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:

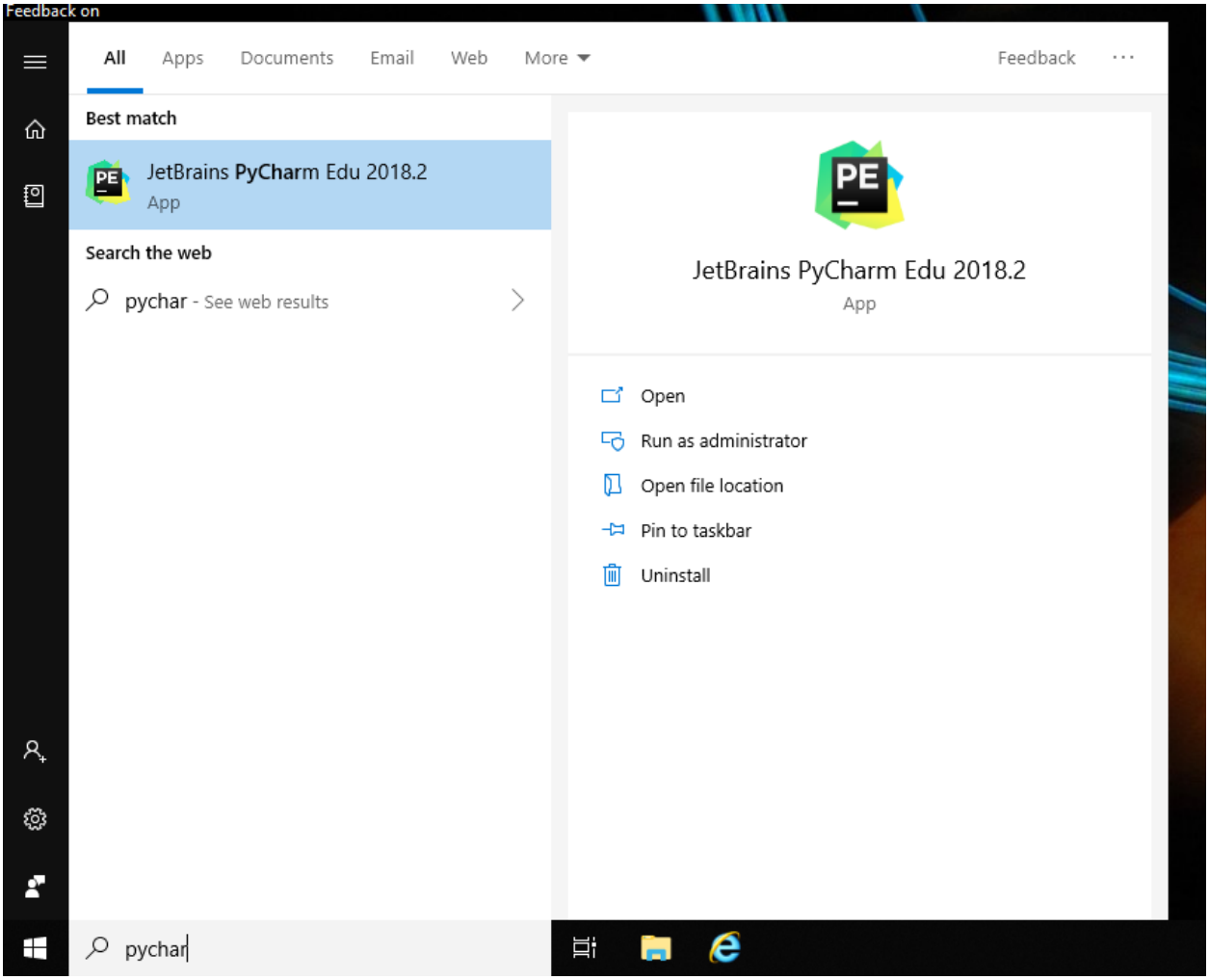

Fig. 2: Searching for and selecting PyCharm in the Start Menu.

