a character with curves, consider this sequence: **Ftjx**.

Can you figure out what character that would draw (don't forget the robot will always start at node 'M')? Yes, that sequence would draw a 'C'. Here's the analysis:

- 'F' means go to node F with the pen up (for straight lines, upper case means 'pen up')
- 't' means draw a counterclockwise arc from the current position (node F) to the new position, node T. (Node T is the same location as node D). This will be drawn with the pen down since arcs will *always* be drawn with the pen down.
- 'j' means move to node J with a straight line motion with the pen down.
- 'x' means draw a counterclockwise arc from the current position (node J) to the new position, node X. The pen will be down, as always for arcs.

Your turn! Can you give a sequence for the drawing of the letter 'Q' as depicted above? There are many possible ways to draw the letter, some of which are equally good, but so that you can compare your solution with mine I ask you to respect these rules:

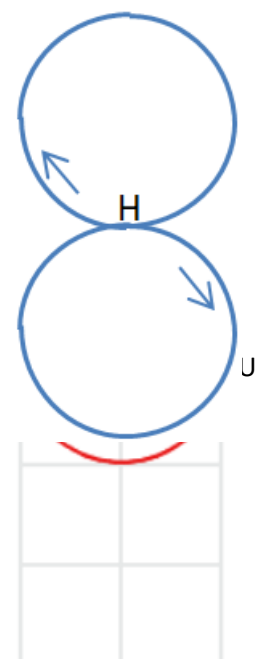
- Assume that the robot starts at the bottom left as usual, at node M, which is below the X in this diagram.
- Start drawing with the pen down at node JW (the X in this image) with an upward movement.
- Make the final short stroke in the bottom right corner with a movement from upper left to lower right.



Finished? Then compare your solution with mine: **JdUIWko**. (The fourth character is 'l' as in 'lion' and not 'l' as in 'India'.) Here's the logic:

- First move to position J with the pen up.
- Then move to node D with the pen down (so lowercase).
- Then arc clockwise to node U (or F or FU – it's all the same place). The pen will be down because it is always down for arcs.
- Then move to node L with the pen down (so lowercase).
- Then arc clockwise to node JW. The pen will be down.
- Then move to node K with the pen up.
- Then move to node O with the pen down (so lowercase).

OK, we're nearly ready to do the coding, but first we need to be aware of certain problems drawing three-quarter circles or full circles. Some of the digit characters (6, 8, 9) contain circles, and in every case the circle touches the central node, node HV. If the robot happens to be at the central node (HV) and we give it the instruction 'V' then we are telling it to draw an arc in the clockwise direction which in this case must be a circle, since it ends where it started. But this instruction is ambiguous – it doesn't make it clear whether the circle should be in the *top* half of the grid or in the *bottom* half (see image). The simplest solution is *never* to give an instruction to draw a *circle* but instead give *two* instructions to draw *semicircles*. For example if you wanted to draw a circle in the *lower* half of the grid, starting at point HV and moving clockwise (as in the character '6', perhaps) then instead of saying 'V' you could say 'YV' (draw a clockwise semicircle to NY then a clockwise semicircle to HV).



Also, if we allow for three-quarter circles to be drawn then we are confronted with another ambiguity. For example, if the robot is in location FU and we give the instruction 'S' then we

telling the robot to make a clockwise arc from node FU to node BS but there are two way of doing that (with arcs of radius 1 grid unit, as always), as shown in the diagram. The easiest way of removing that ambiguity is to do what we did for circles: insist that the motion be broken into two motions, in this case we could split the red three-quarter circle into a semicircle and a quarter circle.

Coding time!

We've used a function **move_straight()** to make the robot move in a straight line from one node to another, sometimes drawing a line as it does so. Let's make a new function **draw_arc()** to ... draw arcs. For each instruction in the form of a character we need to send that instruction to the right function. If the instruction character is a letter in the range A to Q then we know it is an instruction to move in a straight line and if it is an instruction character in the range S to Y then we know it is an instruction to draw an arc.

