PYTHON PROGRAMMING

Key Stage 3

COMPUTING

STUDENT REFERENCE AND EXERCISES

PART II

Using Minecraft Pi and Codecadmy

Minecraft Pi Book

Craig Richardson

June 13, 2013

Contents

Chapter 1

Introduction

Technology is essential for our everyday lives. Understanding technology is not only necessary for a successful career, but it is also necessary for understanding the world we live in. Programming underlies all computer technologies and learning to program is a very powerful skill.

Put simply, by creating programs we can communicate instructions to a computer. It is creative. With programming you can create nearly anything you can think of. To create a program we use a programming language.

This book provides exercises and documentation in the Python programming language. All of the exercises use Minecraft Pi on the Raspberry Pi to provide concrete and fun ways to develop programming skills. Alongside this book it is recommended that students complete the Codecademy Python track. Codecademy is an excellent resource for supporting students as they learn programming with Python. Exercises follow on from Codecademy so that students can further develop their problem solving skills and apply their knowledge in an environment that is fun and challenging.

When learning to program, developing problem solving skills is just as important as remembering a programming language.

1.1 Using this Book

This book develops programming skills through exercises using Minecraft and Python on the Raspberry Pi. It assumes you are using this resource

alongside the Codecademy Python track and provides reference materials that supplement this.

Every exercise in this book is challenging and has a number of suggested extensions. You apply your knowledge of Python to create some really useful programs with Minecraft Pi. Along the way you'll develop problem solving skills that are essential for programming.

Trying out your own ideas is highly encouraged. Use the concepts you've learned to work out how to make your ideas a reality. If you want to do something, but can't work out how to do it yet, try finding it out for yourself by skipping ahead in the book or try searching for help online. Finding things out for yourself is a really important skill.

The reference parts of chapters provide supplementary material based on the Codecademy Python track. These materials are meant to be used when you need extra support to understand a concept or when you're developing a program and need a reference for a certain piece of code. If you're sitting exams you can also use the book to revise concepts and definitions.

Codecademy is an excellent, free, online site for learning to program. You work through programming exercises and learn at the same time. The content of this book matches the content of the Codecademy Python track. It is recommended that you work through each Codecademy lesson before attempting the corresponding exercises in this book. However, if you're feeling really adventurous and want a challenge, don't use Codecademy and try using the reference materials in this book or elsewhere as a starting point for learning how to do things.

1.2 Standards Used in this Book

The following standards are used in this book:

Example code looks like this:

```
1 items = 6
2 # add more items
3 items = items + 5
```

When lines of code are too long to fit the page, the ↪ indicates the code has been pushed onto a new line in order to fit. When copying code with

this symbol ignore the line break and write code as a single line.

Code Syntax Definitions

syntax type

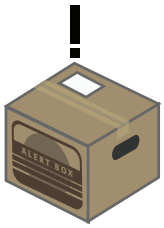
The syntax definition box explains a key piece of code in Python. It also includes a definition of the code in its simplest form and an example.

Expression:

```
1 12
```

Statement:

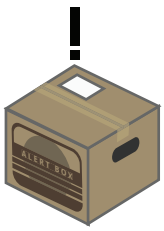
```
1 age = 12
```



ALERT: This is an alert. They usually contain important information, tips and warnings.

Chapter 2

Getting Started



ALERT: This guide assumes you know how to install the Raspbian operating system on your Raspberry Pi, you can login, open a desktop environment (using startx) and can use basic terminal commands. If you do not know how to do any of these things there are plenty of excellent guides online.

This first thing you'll need to do is install Minecraft Pi Edition on your Raspberry Pi. Instructions for this can be found here: `http://pi.minecraft.net/`

Now you need to open a terminal and move to the mcpi directory using `cd mcpi`. Then to run Minecraft Pi use the following command in the terminal `./minecraft-pi`.

Once you've created a new game in Minecraft Pi, familiarise yourself with the controls:

w Move the player forward

a Move the player left

s Move the player backwards

d Move the player right

mouse Change the player direction/camera angle

tab Release the mouse

right click Place block

left click Smash block

scroll wheel Change selected block

space Jump (fly higher when flying)

double tap space Fly (double tap again to stop flying)

shift Crouch (fly lower when flying)

e Block selection menu

1–8 Change block selection from inventory

esc Open the options menu (also releases mouse)

Play around in Minecraft. Build some things. See how different blocks, like water, react when you smash their neighbours.

To run Minecraft Pi you must be using a Desktop environment, you can't run it from the default command line interface that Raspbian boots to be default.

2.1 Programming with Python

Python is a programming language that is suitable for beginners. It is easy to read and write.

There are two versions of Python 2.7 and 3. We will be using 2.7 in this book. There are some minor differences between the two versions. You can find out more online or use a feature in Ninja-IDE (mentioned below) to identify compatibility issues in your code.

2.1.1 The Minecraft Pi API

The Minecraft Pi API allows programmers to interface their code with a Minecraft Pi game. In other words you can write a program that interacts with Minecraft Pi, without changing the game itself.

The API is stored within a subdirectory of the mcpi directory. You will need to copy it to the same directory as your Python code in order to use

it. Navigate to the directory you will save your Python code and run the following terminal command:

```
cp -r ~/mcpi/api/python/* .
```

Don't forget the full stop.

2.1.2 Writing and Running Code

There are a number of choices for writing your Python code with your Raspberry Pi. We'll go over your options. Whichever program you choose to write and run your code is down to preference. Try the different options out and see what you like best.

Whatever option you use to write and run your code, make sure you are in a Minecraft game world when you run it otherwise you will receive an error.

Terminal + nano

The terminal is a way to interact with your Raspberry Pi using text. It uses a command line interface instead of a graphical user interface. You might be used to the graphical environment of your desktop or file browser. A terminal can does the same things, but with text instead of mouse clicks.

To create and edit a python file using the terminal we use the nano text editor. In a terminal move to the directory that you want to create your Python program and run the following command

```
nano filename.py
```

Change the file name to whatever you want.

The Nano text editor uses the cursor to place text. You can't change the cursor position with the mouse, you can only change it with the direction keys. Furthermore you can't use ctrl-c and ctrl-v to copy and paste text.

To save your file you use ctrl-o, followed by the name you want to save your file as. To exit the nano text editor you use ctrl-x. You will be prompted to save your file when exiting.

The nano text editor can be difficult to use for people who are familiar with using graphical user interface instead of a command line interface.

After you have created your file you can run it with the following command in a terminal:

```
python filename.py
```

If you leave out the filename, Python will open as an interactive console, which allows you to write code on the fly instead of running it from a file. This is perfect for quickly testing a piece of code:

```
python
```

To exit the interactive console use ctrl+z.

IDLE

IDLE is the default integrated development environment of Python. An integrated development environment gives you a space to write code and provides other features like debugging and syntax highlighting. There are two versions of IDLE included with the Raspberry Pi, we will be using IDLE, not IDLE 3.

IDLE is a very basic IDE. By default it opens into an interactive console, which is perfect for testing code quickly without the need to save. To create a new file that is not an interactive console, click on file > new window or press ctrl + n.

Save the file using the file menu. Make sure you save the file in same directory as the mcpi api directory.

To run a program that you have written in the text-editor window you click on the run menu then run module or press the F5 key.

You can also your program from the terminal.

Geany

Geany is another IDE that is more robust than IDLE. It is used for programming in a number of programming languages. There are more features in Geany than IDLE and I suggest you explore what it can do.

Geany does not come pre-installed with Raspbian. To install it connect to the internet and run the following commands in a terminal:

```
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install geany
```

Geany will now be available under the programming section of the task bar menu.

Once you've saved your program you can run it from Geany's in-built terminal or from a regular terminal.

Ninja-IDE

Ninja-IDE is relatively less well-known IDE than the other options. It is an extremely well-made IDE that is specifically designed for Python. As a result it has a number of very useful features for Python developers, like compatibility highlighting between Python 2.7 and Python 3.

At the moment Ninja-IDE on the Raspberry Pi is a bit more complicated to install. You can find instructions for it on my website:
http://bit.ly/11BasQ0

There is a run button on the side bar, which will run your program.

You can also open directories, which be managed as a project. There are a number of useful plug-ins and colour customisations, which you can find via the menu.

Chapter 3

Python Syntax

Syntax is the basic set of rules that programming languages use. These rules are Python's equivalent of grammar and punctuation.

Python's syntax is necessary for structuring a program so that the computer can understand the instructions the programmer is giving it. Without syntax the computer would not understand what it was being told to do.

Understanding syntax is necessary as it underpins everything in programming. Without it you can't communicate with the computer.

This chapter covers the fundamentals of Python's syntax. At the start of this chapter there are several programming challenges using Minecraft Pi. Each challenge states the knowledge you require to complete the task. Before starting the exercises it is recommended you complete the first Python tutorial on Codecademy, unless you're feeling adventurous. If you can't quite remember how to use a certain concept we've included reference for everything you have covered over at Codecademy.

As with every chapter in this book you can the reference part of this chapter in a number of different ways:

- look at the references to support your understanding when attempting the exercises
- use it when you need extra help to understand a topic introduced in Codecademy
- refer to it to as a reminder when you're writing a program
- let it help you with revision for your exams.

The book was written to support you when you're learning to program, so use it for whatever support you need.

3.1 Minecraft Exercises

Let's practice variables with Minecraft on the Raspberry Pi. Each of these exercises introduces you to some code that uses variables. These exercises will show you how changes the player's position in Minecraft and also stop the player from destroying blocks. For each exercise we'll tell you the concepts you'll practice and its difficulty. We'll even explain the Minecraft bits of code to you as you go.

When using these exercises, identify where the variables are and how they're being used. This will help you understand what's going on. If you're unsure of something refer back to the explanations earlier in the chapter.

When you think you get it, try the extensions exercises. You can also be creative and try your own ideas by changing, rearranging and combining the code.

3.1.1 Teleport the Player

Skills and knowledge we'll practice in this exercise:

- Variables
- Integers
- Co-ordinates
- The Minecraft API
- Setting the player position

Let's get started with your first program using Python and the Minecraft API. We'll start with something simple, teleporting the player to a new location with integers.

Your character has a position in the Minecraft world. This is represented by three numbers that you can see in the top left corner of the game window. These numbers are known as x, y and z.

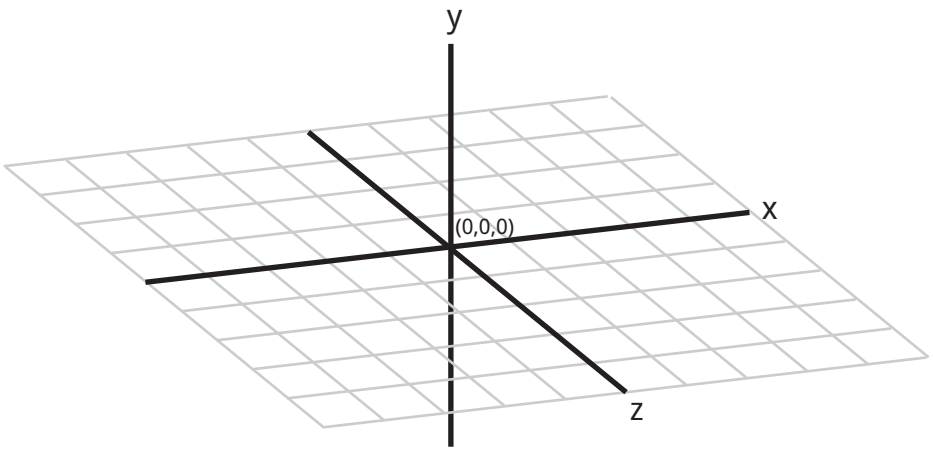


Figure 3.1: 3D Co-ordinates

When you move you'll notice that these numbers change. These variables are co-ordinates and represent your position in Minecraft's 3D world as shown in figure 3.1.1: y represents height and x and z represent your position on a flat plane.

As well as moving your character with the keyboard, you can change their position using Python. We'll take you through the code to do that.

Instructions

Create a new file and name it teleport.py. Move the Minecraft API folder to the same folder, steps to achieve this can be found on page [page number].

Now open the teleport.py file in a text editor. First of all we need two lines of code that connect to our program to Minecraft. You will use these two lines of code in all programs that interact with Minecraft. Add these two lines at the top of your program:

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
```

We'll now create the three integer variables that represent the x, y and z position that we want to teleport our character to. For now let's set our destination to co-ordinates (10, 11, 12).

```
4 x = 10
```

```
y = 11
z = 12
```

Finally we need a single line of code that will move the player.

```
mc.player.setTilePos(x,y,z)
```

This is a function, you'll learn more about functions later. What you need to know is that the `setTilePos(x,y,z)` bit at the end of the line tells Minecraft to change the player's position using the three variables that we just set.

Here's the full code. We've included some comments so that it's easy to understand:

```
#connect to Minecraft
import mcpi.minecraft as minecraft
mc = minecraft.Minecraft.create()

"""set x,y and z variables
to represent coordinates"""
x = 10
y = 11
z = 12

#change the player's position
mc.player.setTilePos(x,y,z)
```

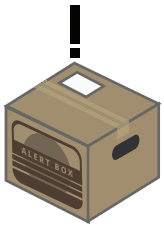
Now let's run the program. Do these steps:

1. Open Minecraft Pi (instructions on page [page number])
2. Open up a terminal and change the directory to where you saved your program (instructions on page [page number]).
3. Type python teleport.py in the terminal and press enter.

Well done! Your program should now run and your character will be teleported to co-ordinates (10,11,12).

Extensions

- Change the values of the x, y and z varaibles
- Use negative values for the x, y and z variables



ALERT: Don't use a value larger than 127 for the x and z variables or a value larger than [find value] for the y variable. The Minecraft Pi world is only small and numbers bigger than this will cause the game to crash [check this]

3.1.2 Teleport the Player Precisely

Skills and knowledge we'll practice in this exercise:

- Variables
- Floats
- Following instructions
- The Minecraft API
- Setting the player position in Minecraft with Floats

In this exercise we'll not give you the code to copy, instead we'll give you help to work out what you have to do. Don't worry, it's pretty similar to the last exercise, with some tiny differences.

In the last exercise you learned how to set the player's position using integers. You may have noticed that the location of the player in the top left of the window has a decimal place. For more precise movement across blocks, the location of the player is actually stored as a float in the game. In this exercise you'll set the player's position using Floats.

Instructions

Using the previous exercise as a guide follow these steps to teleport the player using a float value:

1. create a new file named teleportPrecise.py (make sure the Minecraft API folder is in the same directory)

2. Open the teleportPrecise.py file and add the two lines of code that connect our program to the Minecraft game
3. Define three variables named x, y and z and set their values to floats
4. Add the following line of code `mc.player.setPos(x,y,z)`
5. Open a Minecraft world and run the code

Notice there is a difference between `mc.player.setPos(x,y,z)` and `mc.player.setTilePos(x,y,z)`, which we used in the last exercise:

- `setPos(x,y,z)` uses floats to set the player's position
- `setTilePos(x,y,z)` uses integers to set the player's position

Extension

- Change the values of x, y and z variables. Use a mixture of positive and negative floats.
- See what happens when you only change the decimal values slightly.

3.1.3 Teleportation Tour

Skills and knowledge we'll practice in this exercise:

- Reusing Code
- Changing values
- Time
- Setting the player position

In this exercise we'll practice changing the values of variables. We're going to reuse some code from the first exercises to teleport the player to several locations across the map. The player will teleport to one location, wait a few seconds then teleport to another location.

For this we'll show you how to make Python wait for a few seconds.

Instructions

We'll be using the code from the first exercise. We've copied it here so that it's easier for you. You can use the code from the second exercise if you want and it'll work just the same.

```
1 #connect to Minecraft
2 import mcpi.minecraft as minecraft
3 mc = minecraft.Minecraft.create()
4
5 #set x, y and z variables
6 x = 10
7 y = 11
8 z = 12
9 #change the player's position
10 mc.player.setTilePos(x,y,z)
```

Let's get started. As usual do the following steps. You'll need these steps for every exercise so we're going to stop telling you to do them from now on.

1. Create a new file as usual, name it something simple that explains it's purpose, in this example use tour.py
2. Make sure the Minecraft API is in the same directory
3. Open the file in a text editor

Now let's edit the code:

1. Copy the code from teleport.py into tour.py
2. On line 4 add the following code: `import time`
3. On line 11 add the following code: `time.sleep(5)`
4. Copy lines 5 to 10 and paste them on line 12 onwards
5. Change the values of the x, y and z variables on lines 13, 14 and 15 respectively
6. Open a Minecraft world and run the code

You should see the player teleport to the first location, wait five seconds, then teleport to the second location. The `time.sleep(5)` line makes Python wait for five seconds.

Extensions

- Wait a different amount of time
- Copy the code to move the player as many times as you want
- Change only one variable: you don't have to change every variable every time

3.1.4 Stop Smashing Things

Skills and knowledge we'll practice in this exercise:

- Booleans
- Problem solving
- Immutable Blocks

Here's a nice simple task to finish things off. In Minecraft it's easy to smash blocks. This is useful when you want to smash things, but can be annoying if you've spent ages building something really cool. In this exercise we'll make it so that the player can't smash blocks.

Instructions

With `setting(world_immutable, True)` you can make blocks immutable. Immutable is another way of saying things cannot be changed. Here's the code you need to do this:

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
4 mc.setting(world_immutable, True)
```

After you've created this program and run it you'll notice that you can't smash blocks.

Your task now is to copy your program into a new file and change it to allow the player to smash blocks. Hint: there's a boolean in there.

Extensions

Try firing an arrow with your bow. What happens? Why do you think this is?

End of Exercises

3.2 Variables and Data Types

A variable stores a piece of data in the computer's memory.

There are different types of data that a variable can store including numbers and strings. These data types can store a wide variety of things, from names to GPS co-ordinates.

Variables are one of the most important concepts to learn as they are fundamental to programming.

Variable

concept

Variables store data in a computer's memory. This data can be a number of things such as numbers, letters and symbols. Every variable has a name and a value. The = symbol is used to set the value of a variable. When setting a variable, the variable name is always on the left of the = and the value is always on the right.

Expression:

```
1 variableName = value
```

Statement:

```
1 age = 23
2 height = 162.5
3 canDrive = True
```

When creating a variable you need three things: a variable name, an equals sign = and a value. In this example "speed" is the variable name and 30 is the value:

```
1 speed = 30
```

The variable name always goes on the left of the equals sign and the value on the right.

The name of the variable can be whatever you want, although it is better to name it something that explains its purpose. This makes it easier for the programmer to understand what is going on in the future.

Variables can hold data of various types. In this chapter we will cover three of these data types:

- Integers
- Floats
- Booleans

3.2.1 Integers

You have come across integers almost every day of your life. Integers are whole numbers. For example there might be 12 people in the street, you are going to meet 5 friends, or you've just bought 2 apples. The number of these things are all described with integers.

Integers can represent positive and negative numbers. Negative numbers are number less than 0. They have a - sign before the number.

Using integer in Python is easy. Say that we want 5 cats to take photos of. In Python we can declare an integer variable to represent this like so:

```
1 cats = 5
```

As with all variables, the variable name is written at the start of the line. We then put an equals sign to tell Python that we're assigning a value to a variable. Finally we write the integer value that we want to store in the variable.

The value side of the integer variable should not be written with any spaces, letters or symbols. Otherwise Python will get confused and won't under-

Integers

data type

A data type for whole numbers, which are either positive or negative. For example 10, 32, -6, 194689 and -5 are all integers. 3.14 and 6.025 are not integers as they have decimal places.

Expression:

```
1 12
```

Statement:

```
1 age = 12
```

stand that you want to create an integer. The - symbol is the one exception, which you must use if you want to create a negative value.

To say the temperature is negative 5 degrees would set a variable like so:

```
1 temperature = -5
```

3.2.2 Floats

Not all numbers are whole numbers. Decimal places are used to represent values that can't be described with whole numbers. For example you might have half (0.5) an apple, you hourly pay is £7.34 or the market is 2.6 miles away from your house.

Floating numbers are used instead of integers when more precise data is required.

Integers only makes sense where things are measured in whole numbers, like you have 2 cats. It wouldn't make sense to have two and a half cats.

Floats on the other hand represent numbers where it makes sense to have fractions of numbers. Like the temperature can be -6.3 degrees, 17.4 degrees, 18.9 degrees and so on.

Floats can represent whole numbers, but integers cannot represent numbers with decimal places.

Float

data type

Number values with decimal places, either positive or negative. For example 3.14, 6.025, 105896.7584926, -8.276 and 1.00 are all floats.