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

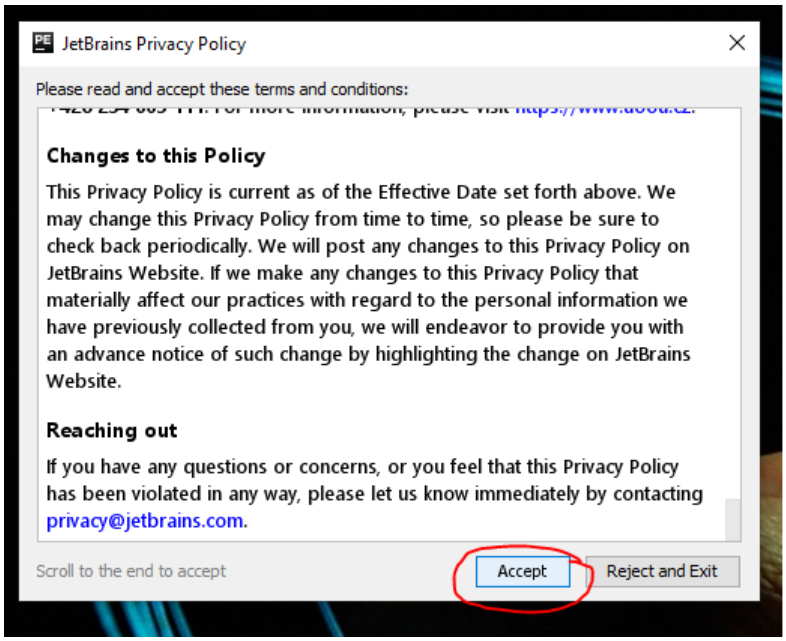

Fig. 3: The terms and conditions dialogue box.

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.