To send the characters to the correct function we can thus modify the main block of code like this (the new lines are highlighted).

```
for char in my_string:
    sound.speak(char)
    print('Letter to write: '+char+'\n', file=stderr)
    # set current_node to equal [0,0] each time we start writing a character
    current_node=[0,0]
    for letter in (sequence[char]+'Q'): # add Q to put space between the characters
        arc_nodes = 'STUVWXY'
        if letter.upper() not in arc_nodes:
            current_node, heading = move_straight(letter, current_node, heading)
        else:
            current_node, heading = draw_arc(letter, current_node, heading)
```

The new code in bold type above is needed so that we can avoid using the *global* keyword – more on that later. Now we can write the **draw_arc()** function definition:

```
def draw_arc(letter, current_node, heading): # so that we can avoid using the global keyword
    pd() # make sure the pen is down
    print('node letter:'+letter+' robot heading:'+str(int(heading)),file=stderr)
    cw = letter.isupper() # uppercase means draw arc clockwise (cw)
    letter = letter.lower() # set the letter to lower case
    distance, direction = get_dist_and_dir(letter)
    if distance==2: # semicircle
        arc_angle = 180
        if cw:
            hs = direction-90 # hs = heading at start of arc
            he = (direction+90)%360 # he = heading at end of arc
        else:
            hs = direction+90
            he = (direction-90)%360
    else: # quarter circle
        arc_angle = 90
        if cw: # if arc is to be drawn clockwise
            hs = direction-45 # hs = heading for start of arc
            he = (direction+45)%360 # he = heading at end of arc
        else:
            hs = direction+45
            he = (direction-45)%360
    turn_angle = hs - heading
    turn_angle = ((turn_angle+180)%360)-180 # get turn angle into range -180° to 180°
    turn(turn_angle) # make the turn!
    steering = (-1+2*cw)*100*sep/(2*scl+sep) # from drawbot code
    degs = arc_angle*(2*scl+sep)*wf/4.32
    print('arc_angle:'+str(arc_angle), end='', file=stderr)
    print(' steering:'+ str(int(steering))+ ' degs:'+str(int(degs)), file=stderr)
    steer_pair.on_for_degrees(steering, sp, degs)
    heading = he # update the heading value now that the robot has moved
    current_node = [node[letter][0], node[letter][1]]
    return current_node, heading # so that we can avoid using the global keyword
```

Notes

- The line **pd()** makes sure the pen is down before drawing the arc since we deduced that the pen will always be down when arcs are drawn.
- The script uses a variable '**cw**' to indicate whether the arc is to be drawn clockwise or not (i.e. counterclockwise). If **cw** is True then the arc is to be drawn clockwise.
- Once we've used the case of the letter sent to the function to determine and record whether the arc is to be drawn clockwise or counterclockwise then we can make sure the letter is lowercase to simplify the coding from that point on.
- If the arc to be drawn is a semicircle (which can be recognized because the destination node will be two units away from the start node) then we can set the arc angle to be 180°.
- Otherwise the arc must be a quarter circle so we can set the arc angle to be 90°.
- The code for both types of arc also calculates what heading the robot must have when it is about to draw the arc ('**hs**' for 'heading at start') and what heading the robot will have once it has completed drawing the arc ('**he**' for 'heading at end').
- The turn angle that the robot needs to turn through before starting to draw the arc is found from the difference between its current heading ('heading') and the heading it must have at the start of the arc ('hs').
- Having made the necessary turn on the spot we calculate the steering value in same way that we calculated it for the drawbot.
- After printing some useful information and moving through the arc we update the 'heading' value with the new heading which we know is equal to '**he**' (heading at end of arc).
- Similarly we update **current_node** to have the value of the node where the robot is now located.

Something else to notice in the above function definition is that, unlike the definition of the **move_straight()** function in the last full script, the **draw_arc()** function definition doesn't use the **global** keyword. You are supposed to have a basic knowledge of Python before starting this course so you should already know what is the function of the **global** keyword and why its use is frowned upon. For a refresher, see book.pythontips.com/en/latest/global_and_return.html. The **global** keyword was used in the the **move_straight()** function definition so that the variables **current_node** and **heading** could be modified (not just read) from within the function. To avoid having to use the **global** keyword, the variables can be passed to the function as *parameters* and then **return** can be used so that the variables in the main script are updated with the modified values. Note that when **return** is used with multiple values, as in the **draw_arc()** function definition, the values are returned as a tuple. Now that we are aware of that technique, let's also use it for the **move_straight()** function. The affected lines are highlighted in green below. The most important thing is to know that *the use of the **global** keyword should be avoided in complex scripts*, except that it may be okay to use it for 'constant variables' (if that concept makes any sense to you!).

