# SourceBots Docs

***Release 0.10.1***

**Aug 07, 2022**

# Contents:

This documentation explains how to use the SourceBots kit and Python API.

There are a number of sections in the documentation offering help for the kit and programming. Under the tutorials section, concepts from across the entire documentation are brought together to help you understand what you can, or need, to do.

# CHAPTER 1

# Tutorials

This section contains guides that will help your use your robotics kit.

## 1.1 Connecting Your Kit

### 1.1.1 Essential hardware

- Raspberry Pi (the board with many USB sockets, HDMI and microUSB)
- Power Board (the board with a fan, many green sockets and two buttons)
- Motor Board (the board with three green sockets on one end)
- Servo Board (the square board with many pins on the side)
- Arduino (a board with a metal USB-B connector)
- a Battery (will be provided later)
- USB Hub

### 1.1.2 Connectors and cables

- 2 x 3.81mm CamCon (for the Raspberry Pi)
- 4 x 5mm CamCon (for the Motors)
- 1 x 5mm CamCon with wire loop (attached to the power board)
- 4 x 7.5mm CamCon (for the Motor and Servo Boards)
- 3 x USB-A to micro-USB cables
- USB-A to USB-B cable (for the Arduino)
- Wire

**Hint:** *CamCons* are the green connectors used for power wiring within our kit.

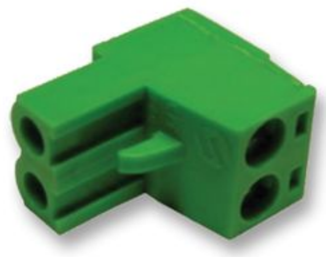


Fig. 1: A CamCon connector (more information).

### 1.1.3 Peripherals

- Servo motor
- 2 x Motors
- Ultrasound distance sensors
- USB memory stick

### 1.1.4 Tools you'll need

- Pliers
- Wire Cutters
- Wire Strippers
- Screwdriver (2mm flat-head)

You will need to fetch any other needed tools/supplies yourself.

### 1.1.5 Important notes before you start

**Warning:** Make sure to read all these **before** you start assembly.

- Do not disassemble/reassemble your kit without first switching it off by pressing the red button.
- Always be careful handling your battery, only ever plug it into the power board (the board with a fan).
- Check your kit thoroughly before switching it on again. If something is connected up incorrectly when the kit is powered up, it may break the kit!
- When making your own wires, especially those with CamCons on the end, always double-check that the correct connections are made at either end (positive to positive, ground to ground, etc.) before plugging in the cable or plugging in the battery and switching things on. Don't be afraid to ask someone to check your connections.
- Colour coding is key; please use *red* for wires connected to power (say 12V or 5V), *black* for wires connected to ground (0V) and *any other colour* for motors.

### 1.1.6 How it all fits together

The first step of your robot is assembly! Here we'll guide you step-by-step on how to connect things up. You'll be cutting your own wires here!

1. Connect the Raspberry Pi to the Power Board using two 3.81mm (small) CamCons. Please make sure that you check the polarity of the connector on the tab.
2. Connect the USB hub to the Pi by plugging it into any one of its four USB sockets.
3. Connect the Power Board to the Pi via one of the black micro-USB cables; the standard USB end goes into any USB socket on the Pi or connected USB hub, the micro-USB end into the Power Board.
4. Connect the Motor Board to the Power Board by screwing the two 7.5mm (large) CamCons provided onto the opposite ends of a pair of wires, ensuring that positive connects to positive and ground to ground, and then plugging one end into the appropriate socket of the Motor Board and the other into a high power socket (marked `H0` or `H1`) on the side of the Power Board.
5. Connect the Motor Board to the Pi by way of another micro-USB cable; the big end goes into any USB socket on the Pi or connected USB hub, the micro-USB end goes into the Motor Board.
6. Connect the Arduino to the Pi by way of the USB-A (rectangle) to USB-B (square-like) cable.

**Warning:** Please don't connect the Arduino to the Raspberry Pi via the USB Hub. If the Arduino is not connected *directly* to the Pi, you may have issues with getting enough power to it.

7. Connect the Servo Board to the Power Board by screwing the two 7.5mm (large) CamCons onto the opposite ends of a pair of wires, ensuring that positive connects to positive and ground to ground, and then plugging one end into a low power socket on the side of the Power Board and the other into the 12V socket on the servo board.
8. Connect the Servo Board to the Pi by way of another micro-USB cable; the USB A (rectangle) end goes into any USB socket on the Pi or connected via the USB hub, the micro-USB end goes into the Servo Board.
9. To connect the motors, first screw two 5mm (medium) CamCons provided onto the opposite ends of a pair of wires. You can then use this cable to connect a motor to the `M0` or `M1` port on the motor board.
10. To connect a servo, push the three pin connector vertically into the pins on the side of the servo board. The black or brown wire (negative) should be at the bottom.
11. At this point, check that everything is connected up correctly (it may be helpful to ask a facilitator to check that all cables are connected properly).
12. Connect the Power Board to one of the blue LiPo batteries by plugging the yellow connector on the cable connected to the Power Board into its counterpart on the battery.
13. If there is not one plugged in already, a loop of wire should be connected to the socket beneath the On|Off switch. Check that the Power Board works by pressing the On|Off switch and checking that the bright LED on the Raspberry Pi comes on green.

## 1.2 Setting up the PyCharm IDE

This tutorial will guide you through setting up the PyCharm editor with support for our robot software. This means that the editor can do a better job of checking your code and suggesting as you type, saving you wasting time transferring your code to your robot only to find that a variable name has been mispelt, or something equally menial!

First, open up PyCharm from the Start Menu:

It may ask you to read and accept their terms and conditions before you can proceed:

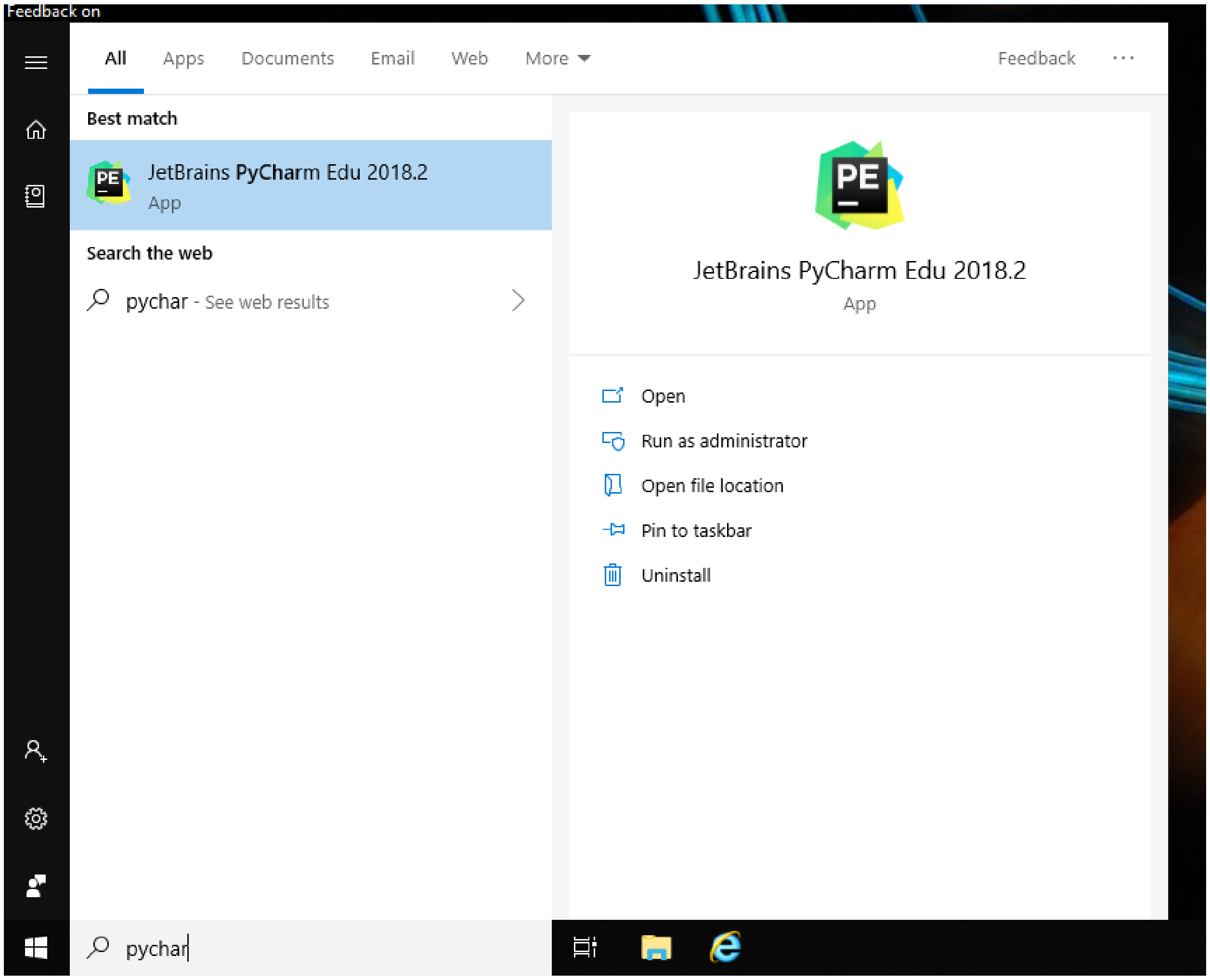
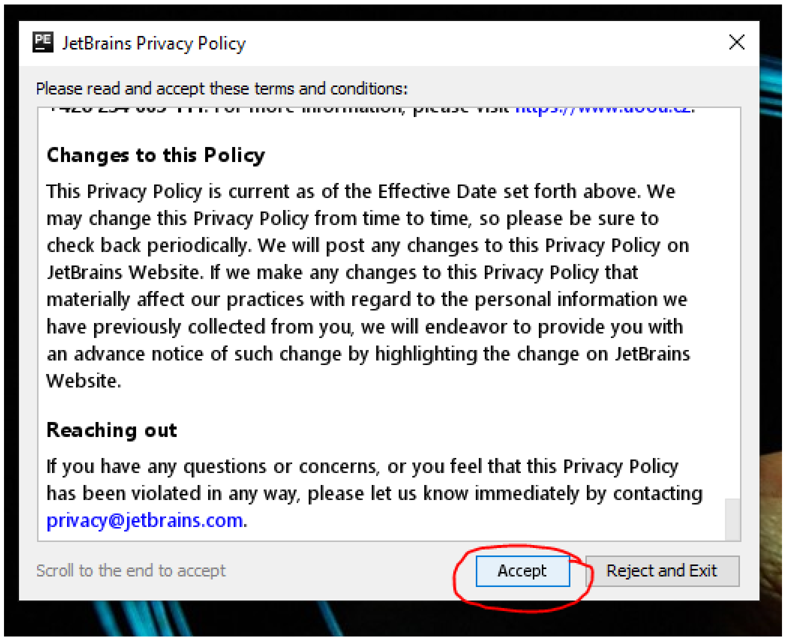


Fig. 2: Searching for and selecting PyCharm in the Start Menu.


JetBrains Privacy Policy

Please read and accept these terms and conditions:

**Changes to this Policy**

This Privacy Policy is current as of the Effective Date set forth above. We may change this Privacy Policy from time to time, so please be sure to check back periodically. We will post any changes to this Privacy Policy on JetBrains Website. If we make any changes to this Privacy Policy that materially affect our practices with regard to the personal information we have previously collected from you, we will endeavor to provide you with an advance notice of such change by highlighting the change on JetBrains Website.

**Reaching out**

If you have any questions or concerns, or you feel that this Privacy Policy has been violated in any way, please let us know immediately by contacting privacy@jetbrains.com.

Fig. 3: The terms and conditions dialogue box.

Firstly, we will need to create a new project.

Next, you will need to configure the environment for your project. This ensures that we are using a compatible version of Python (3.6 or later), and that Pycharm is aware of the other code running on, and controlling your robot.

At the top of the dialogue box, choose an appropriate location to save your code.

**Hint:** If you are using a university computer, files that are placed in your `H:/` drive will be synchronised and backed up across any other university computer.

Select *New Environment using Virtualenv*, and Python 3.6 for the base interpreter.

**Warning:** The location of Python 3.6 may vary by computer. Ask a volunteer for help if you are stuck.

Once PyCharm has finished loading, we need to create a new `main.py`.

Right-click on your project name, select New -> Python File.

You will need to name your file `main.py` or your robot will not recognise it.

In order to get suggestions as you type your code, you'll need to install the same Python package that is used on your robot, which is called `sbot`. You can do this by opening PyCharm's settings, navigating to the "Project Interpreter" tab, and pressing the Install button:

Enter "sbot" into the search bar and ensure it is selected in the pane on the left, then press "Install Package". This will take some time, so wait for the green success message.

You are now ready to program your robot. Pycharm will give you auto-suggestions and let you know if your code is mis-spelt or has other common errors.