Expression:

```
1 17.5
```

Statement:

```
1 tax = 17.5
```

You can link this with Maths. Integers are used for discrete data and floats are used for continuous data.

In Python declaring a float variable is achieved in the same as declaring all other variable types, except you include a decimal point within the number value. For example to say we have 1.34 litres of water:

```
1 litresOfWater = 1.34
```

To create a negative float you precede the number with a - symbol:

```
1 temperature = -4.37
```

3.2.3 Booleans

Booleans are an interesting data type. There are only two possible values for a boolean, True or False.

Think of a basic light switch, it can be either on or off. This is how a boolean works, it is either True (on) or False (off).

In Python we can declare a boolean variable like this to represent that the light is on:

```
1 light = True
```

When the light is off we would write this:

```
1 light = False
```

Boolean

data type

Data that is either True or False. Also represented as 1 or 0, On or Off.

Expression:

```
1 True
2 False
```

Statement:

```
1 canFly = True
2 isBird = False
```

The first letter of True and False values must always be capitalised.

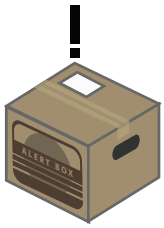
Booleans have a number of uses. They are particularly useful for representing answers to questions with either True or False. For example, is someone asleep? Is the dog hungry? Is it raining?

3.2.4 Changing Values of Variables

You can change the value of a variable at any time. You do this in the same way as declaring a variable, with an equals sign. For example, say we declared a variable "cats" as 5 and we then wanted to change it to 10 as we bought more cats. It would look like this in Python:

```
1 cats = 5
2 cats = 10
```

The cats variable started out as 5, but then it is changed to 10. When it is changed the cats variable forgets the old value and remembers the new one. Pretty simple.



ALERT: Python reads programs in line order, starting at the top and ending at the bottom. In other words it will execute code on the first line, then the second line and so on. Keep this in mind when changing the value of a variable.

3.3 Whitespace and Statements

Python needs to know when to stop reading one instruction in your code and when to start reading the next one. This is where statements and white space come in.

Think of sentences in English. If I didn't include any full stops in my sentences you would find it very difficult to understand what I was trying to tell you. Take following text:

> There are no dragons in the pub there is a witch

That is quite difficult to understand. However with the correct punctuation:

> There are no dragons. In the pub there is a witch.

See what I mean? By using full stops my sentence becomes easier for you to understand. It makes it easier to communicate. And that is what syntax in programming is about, clearly communicating instructions to the computer.

3.3.1 Statements and Line Breaks

Think of a single instruction in your code as a sentence. To end a sentence in English you use a full-stop. Instead of a full stop, Python uses a new line to indicate the end of an instruction.

Each instruction on a new line in Python is called a statement. Python requires a new line between each statement so that it can see where one

statement ends and another begins.

For example if we wanted to create a three variables in Python, we would write it like this:

```
1 socks = 12
2 human = True
3 age = 25
```

Notice how each statement is on a new line. This means Python can understand that you want to create three variables.

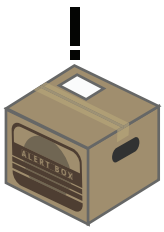
If you don't put each statement on a new line Python will get confused:

```
1 socks = 12 human = True age = 25
```

This code will confuse Python and it will not be able to follow your instructions. It doesn't know where one statement starts and another begins. When Python is confused it won't do what you want it to do. It will tell you it is confused with an error message.

3.3.2 Indentation

Sometimes it is necessary to indicate that some code belongs in a group. This is important for reusing code, trying different options and repeating the same instructions. Python uses indentation and spaces for these things.



ALERT: Indentation is important in Python. We won't cover this topic in depth until later. For the moment it is important to remember not to use the tab key or put in lots of spaces at the start of a line as this will confuse Python, cause errors, or make your program do something unexpected.

3.4 Comments

Comments are statements in your code that the Python interpreter ignores. Statements that are comments don't do anything. They are useful though.

Comments can be used to write notes in your code so that you can read what pieces of code do in the future.

They can also be used to stop the interpreter reading statements in your code. This is useful when you're testing or trying to find bugs and errors.

In Python there are two ways to write comments: on a single line or on multiple lines.

.....

Single Line Comments

comment

Comments tell the Python interpreter not to read statements with comments. Single line comments only block a single line from the Python interpreter. The # symbol indicates the start of a comment.

Expression:

```
1 #Commented text
```

Statement:

```
1 #Sets characteristics of a person
2 name = "Helen" #person's name
```

.....

3.4.1 Single Line Comments

Let's look at different ways to use the single line comment:

```
1 #We have 5 cats
2 cats = 5
```

In this code the Python interpreter would ignore the first line, but would read the second line.

If you wanted the interpreter to ignore the second statement as well you would put a hash sign at the start of the line like this:

```
1 #We have 5 cats
2 #cats = 5
```

In this way Python would not create the variable cats and not set it to 5. The comment makes it ignore that statement.

You don't have to put a comment at the start of a line. You can put it at the end, as long as there is a complete statement on that line. For example:

```
1 cats = 5 #We have 5 cats
```

This would work and create a cats variable with a value of 5.

However if we were to move the comment symbol earlier in the line (e.g. before the 5) we would get an error. The error would occur because the statement is not complete:

```
1 cats = #5 We have 5 cats
```

Python would get confused as it expects you to include a value, but there isn't one as it ignores everything after the start of the comment. Avoid this.

3.4.2 Multi-line Comments

Sometimes you want Python to ignore several lines of code. You might have a long explanation of what the code does or you simply want to block it from reading large sections of code while you're debugging. Writing the comment symbol at the start of each line is tedious and time consuming for a large number of lines. We therefore use multi-line comments to achieve this.

Multi-line comments in Python block out several lines of code, which the interpreter will ignore.

The same principles apply to multi-line comments as single line comments, but the syntax is different. Here is an example:

```
1 """This program takes picture of ducks.
2 No code has been written yet,
3 but we're hopeful it will work"""
```

Multi Line Comments

comment

Comments tell the Python interpreter not to read statements with comments. Multi line comments block out several lines of code.

Expression:

```
1 """Commented
2 statements
3 go here"""
```

Statement:

```
1 """We have five cats
2 Jonesie likes swimming
3 The other four don't"""
```

Multi-line comments in Python start with three double quotation marks on the first line that you want to block. It finishes with three more double quotation marks at the end of the last line you want to block out.

Chapter 4

Maths Operations

Remember the last chapter when you learned how to declare variables? It would be pretty useful to know how to do something with them wouldn't it?

That's what maths operations are for: doing things with variables. Specifically integer and float variables.

From maths you will know about addition, subtraction, multiplication and division. Each of these is known as an operation. Python can do all of these operations and more.

Operations are extremely useful and understanding them will make programming a lot easier.

Don't worry if you're not great at Maths. As long as you understand the very basics, like addition and subtraction, you'll be fine.

If you'd like some extra support with Maths, Khanacademy.com has some excellent tutorials that are easy to understand and allow you to progress at your own pace.

4.1 Minecraft Exercises

Let's get more comfortable with maths operators. In this set of exercises you'll build upon the knowledge of this chapter and the previous chapter. We'll introduce you to creating blocks in Minecraft with Python. This makes it easy to build very complex structures in Minecraft really quickly. Understanding maths operators compliments this well.

As with the previous set of exercises we'll start of with a lot of support in the first exercise and gradually allow you to work things out on your own using the knowledge you're developing.

4.1.1 Stacking Blocks

Skills and knowledge we'll practice in this exercise:

- Integers
- Addition
- Creating a block

Let me introduce you to setBlock(). We use this to create a block in Minecraft. Like setPos() and setTilePos(), setBlock() uses x, y and z values for co-ordinates. It also uses a fourth value, block type. As you'd expect this value states which block type you want to place in the game, for example grass, lava or melon.

Each type of block is represented with an integer. For example grass' value is 2, air's value is 0, water's value is 8 and melon's value is 103. For a full list of blocks and their integer values see page [page number].

To use setBlock(), inside the brackets we provide the values of the x, y and z coordinates that we want to place the block and block type we want the block to be. Each of these should be seperated by commas. For example we can place a melon block at co-ordinates (6, 5, 28) with the following code:

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
4 mc.setBlock(6, 5, 28, 103)
```

Simple. Of course you can substitute the values in the brackets with variables to get the same effect, like so:

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
```

```
4 x = 6
5 y = 5
6 z = 28
7 blockType = 103
8 mc.setBlock(x, y, z, blockType)
```

In this code we set variables to represent the co-ordinates that we'll place the block at. We also set a variable for the block type. We then provide these variables to setBlock() and Minecraft works its magic.

When you combine this code with maths operators you can do some pretty cool things. Let's start with something simple, creating a stack of blocks.

Instructions

Here's some simple code to create a stack of two blocks in Minecraft:

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
4 x = 6
5 y = 5
6 z = 28
7 blockType = 103
8 mc.setBlock(x, y, z, blockType)
9
10 y = y + 1
11 mc.setBlock(x, y, z, blockType)
```

You should be familiar with this code from above. The difference is on line 10 where we add 1 to the value of y. We then use the same line of code on line 11 as we did on line 8 to create a new block. As the value of y has increased by 1, the second block is higher on the y axis than the first block.

You task is to add another two blocks on top of the current two. So when you run your program there should be a stack of 4 blocks.

Extensions

- Change the Block Type

- Change the increments between blocks
- Change directions to make a square
- Use your imagination and make something cool appear

4.1.2 Super Jump

Skills and knowledge we'll practice in this exercise:

- Addition operator

During the exercises in the last chapter we learned how to change the player's location. In this exercises we'll take this one step further. First we'll find out where the player is and then move them a set number of blocks using operators.

To find the player's location we use getTilePos(). For example we can access the player's location and then set it into our x, y and z variables with the following code:

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
4 position = mc.player.getTilePos()
5 x = position.x
6 y = position.y
7 z = position.z
```

If we wanted to then teleport the player -5 blocks along the x-axis we'd add this line:

```
8 x = x - 5
9 mc.player.setTilePos(x, y, z)
```

Intructions

Your task is to make the player jump ten blocks into the air directly above their current position. To do this you can use the code above and change it slightly.

Extensions

- Add a block below the player after they jump into the air
- Add more steps to the code and change it so that it looks like a stack of blocks is lifting the player into the air

4.1.3 Set Block Below Player

Skills and knowledge we'll practice in this exercise:

- Subtraction
- Shorthand operators

It's time for you to work things out for yourself. In this exercise you'll change the block directly below the player. To achieve this you'll need to use getTilePos() and setBlock().

Instructions

Using the comments below as a guide, write code to set the block directly below the player. You've learned how to get the player's position and set blocks in the previous exercises. Adapt and combine the two.

Try using shorthands operators for this exercise. You can find more about shorthand operators on page [pagenumber].

The comments below outline the structure of the code:

```
1 # connect to Minecraft
2 # get the player's position
3 # define x,y,z variables to store the player's
    ↪ position
4 # define the block type
5 # set y to one block below the player's postion
6 # set a block
```

Extensions

- Build a structure around the player

4.1.4 Speed Building

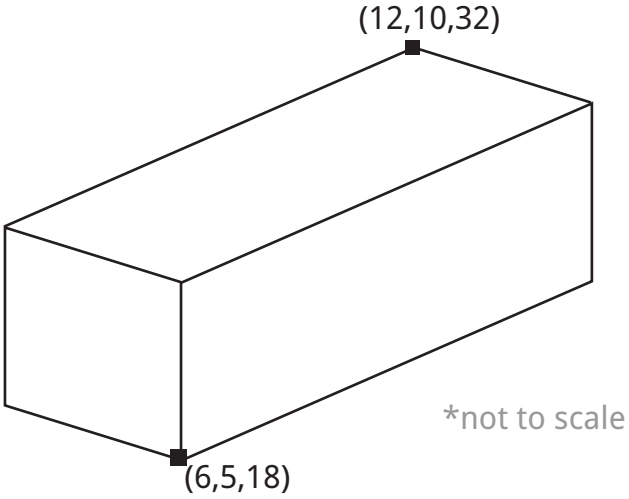
Skills and knowledge we'll practice in this exercise:

- Addition
- Subtraction
- Operator Order

We use setBlock() to create a single block. setBlock() has a friend, setBlocks() who creates several blocks in the shape of a cuboid.

setBlocks() is useful for creating a lot of blocks in a large area, where creating each block one at a time would be too time consuming and tedious.

To use setBlocks() we two sets of co-ordinates and the block type. The first set of co-ordinates states where we want one corner of the cuboid and the second set states where we want the opposite corner. Here's a diagram of where the co-ordinates tell the cuboid to go:



To create the above cuboid we would use the following code:

```
1  import mcpi.minecraft as minecraft
2  mc = minecraft.Minecraft.create()
3
4  x1 = 6
5  y1 = 5
6  z1 = 22
7  x2 = 12
8  y2 = 10
9  z2 = 32
10 blockType = 4
11 mc.setBlocks(x1, y1, z1, x2, y2, z2, blockType)
```

The width, height and length of this cuboid are 6, 5 and 10 respectively.

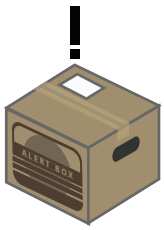
Instructions

Your task is to change the program to create a hollow cuboid at the player's position. In other words you need to create a cuboid made of blocks and then inside that cuboid create another cuboid that is made of air. Kind of like an empty box.

The above code needs to be adapted to include the following functionality:

- Get the player's position

- Set one corner of the cuboid to the player's position
- Set the opposite corner relative to the player's position
- Hollow out the cuboid



ALERT: Build and test this exercise in stages. First get the cuboid to appear next to the player. Instead of x2, y2 and z2 use width, height and length variables to set the size of the cuboid relateive to the player's position. Finally use addition and subtraction to calculate the co-ordinates of the inner cuboid made of air.

Extensions

This code is really useful for making buildings. See if you can add extra bits to the code to achieve the following:

- Add an extra level
- Change the floor to wood
- Add an entrance with a door
- Create two rooms divided by a wall

4.1.5 Proportions

Skills and knowledge we'll practice in this exercise:

- Multiplication
- Division

To finish off this set of exercises we'll end with something simple. We'll reuse code from the last exercise to create a building where its width,

height and length are proportional to one another.

Instructions

Copy and adapt the code in the last exercise so that:

- The length of the building is twice its width
- The height of the building is half its width

Extensions

End of Exercises

4.2 Operators, Expressions and Statements

When writing code it is important to know the difference between operators, expressions and statements.

Operators are bits of code that do something to variables. For example the addition operator adds two variables.

Expressions are small pieces of code that can be used in a variety of places, but cannot do anything unless part of a statement. Expressions can contain operators. For example 2 + 2 is an expression.

A statement is a single line or block of code that does something in your program. It can contain expressions, variables and operators. For example `people = 2 + 2`

When a new Python concept is introduced we will write the syntax as an expression so that you can understand it's most basic form. We'll also demonstrate its usage with a statement so that you can see an example of how it can be used.

4.2.1 Addition

I'm pretty sure you can add two numbers. Python can add numbers too.

You write an addition expression in Python like you would in Maths, with the + operator. Using the addition operator is similar to declaring a variable.

Addition +

operator

The addition operator adds two values. The + operator is used between values for addition.

Expression:

```
1 number1 + number2
```

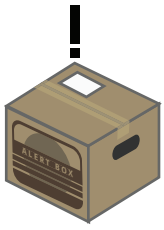
Statement:

```
1 pizzas = 2 + 1
```

For example, we have two shoes and we buy another pair:

```
1 shoes = 2 + 2
```

Can you guess what the value of the shoes variable is in the statement example? If you said 4 then you'd be correct. Python works out what is on the right hand side of the equals sign and then assigns it to the variable. In this case it's 4.



ALERT: When using addition — or any other operators — in your programs remember to write entire statements and not just an expression.

4.2.2 Subtraction

Subtraction is very similar to addition. Instead of a plus sign, you just put a minus sign.

. . . . . .

Subtraction -

operator

Subtracts one value values from another. The - operator is used between two number values for subtraction.

Expression:

```
1 number1 - number2
```

Statement:

```
1 pizzas = 2 - 1
```

. . . . . .

We're out for a run in a field and a cows steal one of our shoes:

```
1 shoes = 4 - 1
```

The value of shoes in the statement is now 3. As with addition, Python works out the operation on the right of the equals sign and sets the variable to that value. This is the same with all operations when using them to set variables.

4.2.3 Multiplication

Multiplication is slightly different than what you're used to in Maths. Instead of an x to multiply two numbers we use a *.

Other than the symbol, multiplication works exactly the same in maths and Python. 2 * 2 still equals 4.

The number of cars parked outside our house has just doubled. We can represent this in Python like this:

```
1 cars = 4 * 2
```

The value of cars in this example is 8. The code multiplied 4 by 2.

Multiplication *

operator

The multiplication operator multiplies two values. The * operator is used between two number values.

Expression:

```
1 number1 * number2
```

Statement:

```
1 seats = 6 * 9 # answer of 54
```

4.2.4 Division

Instead of the ÷ symbol, the symbol for division in Python is a `/`.

As with division in Maths, you put the number that you want to divide on the left of the `/` and the number you want to divide by on the right.

Division /

operator

Divides one value by another. The `/` symbol is used between two number values. The number that you want to divide on the left of the `/` and the number you want to divide by on the right.

Expression:

```
1 number1 / number2
```

Statement:

```
1 hair = 4 / 2 # answer of 2
```

Half of the cars on the street have driven away. There were 8 cars. Here's how we represent this with a division operator in Python:

```
1 cars = 8 / 2
```

The value of cars is 4. We divided 8 by 2.

4.2.5 Exponentials

Exponentials are numbers that are multiplied by themselves a number of times.

You might be familiar with writing exponentials as 2^2 (two to the power of two) which is a short way of saying, two times by itself two times (2 * 2). Another example is 2^4 (two to the power of four), which is two times by itself four times (2 * 2 * 2 * 2).

Exponentials are very important in computing. All computers work using a system called binary, which is made up of powers of 2. You will learn more about binary later.

In Python you use ** as the exponential operator. The number you want multiply goes on the left of the operator (the base) and the number of times you want to multiply it by itself (the exponential) goes on the right.

. . . . . .

Exponential **

operator

Raises one number to the power of another. The ** operator is used for exponentials. The number you want multiply goes on the left of the operator (the base) and the number of times you want to multiply it by itself (the exponential) goes on the right.

Expression:

```
1 number ** toThePowerOf
```

Statement:

```
1 e.g. cube = 2 ** 3
2 # 2 to the power of 3, which equals 8
3
4 square = 4 ** 2
5 # 4 to the power of 2, which equals 16
```

. . . . . .

We need to set out four sets of four rows of four chairs. This is 4 * 4 * 4 or 4^3. Here's the code to work out how many chairs we have:

```
1 chairs = 4 ** 3
```

Our answer should be 64. 4 *4 is 16. 16 * 4 is 64. So we need 64 chairs.

4.2.6 Modulo

The modulo operator gives you the remainder of a division of two integers. For example 7 divides by 3 two times (2 * 3 = 6) with 1 left over (7 - 6 = 1). Therefore the modulo of 7 / 3 is 1.

If a number divides perfectly by the other number, the result will be 0. For example 4 / 2 = 2 remainder 0, therefore the modulo is 0.

When you first learned to divide numbers you may have said things like 6 divided by 4 is eqaul to 1 remainder 2. This is the same concept.

.....

Modulo %

operator

Calculates the remainder of one number when it is divided by another. The % symbol is used as the modulo operator in Python. The number you want to divide goes on the left, and the number you want to divide by goes on the right.

Expression:

```
1 number1 % number2
```

Statement:

```
1 barrels = 10 % 2 # result is 0
2 pigeons = 13 % 5 # result is 3
```

.....

We bought a chocolate bar with 7 pieces and we must share it evenly between 3 people. Any pieces we can't share will be given to your cousin. Here's the code to work out how much your cousin gets:

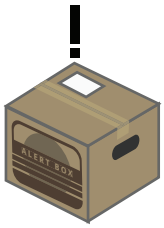
```
1 pieces = 7 % 3
```

The values of pieces should be 3. 7 divides by 3 twice with one left over (3 * 2 = 6, 7 - 6 = 1). So the answer is 1.

4.3 Operator Order

Several maths operators can be used together. For example we could multiply 5 by 2 and then take away 1, all in the same statement.

```
1 turtles = 5 * 2 - 1 #value of 9
```



ALERT: When using combining maths operators together Python will evaluate them from left to right. This order is important to know otherwise you may have unexpected results.
The only maths operator that is not evaluated in this order is the exponential operator. It will always be evaluated before any other operator.