The ready-to-run script below (**writer3.py** in the **part4** folder), includes node sequences for the curly letters **BCDGI** but not for **OPQRSU**. It also contains node sequences for the digits **0-9**. As usual, don't forget to set the wheel factor (**wf**) value to match your version of the EV3 kit. You may also want to write a different string to the one given, using the permitted characters and not exceeding five characters.

```
#!/usr/bin/env python3
# Copyright Nigel Ward, May 2019. See http://ev3python.com
# If necessary, use the separate script to make sure that the
# pen is raised before running this script
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C, MediumMotor
from sys import stderr
from math import sqrt, pi, atan2, degrees
from ev3dev2.sound import Sound
from time import sleep

medium_motor = MediumMotor()
steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
sound = Sound()

my_string = 'BLACK'.upper() # Use only characters ABCDEFGHIJKLMN and 0123456789.
wf = 1 # wheel factor. Use 1 for home version and 0.77 for edu version.
scl = 3 # Scale. Use scl values between 3 and 6.
sp = 20 # speed of steer_pair. Use values between 15 and 25.
sep = 10.5 # effective wheel separation in centimeters.
deg_per_cm = 26.84 * wf # degrees of wheel turn per cm advanced.
deg_per_robot_deg = 2.46 * wf # angle wheel turns through when robot turns one degree.
medium_motor.position = 0 # needed to protect against a bug.
# 'direction' is the direction from the current node to the next node, measured cw from 'north'.
heading = 90 # 'heading' is the direction the robot is facing, measured clockwise from 'north'.
go_forwards = True # initially the robot will move forwards (for positive distance values).
```

```

sequence={ 'A': 'GiMbo', 'B': 'abVgHYm', 'C': 'Ftjx', 'D': 'abUlYm', 'E': 'GhCamo', 'F': 'GhCam' }
sequence.update({ 'G': 'FtjxKlo', 'H': 'aGiCo', 'I': 'AcBnMo', 'J': 'AclW', 'K': 'aCgo' })
sequence.update({ 'L': 'Amo', 'M': 'ahco', 'N': 'aoc' })
sequence.update({ 'O': 'JdUlW', '1': 'DbnMo', '2': 'DUmo', '3': 'DVW' })
sequence.update({ '4': 'Ljco', '5': 'nvgac', '6': 'JXWdU', '7': 'ca', '8': 'NvSVy', '9': 'Jxftu' })

node={ 'a': (0,4), 'b': (1,4), 'c': (2,4), 'd': (0,3), 'e': (1,3), 'f': (2,3), 'g': (0,2) }
node.update({ 'h': (1,2), 'i': (2,2), 'j': (0,1), 'k': (1,1) })
node.update({ 'l': (2,1), 'm': (0,0), 'n': (1,0), 'o': (2,0), 'q': (2.6,0) })
node.update({ 's': (1,4), 't': (0,3), 'u': (2,3), 'v': (1,2), 'w': (0,1), 'x': (2,1), 'y': (1,0) })

def pu():
    if abs(medium_motor.position) > 10: # if pen is not already up
        medium_motor.on_to_position(speed=50, position=0)

def pd():
    if abs(medium_motor.position-180) > 10: # if pen is not already down
        medium_motor.on_to_position(speed=50, position=180)

def get_dist_and_dir(letter):
    dx = node[letter][0] - current_node[0]
    dy = node[letter][1] - current_node[1]
    distance = sqrt(dx**2+dy**2) # Pythagoras!
    angle = atan2(dy,dx) # 'Cartesian' angle in radians ccw from east
    direction = (90-degrees(angle))%360 # convert to degrees cw from north, in range 0-360
    print('distance to node:'+str(distance)+' direction to node:'+str(direction),file=stderr)
    return distance, direction # as a TUPLE

def move_straight(letter, current_node, heading):
    print('node letter:'+letter+' robot heading:'+str(int(heading)),file=stderr)
    if letter.islower(): # lower the pen if necessary
        pd()
    else: # raise the pen if necessary
        pu()
    letter = letter.lower() # set the letter to lower case
    distance, direction = get_dist_and_dir(letter)
    turn_angle = direction - heading
    turn_angle = ((turn_angle+180)%360)-180 # get turn angle into range -180° to 180°
    if abs(turn_angle)>90:
        turn_angle = ((turn_angle+270)%360)-90 # get into range -90° to 90°
        go_forwards = False
    else:
        go_forwards = True
    print('turn_angle:'+str(turn_angle)+' go_forwards:True', file=stderr)
    print('medium motor.position (0=up, 180=down): '+str(int(medium_motor.position))+'\n',file=stderr)
    turn(turn_angle) # make the turn!
    advance(distance, go_forwards) # make the robot advance!
    heading = heading + turn_angle # modify heading since the robot has just turned
    current_node = [node[letter][0], node[letter][1]] # update current_node
    return current_node, heading

def draw_arc(letter, current_node, heading): # so that we can avoid using the global keyword
    pd() # make sure the pen is down
    print('node letter:'+letter+' robot heading:'+str(int(heading)),file=stderr)
    cw = letter.isupper() # uppercase means draw arc clockwise (cw)
    letter = letter.lower() # set the letter to lower case
    distance, direction = get_dist_and_dir(letter)
    if distance==2: # semicircle
        arc_angle = 180
        if cw:
            hs = direction-90 # hs = heading at start of arc
            he = (direction+90)%360 # he = heading at end of arc
        else:
            hs = direction+90
            he = (direction-90)%360
    else: # quarter circle
        arc_angle = 90
        if cw: # if arc is to be drawn clockwise
            hs = direction-45 # hs = heading for start of arc
            he = (direction+45)%360 # he = heading at end of arc
        else:
            hs = direction+45
            he = (direction-45)%360
    turn_angle = hs - heading
    turn_angle = ((turn_angle+180)%360)-180 # get turn angle into range -180° to 180°
    turn(turn_angle) # make the turn!
    steering = (-1+2*cw)*100*sep/(2*scl+sep) # from drawbot code
    degs = arc_angle*(2*scl+sep)*wf/4.32
    print('arc_angle:'+str(arc_angle), end='', file=stderr)
    print(' steering:'+str(int(steering))+ ' degs:'+str(int(degs))+'\n', file=stderr)
    steer_pair.on_for_degrees(steering, sp, degs)
    heading = he # update the heading value now that the robot has moved
    current_node = [node[letter][0], node[letter][1]]