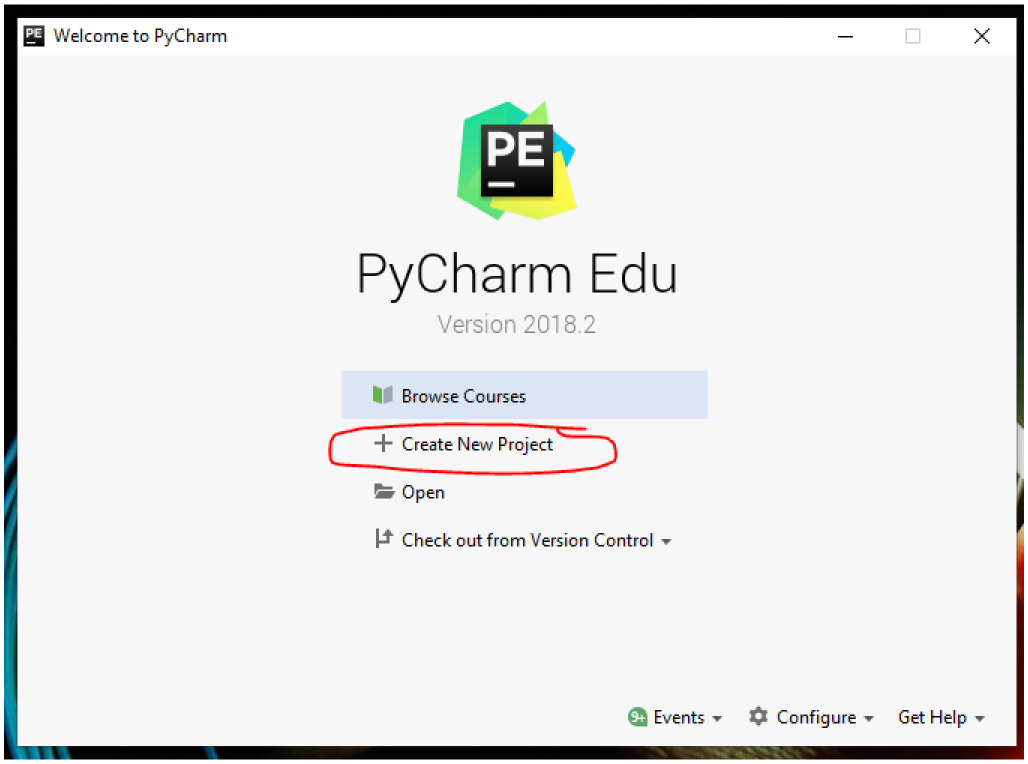
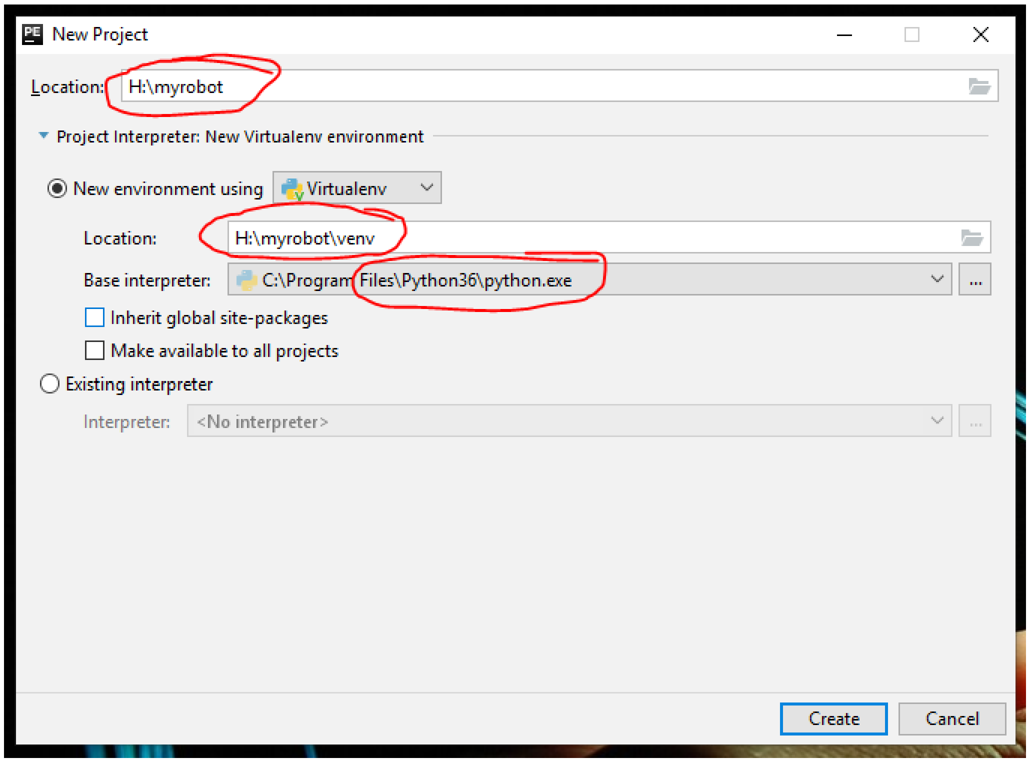


Fig. 4: The welcome dialogue, with the Create New Project button.


New Project
Project Interpreter: New Virtualenv environment
Inherit global site-packages
Make available to all projects
Existing interpreter
Create
Cancel

Location: H:\myrobot

New environment using: Virtualenv

Location: H:\myrobot\venv

Base interpreter: C:\Program Files\Python36\python.exe

Interpreter: <No interpreter>

Fig. 5: Project configuration dialogue

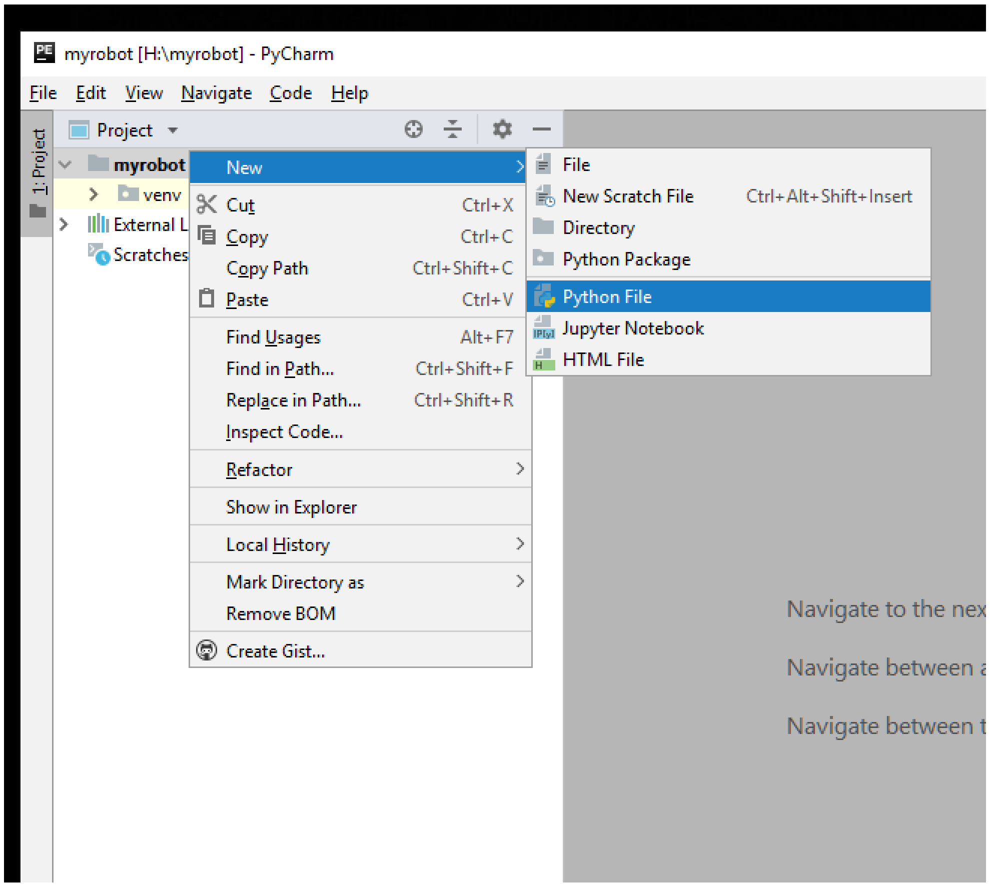


Fig. 6: Create a new Python file

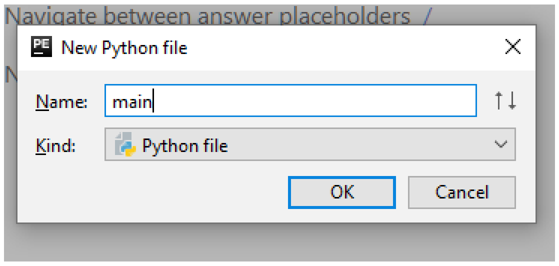


Fig. 7: Naming your file

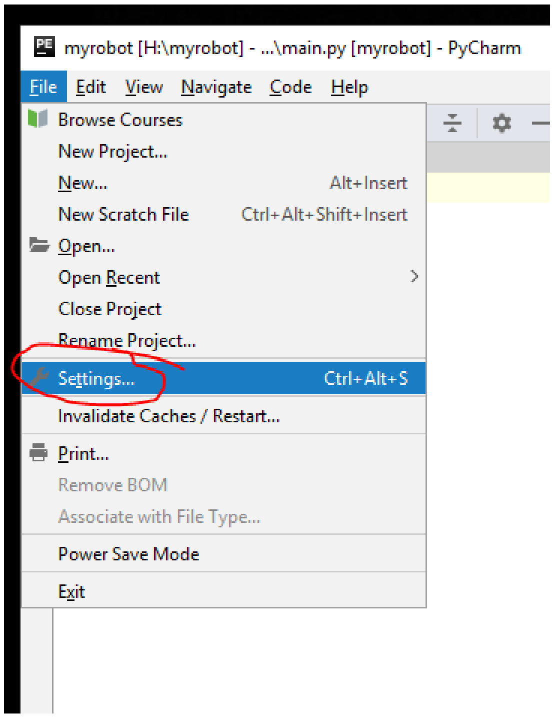


Fig. 8: Opening PyCharm's settings from the File menu.

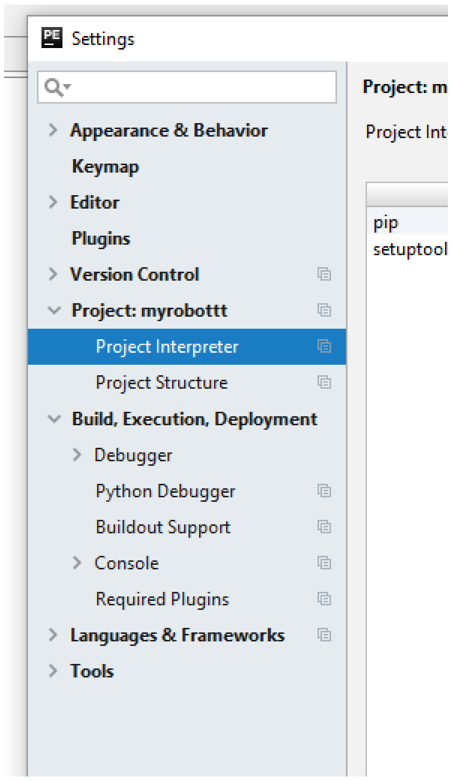


Fig. 9: The Project Interpreter tab in PyCharm's settings.

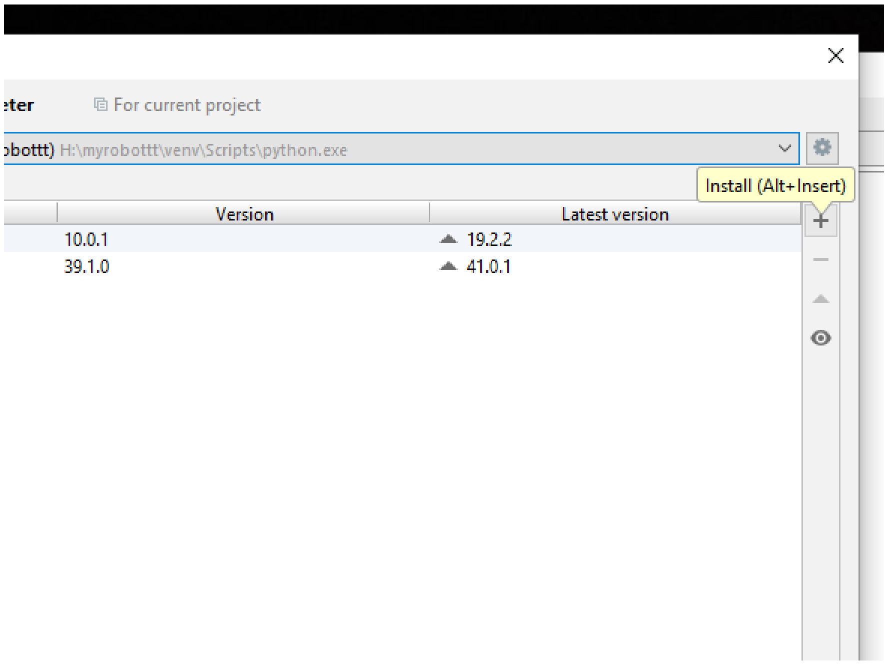


Fig. 10: This button opens the package installation window.

## 1.3 Using Your Kit

**Hint:** In this tutorial, we will be using our API (Application Programming Interface), which is the functions you need to use to make the robot do stuff, so make sure to have a look at it first!

### 1.3.1 Setup

Before you start this tutorial, make sure you have connected your kit together.

There are two lines of code you must have at the very top of your code whenever you want to do something with the robot, those lines are:

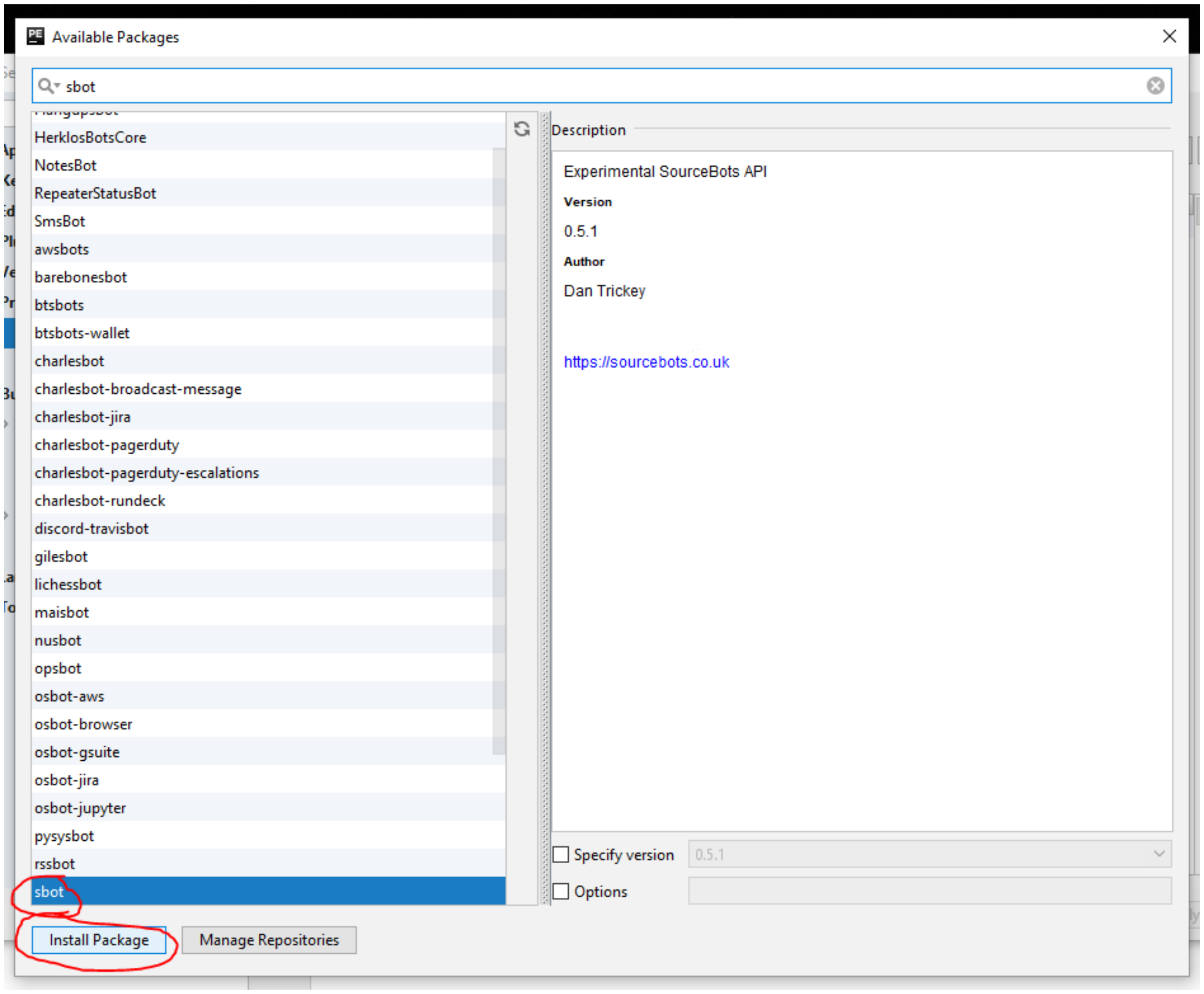
```
from sbot import *

r = Robot()
```

These two lines simply copy in all of our helper functions, and sets up your robot.

Any code below the line with `r = Robot()` won't be run until you hit the black 'Start' button on the power board.

You can run your code by inserting the USB stick into a port onto your robot. The robot will flash a light next to the start button. Press the button to start your code.