To change the order of operators we can use brackets (). Any expressions with operators in brackets will be evaluate before anything else.

For example:

```
1 eggs = 6 * 3 - 2
```

This would normally have a value of 16. However, with brackets...

```
1 eggs = 6 * (3 - 2)
```

...it now has the value 6.

4.4 Interchanging Variables and Values

Wherever you can put a value, you can also put a variable.

For example each house has five cats and there are four houses. You could work out the total number of cats like this:

```
1 totalCats = 5 * 4
```

However, what if the number of cats per house changed quite often? You might use the value 5 many times within you code, making it very repetitive and difficult to change it every time it needs updating.

There is a simpler way. You can replace one or both of the values with variables.

```
1 catsPerHouse = 5
2 totalCats =  catsPerHouse * 4
```

You could also use a variable for the number of houses, like this:

```
1 catsPerHouse = 5
2 houses = 4
3 totalCats = catsPerHouse * houses
```

You can even say that one variable is equal to the value of another variable. Like this:

```
1 cats = 5
2 oldCats = cats
```

This will make both variables hold the same value, i.e. 5. You will find this useful when you need to change the value of one variable, but also store the old value somewhere else. When changing the value of one variable it will not affect the other:

```
1 cats = 5
2 oldCats = cats #sets oldCats to 5
3 cats = 6
4 #oldCats is still 5
```

4.5 Shorthand Operators

Quite often you will want to use an operator on a variable and then store the result in the same variable. For example we might want to add 5 to a variable:

```
1 horses = 6
2 horses = horses + 5 #The value of horses is now 11.
```

As programmers like to achieve things as quickly as possible with little repetition, there is shortcut to achieve this.

Shorthand operators in Python use a maths operator on a variable and reassign the result into the same variable.

There are four shorthand operators:

Shorthand Operators

operator

Shorthand operators use a maths operator on a variable and reassign the result into the same variable. There are four shorthand maths operators: addition (+=), subtraction (-=), multiplication (*=) and division (/=). The variable goes on the left of the statement, the operator in the middle and the value to operate with on the right. [correct code examples]

Expression:

```
variable += value #addition
variable -= value #subtraction
variable *= value #multiplication
variable /= value #division
```

Statement:

```
shoes = 5
shoes += 1 #value of 6
shoes -= 2 #value of 4
shoes *= 2 #value of 8
shoes /= 2 #value of 4
```

- addition (+=)
- subtraction (-=)
- multiplication (*=)
- division (/=)

For example, we can rewrite the above example with the addition shorthand operator:

```
horses = 6
horses +=  5 #The value of horses will now equal
    ↪ 11.
```

Chapter 5

Strings and Console Output

We have already seen three types of variables: integers, floats and booleans. There is also another variable type, strings.

Integers and floats store numerical data and booleans store True and False conditions. We also need to store letters and symbols. This is where strings come in.

Strings are a data type to store characters, symbols and numbers. In this chapter we will show you how to use strings.

Displaying data to the user is an essential part of programming. Until now you haven't been able to tell Python to output data to the user. We'll also learn how to output data to the console in this chapter.

5.1 Minecraft Exercises

In this chapter we covered strings and the console. You've learned how to use the string data type, print things to the console and take input from the console. All of these things can be combined with the Minecraft API.

During this chapter you were also introduced to functions. If you are eagle eyed enough you might have noticed that you've seen functions before. [Minecraft Pi API functions that have been introduced] are all examples of functions. They're all reusable blocks of code that make it easier for you to complete tasks. Pretty cool, huh?

In this set of exercises we'll build on the topics introduced in this chapter and the previous ones. You'll be introduced to printing message to the Minecraft chat using strings and will practice inputting data on the console to create things in the Minecraft world.

5.1.1 Hello Minecraft World

Skills and knowledge we'll practice in this exercise:

- Strings
- Output
- Minecraft Chat

Minecraft Pi edition has a chat window. At the moment the only way to use this chat window is via the API. It's pretty simple to do this. We just need to use the postToChat() function. The postToChat() function takes a string as an argument and posts it to the Minecraft chat window. The following code will post "Hello Minecraft World":

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
4 mc.postToChat("Hello Minecraft World")
```

Instructions

You task is to change the message in the chat to anything you want.

If you can connect to another Raspberry Pi with Minecraft try sending messages via the chat with another person (instructions on connecting to another game of Minecraft Pi or PE on page [page]).

Not the easiest way to chat is it? We'll improve the program in the next exercise.

Extensions

Try out these extension tasks when you're happy with the chat:

- Print block type below player
- Print the player's co-ordinates

5.1.2 Inputting Your Message

Skills and knowledge we'll practice in this exercise:

- Strings
- Input
- Output
- Minecraft Chat

Let's make using the chat easier. Instead of setting the chat message in the program we can use the console to input a message into the program.

Instructions

Follow these steps to create a chat program that uses the console for input:

1. Create a variable and set it to the value of:
   `raw_input("Enter your message: ")`
2. Use the mc.PostToChat() function and use your input variable as an argument
3. Run your program and you should see a prompt in your terminal saying "Enter your message: "
4. Enter your message in the terminal and press enter
5. You should see your message appear in the chat window

Don't forget the first two lines that connect your program to Minecraft!

See how much easier it is to chat using input than hard coding a string value into your program? Good. We're learned something here.

Extensions

- Concatenate some strings and user input together and post it chat
- Call the `raw_input` function several times to collect different pieces of information from the user and post it to the chat

5.1.3 User Name

Skills and knowledge we'll practice in this exercise:

- Strings
- Concatenation
- Upper
- Input
- Output
- Minecraft Chat

When you're playing together with more than two people it can be confusing who is writing a message in Minecraft's chat. There is an obvious solution to this, include the user's name at the start of a message. In this exercise you'll modify the previous exercise to include a username for all messages sent to the chat.

Instructions

This one is a bit more challenging. Using the program you wrote in the last exercise add the following functionality:

1. Take in the user's name as input before taking in their message
2. When posting the message to chat, capitalise the user's name put it before the message

The message posted to chat should be in the following format: "DAVE: I need diamonds."

Extensions

- What happens if you leave your name blank in the input? Why do you think this is?
- Change the username to lowercase characters.

5.1.4 Mad Libs

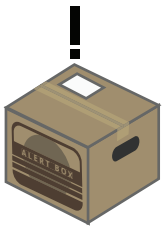
Skills and knowledge we'll practice in this exercise:

- Strings
- Input
- Output
- Placeholders

Mad Libs are well known songs and phrases with certain words swapped out for stupid words. For example Mad Libbing the song Jingle Bells we could swap out the words bells and sleigh for the words chicken and alphabet. Try Googling Mad Libs for some really funny examples. We'll make our own version of Mad Libs and post the result to Minecraft's chat.

Instructions

Use the %s placeholder to swap out words in Jingle Bells or another song and get the user to input random words to replace them. Post the result to chat.



ALERT: The `raw_input()` function can be used as many times as you want in a program. Use this to input the words you want to use for your Mad Lib.

5.1.5 Create a Block with Input

Skills and knowledge we'll practice in this exercise:

- Integers
- Input
- Blocks
- Co-ordinates

The `input()` function allows you to input integers from the console. Using this function several times we can take in several variables from the command line, and say, use them as the arguments for creating a block. Let's do that.

Instructions

Create a block using console input. You'll need four variables that are set using four corresponding inputs from the command line. That's three variables for the co-ordinates and one for the block type. You'll need to use `setBlock()` to create the block.

Extensions

You can change your code to input co-ordinates for:

- Creating a block
- Creating several blocks
- Teleporting the player
- Hollow cuboid with co-ordinates as input
- Printing the block type at the given co-ordinates

Give them all a try if you can. They're really useful when you want to check things during testing your own programs.

5.1.6 Sprint Record

Skills and knowledge we'll practice in this exercise:

- Subtraction
- Concatenation
- Time
- Co-ordinates

This exercise combines variables and the maths operators, that you learned in the previous chapter, with posting messages to the chat.

The exercise is meant to be a lot more challenging than the others. Saying that, we've covered everything you need to create the program in this chapter and the previous chapters.

This program will work out how far the player travels in 10 seconds and displays the results in chat.

Instructions

Here's a step-by-step guide for creating this program:

1. Connect to Minecraft in the usual way
2. Get the player's current position and create variables to record the co-ordinates
3. Wait ten seconds while the player uses the keyboard to run in a direction
4. Get the player's new position and store the values in new variables
5. Compare the difference between the starting position and ending position and post the results to the chat
6. The results should be in the following format "The player has moved x: 10, y: 6 and z: -3"

If you've got the program running, but you're finding it too difficult to switch between the command line and Minecraft fast enough, try adding a three second count down before step 2. Post this countdown to the chat.

Extensions

- Change the console output so that it displays points for the distance moved

End of Exercises

5.2 Strings

Strings are pretty much the same thing as text. When you want to include words or sentences in your program, you use strings.

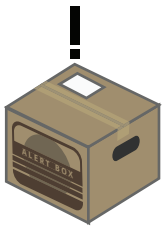
In the string data-type you can store letters, numbers and symbols. All strings are enclosed in speech marks. For example, this is a string:

```
1 "Cats, cats everywhere and not a drop to drink."
```

This is also a string:

```
1 'Turn left after 200 yards!'
```

Each individual letter, number or symbol in a string is called a character. A string has its name as it is a string of characters.



ALERT: When writing a string you can use either type of speech marks. Either ' or ". The two types of speech marks don't mix. You must close a string variable with the same type of quotation marks that you opened it with.

5.2.1 Substrings

The position of each character in a string can be referenced with a number. This is useful as you can get the letter at a certain point in a string.

String

variable type

A variable type that represents text. They include a combination of characters: i.e. letter, numbers and symbols.

Expression:

```
"Your text in here"
'Your text in here'
```

Statement:

```
name = "Edward"
address = "64 Engl Drive, Southumbria, ST6 7HY"
paragraph = "Edward likes swimming. He would like
    ↪ to teach you how to swim."
```

For example, when you want to use a person's initial you would access the character in the first position of their name and ignore all of the other characters.

The position of a character in a string is known as the character's index position.

The first index in a string, instead of being 1 as you would expect, is 0. The second position is 1, the third is 2 and so on. For example the positions of the string "Cats" is:

```
0|1|2|3
C|a|t|s
```

Counting from 0 may seem stupid, but there is a reason. Old computers were really slow and had very small memories. It was faster and more efficient to start counting indexes from 0. Even though computers are much faster these days, counting from 0 has stuck around as it would be too much hassle to change.

Square brackets [] are used to access the character in a string. An integer is put inside the brackets to tell Python which character index you want. For example, when we want the initial of someone's first name:

```
name = "Jim"
initial = name[0] #value of "J"
```

Substring

function

The substring method accesses characters in a string using the index positions of the character. To use a substring, square brackets [ ] are used at the end of a string data-type with an index position as an integer inside the brackets.

Expression:

```
1 "String"[index]
```

Statement:

```
1 firstLetter = "Cats"[0] #vaue of "C"
2 #OR
3 position = 3 #integer with value of 3
4 fourthLetter = "Cats"[position] #value of s
5 #OR
6 name = "Barry"
7 initial = name[0] #value of "B"
```

5.3 String Functions and Methods

Within Python there are reusable bits of code called functions and methods. As these bits of code are reusable it means the programmer doesn't have to rewrite code for common tasks over again.

The difference between a method and a function is subtle and we'll explain it later on in another section.

Strings have their own set of functions and methods. In this section we'll cover four of them:

- len()
- lower()
- upper()
- str()

5.3.1 len()

len()

method

Returns the length of a string as an integer. The string is provided as an argument in the function's brackets.

Expression:

```
len("String")
```

Statement:

```
lengthOfName = len("Anny")
#This sets the variable to the value of 4.
#OR
animal = "Goose"
animalLength = len(animal) #5
```

The len() function returns the length of a string. You use it by putting a string inside of the brackets.

For example if we wanted to find out how many characters are in someone's name:

```
name = "Humbaba"
length = len(name) #value of 7
```

5.3.2 .lower()

The lower() method makes all of the characters in a string lowercase. In other words it replaces any capital letters with the smaller equivalent.

We use it slightly differently to len(). We put lower() at the end of a string with a full stop in the middle. This is called dot notation.

To change someone's name from uppercase to lowercase we use the lower() method like so:

```
name = "VIX"
name = name.lower() #value of "vix"
```

lower()

method

The .lower() method converts a string into lower case letters.

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

5.3.3 .upper()

Another function exists to turn all characters in a string into uppercase. In other words it capitalises everything.

The upper() method works like the lower() method. You put it at the end of a string data-type with a full stop in between. For example:

```
1 animal = "fish"
2 animal = animal.upper() # value of "FISH"
```

5.3.4 str()

Converting one variable type to another is useful. For example sometimes it is necessary to change an integer into a string.

For example you have an integer, say 2, that you want to be treated as a string instead of integer. This is really easy:

```
1 houseNumber = str(2) # value is "2"
```

You can do this with floats and booleans as well.

The str() function really becomes useful when we start using the print operator.

upper()

method

The .upper() method converts a string into upper case letters. HERE!!!

Expression:

```
"String".upper()
```

Statement:

```
upperCaseName = "Pratap Jefferson".upper()#value
    ↪ of "PRATAP JEFFERSON"
#OR
country = "Denmark"
country = country.upper() #value of "DENMARK"
```

str()

function

The str() function turns non-string data types types into strings. It can be used with integers, floats, booleans and other data types. To convert a different variable type to a string, you put the value inside the brackets of the function.

Expression:

```
str(value)
```

Statement:

```
partGuests = str(12) #value of "12"
pi = str(3.14) #value of "3.14"
isFish = str(True)#value of "True"
```

5.4 Print

Until now we have only made variables and changed them. Displaying values to the user is important for user interaction. This is known as output and is where the print operator comes in.

print

operator

The print statement takes a string and displays it on the console.

Expression:

```
1 print value
```

Statement:

```
1 sentence = "These are not the droids you're
    ↪ looking for"
2 print sentence
```

The print operator outputs data to the console.

If you've been using IDLE, the console is the same thing as the interactive shell. Print will display data on a new line in the interactive shell. For those of you who have been using a console or Geany's built-in console, print will display data on a new line in these windows. You can find more information on the console on page [page number]

To use the print statement you start the line with the **print** operator followed by a string:

```
1 print "String"
```

So if we wanted to print the word "chocolate" to the console, our code would look like this:

```
1 print "chocolate"
```

5.4.1 Printing String Variables

As variables can take the place of values, you use print to output variables to the console. For example we have a name string and we want to display it on the console:

```
1 name = "Charles Christopher"
2 print name
```

This outputs `Charles Christopher` to the console. Pretty simple.

You can use the print operator on integer, float and boolean data-types as well.

5.4.2 Joining Strings

Quite often we need to print a combination of a pre-defined string and a variable. We may also need join two or more strings. Concatenation in Python makes it easy to do this.

.....

Concatenation +

operator

When used with strings, the + operator combines the strings into a single string.

Expression:

```
1 "string" + "string"
```

Statement:

```
1 name = "Gilgamesh"
2 print "Your name is " + name
```

.....

Remember the addition operator from earlier? Yes, the +. It is used to add numbers together, but can also be used to concatenate strings.

The + operator is used to concatenate strings together. For example:

```
1 firstName = "Charles"
2 lastName = "Christopher"
3 print firstName + lastName
```

The output will be `CharlesChristoper`. There is no space character in between the values when we print them. We can add this to the third like like so:

```
1 print firstName + " " + lastName
```

What if we want a string that won't change at the start of the output? Simple, we just write the value in like we would any other string:

```
1 print "His name is " + firstName + " " + lastName
```

This will output "His name is Charles Christoper".

5.4.3 Concatenating Integers, Floats and Booleans

Concatenating and printing variable types other than strings on the console is slightly different than strings.

To concatenate two pieces of data they must be string data-types. As the + operator is used for addition and concatenation, the operator will try to add number values, instead of concatenate them. Integers, floats and booleans are not strings so we must change them to strings in order to use concatenate. Can you think how we do this?

We use the str() function to convert these variables into strings (we introduced str() in section [sectionNumber]). For example, if we wanted the output "Age: 23":

```
1 age = 23
2 print "Age: " + str(age)
```

How about joining two numbers instead of adding them? Pretty simple, just remember to include the str() method.

```
1 print str(19) + str(84) #outputs 1984
```

You can do the same with floats and booleans.

Concatenation can be used as many times as you want within a statement. For example:

```
1 print "The year is " + str(19) + str(84)
```

5.4.4 Placeholders in Strings

There is another approach for using variables with a string. This method substitutes placeholders in a string with variables. It uses the % operator.

In a string you place "%s" in locations that you want to substitute with a variable. You then follow the string with a % and brackets containing the

Placeholder %s

operator

The %s operator is used as a placeholder to substitute variables into a string. The string containing the %s placeholder must be followed by a % operator and a list of values in brackets. You need to have the same number of values in the brackets as the number of "%s" in the string. All values in the brackets should be separated by a comma.

Expression:

```
1 "string %s" % (values)
```

Statement:

```
1 noun = "cheeses"
2 verb = "paying"
3 print "These are not the %s you're %s for." %
    ↪ (noun, verb)
```

values that will replace "%s" in the string. Each value should be seperated with a comma. For example:

```
1 name = "Townsen"
2 brothers = "three"
3 print "I am %s. I have %s brothers." % (name,
    ↪ brothers)
```

This outputs "I am Townsen. I have three brothers."

5.4.5 raw_input()

Quite often you will want to provide your program with input. This interaction between the program and the user is a very important as it can change the data that the program uses.

Up until now all of your variables have been set in the program without any interaction from the user when the program is running. It would be nice to be able to change these variables while the program is running.

raw_input()

function

A function that allows the user to input strings into the program via the command line.

Expression:

```
1 raw_input("string")
```

Statement:

```
1 hats = raw_input("How many hats do you have?")
2 print hats
```

One way of interacting with your program is with the raw_input() function. It prints a string to the console and then waits for the user to type a response. The value that the user inputs is then returned to the program. For example:

```
1 name = raw_input("What is your name?")
2 print name
```

In this example the string "What is your name?" is printed to the console. The user then enters a response, and this response is used as the name variable. The variable name is then printed to the console.

5.4.6 input()

Like raw_input(), the input() function allows the user to input data from the command line.

The difference between raw_input() and input() is that raw_input() will always convert the user's value into a string, whereas input() allows other datatypes.

You use the input() function to input integers, floats and booleans. Whereas you use the raw_input function to input strings. The input() function can input strings, but they must be enclosed in speech marks.