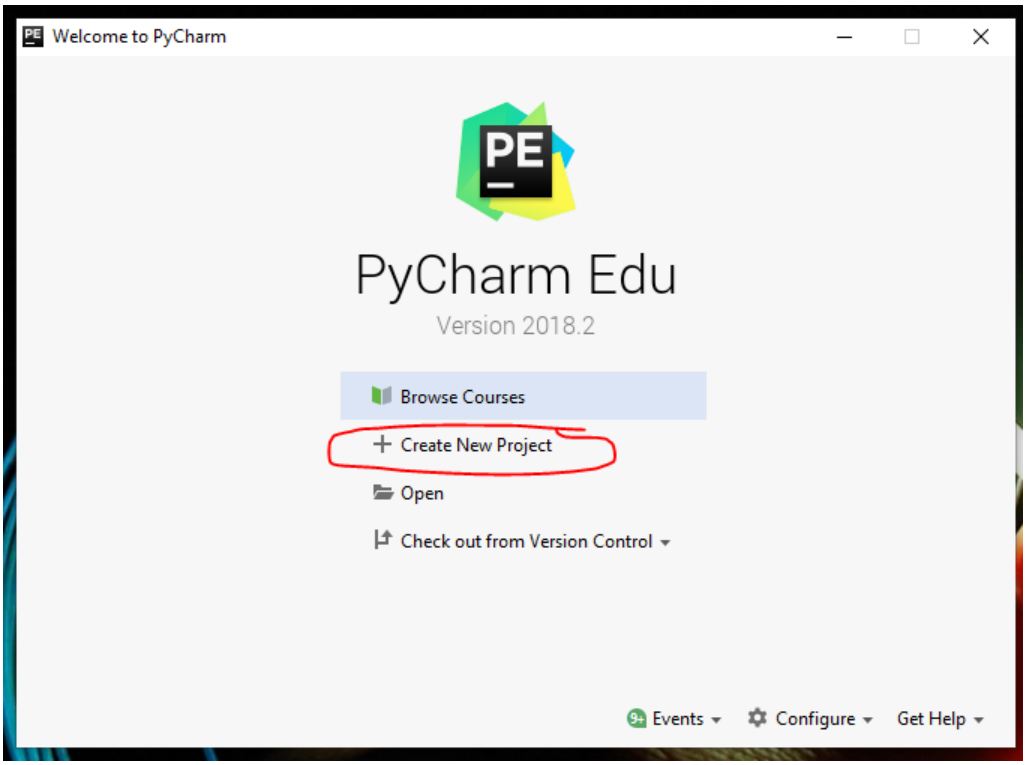

Fig. 4: The welcome dialogue, with the Create New Project button.