**Warning:** Please ensure that you "eject" your drive from your Windows machine, as not doing this can *corrupt* your USB!



Experimental SourceBots API

**Version**

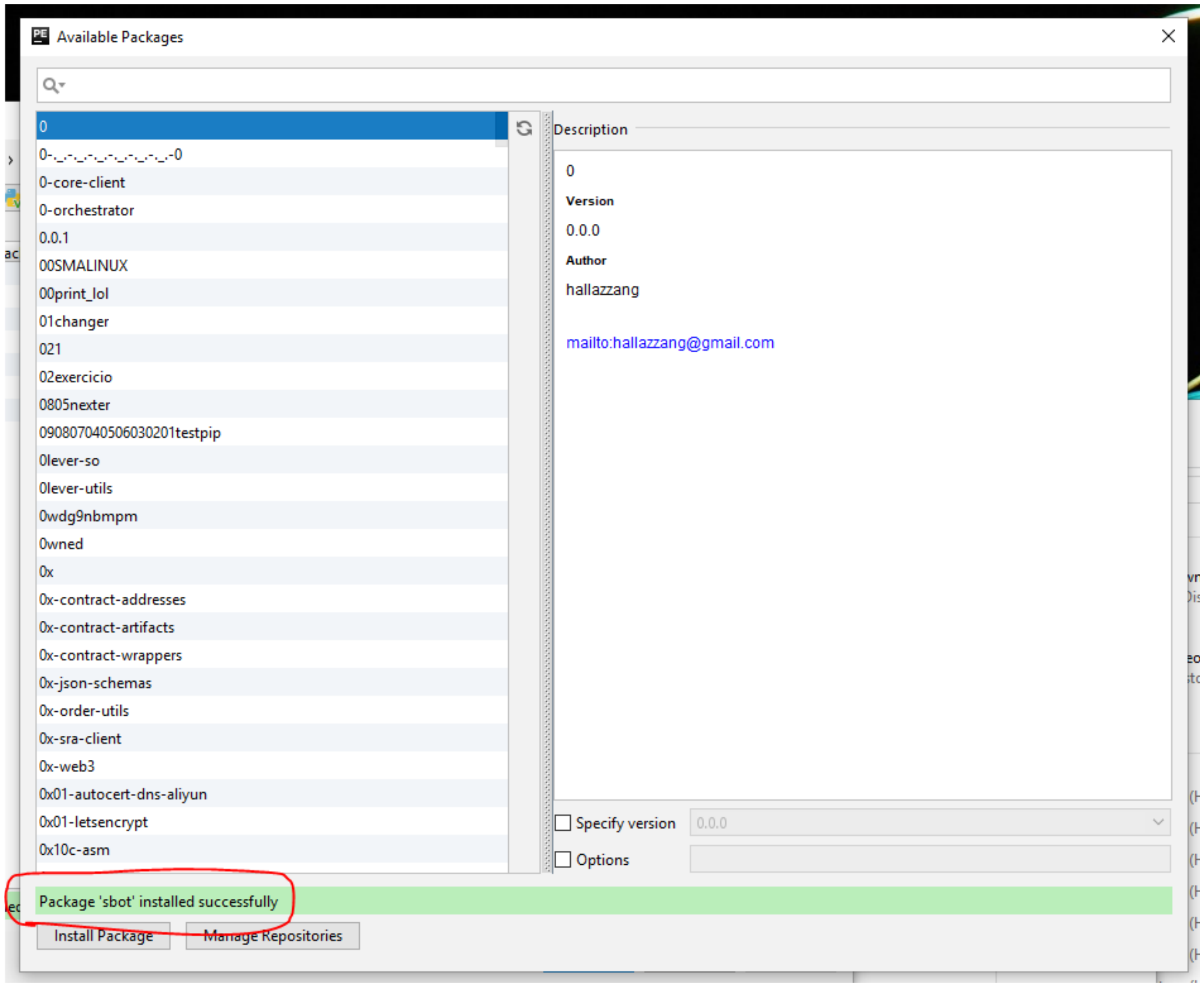
0.5.1

**Author**

Dan Trickey

https://sourcebots.co.uk

Fig. 11: Installing sbot



0
0-._.-._.-._.-._.-._.-._.-0
0-core-client
0-orchestrator
0.0.1
00SMALINUX
00print_lol
01changer
021
02exercicio
0805nexter
090807040506030201testpip
0lever-so
0lever-utils
0wdg9nbmpm
0wned
0x
0x-contract-addresses
0x-contract-artifacts
0x-contract-wrappers
0x-json-schemas
0x-order-utils
0x-sra-client
0x-web3
0x01-autocert-dns-aliyun
0x01-letsencrypt
0x10c-asm

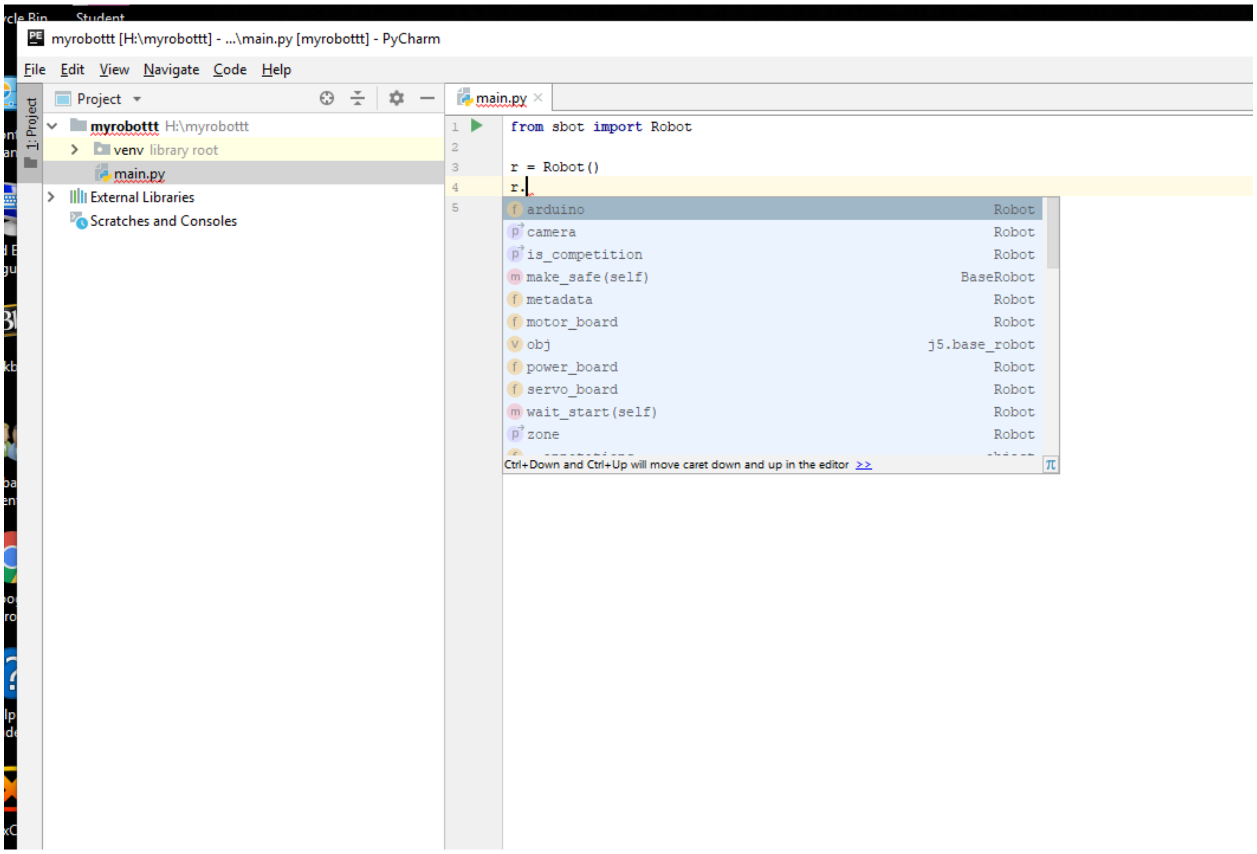
0
Version
0.0.0
Author
hallazzang

mailto:hallazzang@gmail.com

0.0.0

Package 'sbot' installed successfully

Fig. 12: Installation success



```
from sbot import Robot

r = Robot()
r.
```

```
arduino                     Robot
camera                      Robot
is_competition              Robot
make_safe(self)         BaseRobot
metadata                    Robot
motor_board                 Robot
obj                 j5.base_robot
power_board                 Robot
servo_board                 Robot
wait_start(self)            Robot
zone                        Robot
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>
```

Fig. 13: Code auto-suggestions

### 1.3.2 Forwards and Backwards

Start by checking that you can drive your motors forwards and backwards. Doing this is actually very easy; the only thing you need to realise is that a positive number drives the motor in one direction and a negative number drives it in the other direction.

**Warning:** Make sure your robots can turn without danger. If your motors aren't attached to a chassis, make sure they don't have wheels and are sitting in a position which makes it safe for them to turn.

Please remember that the actual movement of the robot depends which way around the motor is mounted on the robot! It's very likely you might need to set one motor to go in reverse in order for your robot to go forwards.

Here's the code:

```
from sbot import *
from time import sleep
r = Robot()
while True:
    # Set motor 0 to 20% power.
    r.motor_board.motors[0].power = 0.2
    # Set motor 1 to 20% power
    r.motor_board.motors[1].power = 0.2

    sleep(1)

    r.motor_board.motors[0].power = 0
    r.motor_board.motors[1].power = 0

    sleep(1)
```

You're hopefully familiar with the first few lines; in fact, the only lines you may not be familiar with are the `r.motor_board.motors[0]...` lines. For a comprehensive reference to the 'motor' object, see the motor page.

But, to summarise:

```
r.motor_board.motors[0].power = x
```

will set the power of the motor connected to output 0 (the `motors[0]` part) on the motor board to `x`, where `x` is a value between `-1` and `1`, inclusive.

Now see if you can turn on the spot, then try driving in various shapes.

Write some code that would make your robot drive in: - a square - a wavy line

### 1.3.3 Changing the speed

When you move your robot, it's likely you'll want your robot to go from stand- still to moving at a high speed. The most obvious way of doing this is to just immediately set the power of the motors from `0` to `1`.

There are 2 problems with doing this:

1. Setting the power from `0` to `1` very quickly draws a very large current from the motor, so much so that it might trip the over-current protection on our power board.
2. Quickly changing the speed can cause the wheels to slip, meaning your robot won't necessarily go the distance or direction you expect it to.

There's a simple solution to this, whenever you speed up or slow down, you should write some code which smoothly changes the motor speed from 0 to the target speed.

Firstly, how do you smoothly change the robot speed?

It's pretty simple once you understand it:

```python
from sbot import *
import time

r = Robot()

for power in range(0, 101):
      r.motor_board.motors[0].power = power / 100
      time.sleep(0.01)
```

This code should smoothly speed up your motor from 0 to 1 in 1 second.

The python `range` function takes in 2 parameters, `from` and `to`. It then simply returns a list of numbers between those two values. It *doesn't* give you the last number. (i.e. `range(0,3)` will give you a list containing 0, 1, and 2) So if you want the last number you'll need to go one further.

The `time.sleep` is there otherwise the code will immediately go to full power.

Now try and write some code that: - Smoothly starts and stops your robot.

### 1.3.4 Servos

Servos are a motor which knows what position it's at. You can tell it an angle and it'll handle turning to that value!

**Warning:** Be warned, most servos can't turn a full 360 degrees! Always check how far it can move before you design a cool robot arm!

Servos can be set to turn to a specific position. Sadly you can't just tell it an angle to turn to in degrees, you can only tell it to go between `-1` and `1`. You'll need to measure the angle yourself and work this out if you need it!

If you plug a servo in channel '0' of the servo board, this code will turn it back and forth from minimum to maximum forever:

```python
from sbot import *
from time import sleep

r = Robot()

r.servo_board.servos[0].position = 1

while True:
    r.servo_board.servos[0].position = -r.servo_board.servos[0].position
    sleep(1)
```

This works because you can get the last position you told the servo to go to with `blah = r.servo_board.servos[0].position`

Now connect 2 servos to your robot. See if you can spell out "Hello" in Semaphore. You will have to think about which way to orient your servos so they can reach all of the positions they need to. You can add paper flags to your servos if you want to.

### 1.3.5 Ultrasound

An Ultrasound Sensor can be used to measure distances.

The sensor sends a pulse of sound at the object and then measures the time taken for the reflection to be heard.

The ultrasound sensors aren't lasers, they have a cone-shaped range, and give you the distance of the nearest large thing. Also ultrasound sensors have both a minimum and a maximum range! Make sure you know what the minimum range is for your sensor by experimenting with it.

```python
from sbot import *
from time import sleep

r = Robot()

while True:
    distance = r.arduino.ultrasound_sensors[4, 5].distance()
    print("Object is {}m away.".format(distance))
    sleep(1)
```

This code will print the distance in metres to the log file every second.

Try write some code that spins your motors forward, but stop when a object closer than 20cm is detected by the ultrasound sensor.

### 1.3.6 Buzzer

The power board on your kit has a piezoelectric buzzer onboard. We can use this to play tunes and make sounds, which can be useful when trying to figure out what your code is doing live.

```python
from sbot import *
from time import sleep

r = Robot()

# Play a tone of 1000Hz for 1 second.
r.power_board.piezo.buzz(1, 1000)

# Play A7 for 1 second.
r.power_board.piezo.buzz(1, Note.A7)
```

**Hint:** Notes from `C6` to `C8` are available. You can play other tones by looking up the frequency here.

**Warning:** Calling `buzz` is non-blocking, which means it doesn't actually wait for the piezo to stop buzzing before continuing with your code. If you want to wait for the buzzing to stop, add a `sleep` afterwards. If you send more than 32 beeps to the robot too quickly, your power board will crash!

### 1.3.7 Building a Theremin

A Theremin is a unusual musical instrument that is controlled by the distance your hand is from its antennae.

Can you use your ultrasound sensor and buzzer to build a basic Theremin?

Fig. 14: A Moog Etherwave, assembled from a theremin kit: the loop antenna on the left controls the volume while the upright antenna controls the pitch.

Here's some code to help you get started:

```
from sbot import *
from time import sleep

r = Robot()

while True:
    distance = ...

    pitch_length = ...

    # Remember, humans can hear between about 2000Hz and 20,000Hz
    pitch_to_play = ...

    r.power_board.piezo.buzz(pitch_length, pitch_to_play)
    sleep(pitch_length)
```

### 1.3.8 Inputs and Outputs

The Arduino has some pins on it that can allow your robot to sense it's environment.

We will investigate how these work in more detail in the electronics labs, but we can run some code anyway.

```
from sbot import *
from time import sleep

r = Robot()

# Turn on the pins
for pin in r.arduino.pins:
    pin.mode = GPIOPinMode.DIGITAL_OUTPUT
    pin.digital_state = True

# Flash all of the pins.
while True:
    pin.digital_state = not pin.digital_state
    sleep(0.5)
```

## 1.4 Electronics Labs

The Electronics Labs can be downloaded here: `Electronics Labs`.

## 1.5 Python: A Whirlwind Tour

In this tutorial, we'll introduce the basic concepts of programming, which will be central to the programs that you will run on your robot. There are many different languages in which computers can be programmed, all with their advantages and disadvantages, but we use Python, specifically 3.6. We chose Python because it's easy to learn, but also elegant and powerful.