```
1 bill = 120
```

input()

function

A function that allows the user to input data, including integers, floats and booleans, into the program via the command line.

Expression:

```
1 input("string")
```

Statement:

```
1 hats = input("How many hats do you have?")
2 print hats
```

```
2 people = input("How many people are sharing the
    ↪ bill?")
3 print "The cost each is " + str(bill / people)
```

5.5 Date and Time

Python has functions that allow you to use the date and time as variables. Let's briefly cover the basics of date and time.

Unlike the string functions we were introduced to earlier, the date and time functions aren't included in every Python program. They are stored in a separate location called a module. A module is a collection of functions grouped around a similar theme, in this case date and time. In order to use this module we must import them.

To use the date and time in your code you need to write this statement first:

```
1 from datetime import datetime
```

[finish this bit]

5.5.1 Getting the Current Date and Time

[This hasn't been written yet]

datetime.now()

For example: print datetime.now()

Extracting Information

.month .day .year .hour

Chapter 6

Comparators and Control Flow

Making decisions is very important for everyday life. If it's cold outside you wouldn't wear shorts, you'd wear a coat. If you're hungry you'd eat something. We make decisions based on conditions. Programs can also make decisions using conditions.

In Python control flow is used to decide whether a piece of code will execute or not. This allows us to change the outcome of the program depending on conditions. For example you might want to display a warning message if the user inputs an incorrect data type or create an inventory system that tells the user to restock an item if there are less than 5 in stock.

In this chapter we will first cover comparators and binary operators, which are necessary for determining whether a condition is True or False. We will then cover if statements, which execute a block of code based on whether a condition is True or False.

6.1 Minecraft Exercises

Comparators and if statements are really useful with Minecraft.

6.1.1 Swimming

Skills and knowledge we'll practice in this exercise:

- Equal-to comparator

We're going to make a program that states whether the player is in water. We'll use comparators to achieve this. The results will be posted to Minecraft chat.

To find out the block type at certain co-ordinates we use the getBlock() function. This function takes the co-ordinates as three arguments and returns the block type as an integer. Take the following code sample:

```
1 blockType = mc.getBlock(10, 18, 13)
```

If the block type at co-ordinates (10,18,13) is a melon (value 109), the blockType variable will now hold a value of 109.

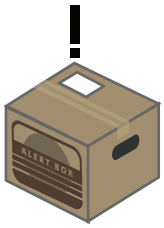
Instructions

To create this program follow these steps:

1. Create a variable that stores the player's position.
2. Create a variable that uses the getBlock() function to store the block type at the player's position. Name this variable blockType.
3. Using the equal to comparator (==) compare the blockType variable to the value 9, which is the value of water. Store this comparison in a variable called swimming.
4. Print the value of the swimming variable to chat.

When the player is in water, the chat will say True. When the player isn't in water the chat will say False.

Try changing the chat message so that it's more user friendly.



ALERT: At the moment you will not be able to run this program in real-time. You must run the program every time you want to make the program work. This applies to all of the other exercises in this chapter. You'll learn how to update the program in real-time in the loops chapter.

Extensions

Try these extension exercises if you want to take this exercise further:

- Find out if the player is standing in a tree

6.1.2 Do you want to stop smashing things?

Skills and knowledge we'll practice in this exercise:

- Input
- if statement
- else statement
- Equal to
- Immutable Blocks

Remember back to the first chapter, we made a program that stopped the player from smashing blocks. In other words blocks were made immutable. The program was useful as it allowed you to protect your precious creations from accidents or vandals. As useful as the program is it became a bit cumbersome when you wanted to turn immutable off, which required a second program.

Using an if statement, an else statement and console input we can make a program that handles turning immutable on and off. Our program will ask whether you want the blocks to be immutable and then set immutable to True or False, depending on your response.

Instructions

In this exercise we'll introduce you to planning programs with a flowchart. You learned about flow charts in chapter [chapter number]

We'll work through the flowchart step-by-step and give you the code for each step:

1. Start at the top of the diagram, where it says start. Follow the arrow to the next box.
2. A rectangle represents a process, in this case we're asking the a question
3. The next box is an input, in this case we're taking the user's response to our question. This box and the above process can be combined when we're writing Python code:

```
4 response = raw_input("Do you want blocks to be
    ↪ immutable?")
```

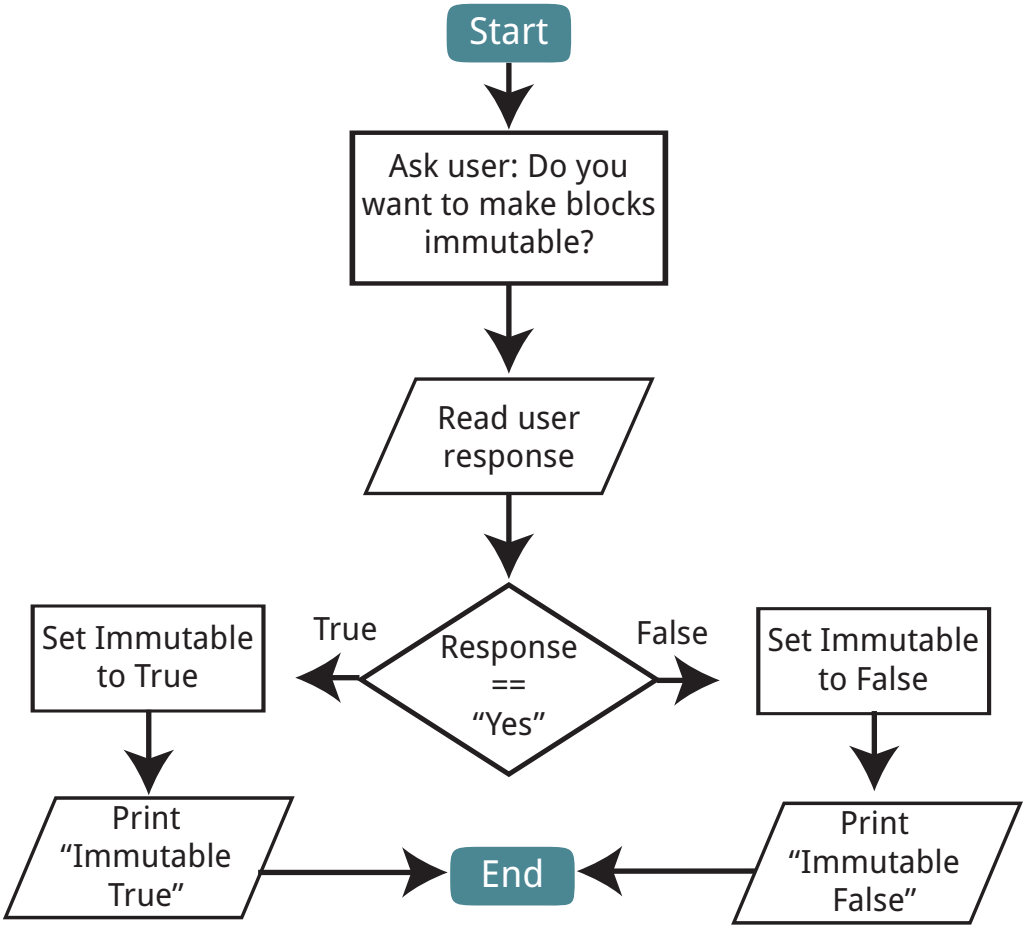
4. The next item is a condition that determines whether the user's response was "Yes".
5. When the response is True we follow the arrow labelled True.
6. This step sets immutable to True in Minecraft.
7. We then print a statement to the user to say that immutable is now True. We'll use an if statement for steps 4–7:

```
5 if response == "True":
6     mc.setting("world_immutable, True)
7     mc.postToChat("World is immutable")
```

8. Going back to the comparator, if the comparison is False, we set immutable to False and tell the user this. Effectively we do the opposite of steps 4–7. We can represent the False flow with the following code:

```
8 else:
9     mc.setting("world_immutable, False)
10    mc.postToChat("World is mutable")
```

Copy the code and make it complete with the code at the top of the program that we use every exercise.


Start
Ask user: Do you want to make blocks immutable?
Read user response
Response == "Yes"
True
False
Set Immutable to True
Set Immutable to False
Print "Immutable True"
Print "Immutable False"
End

Extensions

- You'll notive that you get the same result for entering "No" as you would for entering non-sensical input, like "banana". Extend the if statement to handle incorrect input. Include an elif statement to handle the "No" response and change the else code to ask for correct input.
- Use boolean operators to accept different variation of "Yes" and "No", such as lower case "yes", upper case "YES" and a single character response "Y".

6.1.3 Bring us a shrubbery

Skills and knowledge we'll practice in this exercise:

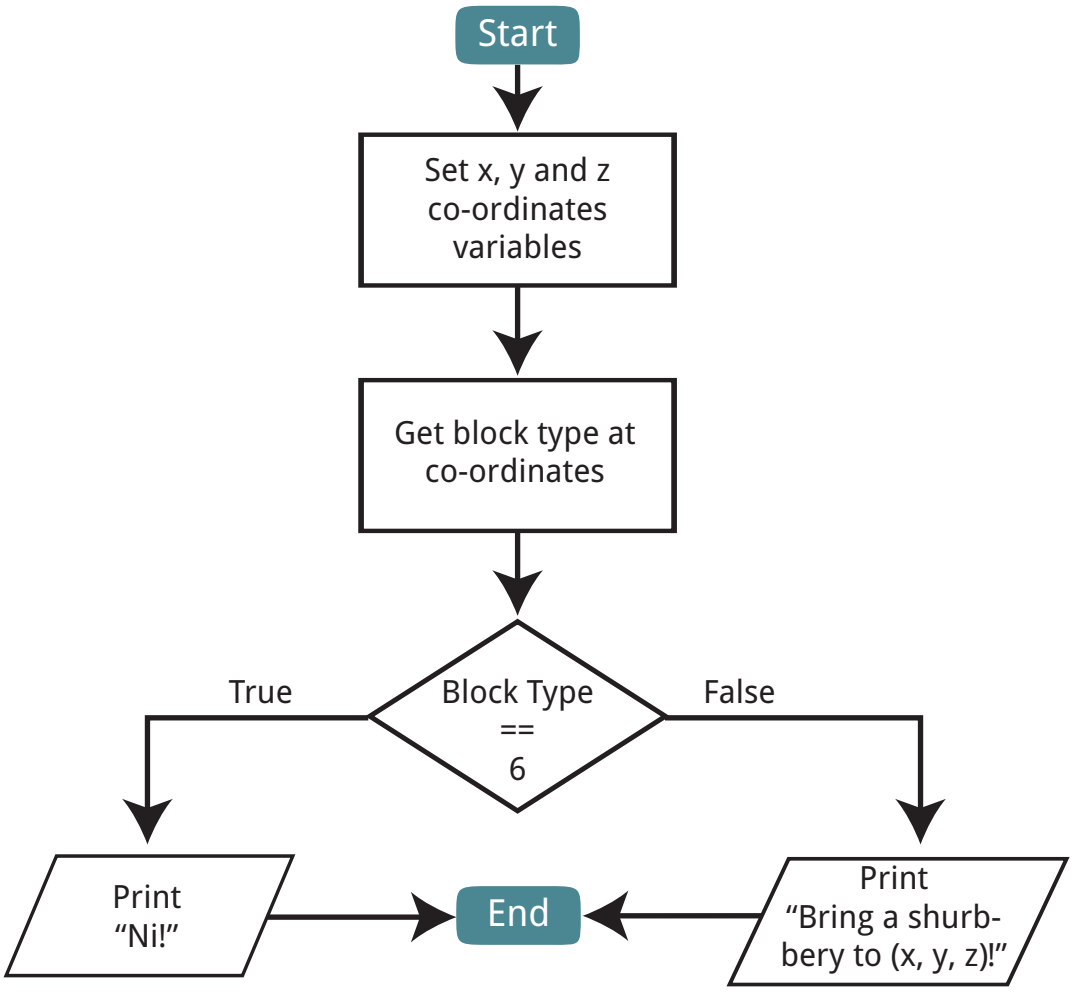
- if statement
- else statement
- Equal-to
- Block type
- Systems modelling

Collecting an item and delivering it to a certain location is a staple of video games. We'll create a simple program that checks whether a certain block has a tree sapling placed on it.

Instructions

Follow the instructions in the flowchart.

A simple tip, you'll need to find a convenient location in the game world and use it's co-ordinates to set the x, y and z variables in the first step. Also, the value of a tree sapling is 6. Also make sure you have set the co-ordinates to the correct location; it's easy to spend ages wondering why it's not working only to realise you were one block off on the x-axis. The rest of the instructions on the flowchart can be achieved with code that you've learned so far.


Start
Set x, y and z co-ordinates variables
Get block type at co-ordinates
Block Type == 6
True
False
Print "Ni!"
End
Print "Bring a shrub-bery to (x, y, z)!"

Extensions

- Check for a different block type, like a diamond.
- Change the body of the if statement do something else, like change the ground to lava or build a house. Be creative.
- Accept leaves (block type 18) as an equivalent for a shrubbery using the OR boolean operator.
- Accept more than one block type with different responses using elif statements.

6.1.4 Take a Shower

Skills and knowledge we'll practice in this exercise:

- if statement
- else statement
- greater than or equal to
- less than or equal to

The best Minecraft houses have a lot of attention to detail. Many people include wooden flooring, fireplaces and pictures in their houses to make them more homely. You're going to go one step further and make a working shower.

Instructions

Create a 3 x 4 block area on the ground. Using an if statement, the less than or equal to comparator and the greater than or equal to comparator work out if the player is in the shower. When the player is in the shower, make water appear several blocks above. When the the player is not in the shower, stop the water.

Extensions

- Make a trap door that opens if the player is on top of it

6.1.5 Secret Passage

Skills and knowledge we'll practice in this exercise:

- Greater than
- Less than
- not equal to
- and operator
- or operator
- if statement
- elif statement
- else statement

Special items in many video games open secret passages. These items are often placed onto small stands, known as plinths. In this exercise we'll create a building with a secret passage that opens when a diamond block is placed on a plinth. When any other type of block is placed on the plinth the floor will turn to lava.

Instructions

Build a cuboid shaped building with an entrance and a secret room behind a wall. Outside the entrance to the building make a single block to represent the plinth.

Work out if the player is inside the walls of the building using greater than and less than comparators alongside binary operators.

When the player is inside the building:

- If the player is inside the building and a diamond block is on the plinth, open a secret passage to the secret room.
- If the player is inside the building and a block that is not a diamond block is on the plinth, make the floor lava. Hint: check that the block is not a diamond and the block is not air.

- If the player is inside the house and there is nothing on the plinth, post "bring us something of value" to the chat.

As this is a more complex program, build it and test it in stages. Make sure each part works before moving on. This will make debugging a lot easier.

Look out for bugs. Test each of the above conditions in different orders.

Extensions

- Something

End of Exercises

6.2 Comparators

Even without a computer we are very good at comparing things. We know that 5 is bigger than 2, 8 and 8 are the same number, and 6 and 12 are not the same number.

Comparators in Python allow us to compare data. There are six comparators in Python:

- Equal to (==)
- Not equal to (!=)
- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)

The comparator goes between two pieces of data. In this case we're using an equal to operator to compare two integer values:

```
1 8 == 2
```

Each comparator will return a boolean value that states whether the condition has been met. As all comparators return a boolean value you can use them anywhere that you can use a boolean value. In the above example the the boolean value is `False`. Keep reading to see how we worked that out.

6.2.1 Equal To

When you want to tell if one value is the same value as the other one, you use the equal to comparator. Equal to compares two values, if they are the same it evaluates to `True`, if they are different it evaluates to `False`.

Equal To

comparator

The equal to operator compares two values to check whether they are the same. When the values are the same the comparison will return the boolean value True, when the values are different we get false.

Expression:

```
value1 == value2
```

Statement:

```
triforce = 3
hasTriforce = triforce == 3 # value of True
hasTriforce = triforce == 2 # value of False
```

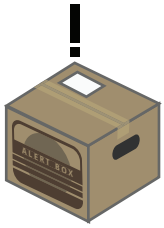
For example:

```
length = 2
width = 2
square = length == width #value of True
```

The square variable is true as the value of the length and width variables are the same value. If they are different we get a False:

```
length = 4
width = 1
square = length == width # value of False
```

The Equal To comparator can be used on all variable types: strings, integers, floats and booleans.



ALERT: Notice how == is used for the comparison instead of =, which is used to set a variable. This is so that Python can tell the difference between a comparison and setting a variable. Try to remember this difference, even the best of us make a mistake by using = instead of == every once in awhile.

6.2.2 Not Equal To

The Not Equal To comparator is the exact opposite of the Equal To comparator.

Instead of comparing whether two values are the same, it compares whether they are different. When the two values are different the comparison will evaluate to True, when they are the same it will evaluate to False.

A rectangle is a four sided shape with different values for length and width. The comparison on line 3 compares the length and width of our shape:

```
1 width = 3
2 length = 2
3 rectangle = width != length #value of True
```

In this example the value of `rectangle` is `True` as `width` and `length` have different values. To make the comparison return `False`, these values need to the same:

```
1 width = 3
2 length = 3
3 rectangle = width != length #value of False
```

The Not Equal To comparator works with strings, integers, floats and booleans.

Not Equal To

comparator

The not equal to operator compares two values to check whether they are not the same. When the variables contain different values, it returns a True boolean value. When they are the same, it returns a False boolean value.

Expression:

```
1 value1 != value2
```

Statement:

```
1 cakes = 5
2 enoughCakes = cakes != 5 # value of False
3 enoughCakes = cakes != 6 # value of True
```

Less Than

comparator

The less than operator compares two values to check whether the first is smaller than the second. When the value on the left is larger than or the same as the value on the right, the result will be false.

Expression:

```
1 value1 < value2
```

Statement:

```
1 cakes = 5
2 enoughCakes = cakes < 5 # value of False
3 enoughCakes = cakes < 4 # value of False
4 enoughCakes = cakes < 6 # value of True
```

6.2.3 Less Than

The Less Than comparator is used to determine whether one value is smaller than another. When the value on the right of the comparator is larger than the other value, True will be returned. When the value on the right

is smaller or the same as the other value, then False will be returned.

A van driving under a bridge needs to know whether it's small enough to fit under:

```
1 vanHeight = 8
2 bridgeHeight = 12
3 willPass = vanHeight < bridgeHeight
4 # Value of True
```

In this case the van will fit as it's smaller than the bridge: 8 is less than 12. Later in its journey, the same van might encounter another bridge that is too low to drive under:

```
1 vanHeight = 8
2 bridgeHeight = 7
3 willPass = vanHeight < bridgeHeight
4 # Value of False
```

This comparator does not work with Strings, though it does work with Integers, Floats and Booleans.

6.2.4 Less Than or Equal To

This comparator works almost the same as the Less Than comparator. It will compare whether the value on the right is less than or the same as the value on the left.

The difference between less than or equal to and less than comparators is that the less than or equal to comparator will return True if both values are the same, where as the less than comparator will return False.

We're presenting our amazing program to a group of people. There are only 30 seats in the room. We can write a program to check if there are enough seats for all the people who want to attend:

```
1 seats = 30
2 people = 30
3 enoughSeats = people <= seats
4 # Value of True
```

In this case there is exactly the right amount of seats, so the `enoughSeats` variable will be `True`; if there were fewer people we'd still have enough seats. Five more people wants to see your amazing program:

Less Than or Equal To

comparator

The less than operator compares two values to check whether the first is smaller than or equal to the second. When the value on the left is smaller than or the same as the value on the right, the result will be True.

Expression:

```
value1 <= value2
```

Statement:

```
cakes = 5
enoughCakes = cakes <= 5 # value of False
enoughCakes = cakes <= 4 # value of True
enoughCakes = cakes <= 6 # value of True
```

```
seats = 30
people = 35
enoughSeats = people <= seats
# Value of False
```

Unfortunately the `enoughSeats` variable is now False so there aren't enough seats for these new people.

This comparator does not work with Strings, though it does work with Integers, Floats and Booleans.

6.2.5 Greater Than

When you need to work out whether one value is bigger than another you use the Greater Than comparator.

When the Greater Than operator is used it will return True when the value on the left is greater than, but not the same as, the number on the right. False will be returned when the left value is smaller than the right value.

Our robot has a lifting limit of 100 bananas. It can't lift more than 99 bananas. As long our robot's lifting limit is greater than the number of banana's it's lifting, they can be lifted:

Greater Than

comparator

The greater than operator compares two values to check whether the first is larger than the second. When the value on the left is larger than the value on the right, the result will be True.

Expression:

```
value1 > value2
```

Statement:

```
seeds = 6
surplusSeeds = seeds > 5 #value of True
surplusSeeds = seeds > 6 #value of False
surplusSeeds = seeds > 7 #value of False
```

```
limit = 100
bananas = 99
canLift = limit > bananas
# value of True
```

The `canLift` variable is `True`. Brilliant, our robot can lift it: 100 is greater than 99. Looks like someone's added another banana to the pile:

```
limit = 100
bananas = 100
canLift = limit > bananas
# value of False
```

Oh no, the limit has been reached. The `canLift` variable is now False: 100 is not greater than 100, it's the same. Our robot can't lift the bananas.

This comparator does not work with Strings, though it does work with Integers, Floats and Booleans.

6.2.6 Greater Than or Equal To

Like the Greater Than comparator the Greater Than or Equal To comparator will determines whether one value is greater than another. Unlike the Greater Than comparator it will also evaluate to True if the values are the same.

Greater Than or Equal To

comparator

The greater than operator compares two values to check whether the first is larger than or the same as the second. When the value on the left is larger than or the same as the value on the right, the result will be True.

Expression:

```
value1 >= value2
```

Statement:

```
seeds = 6
surplusSeeds = seeds >= 5 #value of True
surplusSeeds = seeds >= 6 #value of True
surplusSeeds = seeds >= 7 #value of False
```

We're giving stickers to all of the people who came to see our amazing program presentation. We need to check whether we have enough stickers for everyone who came:

```
stickers = 30
people = 30
enoughStickers = stickers >= people
# value of True
```

We have enough stickers: 30 is the same as 30. If one or more people don't turn up, we'll still have enough stickers. One of your friends thinks the stickers look cool and wants one, there are now 31 people who want stickers:

```
stickers = 31
people = 30
enoughStickers = stickers >= people
# value of False
```

What a shame, we don't have enough stickers: 30 is not greater than 31. Looks like your friend can't have a sticker.

This comparator does not work with Strings, though it does work with Integers, Floats and Booleans.

6.3 Boolean Operators

Combining two or more comparators is often necessary in programs. You might want to determine whether two conditions are true, like a car is red and it costs less than £10,000.

For combining two or more comparisons we use boolean operators. These operators work with both boolean variables types and comparators, which evaluate into booleans.

There are two operators in Python that compare booleans:

- and
- or

There is also a third operator, the not operator that changes the value of a single boolean.

Like comparators, boolean operators can be used anywhere that you would use a boolean value.

Boolean operators are also called logical operators.

6.3.1 and

The and operator is used when you want to check whether two comparisons are both True.

For an expression with an and operator to be True, both comparisons must be True. If either comparison is False, the statement will return False.

This table summarises the results of all of the possible boolean combinations and results when using the and operator:

We want to find out whether a person is older than 18 and owns a car:

and	True	False
True	True	False
False	False	False

```
1 age = 21
2 ownsCar = True
3 eligible = age > 18 and ownsCar == True
4 #value of True
```

The age of the person is greater than 18 and they own a car, so the `eligible` variable evaluates to `True`. If one of these comparisons was `False` then the statement would evaluate to `False`. Like if they don't own a car, but are older than 18:

```
1 age = 25
2 ownsCar = False
3 eligible = age > 18 and ownsCar == True
4 #value of False
```

.....

and

operator

The and boolean operator will return True when the boolean values on either side of it are both true. If one or both of the values are False, then the and Operator will return False.

Expression:

```
1 boolean1 and boolean2
```

Statement:

```
1 trueStatement = True and True
2 #OR
3 name = "Sam"
4 age = "45"
5 canRegister = name != "Jim" and age >= 18
6 #value of True
```

.....

6.3.2 or

The or comparator works slightly differently to and. When either or both comparisons are True, the or expression will return True.

That means that as long as one comparison is True either of the comparisons can be False and the expression will still be True. If neither comparison is True, the expression will evaluate to False.

.....

or

operator

If either conditions are True, then the statement evaluates to True. So one of the booleans can be False and it will still return True. If both booleans are False, then it will return False.

Expression:

```
1 boolean1 or boolean2
```

Statement:

```
1 trueStatement = True or False
2 #OR
3 name = "Jim"
4 age = "32"
5 canRegister = name != "Jim" or age >= 18
6 #value of True
```

.....

This table the possible combinations and results of using the **or** operator with booleans:

or	True	False
True	True	True
False	True	False

If we want to adopt a cat that is either black or ginger we could use this code:

```
1 catColour = raw_input("What colour is the cat?")
2 myCatNow = catColour == "black" or catColor ==
    ↪ "ginger"
3 print "Adopt this cat: " + str(myCatNow)
```

As long the `catColour` is either "black" or "ginger" we'll adopt it. In order for `myCatNow` to return a `False` value, the `catColour` value needs to be neither "black" nor "ginger". So we wouldn't adopt a "blue" cat.

6.3.3 not

The **`not`** operator works differently to the **`and`** and **`or`** operators.

The **`not`** operator changes a boolean to its opposite. It changes a `True` into a `False` and vice versa.

.....

not

operator

The not operator changes condition to their opposites. It makes True values into False values and False values into True.

Expression:

```
1 not boolean
```

Statement:

```
1 lights = 1
2 notEnoughLights = not lights > 2
```

.....

So:

```
1 hungry = not True #This is False
2 sleepy = not False #This is True
```

You can combine the not operator with the other two like this:

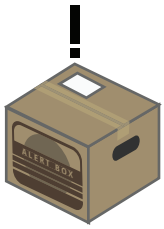
```
3 timeForBed = not hungry and sleepy
4 #value of True
```

The not operator only applies to boolean it is in front of. So for the above example it will reverse the hungry variable and leave the sleepy variable alone.

6.3.4 Boolean Operator Order

It is possible to combine as many boolean operators as you want in a single statement. For example you can use a combination of **and**, **or** and **not** like so:

```
1 flying = True and not False or False
2 #value of True
```



ALERT: There is a certain order that boolean operators are evaluated by Python. If you get the order wrong you might get a result you weren't expecting. This is the order:

1. not
2. and
3. or

So in the above example the **not** `False` part of the statement is evaluated first to create `True`. The and is then evaluated, with True and True evaluating to True. Finally the or is evaluated with True or False becoming True.

Practice creating statements with boolean operators in IDLE and see if you can guess the result of each.

6.4 If, Else and Elif

You're about to learn how to instruct a computer to make decisions. This may not sound like a lot, but it is pretty powerful.

If statements use boolean conditions to decide whether to or not execute some code. These conditions are the same as comparators, which makes comparators really useful.

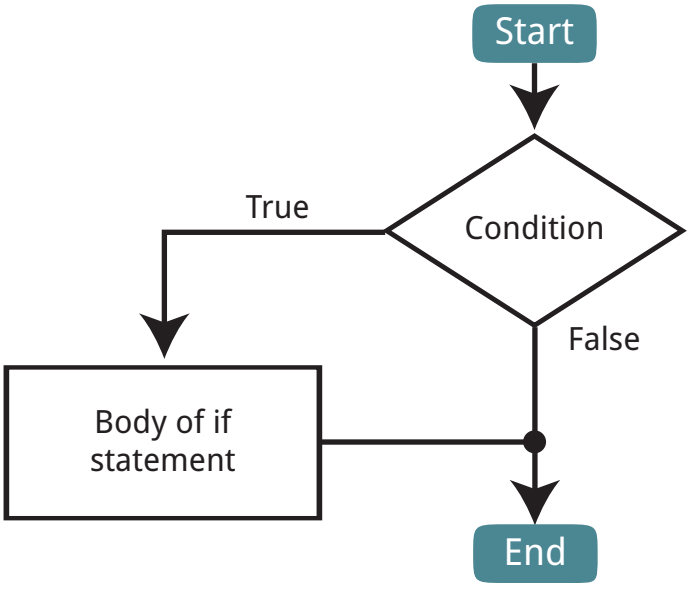
6.4.1 if Statements

In Python, if statements enable the computer to make decisions based on a condition.

An if statement has three parts: an if operator; a condition; and a body of code.

Conditions are the same things as comparators. An if statement will only execute the code contained in its body if the condition is True. So when the condition is True, the body of code will execute then continue through the rest of the program. When the condition is False the body of the if statement is ignored and the program will continue on the line after the if statement.

If statements can be described as a flowchart:



A colon and indentation tells Python which block of code makes up the body of the if statement. A colon comes at the end of the line with the if operator and the condition. This colon tells Python that everything following the colon is the body of the if statement.

The body is the code that will execute when the condition is `True`. The body can have as many lines of code in it, all of which must be indented.

The following code uses an if statement to check whether a password is correct:

if

statement

An if statement tells the computer whether or not to do execute some code based on a condition. Boolean operators are used in if statements to decide whether a condition has been met. The indented code will only run if the boolean is True. When the boolean is False the indented will not run.

Expression:

```
1 if condition:
2     #body of if statement
```

Statement:

```
1 height = 15
2 if height >= 12:
3     print "The height is too high"
```

```
1 password = "cats"
2 attempt = raw_input("Please enter the password")
3 if attempt == password:
4     print "Password is correct"
```

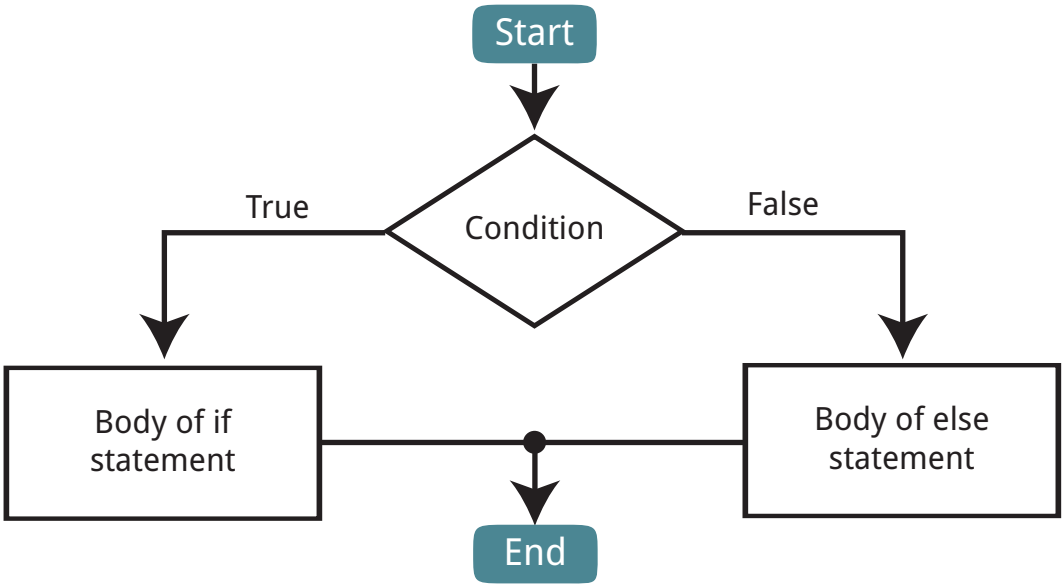
The `attempt == password` expression is the condition as it follows the if operator. The line print `"Password is correct"` makes up the body of the code as it is indented.

The code will only print "Password is correct" if the attempt variable is the same as the password variable. If they are not the same it will not print anything to the console.

6.4.2 else

An if statement will only do something if the boolean condition is True. What if you want it to do something if that condition is False, in addition to when it's True? That's where the else statement comes in.

An else statement works with an if statement. It will execute a block of code when the condition of the if statement evaluates to False. This can be summarised in the following diagram:



else

statement

An else statement works with an if statement. It will execute code if the condition of the if statement is not met, i.e. when the if condition is False

Expression:

```
if condition:
    #body of if statement
else:
    #body of else statement
```

Statement:

```
shells = 12
if shells == 32:
    print "You have all of the shells"
else:
    print "You need more shells"
```

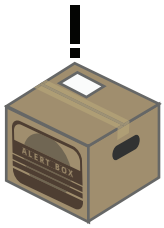
As with an if statement, in Python the else statement uses a colon and indented code to indicate which code belongs to the body of the else statement. The else statement does not have its own condition as it shares this with the if statement.

Returning to the password example using if statements, we can use an else statement to print a message when the password is incorrect.

```
1 password = "cats"
2 attempt = raw_input("Please enter the password")
3 if attempt == password:
4     print "Password is correct"
5 else:
6     print "Password is incorrect"
```

When the attempt variable matches the password variable, the if condition will be true and the program will run the code in the if statement block that prints "Password is correct".

When the attempt variable does not match the password variable, the if condition will be false and the program will run the code in the else statement block that prints "Password is incorrect".



ALERT: An else statement cannot be used without an if statement.

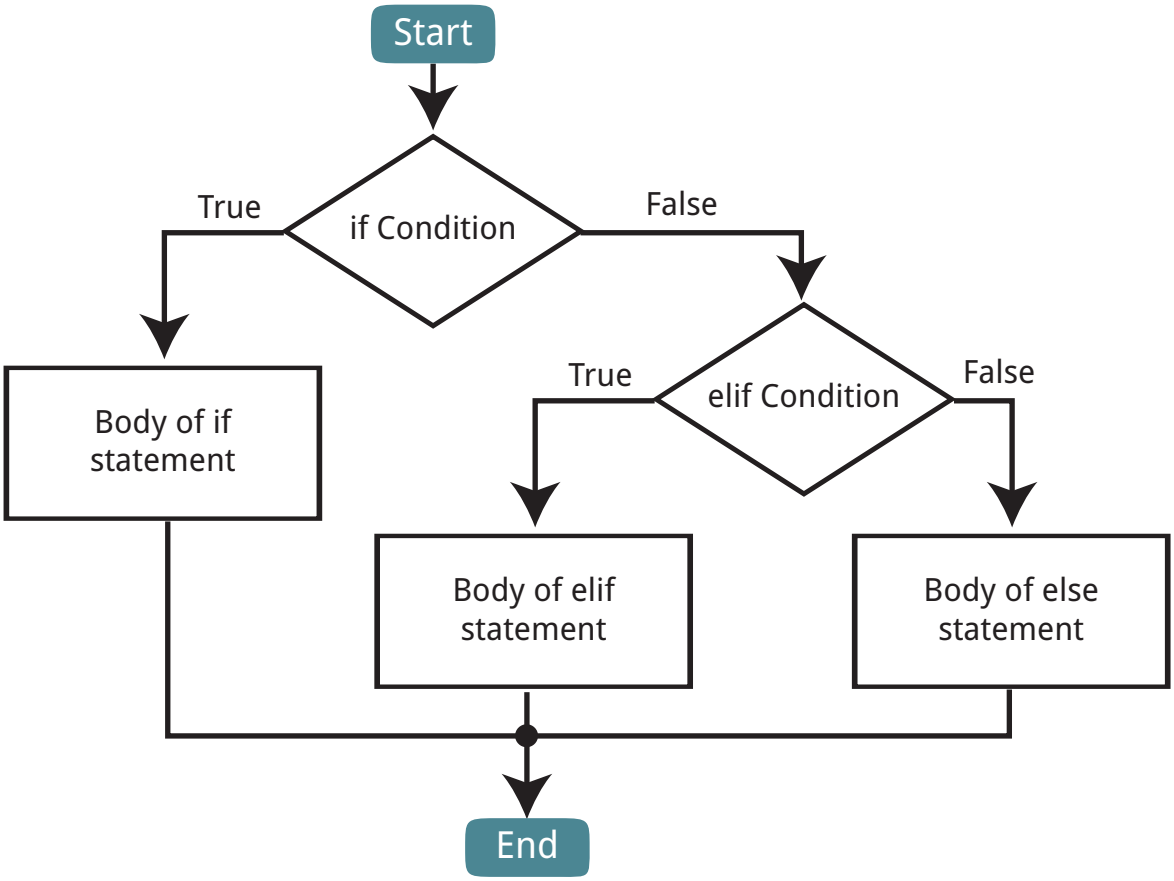
6.4.3 elif

Sometimes you want an else statement to only execute if a condition is met that is different to the condition of the if statement.

The else if statement, or elif in Python, gives you this functionality. An elif statement has its own condition and body. The elif statement will execute when the condition of the if statement is `False` and the condition of the **`elif`** statement is `True`.

When an else statement follows an elif statement it will execute when the if statement is `False` and the elif statement is also `False`.

The following flowchart summarises this:



For example, you could design a program that monitors the temperature of an oven and states whether it is the desired temperature, say 200 degrees celsius. It would need to do three things:

1. when the oven is 200 degrees tell you it's the correct temperature;
2. tell you to increase the heat when the temperature is less than 200; and
3. tell you decrease the heat when the temperature is more than 200.

Here is the code:

```
1 ovenTemp = input("Enter the oven temperature: ")
2 desiredTemp = 200
3 if ovenTemp == desiredTemp:
4     print "Temperature is perfect"
5 elif ovenTemp > desiredTemp:
6     print "Temperature is too hot"
7 elif ovenTemp < desiredTemp:
```

elif

statement

An elif will execute when the if statement condition evaluates to False, and the condition of the elif evaluates to True. It must be used in combination with an if statement.

Expression:

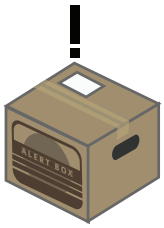
```
if condition:
    #body of if statement
elif condition:
    #body of elif
else:
    #body of else statement
```

Statement:

```
shells = 17
if shells == 32:
    print "You have all of the shells"
elif shells > 16:
    print "You've collected more than half of the
      ↪ shells"
else:
    print "You need more shells"
```

```
    print "Temperature is too cold"
else:
    print "Something weird happened"
```

The code goes through each part of the if statement from the top to the bottom. The else statement will execute when the if statement and elif statement conditions are all false.



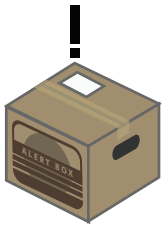
ALERT: You can use as many elif statements as you want, as long as they are after an if statement and before any else statements. The elif statement cannot be used without an if statement.

6.4.4 Nested If statements

It is possible to use if statements within the body of another if statement. This is known as a nested if statement. There are different circumstances where this is useful.

Let's see an example of a nested if statement. We have a simple cash machine that checks if you have enough money and then asks you to confirm your withdrawal if you do. Notice how the nested if statement is indented within the body of the first if statement.

```
1  withdraw = input("How much do you want to
     ↪ withdraw? ")
2  balance = 1,000
3
4  if balance <= withdraw:
5      confirm = raw_input("Are you sure you want to
         ↪ withdraw " + str(withdraw) + "?")
6
7      if confirm == "Yes" or confirm == "y" or
         ↪ confirm == "yes":
8          print "Here is your money"
9          #some more code to give them the money
10 else:
11     print "You do not have enough money"
```



ALERT: The else and elif statements can be used with nested if statements and if statements can be nested within else and elif statements.

6.4.5 Checking For Letters

isalpha()

method

The isalpha() method checks whether a string only contains letters. In other words, it doesn't contain numbers or symbols. If it only contains letters, it returns True, otherwise it returns False.

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

To check whether a string contains only alphabetic characters, i.e. no number or symbols, you use the isalpha() function.

For example:

```
1 "cats".isalpha() #true
2 "9 cats".isalpha() #false
```

This function is useful for validating user input. For example registration forms frequently require you to include your name. Names don't have

numbers or symbols in them, so we need to check whether the name only contains letters. The isalpha() function can be used to check this.

As it returns a boolean value the function can also be used as a condition in an if statement. Here is some code to check whether the user has input a valid name, i.e. a name with only letter values, no numbers or symbols:

```
1 name = raw_input("Please enter your name: ")
2
3 if(name.isalpha()):
4     print "Name accepted. Thank you."
5 else:
6     print "Name must not include numbers or
        ↪ symbols. Letters only."
```

Chapter 7

Functions

Functions are reusable blocks of code that perform specific tasks.

For example, you have a block of code that capitalises the first letter of every sentence in a several strings. You could rewrite (or copy and paste) the capitalisation code every time you need to use it on a new string, however this would be inefficient.

Instead you could write the capitalisation code as a function. Making it easier and faster to repeat the same task multiple times.

We have already seen some functions earlier. Remember str(), raw_input() and len()? They're all pre-written functions that come with Python. You can also create your own functions.

We use functions for the following reasons:

Reusability functions save time. They stop you from rewriting the same code over and over again, making it faster and easier to write a program.

Debugging By containing tasks in groups of code it is easier to identify where a problem originates.

Modularity Functions within the same program can be developed independently of one another without needing to know the internal workings of each one. This makes it easier to share code with other people, whether in a team or through open source software.

Scalability it is easier to increase the amount of data used in code without having to rewrite or copy the code every time. [expand on this]

In this chapter we'll cover how you can write and use your own functions.

Once you're comfortable with functions we'll introduce you to modules. Modules are a collection of functions that are packaged together around a common theme.

7.1 Minecraft Exercises

Functions make it easy to reuse code. You've been using functions in Minedraft for a while, such as...in this set of exercises we'll create our own functions.

Surprise, you've been using modules all along.

7.1.1 A Forest

Skills and knowledge we'll practice in this exercise:

- Creating Functions
- Function Call
- Arguments

A forest is essentially just a bunch of trees. To create a forest in Minecraft we'll make a function that builds a tree and then reuse the function several times to create a forest.

Instructions

This is the basic code that you'll be using.