```

```

return current_node, heading # so that we can avoid using the global keyword

def turn(turn_angle):
    steer_pair.on_for_degrees(steering=100,speed=sp,degrees=turn_angle*degs_per_robot_deg)

def advance(distance, go_forwards):
    degrees = (-1+2*go_forwards)*distance*scl*degs_per_cm
    steer_pair.on_for_degrees(steering=0,speed=sp,degrees=degrees)

for char in my_string:
    sound.speak(char)
    print('Letter to write: '+char+'\n', file=stderr)
    # set current_node to equal [0,0] each time we start writing a character
    current_node=[0,0]
    for letter in (sequence[char]+'Q'): # add Q to put space between the characters
        arc_nodes = 'STUVWXY'
        if letter.upper() not in arc_nodes:
            current_node, heading = move_straight(letter, current_node, heading)
        else:
            current_node, heading = draw_arc(letter, current_node, heading)

```

Here is my result from running the above script with the *home* version:



And here is my result from running the above script with the *education* version:



Neither is perfect, but you've got the message by now. I find it makes quite a difference what type of surface you use: paper, glass etc, so try different surfaces to get the best result. Also, you will probably get better results with text strings that contain more non-curly letters and fewer curly letters.

Here is a sample printout from VS Code's Output giving useful feedback as the writerbot begins to write the letter 'B' by moving to node **a** (pen down) then node **b** (pen down) then node **V** (drawing an arc clockwise with the pen down):

```

Starting: brickrun --directory="/home/robot/part4" "/home/robot/part4/writer3.py"
Started.
-----
Letter to write: B

node letter:a  robot heading:90
distance to node:4.0  direction to node:0.0
turn_angle:-90.0  go_forwards:True
medium_motor.position (0=up, 180=down): 181

node letter:b  robot heading:0
distance to node:1.0  direction to node:90.0
turn_angle:90.0  go_forwards:True
medium_motor.position (0=up, 180=down): 179

node letter:V  robot heading:90
distance to node:2.0  direction to node:180.0
arc_angle:180  steering:63  degs:687
node letter:g  robot heading:270
distance to node:1.0  direction to node:270.0
turn_angle:0.0  go_forwards:True
medium_motor.position (0=up, 180=down): 179

```

Challenge

Update the dictionary called 'sequence' so that it includes node sequences for the missing 'curly' letters **OPQRSU** (and also the letters **TVWXYZ** that you designed previously).