At the end of the tutorial are exercises. The first ones for each section should be quite easy, while the higher-numbered exercises will be harder. Some will be very hard; try these if you're up for a challenge.

Before we begin, a word on learning. The way that you learn to code is by doing it; make sure you try out the examples, fiddle with them, break them, try some of the exercises.

### 1.5.1 Using an interpreter

To run Python programs you need a something called an interpreter. This is a computer program which interprets human-readable Python code into something that the computer can execute. There are a number of online interpreters that should work even on a locked-down computer, such as you will probably find in your college.

If your computer has a compatible browser, go to http://repl.it and select *Python3* from the dropdown. Enter your program in the box on the left, and click the arrow to run it.

If your browser isn't compatible, another good online interpreter can be found at http://codeskulptor.org. It's very similar; simply enter your program into the left pane and click the play button to run it. The output will appear in the right pane.

Whichever you choose, test it with this classic one line program:

```
print("Hello World!")
```

The text `Hello World!` should appear in the output box.

There's nothing particularly wrong with online interpreters for our needs, but if you want to use Python for something more advanced you'll want an interpreter which runs directly on your computer. Downloads are available for all common OS's from the website.

### 1.5.2 Statements

A statement is a line of code that does something. A program is a list of statements. For example:

```
x = 5
y = (x * 2) + 4
print("Number of bees:", y - 2)
```

The statements are executed one by one, in order. This example would give the output `Number of bees:  12`.

As you may have guessed, the `print` statement displays text on the screen, while the other two lines are simple algebra.

#### Strings

When you want the interpreter to treat something as a text value (for example, after the `print` statement above), you have to surround it in quotes. You can use either single (`'`) or double (`"`) quotes, but try to be consistent. Pieces of text that are treated like this are called 'strings'.

### Comments

Placing a hash (#) in your program ignores anything after the hash.

For example:

```
# This is a comment
print("This isn't.")  # But this is!
```

You should use comments whenever you think that it is not completely clear what a statement or block of statements does, especially as you are working in teams! Also bear in mind the varying coding skills of your team. You might be the best coder in your team, but what if you were taken ill the day before the competition, and your team-mates had to fix your code?

Comments are also useful for temporarily removing statements from your code, for testing:

```
x = 42
#x = x - 4
print("The answer is", x)
```

This example would output `The answer is 42`, as the subtraction is not executed.

## 1.5.3 Variables

Variables store values for later use, as in the first example. They can store many different things, but the most relevant here are numbers, strings (blocks of text), booleans (`True` or `False`) and lists (which we'll come to later).

To set a variable, simply give its name, followed by = and a value. For example:

```
x = 8
my_string = "Tall ship"
```

You can ask the user to put some text into a variable with the `input` function (we'll cover functions in more detail later):

```
name = input("What is your name?")
```

### Identifiers

Certain things in your program, for example variables and functions, will need names. These names are called 'identifiers' and must follow these rules:

- Identifiers can contain letters, digits, and underscores. They may not contain spaces or other symbols.
- An identifier cannot begin with a digit.
- Identifiers are case sensitive. This means that `bees`, `Bees` and `BEES` are three different identifiers.

## 1.5.4 Code blocks and indentation

Python is reasonably unique in that it cares about indentation, and uses it to decide which statements are referred to by things like `if` statements.

In most other programming languages, if you don't indent your code it will run just fine, but any poor soul who has to read your code will hunt you down and hit you around the head with a large, wet fish. In Python, you'll just get an error, which we're sure you'll agree is preferable.

A group of consecutive statements that are all indented by the same distance is called a block. `if` statements, as well as functions and loops, all refer to the block that follows them, which must be indented further than that statement. An example is in order. Let's expand the first `if` example:

```
name = input("What is your name?")
email = "Bank of Nigeria: Tax Refund"
if name == "Tim":
    print("Hello Tim.")
    if email != "":
        print("You've got an email.")

        # (blocks can contain blank lines in the middle)
        if email != "Bank of Nigeria: Tax Refund":
            print("Looks legitimate, too!")
    else:
        print("No mail.")

else:
    print("You're not Tim!")

print("Python rocks.")
```

Output (for "Tim" as before):

```
Hello Tim.
You've got an email!
Python rocks.
```

To find the limits of an `if` statement, just scan straight down until you encounter another statement on the same indent level. Play around with this example until you understand what's happening.

One final thing: Python doesn't mind *how* you indent lines, just so long as you're consistent. Some text editors insert indent characters when you press tab; others insert spaces (normally four). They'll often look the same, but cause errors if they're mixed. If you're using an online interpreter, you probably don't need to worry. Otherwise, check your editor's settings to make sure they're consistent. Four spaces per indent level is the convention in Python. We'll now move on from this topic before that last sentence causes a flame war.

### 1.5.5 Lists

Lists store more than one value in a single variable and allow you to set and retrieve values by their position ('index') in the list. For example:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
print(shopping_list[0])
shopping_list[3] = "Magazine"
print(shopping_list[2])
print(shopping_list[3])
```

Output:

```
Bread
PNP Transistors
Magazine
```

**Warning:** Like most other programming languages, indices start at 0, not 1. Due to this, the last element of this four-element list is at index 3. Attempting to retrieve `shopping_list[4]` would cause an error.

You can find out the length of a list with the `len` function, like so:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
print("There are", len(shopping_list), "items on your list.")
```

Finally, you can add a value to the end of a list with the `append` method:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
shopping_list.append("Mince pies in October")
print(shopping_list)
```

The values in a list can be of any type, even other lists. Also, a list can contain values of different types.

There are various other useful data structures that are beyond the scope of this tutorial, such as dictionaries (which allow indices other than numbers). You can find out more about these in python's documentation.

### 1.5.6 `while` loops

The `while` loop is the most basic type of loop. It repeats the statements in the loop while a condition is true. For example:

```
x = 10
while x > 0:
    print(x)
    if x == 5:
        print("Half way there!")

    x = x - 1

print("Zero!")
```

Output:

```
10
9
8
7
6
5
Half way there!
4
3
2
1
Zero!
```

The condition is the same as it would be in an `if` statement and the block of code to put in the loop is denoted in the same way, too.

### 1.5.7 `for` loops

The most common application of loops is in conjunction with lists. The `for` loop is designed specifically for that purpose. For example:

```
shopping_list = ["Bread", "Milk", "PNP Transistors", "Newspaper"]
for x in shopping_list:
    print("[ ]", x)
```

The code is executed once for each item in the list, with `x` set to each item in turn. So, the output of this example is:

```
[ ] Bread
[ ] Milk
[ ] PNP Transistors
[ ] Newspaper
```

Unfortunately, this method doesn't tell you the index of the current item. `x` is only a temporary variable, so modifying it has no effect on the list itself (try it). This is where the `enumerate` function comes in (see *Calling functions*). It tells us the index of each value we loop over. An example with numbers:

```
prices = [4, 5, 2, 1.50]
# Add VAT
for index, value in enumerate(prices):
    prices[index] = value * 1.20

print(prices)
```

Output:

```
[4.8, 6.0, 2.4, 1.7999999999999998]
```

### 1.5.8 Calling functions

Functions are pre-written bits of code that can be run ('called') at any point. The simplest functions take no parameters and return nothing. For example, the `exit` function ends your program prematurely:

```
x = 10
while x > 0:
    print(x)
    x = x - 1
    if x == 5:
        exit()   # not supported in repl.it!
```

This will output the numbers 10 to 6, and then stop. Not very useful. However, most functions take input values ('parameters') and output something useful (a 'return value'). For example, the `len` function returns the length of the given list:

```
my_list = [42, "BOOMERANG!!!", [0, 3]]
print(len(my_list))
```

Output:

```
3
```

Combined with the `range` function, which returns a list of numbers in a certain range, you get a list of indices for the list (you might want to look back at that second `for` example).

```
my_list = [42, "BOOMERANG!!!", [0, 3]]
print(range(len(my_list)))
```

Output:

```
[0, 1, 2]
```

The `range` function can also take multiple parameters:

```
print(range(5))            # numbers from 0 to 4.
print(range(2, 5))          # numbers from 2 to 4.
print(range(1, 10, 2))      # odd numbers from 1 to 10
```

Output:

```
[0, 1, 2, 3, 4]
[2, 3, 4]
[1, 3, 5, 7, 9]
```

There are many built-in functions supplied with Python (see *appendix*). Most are in 'modules', collections of functions which have to be imported. For example, the `math` module contains mathematical functions. To use the `sin` function, we must import it:

```
import math

print(math.sin(math.pi / 2))
```

## Defining functions

Of course, you'll want to make your own functions. To do this, you precede a block of code with a `def` statement, specifying an identifier for the function, and any parameters you might want. For example:

```
def annoy(num_times):
    for i in range(num_times):
        print("Na na na-na na!")

annoy(3)
```

The output would be three annoying lines of `Na na na-na na!`.

To return a value, use the `return` statement. A rather trivial example:

```
def multiply(x, y):
    return x * y

print(multiply(2, 3))
```

## Using functions effectively

Without functions, most programs would be very hard to read and maintain. Here's an example (admittedly a little contrived):

```
my_string = "All bees like cheese when they're wearing hats."
x = 0
for c in my_string:
    if c == "a":
        x = x + 1

y = 0
for c in my_string:
    if c == "e":
        y = y + 1
```

Before we explain the example, try and figure out what it does. What do `x` and `y` represent?

Now, let's refine it with functions:

```
def count_letter(string, l):
    x = 0
    for c in string:
        if c == l:
            x = x + 1

    return x

my_string = "Bees like cheese when they're wearing hats."

x = count_letter(my_string, "a")
y = count_letter(my_string, "e")
```

This version has a number of advantages:

- It's far more obvious what the program does.
- The program is shorter, and cleaner.
- The code for counting letters in a string is in only one place, and can be reused.

The last point has another advantage. There's a bug in this program: upper-case letters aren't counted. It's easy to fix, but in the function version we only have to apply the fix in one place. True, it would only be two places in the original, but in a major program, it could be thousands.

You should try and use functions wherever you see multiple lines of code that are repeated, or find yourself writing code to do the same thing (or a similar thing) more than once. In these situations, look at the relevant bits of code and try to think of a way to put it into a function.

### 1.5.9 Scope

When you set a variable inside a function, it will only keep its value inside that function. For example:

```
x = 2

def foo():
    x = 3
    print("In foo(), x =", x)

foo()
print("Outside foo(), x =", x)
```

Output:

```
In foo(), x = 3
Outside foo(), x = 2
```

This can get quite confusing, so it's best to avoid giving variables inside functions ('local' variables) the same identifier as those outside. If you want to get information out of a function, `return` it.

This concept is called 'scope'. We say that variables which are changed inside a function are in a different scope from those outside.

You can have functions within functions, and this can actually be quite useful. In this situation, each nested function will also have its own scope.

### 1.5.10 Exercises: variables and mathematics

#### Average calculator

The first two lines of this program put two numbers entered by the user into variables `a` and `b`. The `int` function converts a string (a piece of text like the ones returned by `input` e.g. `"42"`) into a number (e.g. `42`). Replace the comment with code that averages the numbers and puts them in a variable called `average`.

```python
string1 = input("Enter first number: ")
string2 = input("Enter second number: ")

a = int(string1)
b = int(string2)

# Store the average of a and b in the variable `average`

print("The average of", a, "and", b, "is", average)
```

Run your code and check that it works.

#### Distance calculator

Write a program which uses `input` to take an X and a Y coordinate, and calculate the distance from (0, 0) to (X, Y) using Pythagoras' Theorem. Put the code into an interpreter and run it. Does it do what you expected?

**Tip:** You can find the square root of a number by raising it to the power of 0.5, for example, `my_number ** 0.5`.

**Extension:** can you adapt the program to calculate the distance between any two points?

#### Booleans and `if` statements

A boolean value is either `True` or `False`. For example:

```python
print(42 > 5)
print(4 == 2)
```

Output:

```
True
False
```

< and == are operators, just like + or `*`, which return booleans. Others include <= (less than or equal to), >, >= and != (not equal to). You can also use `and`, `or`, and `not` (see the *Operators* appendix).

`if` statements execute code only if their condition is true. The code to include in the `if` is denoted by a number of indented lines. To indent a line, press the tab key or insert four spaces at the start. You can also include an `else` statement, which is executed if the condition is false. For example:

```
name = input("What is your name?")
if name == "Tim":
    print("Hello Tim.")
    print("You've got an email.")
else:
    print("You're not Tim!")

print("Python rocks!")
```

If you typed "Tim" at the prompt, this example would output:

```
Hello Tim.
You've got an email.
Python rocks!
```

Having another `if` in the `else` block is very common:

```
price = 50000 * 1.3
if price < 60000:
    print("We can afford the tall ship!")
else:
    if price < 70000:
        print("We might be able to afford the tall ship...")
    else:
        print("We can't afford the tall ship. :-(")
```

So common that there's a special keyword, `elif`, for the purpose. So, the following piece of code is equivalent to the last:

```
price = 50000 * 1.3
if price < 60000:
    print("We can afford the tall ship!")
elif price < 70000:
    print("We might be able to afford the tall ship...")
else:
    print("We can't afford the tall ship. :-(")
```

Both output:

```
We might be able to afford the tall ship...
```

### 1.5.11 Exercises: `if` statements and blocks

#### So many `if`s

Without running it, work out what output the following code will give:

```
some_text = "Duct Tape"
if 5 > 4:
```

(continues on next page)

(continued from previous page)

```
    print("Maths works.")
    if some_text == "duct tape":
        print("The case is wrong.")
    elif some_text == "Duct Tape":
        print("That's right.")
    else:
        print("Completely wrong.")
else:
    print("Oh-oh.")
```

Run the code and check your prediction.

#### Age detection tool

Write a program that asks the user for their age, and prints a different message depending on whether they are under 18, over 65, or in between.

### 1.5.12 Exercises: lists and loops

#### A better average calculator

Write a program which calculates the average of a list of numbers. You can specify the list in the code.

**Extension:** You can tell when a user has not entered anything at a `input` prompt when it returns the empty string, `""`. Otherwise, it returns a string (like "42.5"), which you can turn into a number with the `float` function. Additionally, extend your program to let the user enter the list of values. Stop asking for new list entries when they do not enter anything at the `input` prompt. Example of how to recognize when a user doesn't enter anything:

```
var = input("Enter a number: ")
if var == "":
    print("You didn't enter anything!")
else:
    print("You entered", float(var))
```

#### Fizz buzz

Write a program which prints a list of numbers from 0 to 100, but replace numbers divisible by 3 with "Fizz", numbers divisible by 5 with "Buzz", and numbers divisible by both with "Fizz Buzz".

**Extension:** create a list of numbers, and replace a number with "Fuzz" if it is a multiple of any number in the list.

#### Trees and triangles

You can combine (or 'concatenate') strings in Python with the + operator:

```
str = "Hello "
str = str + "World!"
print(str)
```

Write a program that asks the user for a number, and then prints a triangle of that height, with its right angle at the bottom left. For example, given the number 3, the program should output:

```
*
**
***
```

Try the same, but with the right angle in the top-right, like so (again, for input 3):

```
***
 **
  *
```

**Extension:** print out a tree shape of the given size. For example, a tree of size 4 would look like this:

```
   *
  ***
 *****
*******
   *
   *
```

### 1.5.13 Exercises: functions

#### Trigonometry

Write a program that takes as input an angle (in radians) and the length of one side (of your choice) of a right-angled triangle. Print out the length of all sides of the triangle.

You'll need the functions contained in the `math` module (docs here)

**Note:** Python uses radians for its angles. If you are not comfortable with radians, you can use the `radians` function in the `math` module to convert to radians from degrees.

**Extension:** you can return multiple values from a function like so:

```
def foo():
    return 1, 2, 3

x, y, z = foo()
```

Wrap your triangle calculation code in a function.

#### Greeting

Write a function that takes a name as an input, and prints a message greeting that person.

#### Average function

Wrap the code for your average calculator from the Lists and Loops exercises in a function that takes a list as a parameter and returns its average.

### 1.5.14 What to do next

As mentioned at the start, there are loads of Python exercises out there on the Web. If you want to learn some more advanced concepts, there are more tutorials out there too. Here's our recommendations of tutorials from Codecademy.

### 1.5.15 Appendices

#### Operators

There are three types of operators in Python: arithmetic, comparison, and logical. I'll list the most important.

##### Arithmetic

The usual mathematical order (BODMAS) applies to these, just like in normal algebra.

**+, -, *, /** Self-explanatory.

**%** Remainder. For example, `5 % 2` is 1, `4 % 2` is 0.

****** power (e.g. `4 ** 2` is 4 squared)

##### Comparison

These return a boolean (`True` or `False`) value, and are used in `if` statements and `while` loops. These are always done after arithmetic.

**==, !=** equal to, not equal to

**<, <=, >, >=** less than, less than or equal to, greater than, etc.

**in** returns true if the item on the left is contained in the item on the right. The items can be strings, lists, or other objects. For example:

```
if "car" in "Scarzy's hair":
    print("Of course.")
```

```
if 7 in [2, 35, 7, 8]:
    print("Found a seven!")
```

##### Logical

These operators are `and`, `or`, and `not`. They are done after both arithmetic and comparisons. They're pretty self-explanatory, with an example:

```
x = 5
y = 8
z = 2

if x == 5 and y == 3:
    print("Yes")
else:
    print("No")
```

(continues on next page)

(continued from previous page)

```
print(x == 5 or not y == 8)          # could use y != 8 instead
print(x == 2 and y == 3 or z == 2)   # needs brackets for clarity!
```

Output:

```
No
True
True
```

When more than one boolean operator is used in an expression, `not` is performed first (as it works on a single operand). After this, `and` is done before `or`, but you should use brackets instead of relying on that fact, for readability. So, the last line of the example should read:

```
print((x == 2 and y == 3) or z == 2)
```

### Built-in functions

A lot of functions are defined for you by Python. Those listed in the docs are always available, and are the most commonly used, including `len`, `range`, and enumerate.

Others are contained in modules. To use a function from a module, you must `import` that module, like so:

```
import math
print(math.sqrt(4))
```

One of the most useful modules for the moment will be the `math` module, see the python docs for more info.

# CHAPTER 2

# API Documentation

## 2.1 Arduino API

The Arduino provides a total of 18 pins for either digital input or output (labelled 2 to 13) and 6 for analogue input (labelled A0 to A5).

### 2.1.1 Accessing the Arduino

The Arduino can be accessed using the `arduino` property of the `Robot` object.

```
my_arduino = r.arduino
```

You can use the GPIO *(General Purpose Input/Output)* pins for anything, from microswitches to LEDs. GPIO is only available on pins 2 to 13 and A0 to A5 because pins 0 and 1 are reserved for communication with the rest of our kit.

### 2.1.2 Pin mode

GPIO pins have four different modes. A pin can only have one mode at a time, and some pins aren't compatible with certain modes. These pin modes are represented by an enum which needs to be imported before they can be used.

```
from sbot import GPIOPinMode
```

**Hint:** The input modes closely resemble those of an Arduino. More information on them can be found in their docs.

#### Setting the pin mode

You will need to ensure that the pin is in the correct pin mode before performing an action with that pin.

```
r.arduino.pins[3].mode = GPIOPinMode.DIGITAL_INPUT_PULLUP
```

You can readabout the possible pin modes below.

### GPIOPinMode.DIGITAL_INPUT

In this mode, the digital state of the pin (whether it is high or low) can be read.

```
r.arduino.pins[4].mode = GPIOPinMode.DIGITAL_INPUT

pin_value = r.arduino.pins[4].digital_read()
```

### GPIOPinMode.DIGITAL_INPUT_PULLUP

Same as `GPIOPinMode.DIGITAL_INPUT`, but with an internal pull-up resistor enabled.

```
r.arduino.pins[4].mode = GPIOPinMode.DIGITAL_INPUT_PULLUP

pin_value = r.arduino.pins[4].digital_read()
```

### GPIOPinMode.DIGITAL_OUTPUT

In this mode, you can set binary values of `0V` or `5V` to the pin.

```
r.arduino.pins[4].mode = GPIOPinMode.DIGITAL_OUTPUT
r.arduino.pins[6].mode = GPIOPinMode.DIGITAL_OUTPUT

r.arduino.pins[4].digital_write(True)
r.arduino.pins[6].digital_write(False)
```

You can get the last value you digitally wrote using `last_digital_write`.

```
pin_state = r.arduino.pins[4].last_digita_write
```

### GPIOPinMode.ANALOGUE_INPUT

Certain sensors output analogue signals rather than digital ones, and so have to be read differently. The arduino has six analogue inputs, which are labelled `A0` to `A5`; however pins `A4` and `A5` are reserved and cannot be used.

**Hint:** Analogue signals can have any voltage, while digital signals can only take on one of two voltages. You can read more about digital vs analogue signals here.

```
from sbot import AnaloguePin

r.arduino.pins[AnaloguePin.A0].mode = GPIOPinMode.ANALOGUE_INPUT

pin_value = r.arduino.pins[AnaloguePin.A0].analogue_read
```

**Hint:** The values are the voltages read on the pins, between 0 and 5.

**Warning:** Pins `A4` and `A5` are reserved and cannot be used.

### 2.1.3 Ultrasound Sensors

You can also measure distance using an ultrasound sensor from the arduino.

```
# Trigger pin: 4
# Echo pin: 5
u = r.arduino.ultrasound_sensors[4, 5]

time_taken = u.pulse()

distance_metres = u.distance()
```

**Warning:** If the ultrasound signal never returns, the sensor will timeout and return `None`.

## 2.2 Motor Board API

The kit can control multiple motors simultaneously. One Motor Board can control up to two motors.

### 2.2.1 Accessing the Motor Board

If there is exactly one motor board attached to your robot, it can be accessed using the `motor_board` property of the `Robot` object.

```
my_motor_board = r.motor_board
```

**Warning:** If there is more than one motor board on your kit, you *must* use the `motor_boards` property. `r.motor_board` *will cause an error*. This is because the kit doesn't know which motor board you want to access.

Motor boards attached to your robot can be accessed under the `motor_boards` property of the `Robot`. The boards are indexed by their serial number, which is written on the board.

```
my_motor_board = r.motor_boards["SRO-AAD-GBH"]
my_other_motor_board = r.motor_boards["SR08U6"]
```

### 2.2.2 Controlling the Motor Board

This board object has an array containing the motors connected to it, which can be accessed as `motors[0]` and `motors[1]`. The Motor Board is labelled so you know which motor is which.

```
my_motor_board.motors[0]
my_motor_board.motors[1]
```

### 2.2.3 Powering motors

Motor power is controlled using pulse-width modulation (PWM). You set the power with a fractional value between `-1` and `1` inclusive, where `1` is maximum speed in one direction, `-1` is maximum speed in the other direction and `0` causes the motor to brake.

```
my_motor_board.motors[0].power = 1
my_motor_board.motors[1].power = -1
```

These values can also be read back:

```
my_motor_board.motors[0].power
>>> 1

my_motor_board.motors[1].power
>>> -1
```

**Warning:** Setting a value outside of the range `-1` to `1` will raise an exception and your code will crash.

**Danger:** Sudden large changes in the motor speed setting (e.g. `-1` to `0`, `1` to `-1` etc.) will likely trigger the over-current protection and your robot will shut down with a distinct beeping noise and/or a red light next to the power board output that is powering the motor board.

#### Special values

In addition to the numeric values, there are two special constants that can be used: `BRAKE` and `COAST`. In order to use these, they must be imported from the `sbot` module like so:

```
from sbot import BRAKE, COAST
```

##### BRAKE

`BRAKE` will stop the motors from turning, and thus stop your robot as quick as possible.

**Hint:** `BRAKE` does the same as setting the power to `0`.

```
from sbot import BRAKE

my_motor_board.motors[0].power = BRAKE
```

#### COAST

COAST will stop applying power to the motors. This will mean they continue moving under the momentum they had before.

```
from sbot import COAST

my_motor_board.motors[1].power = COAST
```

## 2.3 Power Board API

The power board can be accessed using the power_board property of the Robot object.

```
my_power_board = r.power_board
```

### 2.3.1 Power outputs

The six outputs of the power board are grouped together as power_board.outputs.

The power board's six outputs can be turned on and off using the power_on and power_off functions of the group respectively.

**Hint:** power_on is called when you setup your robot, so this doesn't need to be called manually. The ports will come on automatically as soon as your robot is ready, before the start button is pressed.

```
r.power_board.outputs.power_off()
r.power_board.outputs.power_on()
```

You can also get information about and control each output in the group. An output is indexed using the appropriate PowerOutputPosition.

```
from sbot import PowerOutputPosition

r.power_board.outputs[PowerOutputPosition.H0].is_enabled = True
r.power_board.outputs[PowerOutputPosition.L3].is_enabled = False

boolean_value = r.power_board.outputs[PowerOutputPosition.L2].is_enabled

current_amps = r.power_board.outputs[PowerOutputPosition.H1].current
```

**Warning:** The motor and servo boards are powered through these power outputs, whilst the power is off, you won't be able to control your motors or servos. They will register as a missing board and your code will break if you try and control them.

### 2.3.2 Battery Sensor

The power board has some sensors that can monitor the status of your battery. This can be useful for checking the charge status of your battery.

```
battery_voltage = r.power_board.battery_sensor.voltage
battery_current_amps = r.power_board.battery_sensor.current
```

### 2.3.3 Buzzing

The power board has a piezo sounder which can buzz.

The `buzz` function accepts multiple parameters, depending on what you want to play. The first argument is the duration of the beep, in seconds. The later arguments are either the note you want to play, or the frequency of the buzzer (in Hertz). You have to specify which of note or frequency you're passing using a keyword argument, your code will fail otherwise.

Theoretically, the piezo buzzer will buzz at any provided frequency, however humans can only hear between 20Hz and 20000Hz.

The `Note` enum provides notes in scientific pitch notation between `C6` and `C8`. You can play other tones by providing a frequency.

**Hint:** Calling `buzz` is non-blocking, which means it doesn't actually wait for the piezo to stop buzzing before continuing with your code. If you want to wait for the buzzing to stop, add a `sleep` afterwards! If you send more than 32 beeps to the robot too quickly, your power board will crash!

```
from sbot import Note

# Buzz for half a second in D6.
r.power_board.piezo.buzz(0.5, Note.D6)

# Buzz for 2 seconds at 400Hz
r.power_board.piezo.buzz(2, 400)
```

### 2.3.4 Start Button

You can manually wait for the start button to be pressed, not only at the start.

```
r.wait_start()
```

This may be useful for debugging, but be sure to remove it in the competition, as you won't be allowed to touch the start button after a match has begun!

## 2.4 Servo Board API

The kit can control multiple servos simultaneously. One Servo Board can control up to twelve servos.

### 2.4.1 Accessing the Servo Board

The servo board can be accessed using the `servo_board` property of the `Robot` object.

```
my_servo_board = r.servo_board
```

This board object has an array containing the servos connected to it, which can be accessed as `servos[0]`, `servos[1]`, `servos[2]`, etc. The servo board is labelled so you know which servo is which.

**Hint:** Remember that arrays start counting at 0.

### 2.4.2 Setting servo positions

The position of servos can range from `-1` to `1` inclusive:

```
# set servo 1's position to 0.2
r.servo_board.servos[1].position = 0.2

# Set servo 2's position to -0.55
r.servo_board.servos[2].position = -0.55
```

You can read the last value a servo was set to using similar code:

```
last_position = r.servo_board.servos[11].position
```

**Attention:** While it is possible to retrieve the last position a servo was set to, this does not guarantee that the servo is currently in that position.

### 2.4.3 How the set position relates to the servo angle

**Danger:** You should be careful about forcing a servo to drive past its end stops. Some servos are very strong and it could damage the internal gears.

The angle of an RC servo is controlled by the width of a pulse supplied to it periodically. There is no standard for the width of this pulse and there are differences between manufacturers as to what angle the servo will turn to for a given pulse width. To be able to handle the widest range of all servos our hardware outputs a very wide range of pulse widths which in some cases will force the servo to try and turn past its internal end-stops. You should experiment and find what the actual limit of your servos are (it almost certainly won't be -1 and 1) and not drive them past that.

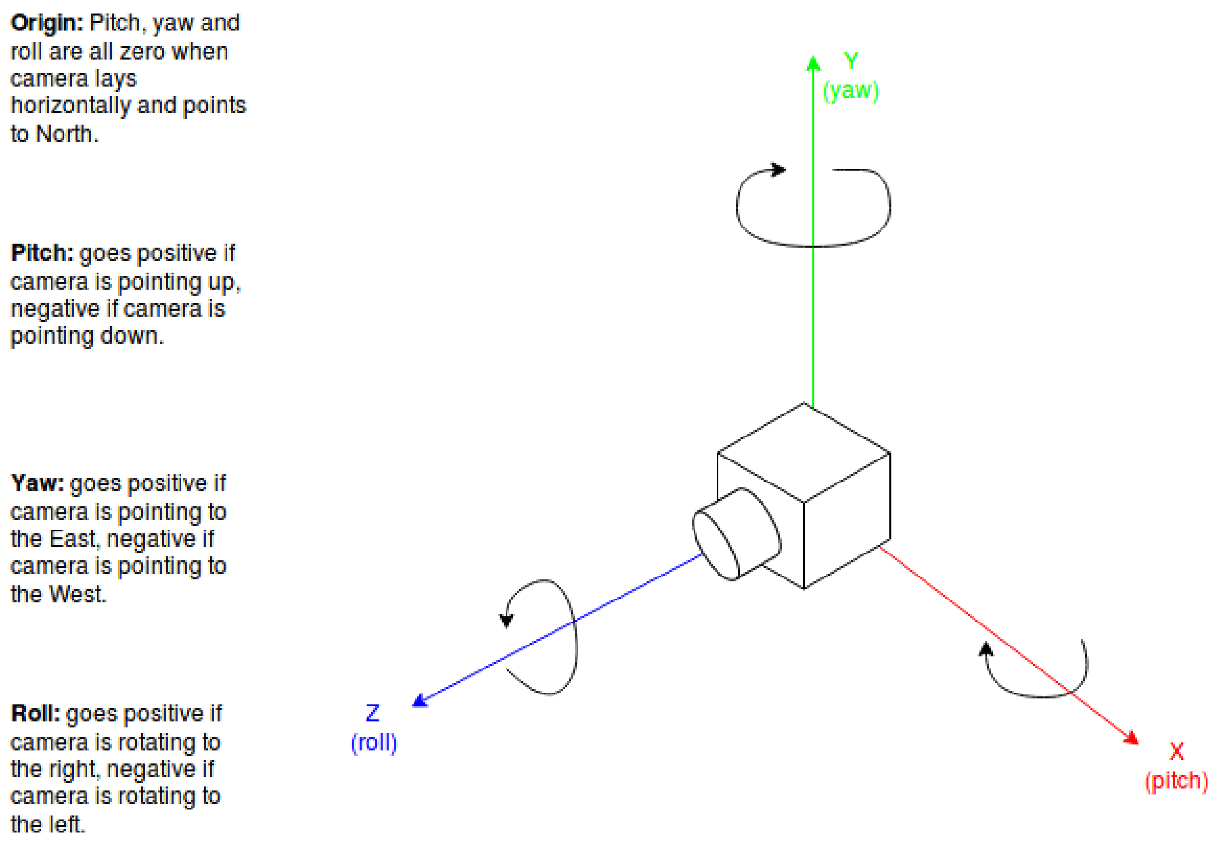
## 2.5 Vision

### 2.5.1 Orientation

Orientation represents the rotation of a marker around the x, y, and z axes. These can be accessed as follows:

- `rot_x` / `pitch` - the angle of rotation in radians counter-clockwise about the Cartesian x axis.
- `rot_y` / `yaw` - the angle of rotation in radians counter-clockwise about the Cartesian y axis.
- `rot_z` / `roll` - the angle of rotation in radians counter-clockwise about the Cartesian z axis.

Rotations are applied in order of z, y, x.



**Origin:** Pitch, yaw and roll are all zero when camera lays horizontally and points to North.

**Pitch:** goes positive if camera is pointing up, negative if camera is pointing down.

**Yaw:** goes positive if camera is pointing to the East, negative if camera is pointing to the West.

**Roll:** goes positive if camera is rotating to the right, negative if camera is rotating to the left.

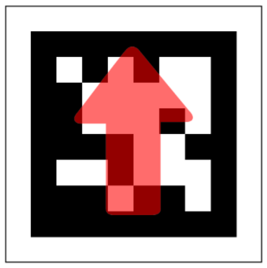
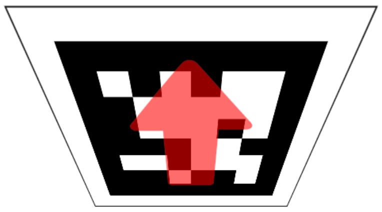
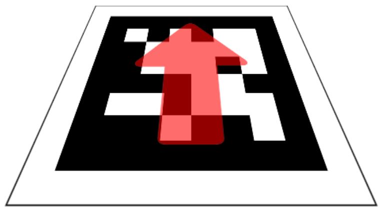
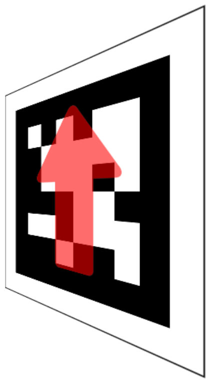
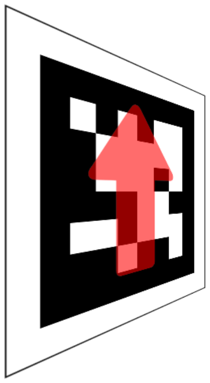
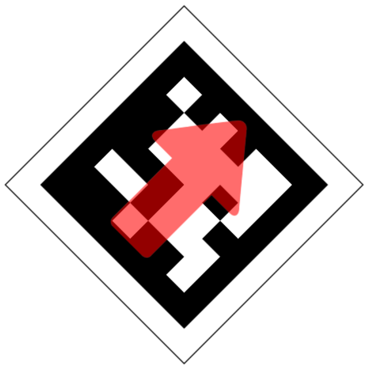
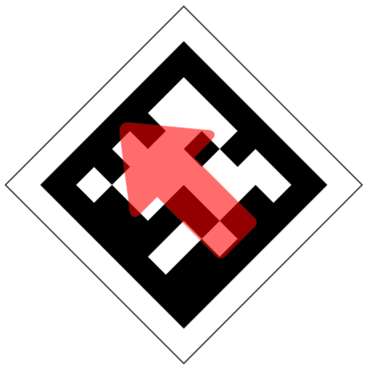
Fig. 1: Yaw Pitch and Roll (Image source: Peking University)

```
markers = r.camera.see()

for m in markers:
    print(m.orientation.rot_x)  # Angle of rotation about x axis.
    print(m.orientation.rot_y)  # Angle of rotation about y axis.
    print(m.orientation.rot_z)  # Angle of rotation about z axis.
```

**Note:** In our use case the z axis always faces the camera, and thus will appear as a clockwise rotation

### Examples

The following table visually explains what positive and negative rotations represent.

| | | | |
|---|---|---|---|
| 0 in all axes | |  | |
| $\pi/4$ in rot_x |  | $-\pi/4$ in rot_x |  |
| $\pi/4$ in rot_y |  | $-\pi/4$ in rot_y |  |
| $\pi/4$ in rot_z |  | $-\pi/4$ in rot_z |  |

### 2.5.2 Position

Your robot supports three different coordinates systems for position:

- Cartesian
- Spherical
- Cylindrical

The latter are both variants of a Polar Coordinates system.

#### Cartesian

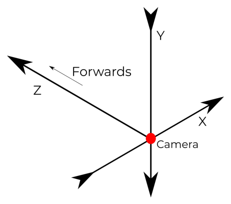


Fig. 2: The cartesian coordinates system

The cartesian coordinates system has three *principal axes* that are perpendicular to each other.

The value of each coordinate indicates the distance travelled along the axis to the point.

The camera is located at the origin, where the coordinates are `(0, 0, 0)`.

```
markers = r.camera.see()

for m in markers:
    print(m.position.cartesian.x)  # Displacement from the origin in metres, along x␣
↪axis.
    print(m.position.cartesian.y)  # Displacement from the origin in metres, along y␣
↪axis.
    print(m.position.cartesian.z)  # Displacement from the origin in metres, along z␣
↪axis.
```

**Hint:** The *y* axis decreases as you go up. This matches convention for computer vision systems.

#### Spherical

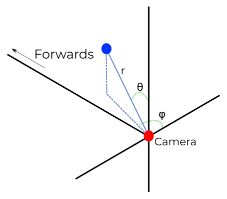


Fig. 3: The spherical coordinates system

The spherical coordinates system has three values to specify a specific point in space.

- `r` - The *radial distance*, the distance from the origin to the point, in metres.
- $\theta$ (theta) - The angle from the azimuth to the point, in radians.
- $\phi$ (phi) - The polar angle from the plane of the camera to the point, in radians.

The camera is located at the origin, where the coordinates are `(0, 0, 0)`.

```
markers = r.camera.see()

for m in markers:
    print(m.position.spherical.r)  # Distance from the origin in metres
    print(m.position.spherical.theta)  # The angle from the azimuth to the point, in␣
↪radians.
    print(m.position.spherical.phi)  # The polar angle from the plane of the camera␣
↪to the point, in radians.
```

**Hint:** You can use the `math.degrees` function to convert from radians to degrees.

**Note:** When searching for spherical coordinates, you may find a references with phi and theta the other way around. This is due to there being *two* conventions for this. We use the ISO 80000-2 16.3 system, as often found in physics.

## Cylindrical

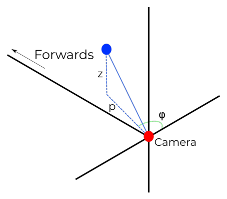


Fig. 4: The cylindrical coordinates system

The cylindrical coordinates system has three values to specify a point in space.

- $\rho$ (rho) - The axial distance from the origin, in metres.
- $\phi$ (phi) - The polar angle from the plane of the camera to the point, in radians.
- `z` - The height of the point from the plane of the camera.

```
markers = r.camera.see()

for m in markers:
    print(m.position.cylindrical.p)  # The axial distance from the origin.
    print(m.position.cylindrical.phi)  # The polar angle from the plane of the camera␣
↪to the point, in radians.
    print(m.position.cylindrical.z)  # The height of the point from the plane of the␣
↪camera, in metres.
```

**Note:** Whilst $\rho$ is technically rho, we denote it as `p` in the API to make it easier to type.

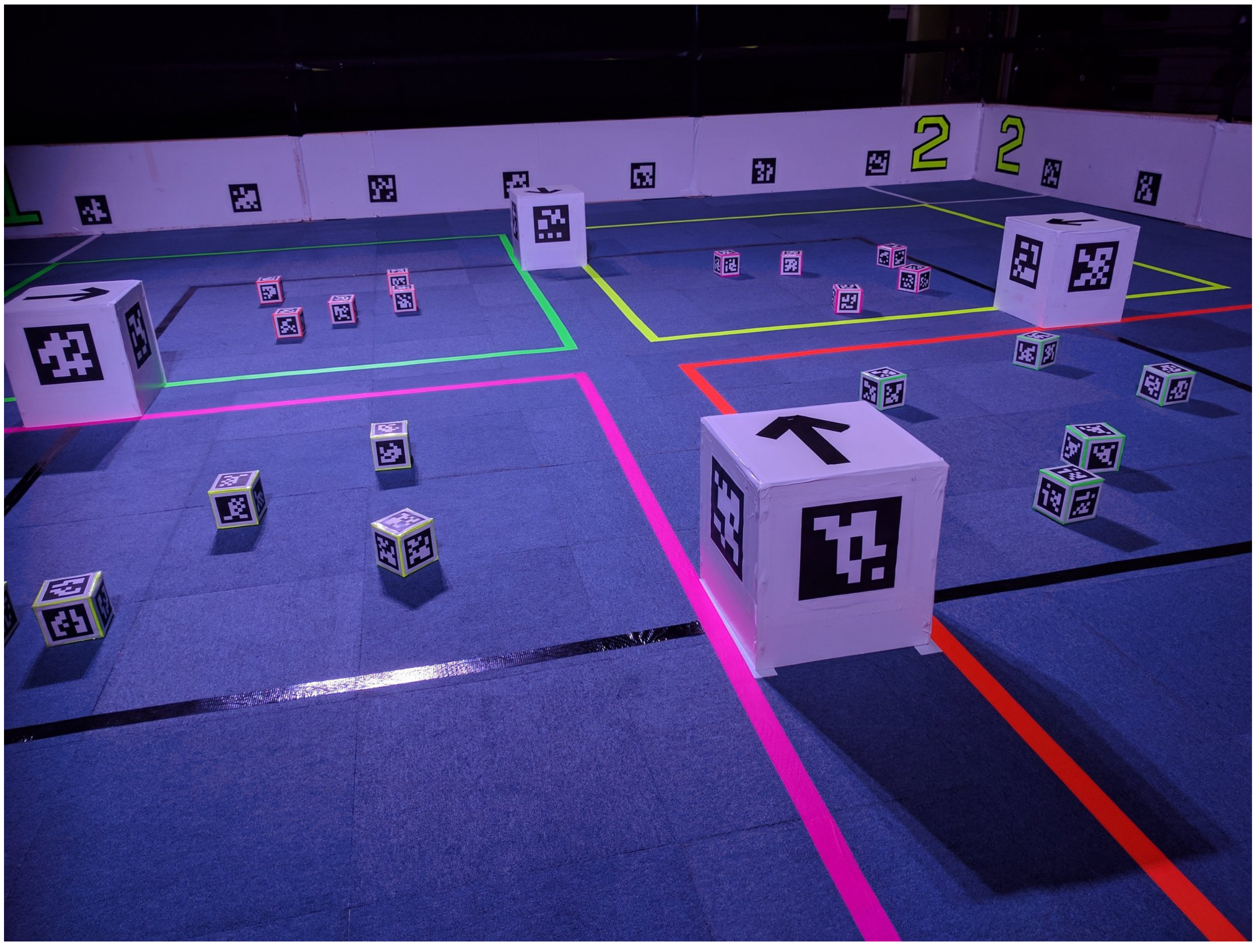


Fig. 5: An arena with Fiducial Markers.

Your robot is able to use a webcam to detect Fiducial Markers. Specifically it will detect AprilTags, using the `36H11` marker set.

Using Pose Estimation, it can calculate the orientation and position of the marker relative to the webcam. Using this data, it is possible to determine the location of your robot and other objects around it.

### 2.5.3 Searching for markers

Assuming you have a webcam connected, you can use `r.camera.see()` to take a picture. The software will process the picture and returns a list of the markers it sees.

```
markers = r.camera.see()
```

**Hint:** Your camera will be able to process images better if they are not blurred.

### 2.5.4 Saving camera output

You can also save a snapshot of what your webcam is currently seeing. This can be useful to debug your code. Every marker that your robot can see will have a square annotated around it, with a red dot indicating the bottom right corner of the marker. The ID of every marker is also written next to it.

Snapshots are saved to your USB drive, and can be viewed on another computer.

```
r.camera.save("snapshot.jpg")
```

### 2.5.5 Markers

The marker objects in the list expose data that may be useful to your robot.

#### Marker ID

Every marker has a numeric identifier that can be used to determine what object it represents.

```
markers = r.camera.see()

for m in markers:
    print(m.id)
```

#### Position

Each marker has a position in 3D space, relative to your webcam.

You can access the position using `m.bearing` and `m.distance`.