```
1 import mcpi.minecraft as minecraft
2 mc = minecraft.Minecraft.create()
3
4 def growTree(x,y,z):
5     #Write your code here
6
7 pos = mc.player.getTilePos()
8 x = pos.x
```

```
 9 y = pos.y
10 z = pos.z
11
12 growTree(x + 1, y, z)
```

The growTree() function created in this code takes arguments that represent co-ordinates. Your task is to write code in the body of of the function that creates a tree at the given co-ordinates. Use the setBlock() and setBlocks() functions to do this.

Once you've got something resembling a tree appearing write several more calls to the function with different arguments. You should create at least three rows of three trees in front of the player each time you run your program.

Extensions

- Randomise the tree height, distance apart and number of trees

7.1.2 Arming TNT

Skills and knowledge we'll practice in this exercise:

- Arguments
- If statements

There's something about the setBlock() and setBlocks() methods that you might not know. They can take an extra argument. You're used to using these methods with arguments for co-ordinates and block type. There's also an extra argument at the end that sets the block state.

Each block has 16 states. Wool for example has a different colour for every state. TNT (block id 46) is explosive in block state 1. There are 16 states for every block, but not all of them are different. To set the state of a blocks we provide these functions with an extra argument after the others. The code below creates armed TNT:

```
4 mc.setBlock(10, 3, -4, 46, 1)
5
6 mc.setBlocks(11, 3, -4, 20, 6, -8, 46, 1)
```

Your mission is to create a function that checks whether a certain block is TNT. If it is, make the TNT explosive.

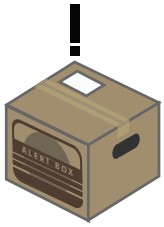
Instructions

Using the skeleton code below write your program:

```
1 # create a function, co-ordinates as arguments
2 # get the block type
3 # check block type is TNT
4 # create armed TNT if it is
5 # otherwise post "not TNT"
```

Once you've written the program to check whether a certain location is TNT, design some tests by calling the function to see if it works. Make sure you test the function on TNT blocks and non-TNT blocks

One way to test it is placing TNT at different locations and calling the function with their co-ordinates. You could also get the player's position and pass it to the function. This way you can test it by standing on the TNT.



ALERT: Once you have armed the TNT you must hit it to make it explode. Don't test your code on multiplayer as setBlock() doesn't work very well on multiplayer.

extensions

- Try finding out all the blocks and their different states.
- Increment states instead of just setting it to 1

7.1.3 Wool Colour

Skills and knowledge we'll practice in this exercise:

- arguments
- funcitons
- return
- If statements
- elif statements
- else statements

Wool (block id 35) has many uses in Minecraft, due to its different colours. As explained in the last exercise, it has 16 different colours that are accessed using an optional block state argument in the setBlock() and setBlocks() methods. However, it's difficult to remember these block states and why would you want to when a program can remember for you? Let's make a program that remembers the block states of wool.

Instructions

You'll need to do some detective work here. First things first, you'll need to find out the colours for each block state of wool are. Achieve this in whatever way you prefer. Here's one to start you off: pink is state 6.

Once you've worked out the block ids, create a function that returns the state value of a colour based on a string argument. For example providing the argument "pink" will return the value 6. It must return the correct value for each colour.

You'll also need to post a useful error message to chat if the argument is not a valid colour.

7.1.4 Turtle

Skills and knowledge we'll practice in this exercise:

- Functions
- Arguments
- Global variables

Logo is a programming language designed for learning. In logo you control an on screen turtle or robotic turtle that draws lines. To make the turtle move it is given commands like forwards 5, which will make it move five pixels forward. Likewise backwards 5 would make the turtle move backwards 5 pixels. In this exercise you'll create a program that works similar to the turtle in logo.

This exercise use global variables, which is not covered at Codecademy. When changing the value of a variable in a function Python needs to know whether it is specific to that function (local) or if it accessible to every other function (global). If the variable name exists outside of function in the main body of the code you must include the **`global`** operator alongside the variable in order to change its value in your function. For example:

```
1 x = 10
2
3 def doubleX()
4     global x
5     x = x * 2
```

Instructions

Build a program using functions that achieves the following list of requirements:

- The forward function moves the player a number of blocks forward along the x-axis
- The backward function moves the player a number of blocks backwards along the x-axis

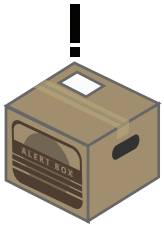
- The right function moves the player a number of blocks along the z-axis in a positive direction
- The left function moves the player a number of blocks along the z-axis in a negative direction
- The up function raises the player a number of blocks along the y-axis
- the down function lowers the player a number of blocks along the y-axis
- The penDown function sets the draw variable to true
- the penUp function sets the draw variable to false

For all of the above functions, the following must apply:

- If the draw variable is true, moving the player will create blocks
- If the draw variable is false, moving the player will not create blocks
- After a function runs it must wait 0.1 seconds before moving on

Once you have created a program that achieves the above functionality, call the functions to draw the following items:

1. Square
2. Three parallel lines
3. Smiley face



ALERT: If you're familiar with logo, you should notice our program works differently. Using right and left in Logo rotate the turtle by an angle. Our program doesn't do this. Instead these functions should move the turtle in a direction.

Extensions

- Add functions to change the colour of the pen
- Add a rubber mode that erases blocks instead of placing them

7.1.5 Import Block Module

Let's finish with something simple. The **`import`** operator allows you to use other Python modules to extend the capabilities of your program. You've in fact been using this for ages. The line at the start of everyone of your Minecraft programs uses the **`import`** operator:

```
1 import mcpi.minecraft as minecraft
```

This module comes with Minecraft and allows you to interface Minecraft with your program. Another module comes with Minecraft, the block module. Up until now you have used integers to represent the block type of blocks. The block module allows you to replace these integers with preset variables. For example the `AIR` variable replaces the integer 0 in the following code:

```
1 import mcpi.block as block
2 import mcpi.minecraft as minecraft
3 mc = minecraft.Minecraft.create()
4
5 mc.setBlock(0, 0, 0, block.AIR)
```

Using variables instead of integers makes block values easier to remember.

Instructions

In order to use the variables of the block module you will need to out what they are. Find the file that contains the block module. Hint: it's name is block.py and it is somewhere in the mcpi folder.

End of Exercises

7.2 Function syntax

Now we know the usefulness of functions, let's see how we use them.

function definition

statement

Functions are reusable blocks of code. They contain a def operator, a name and arguments.

Expression:

```
1 def functionName(arguments):
2     #body of the function goes here
```

Statement:

```
1 def addTwoNumbers(number1, number2):
2     sum = number1 + number2
3     print str(sum)
```

All function definition statements contain four things:

1. def operator
2. A function name (also called an identifier)
3. Optional arguments in brackets () followed by a colon
4. Body of the function

The def operator, which is an abbreviation of define, tells Python that you are writing a function.

The function's name is used to identify the function. Each function needs a different name, otherwise Python doesn't know which block of code you want to use. In the following example codeName is the name of the function:

```
1 def codeName(agentName):
2     print "Code Name: " + agentName
```

Arguments are values that are passed to the function when it is used. They tell the function what values to use for specific variables when it runs. In the above example `agentName` is an argument. Multiple arguments are separated by commas.

A function definition does not need to have arguments. You can leave the brackets empty to indicate that you don't want the function to accept ar-

guments. For example here is a function that doesn't take any arguments and simply prints "Hello":

```
1 def printHello():
2     print "Hello"
```

Don't forget the colon at the end of the line. The lines that follow the colon are the body of the function: the code that will run when the function is called.

The body of the function is always indented by one tab or four spaces, never a mixture of the two. A function can contain as many statements as you want. It can also include if statements and everything else we've covered so far. When you reach the end of the function code you stop indentation.

7.2.1 Calling a function

To use a function, also known as calling a function, you write the name of the function with any arguments it requires in brackets.

Calling a Function

expression

In order to use a function the code must call it. The statement should include the function name and any arguments it requires. Multiple arguments are separated by commas.

Expression:

```
1 functionName(arguments)
```

Statement:

```
1 addTwoNumbers(5, 3)
```

To call the codeName() and printHello() functions we defined earlier, we would use the following code:

```
1 codeName("Solid Snake")
2
3 printHello()
```

7.2.2 Return

There are two types of functions: those that return a value and those that don't return a value. So far we've created functions that don't return a value. Let's have a look at the ones that do return a value.

Returning a value from a function is very useful. For example we have a heart rate monitor that takes the pulse of a person. We can use a function to return data from the heart rate monitor. Without the return operator we would not be able to access data sent from hear rate monitor.

When making your own functions, you use the return operator to return the value of a variable. For example the following code returns a value that is the people argument plus 2:

```
1 def plusTwo(people):
2     return people + 2
```

.....

return

statement

The return statement is used within a function to return a value. It will return the value to the position the function was called and can be used like any other data.

Expression:

```
1 def functionName(arguments):
2     #body of function
3     return variable
```

Statement:

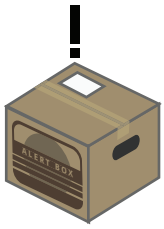
```
1 def addFullStop(string):
2     return string + ".”
3
4 #calling the funciton
5 sentence = addFullStop(”Hello”)
```

.....

To use a function that returns a value, you call it in a place that would expect a value. For example we'll call the plusTwo() function:

```
1 total = plusTwo(6) #value will be 8
```

Functions that return values can be used in statements to set the values of variables. They can be used anywhere that you are expected to put a value.



ALERT: The return operator determines whether a function will return a value or not. Functions that do not return a value cannot be used to set values. Instead they form a statement by themselves.

7.2.3 Multiple Arguments

We have already seen that you can use commas to separate multiple arguments in a function. What if you don't know how many arguments the function call will provide?

Well, Python has a very handy way of accepting an unknown number of arguments: an * operator at the start of the argument name when the function is defined.

We have a program to print the values passed to the function:

```
1 def createList(*items):
2     print items
```

It could be called like this...

```
1 createList("Grover", "Elmo", "Snuffy")
```

...this...

```
1 createList("Grover")
```

...or this...

```
1 createList("Hawk", "Fox", "Snake", "Hippo",
    ↪ "Monkey", "Frog", "Snake", "Jim")
```

The * actually tells the function to expect an list of unspecified length. We will cover lists in the lists chapter, which will make the * operator more useful.

Unspecified Number of Arguments *

operator

When used with arguments the * operator tells a function to expect any number of arguments.

Expression:

```
def functionName(*arguments):
    #body of function
    #optional return variable
```

Statement:

```
def printMembers(*members):
    print members

printMembers("Fox", "Slippy", "Falco", "Peppy")
```

7.3 Modules

Modules are reusable collections of functions. Functions are bundled together into modules so that similar functions are easily shared together.

For example you might have a module for images. The module most likely contains functions that load, save and modify images.

In this section we'll acquaint ourselves with the ways that we can include and use modules in our programs.

7.3.1 Import

In order to use the functions in a module you need to import them into your program.

A module that deals with maths is called the math module, surprisingly. Here's how we import it:

```
import math
```

import

statement

Import is used to include modules in Python programs.

Expression:

```
1 import moduleName
2 #Using a module function
3 moduleName.function(arguments)
```

Statement:

```
1 import math
2 #using a function in a module
3 print math.sqrt(4)
```

Once you've included the module in your program you can then use the functions in the module. After importing a module, to use a module's function you use dot notation. In other words you include the module name, a dot and the function that you want to use from that module.

For example square root (sqrt) is a function in the math module that returns the square root of an argument.

```
1 print math.sqrt(4) #prints 2
```

To find out all of the functions from a module you can use the pydoc module. This module contains all of the documentation on other modules. To access the documentation we import the pydoc module and use the help() function. The name of the module you want to find out about is provided as a string argument to the help() funciton. Here's an example to access the math module documentation:

```
1 import pydoc
2 pydoc.help("math")
```

7.3.2 from

Sometimes you only need one function from a module. To do this you use the from operator.

This allows you to access the function without prefacing it with the module name and dot notation i.e. you would just write function() instead of module.function().

. . . . . .

from

keyword

The form statement is used to import specific functions from a module, instead of all functions from that module.

Expression:

```
1 from moduleName import function
2 #Using a module function
3 function(arguments)
```

Statement:

```
1 from math import sqrt
2 #using a function in a module
3 print sqrt(4)
```

. . . . . .

For our math example you would change the code to this:

```
1 from math import sqrt
2 print sqrt(4) #prints 2
```

You can import more than one function from a module you using the from operator. Just simply separate the funciton names with a comma.

```
1 from math import sqrt, sin
```

7.3.3 Import All *

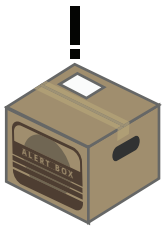
You can also import the entire library using the from module. This is achieved with the * in the place of the function's name.

```
1 from module import *
```

This removes the need to use dot notation and reference the name of the module as before. So we could use the all the funcitons from the math module without the use of dot notation:

```
1 from math import *
2 print sqrt(4)
3 print sin(6)
```

There is a risk to using the * operator.



ALERT: Sometimes different modules share the same function names. When this happens Python will get confused which module the function should be taken from.

It is therefore recommended not to use this method of importing every function modules, unless you are clear that there will be no conflicts.

7.4 Built-in Functions and Methods

Earlier we covered some string methods and functions. Methods are effectively the same thing as functions, with one slight difference, which we will discuss when we cover objects.

In this section of Codecademy we were introduced to four new methods:

- max()
- min()
- abs()
- type()

7.4.1 max()

The max() function returns the highest value (maximum) in a list of arguments:

```
1 highest = max(7,9,2,3) #value of 9
```

max()

function

Takes a list of numerical values as arguments and returns the highest value.

Expression:

```
1 max(values)
```

Statement:

```
1 highest = max(10, 100, 63)
```

7.4.2 min()

min()

function

Takes several numerical values as arguments and returns the lowest value.

Expression:

```
1 min(values)
```

Statement:

```
1 lowest = min(10, 100, 63)
```

To return the lowest value (minimum) in a list of arguments we use the min() function:

```
1 lowest = min(7,9,2,3) #value of 2
```

7.4.3 abs()

To return the number's distance from zero as a positive integer, we use the absolute function, abs():

abs()

function

Takes a single numerical value as an argument and returns its absolute distance from 0 as a positive integer. This makes the function useful for turning negative numbers into positive numbers.

Expression:

```
1 abs(value)
```

Statement:

```
1 absolute = abs(-100)
```

```
1 distance = abs(-231) #value of 231
```

7.4.4 type()

The type() function returns the variable type of a value, whether it's a string, integer, boolean, float or another type we haven't covered yet.

type()

function

Takes an argument and returns its data type.

Expression:

```
1 type(value)
```

Statement:

```
1 userInput = input("Enter a value: ")
2 print type(userInput)
```

There are many occasions where you will need to find out the data type of a value. You use the type() function for this.

For example:

```
1 print type(19)
```

This code will return the variable type in the following format: <type 'int'>. The type function is useful when checking two variables of the same data type.

Chapter 8

Lists and Dictionaries

Lists in the real world are a collection of items. You may be used to shopping lists or a list of instructions. They are used to remember a group of items or so that you can work through steps in a certain order. Lists in Python are very similar.

In Python lists are used to store a collection of data within a sequence. A list can store several types of data, including strings, numbers, booleans and even other lists.

Normally variables can only hold one value. Lists are useful as they allow you to store several values in a single variable.

Lists are also known as arrays. Referring to a list as an array is very common.

8.1 Minecraft Exercises

8.1.1 Glitching Sign

Skills and knowledge we'll practice in this exercise:

- lists
- for loop

Lists store several values in a single variable. This is useful for animations and movement. You're going to create a program that places a sign in front of the player and makes it spin around erratically.

As introduced in the previous set of exercises, each block has 16 states that can be accessed with an optional variable in the setBlock() and setBlocks() methods. Each of the the sign block's states represent the angle of the sign. So state 0 makes it face forward, state 1 makes it turn a bit, state 2 makes it turn more and so on until it's spun all the way around.

Our program will use a list of integers and a loop to make the sign turn erratically in front of the player.

Instructions

Complete the code below. Some code has been provided and the comments describe what else you need to complete. The states list needs to contain a list of integers for the sign to change angle. Use the description above to help you plan what you need to do.

```
#connect to Minecraft
import time
# get player position
# variables for sign position
x =
y =
z =
# list of sign states
states = [ ]

for state in states:
    mc.setBlock(x, y, z, 35, state)
    time.sleep(0.2)
```

Try out different values in the list. See if you can get the sign to rotate normally in one spin without glitching.

Extensions

- Alternate the list between the same two values to make the sign wiggle

- Make a group of signs spin in sync

8.1.2 Block by Numbers

Skills and knowledge we'll practice in this exercise:

- index positions

Painting by numbers is probably a better analogy.

Often for designers keeping things simple by using a limited number of colours and materials is essential for good design. Think of a logo, it's easier to remember simpler logos with only a few colours and elements than it is to remember complex logos.

In a similar way that designers use a limited colour palette, we're going to use a limited palette of blocks to create a simple picture

Instructions

Here is a list of 5 different block types.

```
1 blocks = [47, 20, 103, 81, 57]
```

Using index positions place the blocks to create the following sequences:

- bookshelf, glass, melon, cactus, diamond
- diamond, diamond, cactus, melon, bookshelf
- melon, glass, melon, glass, melon
- bookshelf, bookshelf, glass, bookshelf, bookshelf
- diamond, cactus, melon, glass, bookshelf

Make sure you place the sequences of blocks on the same co-ordinates.

Extensions

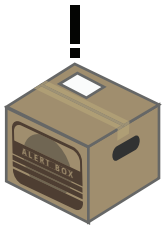
- Change the values of the items in the array

8.1.3 Team Camera

Skills and knowledge we'll practice in this exercise:

- lists
- list indexes

Playing together with other people in Minecraft is fun. Of course you can build things together. With the Minecraft Pi API you can also find out where the other players are and watch them. We'll create a program that allows you to follow other players with a camera.



ALERT: This exercise works best when you have several players connected to the same game world.

The `getPlayerEntityIds()` method returns a list of integers that identify each player currently in the game world. Here's the code used in a program that posts the number of players connected:

```
4 players = mc.getPlayerEntityIds()
5 mc.postToChat(str(len(players)))
```

A set of method exist that control the player's camera. The `setFollow()` method can be used to make the camera follow another player around the world. The player's ID must be provided as an argument in order to achieve this. For example:

```
4 playerId = 150
5 mc.camera.setFollow(playerId)
```

The `setNormal()` method sets the camera view to the normal view of the player. An argument is optional. If no argument is provided the camera will be set to the normal view of your player. When a player ID integer is provided as an argument the camera will change to the normal view of that player. For example:

```
4 mc.camera.setNormal()
```

Instructions

Using the `getPlayerIds` method get the list of all the connected players. Then using list indexes to access the player IDs, change the camera view to follow the second player in the list. Wait 10 seconds before changing the camera back to normal.

If you're playing by yourself set the camera to follow the first player in the list.

Extensions

- Try smashing blocks or placing new ones while in another player's normal view. See how annoying you can be.
- Cycle through the camera views of all of the other players.
- Surround every player with a building.

8.1.4 Dictionary of Wool

Skills and knowledge we'll practice in this exercise:

- dictionaries

In programming there are many different ways to achieve exactly the same thing. In some circumstances for example we can use a dictionary to serve the same purpose as in if statement.

Using a dictionary we are going to remake an exercise from earlier. On page [pagenumber] we used a function with if statements to return the different state IDs for wool based. A string was used to input the human readable colour e.g. ”red”.

Instructions

Here’s a tiny bit of code to start you off:

```
1 wool = {'pink': 6}
```

Complete the dictionary of wool colours and add the following functionality to the above code:

1. Takes a colour in a string as user input
2. Prints the integer value of a wool colour based on user input

Extensions

- What happens when you provide a string that isn’t a dictionary key?

8.1.5 Hacking a Friend’s Game

Skills and knowledge we’ll practice in this exercise:

- Networks
- Lists
- IP Addresses

You can play together with friends on Minecraft Pi. Players must be connected together on the same network and can join each other’s games using the Join Game option on the menu.

Even better than playing together with your friends on Minecraft Pi is hacking their games with the Minecraft API and Python. Effectively you'll be sending commands over the network to your friend's version of Minecraft as they're playing.

To complete this exercise you'll need to have several Raspberry Pi connected on a the same network.

In order to connect to another Raspberry Pi, you'll need its IP address. An IP address uniquely identifies each computer on a network, like a telephone number:

1. Get your friend to open a terminal `ifconfig`
2. Copy down the IP address, it should look similar to this `192.168.1.69`
3. In a terminal on your computer test the connection between your computer and your friend's with `ping [IP Address]` e.g. `ping 192.168.1.69`
    - If your terminal states that bytes have been received, the connection is working
    - If you get an error message, the connection is not working and you must fix it before you can proceed

If the connection between your computer and your friend's computer is working you can now connect to their Minecraft game. Make sure they're in the game world before proceeding.

Remember the two reusable lines of code that we use with every Minecraft Pi program? Copy the second line and give it a new variable name. This will be the connection to your friend's Pi. Include their IP address as a string argument in the create() method.