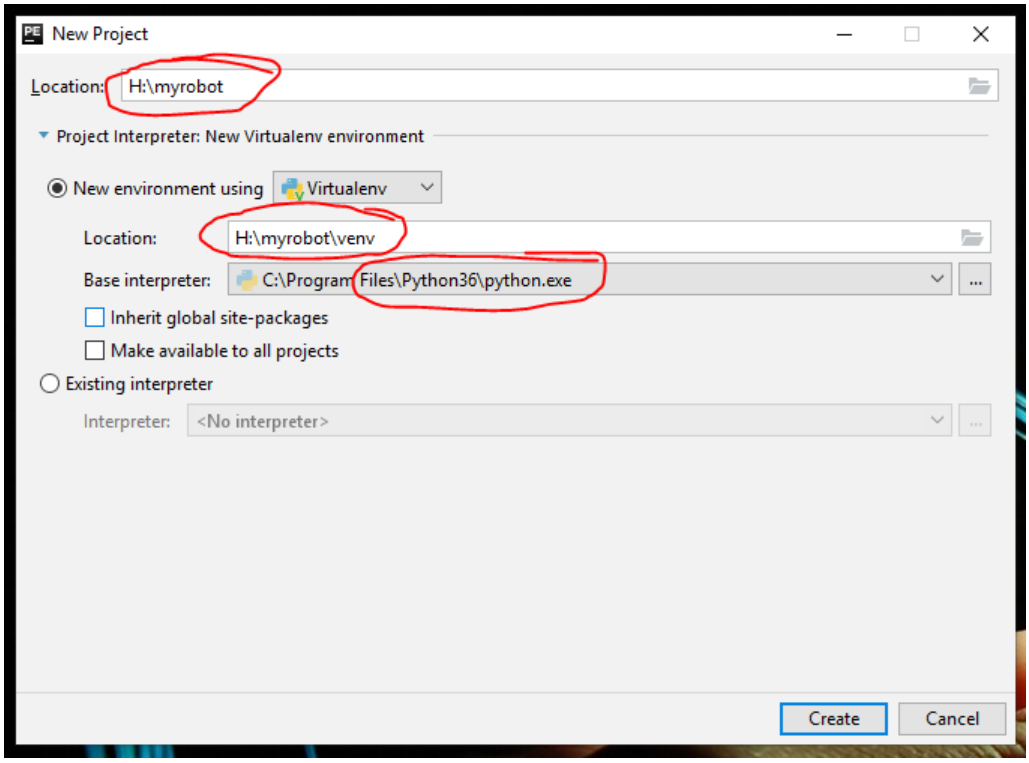

Fig. 5: Project configuration dialogue

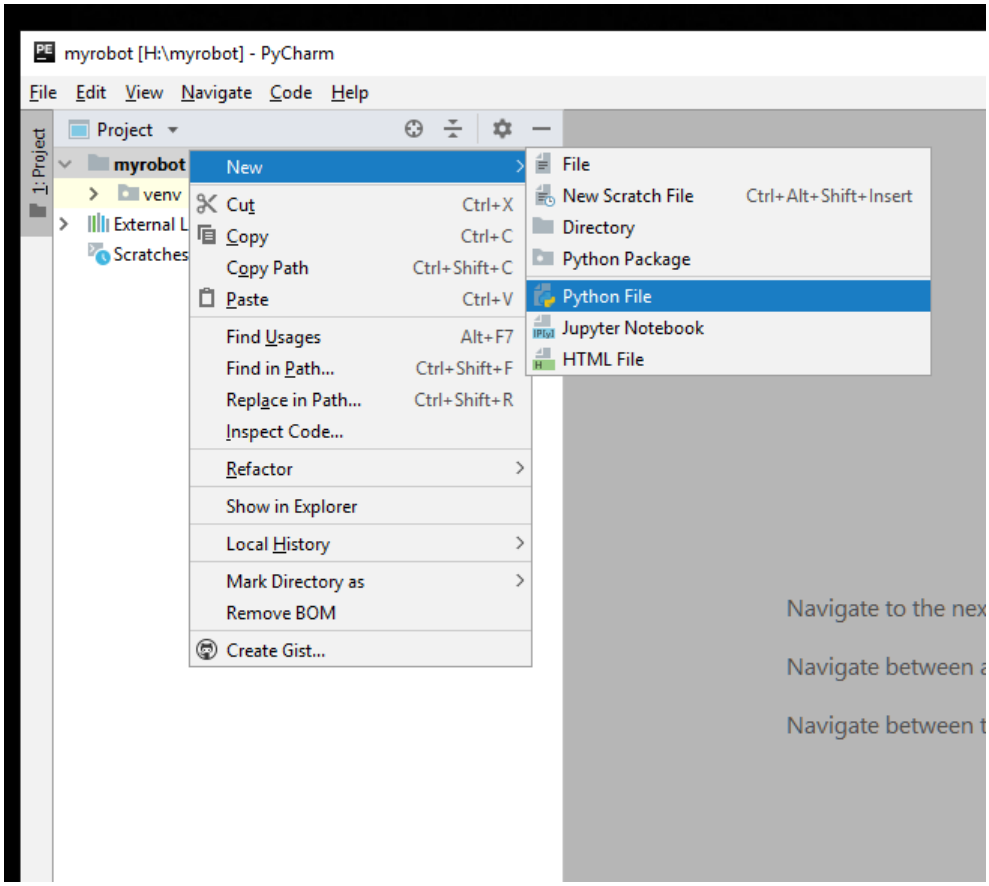

Fig. 6: Create a new Python file

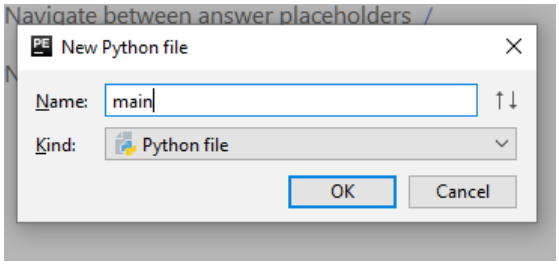

Fig. 7: Naming your file

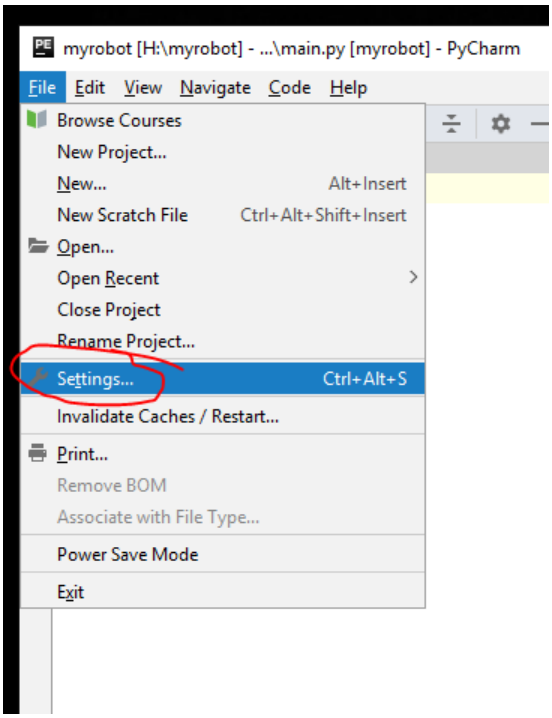

Fig. 8: Opening PyCharm's settings from the File menu.