Difficult Challenge

Currently we have one script to make sure the pen is in the 'up' position and a separate script to do the actual writing. The two scripts could be combined into one. The user would use the left and right buttons to turn the cam if necessary and then press the Enter button to instruct the robot to start writing. Note that no solution is provided for this exercise!

Very difficult challenge

For the straight line moves we used clever code that sometimes made the robot move backwards to reduce the amount of turning on the spot and thus improve accuracy. Make the robot also move backwards when drawing arcs when doing so allows for less turning on the spot and thus improved accuracy. Note that no solution is provided for this final exercise!

Conclusion

That brings us to the end of this course. You've learnt so much!

- You've learnt that Python is probably the best choice of textual programming language to learn (but you knew that already).
- You've learnt that Python coders in the States earn more than 100 thousand dollars per year on average – find out how much they earn in your country and think about how much you can boost your career prospects by deepening your Python skills.
- Even if you don't plan to become a professional coder, learning to program will sharpen your thinking skills – coding does not forgive sloppy thinking!
- You've learnt that Visual Studio Code is a great multiplatform code editor used by many professional coders
- How to install Visual Studio Code on your computer
- How to add extensions such as the EV3 extension
- How to modify the VS Code settings so that the 'Download and Run' configuration is always available for your EV3 Python scripts.
- How to set up VS Code such that you can switch easily between running non-EV3 programs running locally and EV3 programs running remotely
- How to flash an alternate EV3 operating system (EV3dev) to an SD card

- How to download all the official Lego sounds and images to the SD card so that they can be used by your EV3 Python scripts.
- How to write code to interact with both types of EV3 motor.
- How to write code to interact with every type of sensor that is included in the home edition or the education edition of the EV3 kit.
- How to write code to interact with the EV3's buttons, screen and loudspeaker.
- How to assemble the official Educator Vehicle (education set) or a similar vehicle (home set).
- How to assemble a drawbot/writerbot.
- This course is very cheap compared to an EV3 but has allowed you to squeeze lots of extra value out of your rather expensive kit. Not only have you learnt how to use it with a different and much more serious programming language but you have learnt about many new functions are not available in the standard Lego software, such as speech synthesis and vastly improved handling of variables and text.
- You have learnt (the hard way!) that when robots operate in the messiness of the real world their behavior tends to be somewhat approximate and unpredictable, unlike the perfectly predictable, perfectly accurate behaviors of programs that only output to a screen.
- Learning about robots is vital, of course, in a world where robots and artificial intelligence are set to have a huge impact on human society over the coming decades. Maybe you'll never be a professional robot programmer but knowing how robots are programmed may make you feel more comfortable about the prospect of sharing your world with them.
- Perhaps most important of all, you've had lots of practice studying and modifying Python scripts, and thus deepening your knowledge of Python. A good knowledge of Python is extremely valuable in the workplace – remember how I said that in the US Python coders make more than 100 thousand dollars a year, on average?



You've invested a few hours and a few dollars in this course and I hope you feel it was worth it. If so then please give a favorable evaluation and be sure to tell your friends. Also, watch out for a sequel to this course – you never know!

You now have the skills to write your own original EV3 Python programs and I wish you many hours of satisfaction with that!

Message for teachers

As a professional teacher, I have a special message to teachers who have decided to use EV3 Python with their classes or are thinking about doing so:

- VS Code is very suitable for all your Python programming classes and since it's made by the world's biggest software company it's well supported and regularly updated even though it's a free program.
- This course was made with classroom use in mind – not only is the course compatible with the education version of the EV3 (unlike many books and courses out there) but all the exercises except the drawbot/writerbot use the same 'Educator vehicle' base (sometimes with attachments) so there is a strong focus on coding as opposed to dismantling and reassembling different models.
- This course includes a number of challenges where students have to actively write or modify scripts.
- A section on programming robots could (and should) be the perfect fun-filled finale of any Python course! Don't forget that writing code for robots is very different from writing code that just outputs to the screen. Robots behave predictably only when operating in highly controlled environments such as factories and robots of the future that will share our messy human environment are going to need very careful coding if they are not going to behave unpredictably and dangerously. When writing code for robots, a conventional coding mentality isn't good enough – take another look at that video of those carefully programmed robots crashing out of the DARPA Robotics Challenge Finals youtu.be/g0TaYhipOfo Consistent success with programing robots requires a different mentality and this adds a new dimension to the world of coding! On that note I wish you 'Au revoir – until the next time!'