```
markers = r.camera.see()

for m in markers:
    print(m.bearing)   # Bearing to the marker from the origin, in radians
    print(m.distance)  # Distance to the marker from the origin, in metres
```

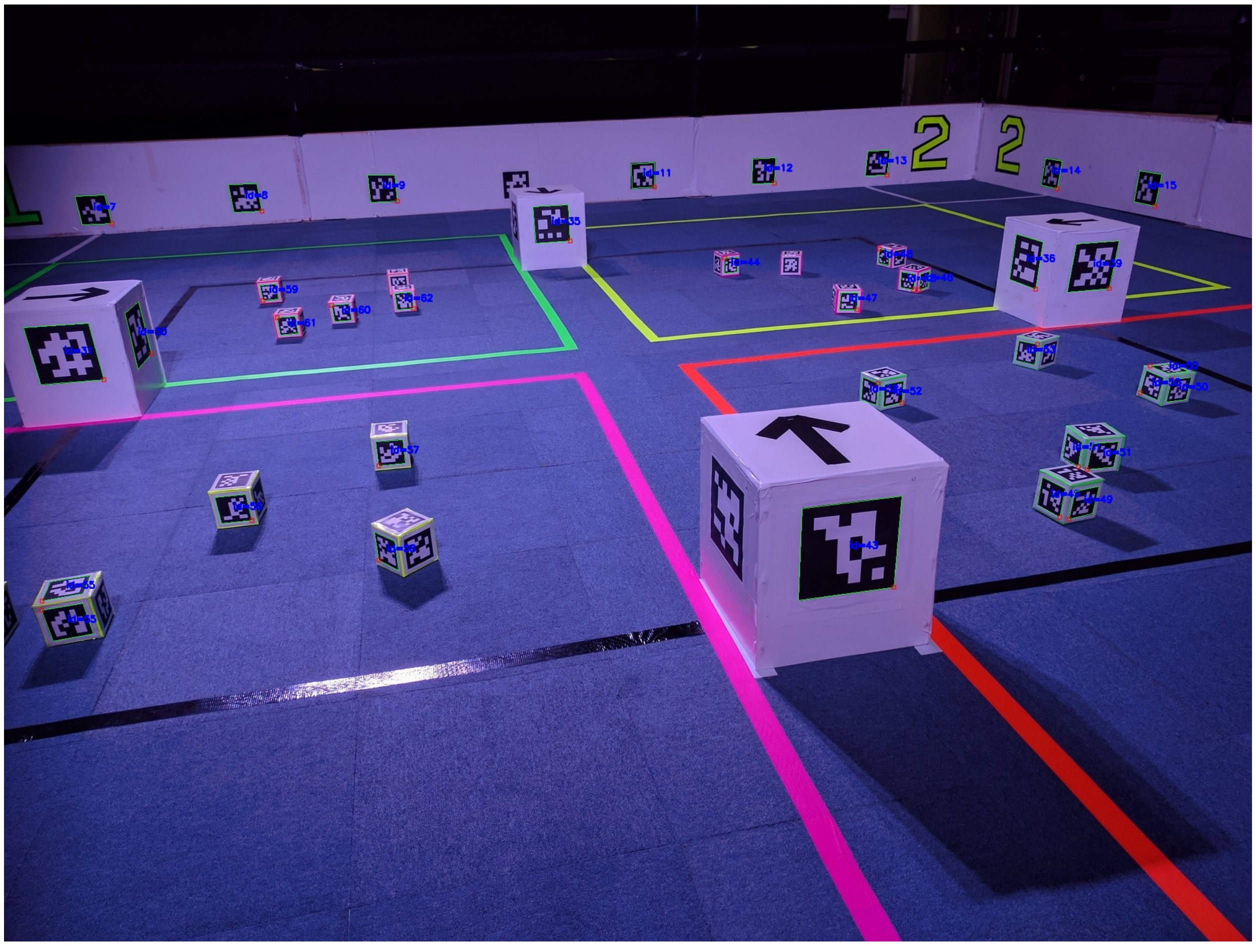


Fig. 6: An annotated arena with Fiducial Markers.

- m.bearing is equivalent to m.position.cylindrical.phi.
- m.distance is equivalent to m.position.cylindrical.p.

For further information on position, including how to use m.position and the coordinate systems, see Position.

It is also possible to look at the Orientation of the marker.

**Hint:** You can use the math.degrees function to convert from radians to degrees.

#### Pixel Positions

The positions of various points on the marker within the image are exposed over the API. This is useful if you would like to perform your own Computer Vision calculations.

The corners are specified in clockwise order, starting from the top left corner of the marker. Pixels are counted from the origin of the image, which conventionally is in the top left corner of the image.

```
markers = r.camera.see()

for m in markers:
    print(m.pixel_corners)  # Pixel positions of the marker corners within the image.
    print(m.pixel_centre)  # Pixel positions of the centre of the marker within the␣
↪image.
```

## 2.6 Game State

### 2.6.1 Mode

Your robot will behave slightly differently between testing and in the arena.

The main difference is that your robot will stop when the match is over, but you may also want to make your own changes to your robot's behaviour.

Your robot can be in 1 of 2 modes: DEVELOPMENT and COMPETITION. By default, your robot will be in DEVELOPMENT mode:

```
r.is_competition
>> False
```

During competition mode, your robot will stop executing code at the end of the match.

### 2.6.2 Zone

Your robot will start in a corner of the arena, known as its starting zone. The number of zones depends on the game. Each zone is given a number, which you can access with the zone property:

```
r.zone
>> 2
```

During a competition match, a USB drive will be used to tell your robot which corner it's in. By default during development, this is 0.

## 2.7 Advanced API

This page documents parts of the API that are for more advanced use. Most students will never need to use features from this page.

**Warning:** These features are for advanced users only.

### 2.7.1 Debug Mode

It is possible to run your robot in "Debug Mode".

In "Debug Mode", your robot will print more information about what it is doing.

```
from sbot import Robot
r = Robot(debug=True)
```

**Warning:** Debug mode is quite verbose. It will print a lot of information that you will not need.

### 2.7.2 Console Mode

It is possible to test your code in "console mode".

This allows you to see every interaction that your robot makes with its hardware interactively. It may be useful to help debug your code.

#### Prerequisites

You need to have the `sbot` library available on your computer.

This can be achieved by following the PyCharm Tutorial or by installing `sbot` from PyPI.

#### Using Console Mode

Console mode is activated by instantiating your `Robot` object with a `ConsoleEnvironment`.

```
from sbot import Robot
from sbot.env import ConsoleEnvironment
r = Robot(environment=ConsoleEnvironment)
```

Robot code that is using Console Mode should be executed on a computer only.

- Run `python3 main.py` in a terminal window.
- Execute the code using PyCharm.

**Danger:** Running Console Mode code on a robot will result in a non-functioning robot.

Console Mode will print the current actions of your robot, and prompt you for information about it.

Programming your robot is done in Python, specifically version 3.7.4. You can learn more about Python from their docs, and our whirlwind tour.

## Setup

The following two lines are required to complete initialisation of the kit:

```
from sbot import Robot

r = Robot()
```

Once this has been done, this `Robot` object can be used to control the robot's functions.

The remainder of the tutorials pages will assume your `Robot` object is defined as `r`.

## Running your code

Your code needs to be put on a USB drive in a file called `main.py`. On insertion into the robot, this file will be executed. The file is directly executed off your USB drive, with your drive as the working directory.

To stop your code running, you can just remove the USB drive. This will also stop the motors and any other peripherals connected to the kit.

You can then reinsert the USB drive into the robot and it will run your `main.py` again (from the start). This allows you to make changes and test them quickly.

**Hint:** If this file is missing or incorrectly named, your robot won't do anything. No log file will be created.

## Start Button

After the robot has finished starting up, it will wait for the *Start Button* on the power board to be pressed before continuing with your code, so that you can control when it starts moving. There is a green LED next to the start button which flashes when the robot is finished setting up and the start button can be pressed.

## Running Code before pressing the start button

If you want to do things before the start button press, such as setting up servos or motors, you can pass `wait_start` to the `Robot` constructor. You will then need to wait for the start button manually.

```
r = Robot(wait_start=False)

# Do your setup here

r.wait_start()
```

## Logs

A log file is saved on the USB drive so you can see what your robot did, what it didn't do, and any errors it raised. The file is saved to `log.txt` in the top-level directory of the USB drive.

**Warning:** The previous log file is deleted at the start of each run, so copy it elsewhere if you need to keep hold of it!

### Serial number

All kit boards have a serial number, unique to that specific board, which can be read using the `serial` property:

```
r.power_board.serial
>>> 'SRO-AA2-7XS'
r.servo_board.serial
>>> 'SRO-AA4-LG2'
r.motor_board.serial
>>> 'SRO-AAO-RV2'
```

### Included Libraries

Python already comes with plenty of built-in libraries to use. We install some extra ones which may be of use:

- numpy

**Hint:** If you would like an extra library installed, go and ask a volunteer to see if we can help.

CHAPTER 3

# Kit Documentation

## 3.1 Arduino

This board allows you to control GPIO pins and analogue pins. More specifically, it's an Arduino Uno.

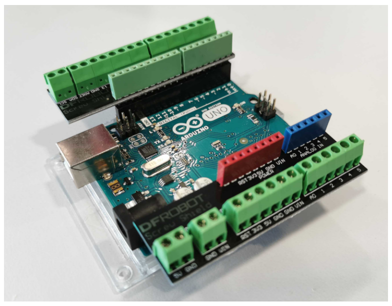


Fig. 1: Arduino Uno

### 3.1.1 Headers

We have supplied 2 screw terminal headers for your Arduino, allowing you to easily, and securely attach your sensors.

### 3.1.2 The reset button

The reset button allows you to instantly reboot the Arduino in case it isn't working. This is not a guaranteed fix, but may solve some problems.

### 3.1.3 GPIO Pins

The Arduino allows you to connect your kit to your own electronics. It has fourteen digital I/O pins, and six analogue. The analogue pins can read an analogue signal from 0 to 5V. The board also has a couple of ground pins, as well as some pins fixed at 3.3V and 5V output.

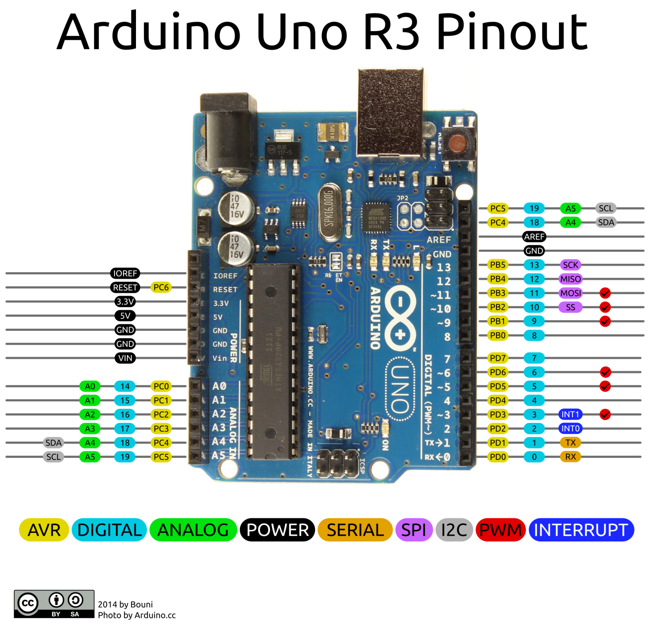


Fig. 2: Pin Map

### 3.1.4 Ultrasound Sensors

Ultrasound sensors are a useful way of measuring distance. Ultrasound sensors communicate with the kit using two wires. A signal is sent to the sensor on the trigger pin, and the length of a response pulse on the echo pin can be used to calculate the distance.

**Warning:** Ultrasound should only be considered accurate up to around two metres, beyond which the signal can become distorted and produce erroneous results.

The sensor has four pin connections: ground, 5V (sometimes labelled *vcc*), *trigger* and *echo*. Most ultrasound sensors will label which pin is which. The ground and 5V should be wired to the ground and 5V pins of the Arduino respectively. The trigger and echo pins should be attached to two different digital IO pins. Take note of these two pins, you'll need them to use the sensor.

**Hint:** If the sensor always returns a distance of zero, it might mean the *trigger* and *echo* pins are connected the wrong way! Either change the pin numbers in the code, or swap the connections.

### 3.1.5 Designs

The schematic diagrams for the Arduino is below, as well as the source code of the firmware on the Arduino. You do not need this information to use the board but it may be of interest to some people.

- Arduino Uno Schematic
- Firmware Source

## 3.2 Batteries

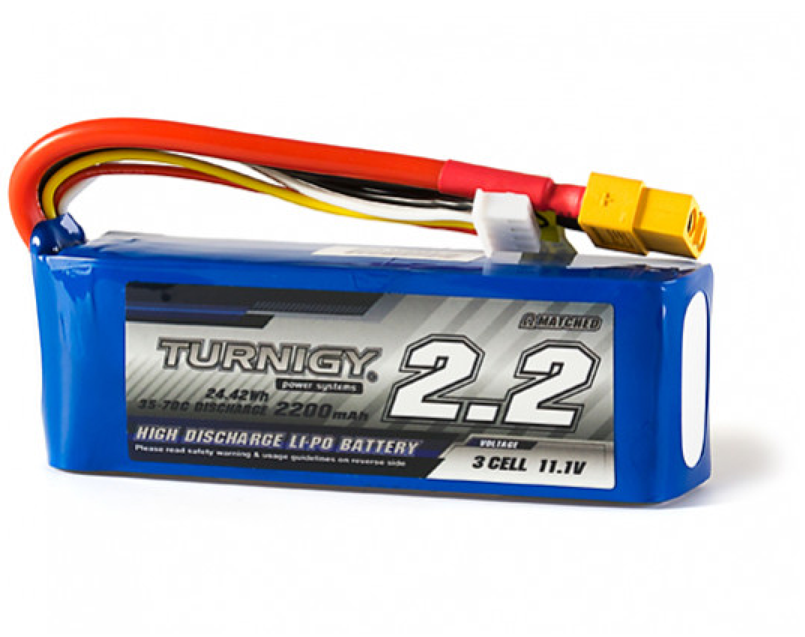


Fig. 3: LiPo Battery

Your robot uses lithium-ion polymer (LiPo) batteries. These are similar to those used in laptops, and are small and light for the amount of energy they contain. This is great for your robot but it is vital to treat such a high concentration of energy with respect. If you do not, there is a serious risk of fire and injury. To avoid this, you should follow the safety information on this page closely, at all times.

**Warning:** You must not use any batteries, chargers, bags or cables not explicitly authorised by SourceBots.

### 3.2.1 Warnings

- Never leave batteries unattended when they are in use.
- Always place the batteries in the provided bag when not in use.
- If a battery has any cuts, nicks, exposed copper on wires or is bulging to the point of no longer being squishy, contact us immediately.

**Danger:** Disobeying the above warnings greatly increases the risk of damage to your battery, which can have **fatal consequences**. If you have any questions or doubts, talk to a member of staff immediately.

### 3.2.2 Storing batteries

When your batteries are not actively in use, they should be safely stored. You must disconnect the batteries from all electrical equipment, and place them in the battery charging bag. You should then store the charging bag in a safe location.

### 3.2.3 Operating batteries

To use your batteries, you must connect them to the power board. Do not tamper with the cable or connect the batteries to anything other than the power board (or the charger when charging).

During operation, the battery is protected by over-current protection and a fuse in the power board. If any equipment is short circuited, the over-current protection will activate, protecting the battery. In extreme circumstances the fuse may blow to prevent damage to the battery. This is an important safety feature: do not, under any circumstances, bypass the fuse. It is not user serviceable and if it has blown then the power board must be replaced. If you suspect the fuse has blown then please contact us straight away.

Mechanical damage to a battery can be dangerous, and a puncture or large force applied to a battery causes a serious risk of fire. To avoid this, your battery should be shielded from mechanical damage while you operate it. Secure your battery to your robot, so that it does not move or fall off while the robot moves. You should also build a compartment for the battery to be placed in, so that accidental collisions do not damage the battery.

### 3.2.4 Flat batteries

When the battery has been almost completely discharged, the Power Board will automatically turn off and the LED marked “Power / Flat Battery Indicator” in the diagram will flash red and green. You should immediately disconnect the battery. Flat batteries should be given to a faciliator in exchange for a freshly charged battery.

## 3.3 Motor Board

The Motor Board can be used to control two 12V DC motors. These can be used for moving your robot, although don’t feel you are limited to using them for this purpose.

The speed and direction of the two outputs are controlled independently through the USB interface. The USB interface is isolated from the rest of the board to prevent damage to the host in the case of a board failure. Due to this isolation the board must have power applied to both the power connector (from the 12V outputs on the power board) and the USB port. If the board does not have power applied to the power connector then the kit will report that there is a problem with the motor board.

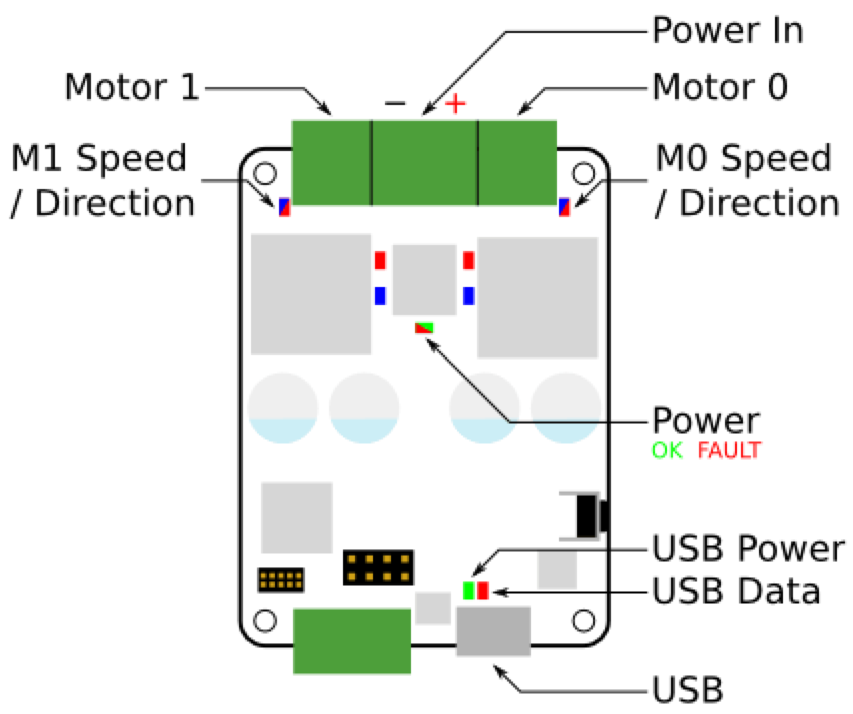


Fig. 4: Board Diagram

### 3.3.1 Board diagram

### 3.3.2 Indicators

| LED | Meaning | Initial power-up state |
|---|---|---|
| Power | The board is powered | On |
| M0/M1 Speed/Direction | Brightness indicates speed, colour indicates direction | Off |
| USB Power | The USB interface is powered | On |
| USB Data | Data is being transferred to/from the board | Off |

### 3.3.3 Case dimensions

The case measures 70x84x20mm. Don't forget that the cables will stick out.

### 3.3.4 Specification

| Parameter | Value |
|---|---|
| Nominal input voltage | 11.1V $\pm$ 15% |
| Absolute maximum input voltage | 16V |
| Minimum input voltage | 9V |
| Output voltage | 11.1V $\pm$ 15% |
| Continuous output current per channel | 10A |
| Peak output current[1] | 20A |
| UART connection voltage[2] | 3.3–5V |

### 3.3.5 Designs

You can access the schematics and source code of the firmware on the motor board in the following places. You do not need this information to use the board but it may be of interest to some people.

- Full Schematics
- Firmware Source
- Hardware Source

## 3.4 Power Board

The Power Board distributes power from the battery to the rest of the kit. It provides six individual general-purpose 12V power outputs along with a separate power connector for the Raspberry Pi.

It also holds the internal On | Off switch for the whole robot as well as the Start button which is used to start your robot code.

### 3.4.1 Board diagram

### 3.4.2 Connectors

There are six power output connectors on the board, labelled L0–L3, H0, and H1. These can supply around 11.1V (±15%). The "H" connectors will supply more current than the "L" connectors.

They should be used to connect to the motor board power input, though can also be used to power other devices. These are enabled when your robot code is started and can also be turned on or off from your code.

There are two 5V connectors that can be used to connect low-current devices that take 5V inputs, such as the Raspberry Pi.

There is also a Micro USB B connector which should be used to connect the Raspberry Pi for control of the power board.

Finally, there are connectors for external Start and On|Off switches. You may connect any latching switch for the On|Off switch, or a push-to-make button for the Start button.

**Hint:** If you intend to use only the internal On|Off switch, a CamCon must be plugged into the On|Off connector with a wire connecting one pin to the other pin on the same connector. Your power board should already have one of these plugged in.

[1] Can be sustained for one second, on a single channel.
[2] If the board is controlled solely via the UART connection, this voltage must be supplied via the UART connector.

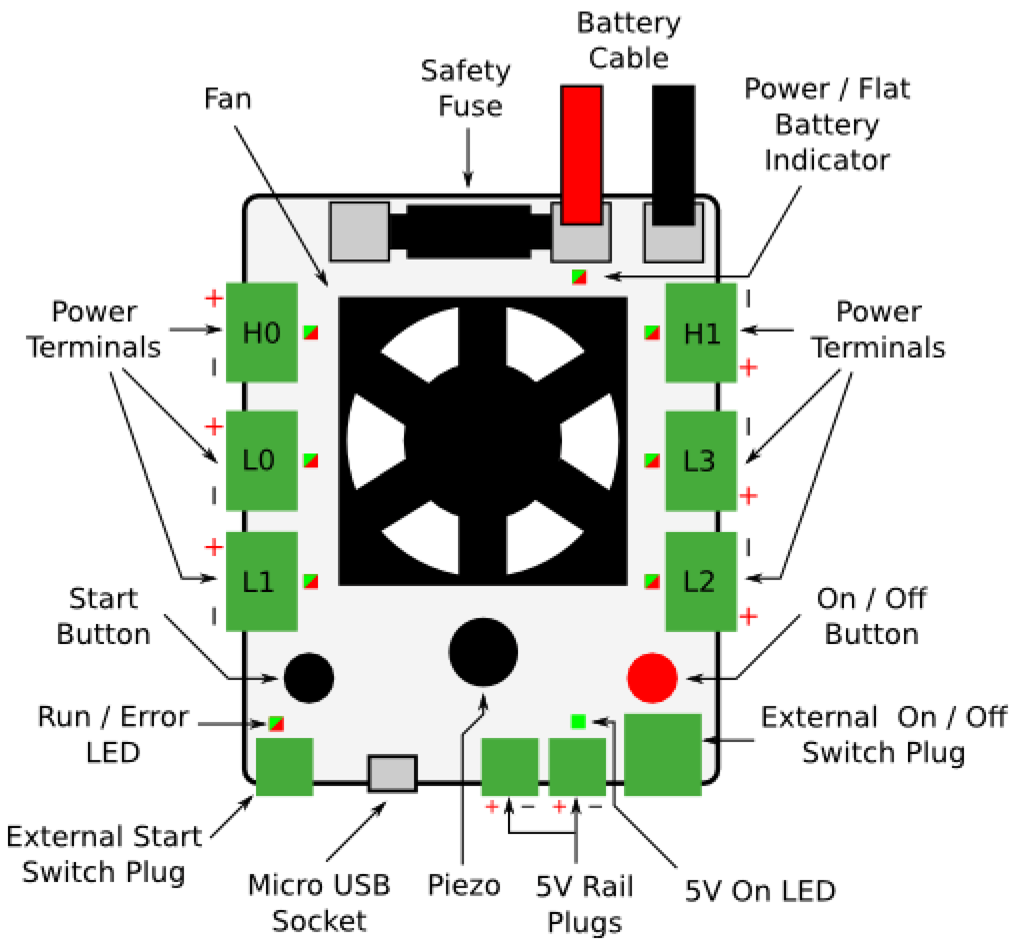


Fig. 5: Power Board Diagram

### 3.4.3 Indicators

| LED | Meaning | Initial power-up state |
|---|---|---|
| PWR \| FLAT | Green when powered Flashing red and green when the battery is low | Green |
| 5V | Green when 5V is being supplied | Green |
| H0-1, L0-3 | Green when the corresponding output is on [1]_Red when the output's current limit is reached | Off |
| RUN \| ERROR | Orange on power-up, or USB reset Flashing green when ready to run Solid green when running or booting | Orange |

On power-up, the Power Board will emit some beeps, which are related to the version of the firmware it has installed.

If the Power Board starts beeping (and all the outputs turn off) then this means that the whole board's current limit has been triggered.

### 3.4.4 Controls

| Control | Use |
|---|---|
| ON\|OFF | Turns the power board on, when used in conjunction with an external switch |
| START | Starts your program |

### 3.4.5 Case dimensions

The case measures 83x99x24mm. Don't forget that the cables will stick out.

### 3.4.6 Specification

| Parameter | Value |
|---|---|
| Main battery fuse current | 40A |
| Overall current limit[2] | 30A |
| High current outputs (H0-1) | 20A |
| Low current outputs (L0-3) | 10A |
| Motor rail output voltage (nominal) | 11.1V ± 15% |
| Total 5V output | 2A |

### 3.4.7 Designs

You can access the schematics and source code of the firmware for the power board in the following places. You do not need this information to use the board but it may be of interest to some people.

- Full Schematics
- Firmware Source
- Hardware Source

[2] If overall current limit is exceeded, the Power Board will turn off and start beeping.

## 3.5 Servo Board

The Servo Board can be used to control up to 12 RC servos. Many devices are available that can be controlled as servos, such as RC motor speed controllers, and these can also be used with the board.

### 3.5.1 Board Diagram

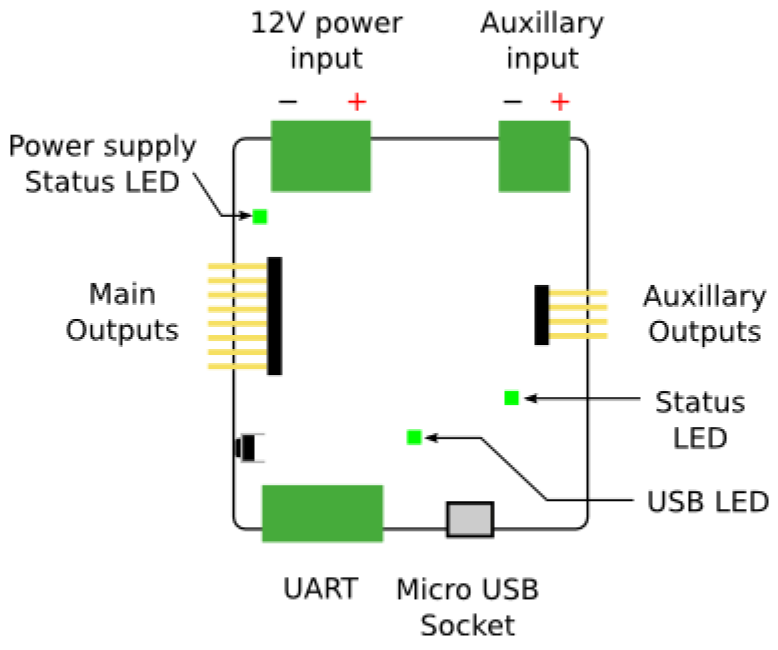


Fig. 6: Board Diagram

### 3.5.2 Indicators

| LED | Meaning | Initial power-up state |
|---|---|---|
| Power | The board is powered | On |

### 3.5.3 Connectors

There are 8 servo connections on the left-side of the board, and 4 on the right. Servo cables are connected vertically, with 0V (the black or brown wire) at the bottom of the board.

For the servo board to operate correctly, you must connect it to the 12V power rail from the power board. A green LED will light next to the servo board 12V connector when it is correctly powered.

### 3.5.4 Case Dimensions

The case measures 68x68x21mm. Don’t forget that the cables will stick out.

### 3.5.5 Range of servos

When using the majority of servos that we supply, you will find that the servo will only turn about 90 degrees.

### 3.5.6 Specification

| Parameter | Value |
|---|---|
| Number of servo channels | 12 |
| Nominal input voltage | 11.1V ± 15% |
| Output voltage | 5.5V |
| Maximum total output current[1] | 10A |

### 3.5.7 Designs

You can access the schematics and source code of the firmware on the servo board in the following places. You do not need this information to use the board but it may be of interest to some people.

- Full Schematics
- Firmware Source
- Hardware designs

## 3.6 Raspberry Pi

**Danger:** Do not reflash or edit the SD card. This will stop your robot working!

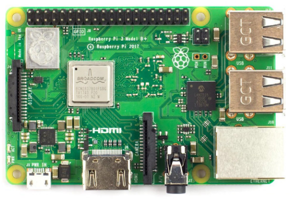


Fig. 7: Raspberry Pi 3B+

The brain of your robot is a Raspberry Pi 3 / 3B+. This handles the running of your python code, recognition of markers and sends control commands to the other boards.

**Warning:** The model of Pi you have will make no difference to the functioning of your robot.

### 3.6.1 Power Hat

Your Raspberry Pi has a Pi Power Hat mounted on the top. This allows you to connect power to it using a 3.81mm CamCon.

[1] If the auxiliary input is connected, outputs 8-11 have an independent maximum current.

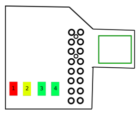


Fig. 8: Pi Power Hat

### Indicator LEDs

There are 4 indicator LEDs on the Pi Power Hat.

All LEDs will turn on at boot. After the Pi detects a USB stick, the LEDs work as follows:

- 1. This LED will illuminate bright green when your Raspberry Pi is on. You may also notice it flicker during boot.
- 2. This LED will illuminate green when your code has finished without error.
- 3. This LED will illuminate yellow whilst your code is running.
- 4. This LED will illuminate red if your code has crashed.

**Hint:** The LEDs may take a few seconds to update after you insert or remove your USB.

## 3.6.2 Technical Details

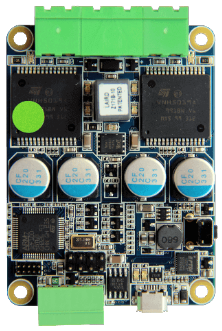
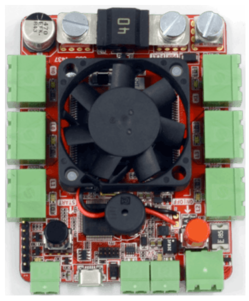
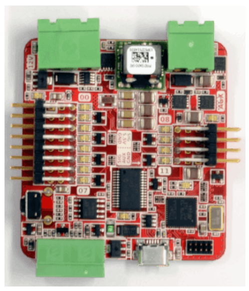
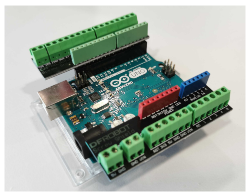
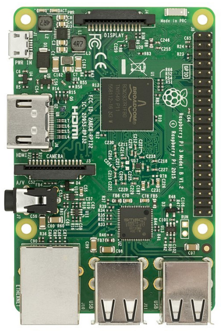
Your robot is running a customised version of the Raspbian operating system.

When a USB stick is inserted, the SourceBots software will look for a *main.py*, and then execute it.

The output of your code is written to a `log.txt` on the USB stick, and also logged to the *systemd journal*.

### Overview

Your kit consists of 6 main “boards”.

| Motor Board | Power Board | Servo Board |
|---|---|---|
|  |  |  |
| Arduino | Raspberry Pi | Webcam |
|  |  | |

It should be noted that only the Raspberry Pi and Power Board are required for your kit to work. The other boards will provide useful features to help your robot work.

You will also need a Battery to use your kit, although this may not be supplied to you immediately.

## 3.7 Other Parts

In addition to the boards, your kit will also contain:

| Part | Quantity | Specification |
|---|---|---|
| USB Hub | 1 | |
| Micro USB Cable | 3 | |
| USB B Cable | 1 | |
| Large CamCon | 4 | 7.5mm |
| Medium CamCon | 5 | 5mm |
| Small CamCon | 2 | 3.81mm |

CHAPTER 4

# How to use the docs

Within this documentation, you will come across a number of boxes like this:

**Hint:** Read the docs!

**Attention:** Directives at large.

**Caution:** Don't take any wooden nickels.

**Danger:** Mad scientist at work!

**Error:** Does not compute.

It would be advisable to take note of these, they contain useful and important information.

# CHAPTER 5

# Indices and tables

- genindex
- modindex
- search