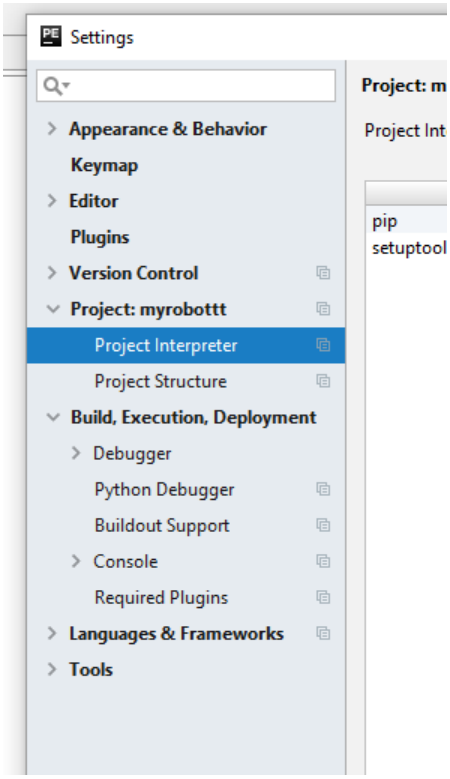

Fig. 9: The Project Interpreter tab in PyCharm's settings.

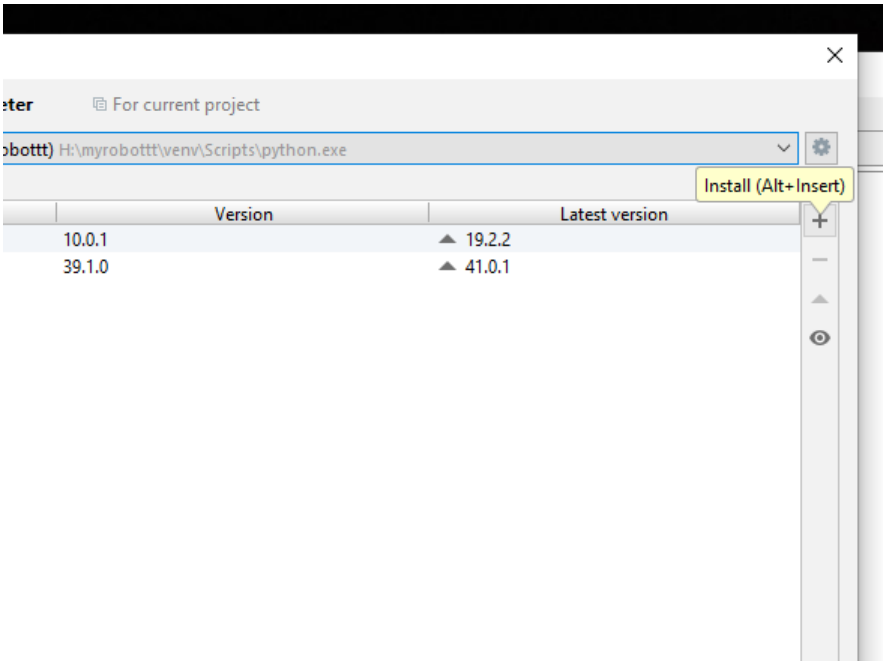

Fig. 10: This button opens the package installation window.

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!