```
1 import mcpi.minecraft as minecraft
2 #local connection
3 mc = minecraft.Minecraft.create()
4 #friends connection, change IP address as
    ↪ appropriate
5 friendsGame =
    ↪ minecraft.Minecraft.create("192.168.1.69")
6
7 friendsGame.player.setTilePos(16,16,16)
```

You can now mess around with your friend's game. Just replace the `mc`

variable name that usually proceeds a Minecraft function with the variable name of your friend's connection.

Instructions

After testing the above, complete the following:

1. Create a list of all of the IP addresses of Minecraft Pi players on your network.
2. Connect to all of their games and store these connections in a list.
3. Using a loop to run any piece of code you want on all of the players
4. Use index positions to target specific people

Extensions

- Create multi-player games using this technique

End of Exercises

8.2 Lists

Lists are sequences of data. This data can be a variety of things including variables and even other lists.

8.2.1 Defining a list

Making a list is easy. You use the square brackets (also called parentheses) to define a list. Within the list you can have any number of items, even no items.

The entire list should be enclosed in square brackets and each item is separated by a comma.

Defining a list

data-type

A list is a series of values. They are enclosed in square brackets [ ] and separated by commas.

Expression:

```
1 [variable1, variable2, variable3]
```

Statement:

```
1 team = [”Pirate”, ”Monkey”, ”Robot”, ”Ninja”]
```

For example, we might want a list of ingredients for our noodle soup:

```
1 noodleSoup = [”water”, ”soy sauce”, ”spring
    ↪ onions”, ”noodles”, ”beef”]
```

You can create an empty list like this:

```
1 emptyList = []
```

You would use an empty list when you want to add values later in your program.

8.2.2 Accessing a list item

To access a value in a list, we reference the item’s position in the list, known as it’s index.

Using our noodle soup example we can access the first item in the list like this:

```
1 print noodleSoup[0] #prints the value ”water”
```

Notice that the first index in a list is 0. The second item is index 1, third is index 2 and so on. This is because computers count from 0 when using lists, whereas we are used to counting from 1.

You may remember indexes from the substring section on page [pagenumber]. This is exactly the same thing. Strings are a special type of list that

contains characters, so you can access their index position in exactly the same way.

List Index

data-type

Each item in a list is referenced with an index position. The index positions in a list start at 0. The value stored in each index can be accessed using square brackets [ ].

Expression:

```
1 listName[index]
```

Statement:

```
1 team = ["Pirate", "Monkey", "Robot", "Ninja"]
2 print team[1] #prints "Monkey"
```

8.2.3 Changing a list item

It is possible to change the value of an item in a list. To do this we use the item's index position and set it's value in exactly the same way we would set the value of a variable.

We want to change the beef in our noodle soup to chicken. The beef is fifth item our list so it has an index of 4. We can easily change it to chicken like so:

```
1 noodleSoup[4] = "chicken"
```

8.3 List Capabilities and Functions

Lists have a set of functions that allow you to manipulate them. These functions include common operations like adding an item to the list or deleting an item.

As we can change the contents of lists in Python we describe them as "mutable". Some other programming languages, such as Java, do not allow

Changing a list item

data-type

The values of list items can be changed using their index position and setting their value in the same way that you'd change a variable's value.

Expression:

```
listName[index] = value
```

Statement:

```
team = ["Pirate", "Monkey", "Robot", "Ninja"]
team[2] = "Snake"
```

you to modify lists once you create them, in which case you would describe the lists as immutable.

8.3.1 Adding an item

append()

method

The append() method adds a new item to the end of a list. The value of the item is included as an argument.

Expression:

```
"String".lower()
```

Statement:

```
lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
#OR
country = "BRAZIL"
country = country.lower() #value of "brazil"
```

You can add an item to the end of a list using the .append() function.

Our noodle soup would be nice with some vegetables. To do this we simply use the .append() function:

```
1 noodleSoup.append("vegetables")
```

The noodle soup list now contains the a vegetables string as the last item in the string.

8.3.2 List Length

.....

len()

function

When a list is used as an argument, the len() function returns the number of items in that list.

Expression:

```
1 len(list)
```

Statement:

```
1 team = ["Pirate", "Monkey", "Robot", "Ninja"]
2 print len(team) #value of 4
```

.....

You can find out the number of items in a list by using the len() function. For example:

```
1 noodleSoup = ["water", "soy sauce", "spring
    ↪ onions", "noodles", "beef", "vegetables"]
2 print len(noodleSoup) #prints 6
```

When getting the length of a list, the computer counts from 1, not 0. This is unlike when accessing items in a list using indexes, which counts from 0.

8.3.3 List Slicing

We can access segments of a list using list slices. Slices return the items in a list from one index position to another. For example we might want to

list slice

operator

List slicing is used to access sections of a list, without access every item. To slice a list the : operator is used.

Expression:

```
listName[startIndex:cutOffIndex]
```

Statement:

```
team = ["Pirate", "Monkey", "Robot", "Ninja"]
eliteTeam = team[0:2]
#value of ["Pirate", "Monkey"]
```

access all of the items from the third index position to the end and ignore all other items.

To do this you use the normal syntax of a list with a colon. The colon goes between two values. the first item states which item will be the first in the new list. The second value is the index that the new list must cut off.

How would we print the second, third and fourth items (index positions 1, 2 and 3) in the noodle soup list? Simple:

```
print noodleSoup[1:4]
```

The value at the cut off index will not be included in the new list. It is exclusive. In other words the item at this position will not be included; the slice will stop before it reaches the item at this position. For example, if you want the last item in the list to be index number 5, you would use 6 for the second value in your slice.

Slicing From Start or End

By leaving out one of the values in a list slice, you can either slice from the start or to the end of the list.

To slice from the start you omit the first value:

```
firstHalf = noodleSoup[ : 3]
```

To slice to the end you omit the second value:

```
1 secondHalf = noodleSoup[3 : ]
```

8.3.4 Searching

You can find the index position of a value in a list using the index() method.

.....

index()

method

The index() method is used to search for the value of an item in the list. It takes the value that you want to find as an argument and returns its index position. If there is no item in the list with that value you will receive an error.

Expression:

```
1 listName.index(value)
```

Statement:

```
1 team = ["Pirate", "Monkey", "Robot", "Ninja"]
2 search = team.index("Robot")
3 #value of 2
```

.....

For example searching for the value "noodles" in our noodle soup list will return the value 3 as it is in index 3 of our list:

```
1 noodleSoup = ["water", "soy sauce", "spring
    ↪ onions", "noodles", "beef", "vegetables"]
2 whereAreTheNoodles = noodleSoup.index["noodles"]
```

If the item you are searching for does not exist in the list you will get an error.

8.3.5 Inserting an Item

It is possible to insert an item into a list. This will place it between two existing items.

insert()

method

To insert an item anywhere in a list you use the insert() method. This method takes the index position and value of the item you want to insert as arguments.

Expression:

```
1 listName.insert(index, value)
```

Statement:

```
1 team = ["Pirate", "Monkey", "Robot", "Ninja"]
2 team.insert(2, "Snake")
```

To insert an item into a list we use the .insert() function. This function takes two arguments, the index position where you want to insert the item and the value that you want to insert.

For example, we want to add pepper into our noodle soup list, in the third index position:

```
1 noodleSoup.insert(3, "pepper")
```

The updated list will hold the following values after the insert:

```
1 ["water", "soy sauce", "spring onions", "pepper",
    ↪ "noodles", "beef", "vegetables"]
```

8.3.6 Removing an Item

Sometimes you want to get rid of an item in a list. We use the .remove() function for this. The index value goes in the function as an argument.

For example we want to remove the beef item in index position 5 of our noodleSoup list:

```
1 noodleSoup.remove(5)
```

Use this in combination with the .index() function if you want to find the index position of an item and then remove it:

remove()

method

To remove an item from a list, the remove method is used. The index position of the item to be removed needs to be provided as an argument.

Expression:

```
1 listName.remove(index)
```

Statement:

```
1 team = ["Pirate", "Monkey", "Robot", "Ninja"]
2 team.remove(2)
```

```
1 beefPosition = noodleSoup.index("beef")
2 noodleSoup.remove(beefPosition)
```

8.3.7 Looping through a list

Accessing each item in a list is a very common task in programming. You could copy and modify the code for each item in the list, however this is inefficient and you will have to change the code every time you change the length of the list.

A for loop can repeat the same block of code on every item of a list. This allows reuse of a single piece of code without the need to copy and modify it for every item in a list. To repeat the same code for every item in a list, the for loop comes in very handy.

If we wanted to print every item in our noodle soup list we would use the following code:

```
1 noodleSoup = ["water", "soy sauce", "spring
    ↪ onions", "pepper", "noodles", "beef",
    ↪ "vegetables"]
2
3 for ingredient in noodleSoup:
4     print ingredient
```

You use the for operator to tell Python you are using a loop.

for loop

statement

For loops exist so that you can repeat the same code several times. They are useful with lists as they can access each item in a list and execute the same code on each item.

Expression:

```
for item in list:
    #body of loop
```

Statement:

```
team = ["Pirate", "Monkey", "Robot", "Ninja"]

for member in team:
    print member
```

After the for keyword we have a variable. This variable represents the item that the loop is currently using. The value will change every time the loop starts again, until it has looped through each item in the list. The first time the loop executes the value will be the same as the item in index position 0, the second time it will be the value in index 1, third time is index 2 and so on.

The in operator and the list name at the end of the statement tell the Python which list you are using.

The loop will execute once for each item in the list.

8.3.8 Sorting a list

The sort() method rearranges a list values into ascending order.

To sort a list containing letters into we can use the sort() method:

```
letters = ["b", "d", "c", "a"]
letters.sort()
```

The order of the list will now be ["a", "b", "c", "d"].

sort()

method

To sort a list into ascending order you use the .sort() function. This will sort numbers based on their numerical value and strings in order of the alphabet.

Expression:

```
1 list.sort()
```

Statement:

```
1 team = ”Pirate”, ”Monkey”, ”Robot”, ”Ninja”]
2 team.sort()
```

8.3.9 Adding Together Items in a List

Python has a built in function that adds all of the items in a list. It is called the sum() function.

For example:

```
1 numbers = [1, 29, 7]
2 total = sum(numbers) #value of 37
```

sum()

function

The sum() function adds together all numerical values in a list

Expression:

```
1 sum(list)
```

Statement:

```
1 score = [1, 5, 8, 3]
2 total = sum(score) #value of 17
```

8.4 Dictionaries

Dictionaries are a type of list that uses a different approach to the kind we've been introduced to. Instead of using an index to identify items, dictionaries identify items a set of keys that are defined by the programmer.

Like lists dictionaries are mutable, meaning that their content can be changed.

Dictionaries are lists that use programmer defined keys to reference the items in the dictionary.

8.4.1 Defining a Dictionary

To define a dictionary you use a pair of curly brackets around the items in the dictionary.

For example you can use a dictionary to describe a person. You can use indexes like "name" and "favouriteAnimal" to store information about the person like so:

```
1 person = {'name;: "David", 'age': 42,
    ↪ 'favouriteAnimal': "Snake", 'favouritePlace':
    ↪ "Inside a cardboard box"}
```

.....

Dictionary

list

Dictionaries are a type of list, where each item has a key and a value. The key uniquely identifies each value in the dictionary.

Expression:

```
1 {'key': item, 'key': item}
```

Statement:

```
1 pilots = {'Wing': "Heero", 'Wing Zero': "Quatre",
    ↪ 'Heavy Arms': "Trowa"}
```

.....

A key uniquely identifies each item in a dictionary. Each key is paired with a value using a colon. Items in the dictionary are then separated by commas.

You may have noticed that using dictionaries makes it easier for the programmer to understand what each item in the list represents, i.e. a key like "name" is more expressive of its purpose than an index number like 0.

8.4.2 Accessing Items in Dictionaries

To access the value of an item in a dictionary you use square brackets and a key. The key must be a string, that is it must be in quotation marks.

Accessing a Dictionary Item

list

The values of items in a dictionary are accessed using their key values.

Expression:

```
1 dictionary['key']
```

Statement:

```
1 pilots = {'Wing': "Heero", 'Wing Zero': "Quatre",
    ↪ 'Heavy Arms': "Trowa"}
2 print pilots['Wing']
```

For example to access the value of the name key in the person dictionary:

```
1 agentName = person['name'] #value of "David"
```

8.4.3 Changing/Adding an Item with a Dictionary

To change an item to a dictionary it's pretty simple. You use square brackets with a key to access the item and set it as you would a normal variable. Adding a new item is also possible using this approach.

We want to change the value of the age item in the person dictionary:

Changing/Adding a Dictionary Item

list

The values of dictionary items are changed by accessing their key and changing the value as you would a variable, i.e. with an =.

Expression:

```
1 dictionary['key'] = value
```

Statement:

```
1 pilots = {'Wing': "Heero", 'Wing Zero': "Quatre",
    ↪ 'Heavy Arms': "Trowa"}
2 pilots['Epyon'] = "Zechs" #adds a new pilot
3 pilots['Wing Zero'] = 'Heero' #changes pilot
```

```
1 person['age'] = 43
```

We also want to add a new item called location with the value "USS Discovery":

```
1 person['location'] = "USS Discovery"
```

8.4.4 Deleting Items in Dictionaries

Sometimes you want to delete an item in a dictionary. You use the del operator to do this.

For example, we want to delete the favouriteAnimal item in our person list:

```
1 del dictionaryName['favouriteAnimal']
```

del

operator

The del operator is used to delete an item in a dictionary.

Expression:

```
1 del dictionary['key']
```

Statement:

```
1 pilots = {'Wing': "Heero", 'Wing Zero': "Heero",
    ↪ 'Hevay Arms': "Trowa"}
2 del pilots['Wing']
```

Chapter 9

Functions and Lists

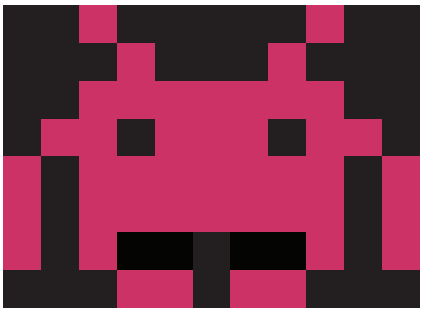
We've learned how to use functions, we've learned how to use lists. Let's combine the two.

In this chapter we'll create our own functions to perform common tasks on lists. We'll also introduce some useful, pre-made functions that are used with lists.

9.1 Minecraft Exercises

9.1.1 Pixel Art

In this exercise you'll use multi-dimensional to create pixel-like images in Minecraft. This may sound mind-blowing, but it's actually quite simple. Take a simple pixel image like a space invader:



As you can see we have a square of smaller squares. Each smaller square is called a pixel. A pixel in our image is either one of two colours, so we

can represent each square with a value, let's say 1 and 0:

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

These values of 1s and 0s can then be stored in a list for each row. Each list containing a row can also be stored as an item in a larger list.

So a pixel image can be stored as a multi-dimensional list. Let's see how we can use this knowledge in Minecraft to create our own pixel art.

Instructions

Create a multi-dimensional list that contains the data for the above space invader. You will need to nest one for loop inside of another for loop in order to work through each item in the rows list and the items in each row.

Extensions

- Multi-colour pixel images

9.1.2 Shadow Castle

Skills and knowledge we'll practice in this exercise:

- for loop

Bad guys in video games hang out in evil looking buildings. Lava and obsidian are pretty evil. Copying is even more evil. Let's copy one of your buildings and make it more evil looking.

You should really use the `getBlocks` method that comes with the Minecraft API. However, `getBlocks()` doesn't work at the moment. We'll need to make our own version to replace it. Here is our version that does the same thing:

```
1  def getBlocks(x1, y1, z1, x2, y2, z2):
2      xhigh = max(x1, x2)
3      xlow = min(x1, x2)
4      yhigh = max(y1, y2)
5      ylow = min(y1, y2)
6      zhigh = max(z1, z2)
7      zlow = min(z1, z2)
8
9      blocks = []
10     for x in range(xhigh - xlow + 1):
11         blocks.append([])
12         for y in range(yhigh - ylow + 1):
13             blocks[x].append([])
14             for z in range(zhigh - zlow + 1):
15                 blocks[x][y].append([])
16                 block = mc.getBlock(xlow + x, ylow
                       ↪ + y, zlow + z)
17                 blocks[x][y][z] = block
18     return blocks
```

Instructions

This exercise is meant to really challenge you. You need to work out how to achieve a set of requirements with code. You have all of the knowledge necessary, you just need to work out how to apply it.

The requirements of your program are:

- Copy a building and store in lists
- Replace all stone block in the copy with obsidian and water with lava
- Place the copy at a new location on the map

You may notice a bug with blocks that have different states, like stairs and wool. Modify the `getBlocks()` function that we've provided for you to fix this bug.

End of Exercises

9.2 Using Functions with Lists

As we have already covered, making your own functions means you can reuse code; saving you time and effort. Lists can be used with functions in a number of ways.

Lists can be provided as arguments, they can be used with loops nested in functions and functions can be used to modify the items in a list.

9.2.1 Lists as Arguments

When using functions we can pass arguments to them. This allows us to provide data to the function, which can change how the function behaves.

It is possible to use lists and dictionaries in functions as an argument type. Remember back to when we were introduced to arguments in functions?

The syntax for using a list as an argument is exactly the same as a normal argument. As is the syntax for using dictionaries as an argument.

Let's make a basic function that prints returns the item in index position 1 of a list:

```
1 def secondItem(list):
2     return list[1]
3
4 animals = [”crow”, ”mantis”, ”snake”]
5 print secondItem(animals) #prints mantis
```

9.2.2 Loops and Lists in Functions

Loops can be used in functions to access each item within a list. The loop works the same as the loop would outside of a function.

Here we have a simple function that uses a loop to print each item in a list individually:

```
1 def printList(list):
2     for item in list:
3         print item
4
5 animals = [”crow”, ”mantis”, ”snake”]
6 printList(animals)
```

9.2.3 Modifying Each List Item

Changing the values of items in list is a common task in programming. Normally you would achieve this by accessing the index position of an item and changing it's value like so:

```
1 list[index] = newValue
```

Changing the value of every item in a list is also very common.

We use a for loop to modify every item in a list, however we need to change the for loop slightly to achieve this. Currently the for loop uses the value of each item in the list every time it loops. To change the value we need to

use the index position of the item instead of its value. The range() function combined with the len() function allows us to do this.

The range() function creates a list of integers between two arguments. The start index and the position it will cut off. The list it creates can be used to represent the index positions of our list.

range() - Two Arguments

function

The range function creates a list of integers. It can take one, two or three arguments. When two are provided the first will determine the first value in the list, the second is where the list will cut off.

Expression:

```
range(startIndex, cutOffIndex)
```

Statement:

```
#prints the values 0 to 9
for index in range(0,10):
    print index
```

For example...