Solutions to some of the 'Challenge' exercises

Challenge: Make a script that sets BOTH pairs of LEDs to glow yellow for 5 seconds.

```
#!/usr/bin/env python3
from ev3dev2.led import Leds
from time import sleep

leds = Leds()

leds.all_off() # Turn all LEDs off. This also turns off the flashing.
sleep(1)
for side in ('LEFT', 'RIGHT'): # Yes, it would be just as easy to do 2 lines with set_color
    leds.set_color(side, 'YELLOW')
    sleep(5)
```

Drawbot challenges

This script is in the **part4** folder with the name **challenges.py**. Pay special attention to the lines in bold type.

```
#!/usr/bin/env python3
# Use the separate script to ensure the cam is pointing up before running this script
from ev3dev2.motor import MoveSteering, OUTPUT_B, OUTPUT_C, MediumMotor
from math import copysign, atan, degrees, sqrt

steer_pair = MoveSteering(OUTPUT_B, OUTPUT_C)
medium_motor = MediumMotor()

shape = 'A' # shape must be one of: pentagram, spiral, squiral, vertex, smiley, A
sp = 25 # speed, try other values if you like
wf = 1 # wheel factor. Use 1 for home version and 0.77 for edu version
degs_per_cm = 26.84 * wf # degrees of wheel turn per cm advanced
degs_per_robot_deg = 2.46 * wf # angle wheel must turn through
# such that the robot turns one degree
def fd(distance):
    steer_pair.on_for_degrees(steering=0, speed=sp, degrees= distance*degs_per_cm)
def bk(distance):
    steer_pair.on_for_degrees(steering=0, speed=sp, degrees=-distance*degs_per_cm)
def lt(angle):
    steer_pair.on_for_degrees(steering=-100, speed=sp, degrees=angle*degs_per_robot_deg)
def rt(angle):
    steer_pair.on_for_degrees(steering= 100, speed=sp, degrees=angle*degs_per_robot_deg)
def arc(angle, r): # r = radius. Negative radius means draw the arc counterclockwise
    steer_pair.on_for_degrees(copysign(100/(0.19*abs(r) + 1),r), sp, angle*42.16*wf*(abs(r)+5.25)/90)
def pu():
    if abs(medium_motor.position) > 10: # if pen is not already up
        medium_motor.on_to_position(speed=50, position=0)
def pd():
    if abs(medium_motor.position-180) > 10: # if pen is not already down
        medium_motor.on_to_position(speed=50, position=180)

if shape == 'pentagram':
    pd()
    for i in range(5):
        if i%2:
            bk(30)
        else:
            fd(30)
        lt(36)
    pu()

elif shape == 'spiral':
    pd()
    for i in range(1,15,2): # i will start at 1, increase in steps of 2 and finish with 13
        arc(180,i)
    pu()

elif shape == 'vertex':
    pd()
    for i in range(2,32,2): # i will start at 2 and finish at 30 but won't include 32
        fd(i)
        rt(93)
```

```

    pu()

elif shape == 'smiley':
    lt(30)
    fd(5)
    pd()
    arc(360,-1.5)
    pu()
    bk(5)
    rt(60)
    fd(5)
    pd()
    arc(360,1.5)
    pu()
    bk(5)
    rt(30)
    bk(5)
    rt(90)
    pd()
    arc(120,-5)
    pu()
    rt(90)
    fd(5)
    rt(90)
    pd()
    arc(360,10)
    pu()

elif shape == 'A':
    scl = 4 # scale. How many cm for each unit of the 4x2 grid
    rt(degrees(atan(0.25)))
    pd()
    fd(scl*sqrt(17))
    lt(2*degrees(atan(0.25)))
    bk(scl*sqrt(17))
    pu()
    rt(degrees(atan(0.25)))
    fd(scl*2)
    lt(90)
    pd()
    fd(scl*2)
    pu()

```