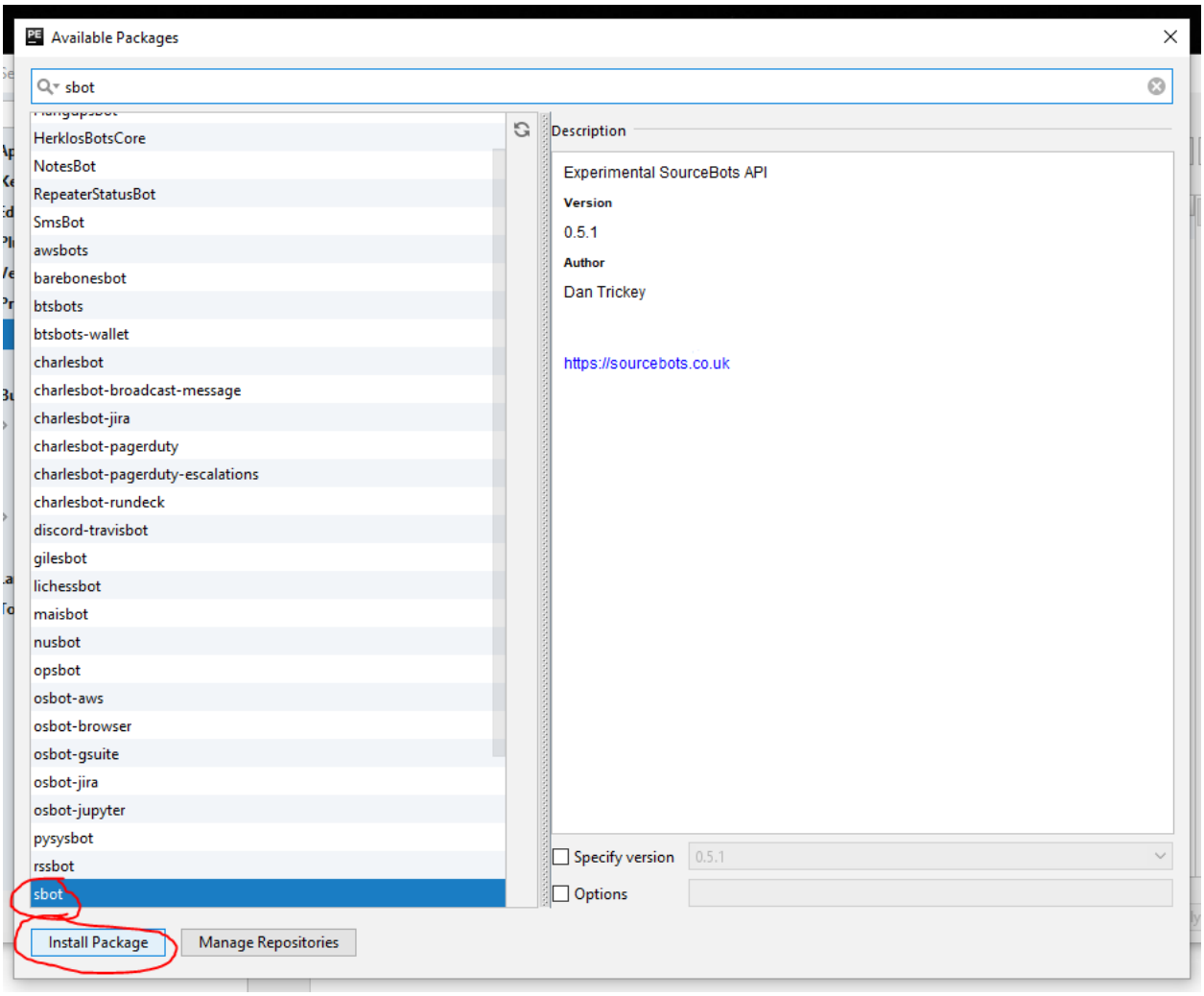

Fig. 11: Installing sbot

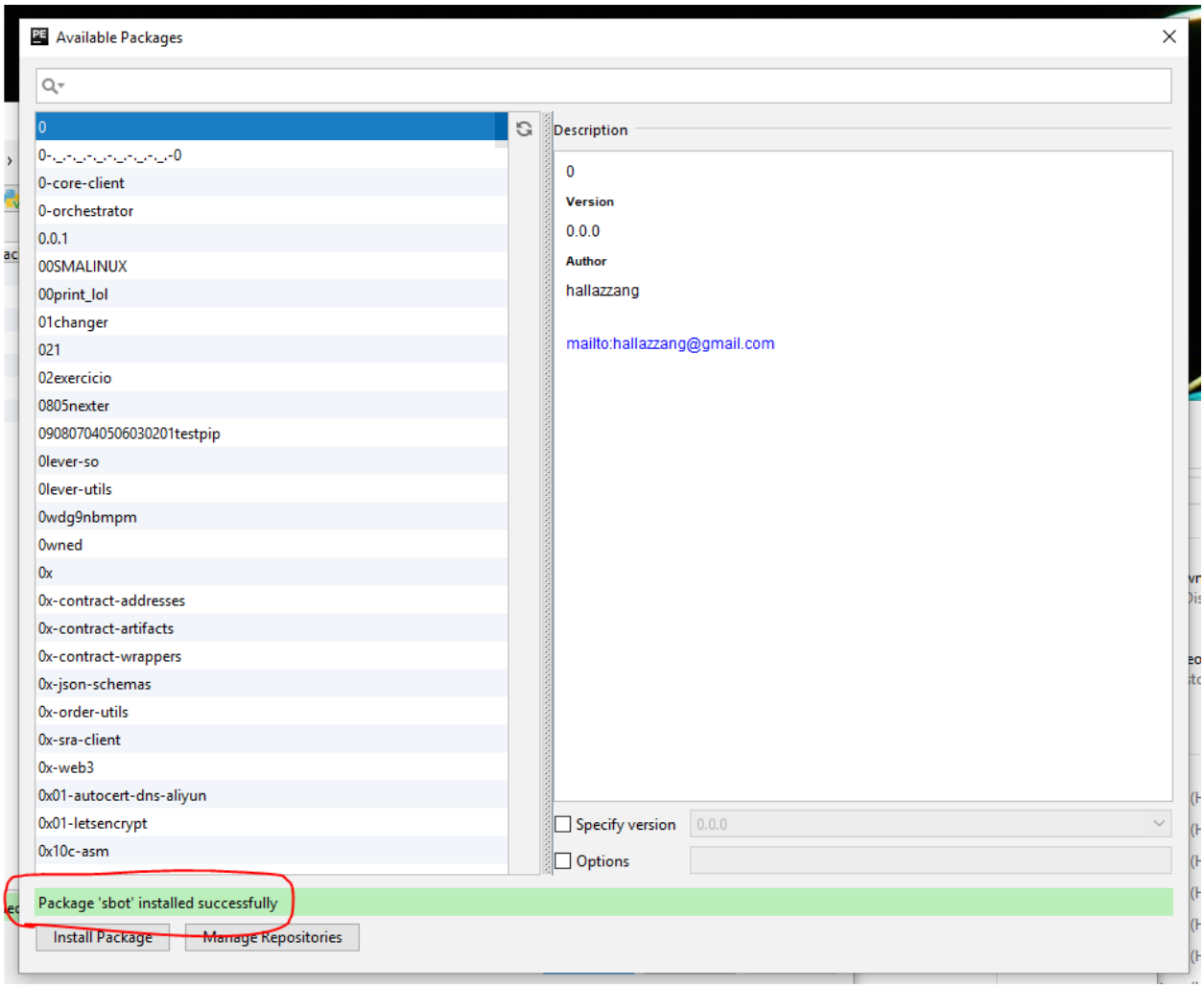

Fig. 12: Installation success

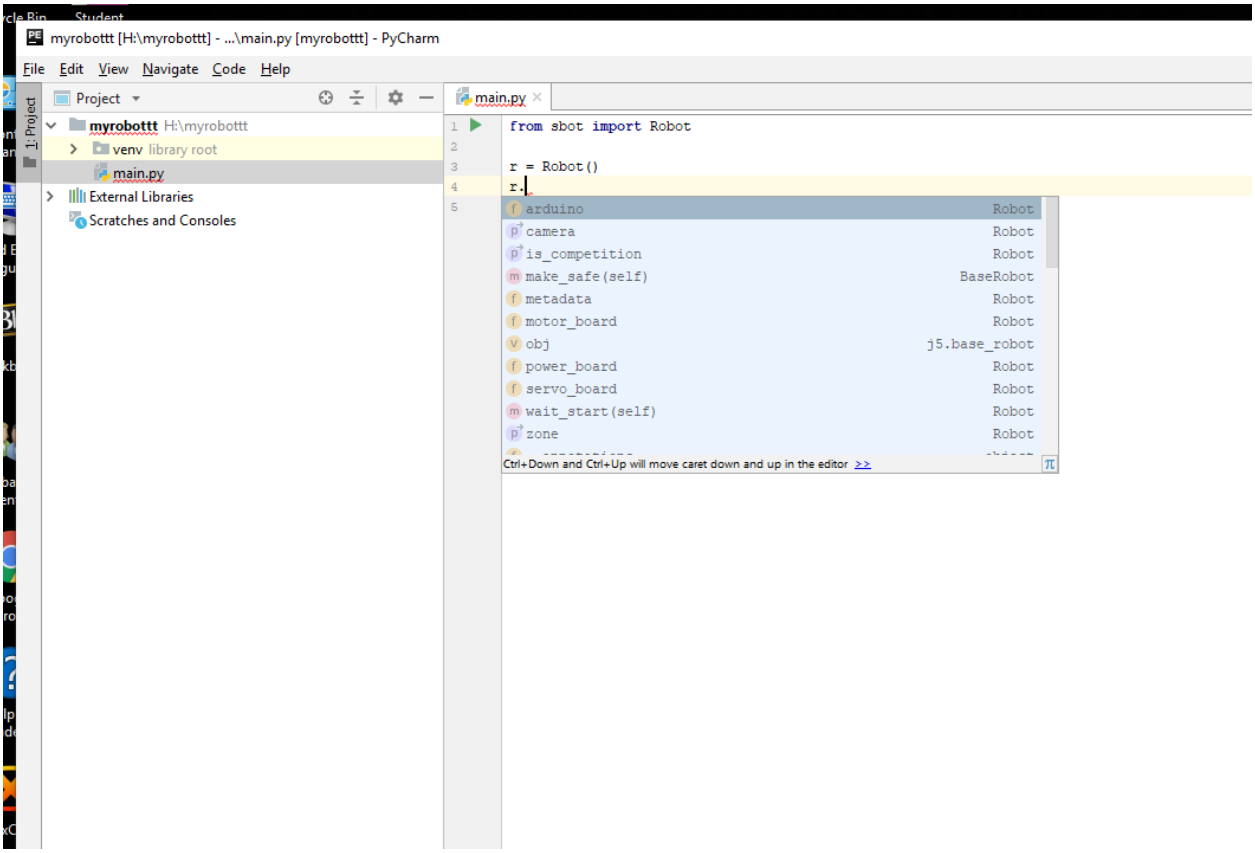

Fig. 13: Code auto-suggestions

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:

```
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:

```
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.

```
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.

```
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.

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.

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.

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

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 `input` function is like `input`, but returns a number (e.g. `42`) when you enter one, rather than a string (like `"42"`).) Replace the comment with code that averages the numbers and puts them in a variable called `average`.

```
a = input("Enter first number: ")
b = input("Enter second number: ")

# 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:

```
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:
    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")

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.