```
aList = range(0, 5)
```

...would create the list:

```
aList = [0, 1, 2, 3, 4]
```

To create a loop that uses the index positions of a list we use the following code:

```
for item in range(0, len(listName)):
    #body of for loop
```

This way the for loop sets the item variable to the index position of the list instead of setting it to its value. For example we could double the value of every item in a list like so:

```
numbers = [3, 6, 2, 8]
for index in range(0, len(numbers)):
    numbers[index] = numbers[index] * 2
```

This would change the values of the numbers list to:

```
[6, 12, 4, 16]
```

9.2.4 Functions to Modify Each Item in a List

You can enclose a loop within a function to modify every item in a list, making the code reusable. Let's concatenate the string "codename: " to the start of every item in our list, then return the list:

```
def codeName(list):
    for index in range(0, len(list)):
        list[index] = "codename: " + list[index]
animals = ["crow", "foxhound", "snake"]
animalsWithCodenames = codeName(animals)
```

9.2.5 More On range()

.....

range() - One Argument

function

The range function creates a list of integers. It can take one, two or three arguments. When one argument is provided it will determine the integer value that the list will cut off.

Expression:

```
range(cutOffValue)
```

Statement:

```
#prints the values 0 to 9
for index in range(10):
    print index
```

.....

The range() function, as we know, creates an list of integers. When used with a for loop and the len() function, It is useful for representing the index positions of another list.

The range() function can be provided with either 1, 2 or 3 arguments. This changes how the function works.

When using one argument the function will start the list with the the value 0 and cut off at the value of the argument provided.

For example:

```
1 agents = range(5) # creates list [0, 1, 2, 3, 4]
```

When using two arguments the list will start at the first argument provided and cut off before the second argument. For example:

```
1 agents = range(2, 5) \# creates [2, 3, 4]
```

With three arguments, the third argument states the size of each increment.

.....

range() - Three Arguments

function

The range function creates a list of integers. It can take one, two or three arguments. When three argument are provided, the first will be the starting value, the second the cut off value and the third is the increment between each value.

Expression:

```
1 range(startValue,cutOffValue,increment)
```

Statement:

```
1 #prints the values 0,2,4,6,8
2 for index in range(0,10,2):
3     print index
```

.....

In other words the value of each new item is usually bigger than the previous value by 1. By changing the increment you can make each next item bigger by the value of the increment. For example this list adds 2 to the previous value to get the next value. We say it increments by two:

```
1 agents = range(3, 10, 2) \# creates [3, 5, 7, 9]
```

9.2.6 Converting a List into a String

Using the join() method you can combine all of the items in a list into a single string.

Join

method

The join function combines all of the items in a list into a string. It takes the list you want to use as an argument. The character that you want to divide each item in the new string precedes the method using dot notation.

Expression:

```
"divider".join(list)
```

Statement:

```
shoppingList = ["shoes", "hat", "trousers"]
print "/".join(shopping)
#prints shoes/hat/trousers
```

The first part of the expression states the divider you want to use between each string, for example a space between each item would be " ", or if you want to use a comma ",". You then use dot notation and the join function. The list you want to join is used as an argument by the join function.

For example:

```
cats = ["fluffy", "charles", "igor", "ginger"]
print " + ".join(cats)
```

This will print "fluffly + charles + igor + ginger".

9.2.7 Splitting a String into a List

It is possible to split a string into a list of strings. To do this we use the split() method on a string. It will return the string as a list.

By default the function will split the string using spaces. It is possible to split the string using another character by providing it as an argument.

Split

method

The split() method splits a string into a list. When no arguments are used the method creates a new item from the string every time it finds a space. When a string argument is provided the method will split the main string whenever it finds the argument string.

Expression:

```
1 "string".split("divider")
```

Statement:

```
1 sentence = "Do a barrell roll"
2 myList = sentence.split()
3 #creates a list with values:
4 #["Do", "a", "barrell", "roll"]
```

For example using "," as an argument would split the string into a new item every time a comma is found in the string.

For example:

```
1 instructions = "Strip the paint/sand the
    ↪ wood/apply a primer/apply the paint"
2 instructionsAsList = instructions.split("/")
3 #value of:
4 ["Strip the paint", "sand the wood", "apply a
    ↪ primer", "apply the paint"]
```

9.3 Using Multiple Lists

Within your program you can use multiple lists together for a number of reasons. It is possible to include lists within lists, combine lists into a single list and use an undefined number of lists as arguments with functions. We'll cover these uses for lists in this section.

9.3.1 Multi-dimensional Lists

Lists can contain any variable type, including other lists.

Think of a list like a number of cardboard boxes. Inside the first box there is a snake, the second box contains a banana and the third a robot. In this example each box represents an index of a list and the item inside it is the value. Boxes can also contain boxes. In the first box we now have 3 boxes, each with a different animal inside. In the same way boxes can contain other boxes, lists can contain other lists.

Here we represent each box from the above example as a list containing another list:

```
1 boxes = [["Snake", "Cat", "Bear"], "banana",
    ↪ "robot"]
```

To access the value of a list with a list we use two sets of square brackets. The first set of brackets contains the index position of the outer list and the second set accesses the index position of the inner lists. To print the "Cat" item in the above example we would use the following:

```
1 print boxes[0][1]
```

Any number of lists can be nested within other lists. You have just seen a two-dimensional list. There are also three-dimensional lists and so on. Lists within can be disorientating so it is recommended you master two-dimensional lists before attempting anything more complex.

9.3.2 Joining Two Lists

When adding two numbers or combining two strings we use the + operator. The + operator can also be used to join two lists.

By joining the two lists we create a new list that contains all of the elements from both lists.

Say we have a list of electronic parts and we want to join it with another list of electronic parts to create a new list:

```
1 gps = ["gps module", "circuit board", "wires"]
2 arduino = ["arduino", "wires", "battery", "jack"]
3 tracker = gps + arduino
```

Multi-Dimensional List

data-type

A mutli-dimensional list is a list within a list. It is defined by by putting one list inside of another. The values of list items within the inner list are accessed using two or more sets of square brackets. The first set of brackets represents the outer list and the second set is the inner list.

Expression:

```
1 list[][]
```

Statement:

```
1 team = [["Jim", "Cat", "Milk"], "Monkey", "Robot",
    ↪ "Ninja"]
2 print team[0][2] #prints Milk
3 print team[2] #prints Monkey
```

Joining Lists +

operator

The + operator can be used to join two lists.

Expression:

```
1 list + list
```

Statement:

```
1 clothesList = ["shoes", "hat", "trousers"]
2 foodList = ["cake", "butter", "fish"]
3 shoppingList = clothesList + foodList
```

The tracker list contains all items from both lists:

```
1 ["gps module", "circuit board", "wires",
    ↪ "arduino", "wires", "battery", "jack"]
```

A function to combine two lists would look like this:

```
1 def combineLists(list1, list2):
2     return list1 + list2
```

9.3.3 Using an Undefined Number of Lists

You may remember the use of the * operators when defining arguments in functions.

The * operator tells the function to expect an undefined number of arguments:

```
1 def functionName(*argument):
2     #body of function
```

When the function receives the arguments it converts them into a single list.

It is possible to use the * operator to pass a function any number of lists. For example:

```
1 def packageElectronics(*list):
2     return list
3
4 gps = [”gps module”, ”circuit board”, ”wires”]
5 arduino = [”arduino”, ”wires”, ”battery”, ”jack”]
6 wireless = [”Electric Imp”, ”April”]
7
8 parcel = packageElectronics(gps, arduino, wireless)
```

The value returned by this function is the list argument without any changes made to it. The value it returns in this example would be:

```
1 [[”gps module”, ”circuit board”, ”wires”],
    ↪ [”arduino”, ”wires”, ”battery”, ”jack”],
    ↪ [”Electric Imp”, ”April”]]
```

If you look carefully you will see that there is one list with three lists inside of it. The syntax is not different from the syntax of a regular list: each list starts with [all items are seperated by commas and the each list ends with].

This is an example of a list within a list. A multi-dimensional list.

Chapter 10

Loops

Loops make it easy to repeat the same piece of code several times.

They have slightly different benefits to functions: functions allow you to reuse pieces of code whereas loops allow you repeat pieces of code.

You have already encountered for loops in the lists section. In this chapter we will cover for loops in more depth as well as introduce a new type of loop.

Iteration and to iterate are other common ways of saying that loops repeat.

10.1 Minecraft Exercises

10.1.1 Midas Touch

Skills and knowledge we'll practice in this exercise:

- while loop
- nested if statement

Midas is a king of legend. Everything he touched turned to gold. You are going to write a program that changes every block below the player to

gold. However, not every block will turn to gold. You need to make sure air and water stay the same. The gold block has a value of 41.

Instructions

Copy and complete the following code using the comments as a guide:

```
# connect to Minecraft
while True:
    # get the player's position
    # if the block below the player is not water
      ↪ or air:
        # change the block to gold
```

To exit the infinite loop you need to stop the program. This can be achieved using control + c in a terminal or the stop button in some IDEs.

10.1.2 Tree Fighter

Skills and knowledge we'll practice in this exercise:

- while loops
- for loops
- variables
- addition

The Minecraft `pollBlockHits` method returns a list of blocks that the player has hit with their sword using the right click on their mouse. As it returns a list we can use this method with a for loop to run some code for every block hit.

As the for loop will end once it has completed all of the items in the block hits list we also need to use a while loop to make the for loop repeat a few times every second. The basic code looks like this:

```
import mcpi.minecraft as minecraft
import time
mc = minecraft.Minecraft.create()

```

```
5 while True:
6     hits = mc.events.pollBlockHits()
7     for hit in hits:
8         print str(hit.pos.x) + ", " +
            ↪ str(hit.pos.y) + ", " + str(hit.pos.z)
9     time.sleep(0.1)
```

This will print out the position of any block hits and the type of block you have hit. Sleep is included so that the loop does not repeat too quickly and make the game run really slowly.

Instructions

Your task is to write a mini-game where the player has to right click on twenty pieces of tree wood with their sword.

Modify the above code so that

- the while loop stops after the player has right clicked on twenty pieces of tree wood
- the number of wood hits is posted to chat after every hit
- after twenty hits a congratulations message is posted to chat and the program stops
- adapt it so that the player can delete blocks even when the world is immutable

Extensions

- Randomly change the block type when you hit a block
- The player can currently cheat by clicking on the same piece of wood several times. Keep a record of which blocks the player has hit and stop them from repeating the same block.

10.1.3 Chat with a Loop

Skills and knowledge we'll practice in this exercise:

- while loop
- break statement
- conditions
- input/output

Earlier in the book when you were introduced to strings, input and output, you were given the task to create a program that posts the user's message to chat. Although this program was really useful, it was quite annoying that you had to rerun the program every time you wanted to post a new message.

In this exercise you'll improve your chat program using a while loop so that users can post as many messages as they want without the need to restart the program.

Instructions

Create a chat program to meet the following requirements:

- User name requested at the start of the program
- User can input message from the terminal
- Posts chat message from the users to Minecraft
- User name is included on all messages
- Program repeats request for message
- Notifications are posted when the user joins and leaves the chat
- When "/exit" is input the user will exit the chat

For hints on the Python concepts you'll need to use, look in the exercise objectives.

Extensions

-

10.1.4 Pyramid

Skills and knowledge we'll practice in this exercise:

- Nested for loops

Loops are really useful for creating structures and building. You can build really complex things by using code. For example the following code builds a triangle:

```
1  import mcpi.minecraft as minecraft
2  mc = minecraft.Minecraft.create()
3
4  pos = mc.player.getTilePos()
5  x = pos.x + 2
6  y = pos.y
7  z = pos.z
8
9  base = 9
10 height = 0
11 sandstone = 24
12
13 while base - (2 * height) > 0:
14     for block in range(base - (2 * height)):
15         mc.setBlock(x + block + height, y +
               ↪ height, z, sandstone)
16     height += 1
```

Instructions

Modify the triangle code so that it builds a Pyramid in the Minecraft world.

Extensions

- hollow it out and include floors every 5 blocks.

10.1.5 Hot and Cold

Skills and knowledge we'll practice in this exercise:

- while loop
- conditions
- if statements
- elif statements
- else statements

Hot and cold is a game you might have played as a child. The idea is that your friend hides an object and you must find it. Your friend gives you hints based on how far away from the object you are. The closer you are the hotter you are and when you're further away you're colder. On fire is means you're right next to the object. Freezing means you're really far away.

Instructions

Using your knowledge of Python and the Minecraft Pi API, create a program that reacreates the Hot and Cold game in Minecraft. The player's objective is to find and smash a diamond block that has been placed randomly in the game world.

You program must end when the player finds and smashes the diamond block. Use Pythagorus to calculate the player's distance to the diamond block.

The code to place a block in a random location has been written for you. It makes sure the diamond block is not placed underground.

```
1 import mcpi.minecraft as minecraft
2 import math
3 import time
4 import random
```

```
5 mc = minecraft.Minecraft.create()
6
7 tileX = random.randint(-127, 127)
8 tileZ = random.randint(-127, 127)
9 tileY = mc.getHeight(tileX, tileZ)
10
11 diamond = 57
12 block = diamond
13 mc.setBlock(tileX, tileY, tileZ, diamond)
14 mc.postToChat("Block set")
```

Extensions

- Hide blocks of dynamite across the map and give them 5 minutes to find them and disarm them using hot and cold

10.1.6 Adapt Exercises

Several exercises from previous lessons required you to rerun a program every time you wanted it to do something. These exercises would benefit from a infinite for loop. Here is a list of exercises that you can adapt to loop:

- Set Block Below Player
- Chat
- Swimming
- Bring us a Shrubbery
- Take a Shower
- Secret Passage
- Arming TNT
- Glitching Signs
- Team Camera

End of Exercises

10.2 While Loop

While loop

loop

While loops iterate over a block of code as long as a condition is True.

Expression:

```
1 while conditon:
2     #body of while
```

Statement:

```
1 userPassword = raw_input("Enter Password: ")
2 password = "Mistletoe"
3 while password != userPassword:
4     userPassword = raw_input("Enter Password: ")
5 print "Password accepted"
```

When the condition is True the while loop will iterate. The while statement works in the following steps:

1. Checks whether the condition is True
2. If the condition is true
   (a) Execute the body of code
   (b) Repeat step 1
3. If the condition is False
   (a) Ignore the body of code
   (b) Continue to the line after the while loop block

While loops are used to repeat blocks of code. They will repeat as long as a condition is True.

This is similar to an if statement. However, the code in the if statement will only execute at most once, whereas the while loop can iterate many times.

For example, this code uses a while loop to print the numbers 1 to 5:

```
1 count = 1
2 while count <= 5:
3     print count
4     count = count + 1
```

Any code you want can go inside the body of the loop, including other loops.

This diagram explains each iteration of the while loop: [Draw a flowchart]

10.2.1 Boolean Operators and While Loops

Boolean operators, like and, or and not, can be used with a while loop when you want the loop to use more than one condition.

For example this loop will iterate while the user has not input the correct password and has had 3 or less attempts to input the correct password:

```
1 password = "cats"
2 input = raw_input("Please enter the password")
3 attempts = 1
4
5 while input != password and attempts <= 3:
6     attempts + 1
7     input = raw_input("Incorrect. Please enter the
        ↪ password")
```

10.2.2 Avoiding Infinite Loops

It is very important that the boolean condition will eventually become False, otherwise the loop will iterate forever and your computer may crash.

There are also some instances where you may want an infinite loop. For example video games use an infinite loop to check user input.

Whether or not you meant to create an inifinite loop, there is a way to get out of it. Pressing control-c while in the command line that is running your program will exit the program and stop it executing.

10.2.3 Break

Sometimes you will want your code to immediately exit a while loop instead of waiting for the condition to become True. The break statement allows you to do this.

break

operator

The break statement is used in a while loop to immediately exit the loop.

Expression:

```
while conditon:
    #body of while
    break
```

Statement:

```
userPassword = raw_input("Enter Password: ")
password = "Mistletoe"
count = 0
while password != userPassword:
    userPassword = raw_input("Enter Password: ")
    count += 1
    if count > 4:
        break
print "Password accepted"
```

The break statement immediately exits a while loop.

It is common to put the break statement within an if statement nested within the loop. This will immediately stop the loop's execution when the if condition is met. For example this code will loop infinitely, ask the user

for input and print that input. The only way to stop the loop is to type "exit":

```
while True:
    userInput = raw_input("Enter a command")
    if userInput = "exit":
        break
    print userInput
```

10.2.4 while/else

Like an if statement, the while loop can be used with an else statement.

.....

else

statement

When used with a while loop an else statement will only execute its body when the while loop's condition is False. It will not execute when a break statement is used.

Expression:

```
while conditon:
    #body of while
else:
    #body of else
```

Statement:

```
temperature = raw_input("Enter the temperature: ")
while temperature < 100:
    print "Water is not boiling"
    temperature = raw_input("Enter the
      ↪ temperature: ")
else:
    print "Water is boiling"
```

.....

The else statement will execute when the condition of a while statement is False. Unlike the body of a while statement, the else statement will execute once and only once. For example:

```
message = raw_input("Please enter a message")

while message != "\exit":
    print message
else:
    print "User has left the chat"
```

When the break statement is used, the body of the else statement will not execute.

10.3 For Loops

We have already been introduced to for loops in the lists section. We'll recover the basics and go into more depth.

.....

for

loop

The for loop iterates through items in a list.

Expression:

```
for item in list:
    #body of code
```

Statement:

```
horsePrices = [1, 8, 5, 4]
total = 0
for horse in horsePrices:
    total += horse
```

.....

The for loop iterates through each item in a list until the end of the list.

Lists can contain any number of items of any data type. The for loop will iterate through each one in index order.

To generate a list containing indexed integers we use the range() function (section [number] to refresh your memory).

10.3.1 Strings as Lists

Strings can be treated like lists. We can access individual characters in a string using their index.

This means we can manipulate strings with for loops like we would a list. The following code uses a for loop to replace every letter a in string with the letter u:

```
orignalString = "cats"
newString = ""

for letter in originalString:
    if letter == "a"
        newString = newString + "u"
    else
        newString = newString + letter
print newString
```

The value of newString after the for loop has executed would be "cuts".

10.3.2 Looping Over a Dictionary

It is also possible to use a for loop with a dictionary.

When using a dictionary with a for loop the syntax is the same, however the key value will be used as the variable each time the loop iterates.

For example, the following code prints the for loop's variable each time the loop iterates. In this case it will print the key of each item in the dictionary:

```
inventory = {'tranquilisers': 5, 'rations': 2,
    ↪ 'boxes': 1}

for key in inventory:
    print key
```

This example will print:

```
tranquilisers
rations
boxes
```

To print the value access the value associated with each you need to use the dictionary[key] syntax as we covered earlier. Let's change the code so it prints the value each time with the key:

```
1 inventory = {'tranquilisers': 5, 'rations': 2,
    ↪ 'boxes': 1}
2
3 for key in inventory:
4     print key + " " + inventory[key]
```

This example will now print:

```
1 tranquilisers 5
2 rations 2
3 boxes 1
```

10.3.3 Using Indexes with For Loops

When using a for loop, not being able to access the index of a list was a problem that we came across earlier. We temporarily solved the problem with the range() function, however there is another way.

The enumerate() function provides a corresponding index for each item in a list.

.....

enumerate

function

The enumerate function is used with for loops to access the index position as well as the corresponding value.

Expression:

```
1 for index, value in enumerate(list):
2     #body of code
```

Statement:

```
1 steps = ["place blue portal", "place orange
    ↪ portal", "walk through portal"]
2 for index, value in enumerate(steps):
3     print str(index) + " " + steps
```

.....

The syntax is a slight modification on the for loop that we're used to. The for loop statement now has two variables — the first is the index, the second is the value — and the enumerate function now surrounds the list that the loop is using.

Here's an example that prints out the index and the value of each item in a list:

```
1 foxhound = ["Decoy Octopus", "Psycho Mantis",
    ↪ "Revolver Ocelot", "Sniper Wolf", "Liquid
    ↪ Snake", "Vulcan Raven"]
2
3 for index, item in enumerate(foxhound):
4     print str(index) + " " + item
```

This will print:

```
1 0 Decoy Octopus
2 1 Psycho Mantis
3 2 Revolver Ocelot
4 3 Sniper Wolf
5 4 Liquid Snake
6 5 Vulcan Raven
```

10.3.4 Zipping Two Lists

It is possible to iterate over two lists simultaneously.

The zip() function creates sets of items from two or more lists. As a for loop iterates it will move through each index of all zipped lists simultaneously.

The syntax once again varies slightly from the regular for loop. There will be the same amount of variables defined in the loop statement as arguments sent to the zip() function. For example, if you have two arguments in the zip() statement you will have two variables; 4 arguments, 4 variables; and so on.

In the following example we zip two lists and print their output together:

```
1 title = ["liquid", "decoy", "psycho", "revolver"]
2 name = ["snake", "octopus", "mantis", "ocelot"]
3
4 for codeTitle, codeName in zip(title, name):
```

zip

function

The zip function allows you to use the values of two lists simultaneously in a for loop.

Expression:

```
1 for item1, item2 in zip(list1, list2):
2     #body of code
```

Statement:

```
1 adjectives = ["sleepy", "hungry", "angry"]
2 nouns = ["rock", "octopus", "spanner"]
3
4 for adjective, noun in zip(adjectives, nouns):
5     print adjective + " " + noun
```

```
5     print codeTitle + " " + codeName
```

This will output:

```
1 liquid snake
2 decoy octopus
3 psycho mantis
4 revolver ocelot
```

When one list contains more items than the other, the for loop will iterate the same number of times as the length of the shorter list. So if you zipped a list of 5 items and another of 3, the for loop would only iterate 3 times.

10.3.5 For/Else Loops

The else statement can also be used with a for loop. It works differently to the else statement used in the while loop and the if statement.

The else statement, when used with a for loop, will execute when the for loop ends naturally. As long as the for loop reaches the end of it's list, it will execute the body of the else statement.

else

operator

When used with a for loop, an else statement will execute when the for loop has reached the end of its list. It will not execute if the for loop stops due to a break statement.

Expression:

```
for items in list:
    #body of for loop
else:
    #body of else statement
```

Statement:

```
group = ["Jatinder", "Steve", "David"]
for member in group:
    print member
else:
    print "list complete"
```

[diagram]

For example

```
foxhound = ["Decoy Octopus", "Psycho Mantis",
    ↪ "Revolver Ocelot", "Sniper Wolf", "Liquid
    ↪ Snake", "Solid Snake", "Vulcan Raven"]

for item in foxhound:
    print item
else
    print "Those are the members of team FOXHOUND"
```

This would print:

```
Decoy Octopus
Psycho Mantis
Revolver Ocelot
Sniper Wolf
Liquid Snake
Vulcan Raven
Those are the members of team FOXHOUND
```

10.3.6 Breaking a For/Else Loop

Using a break statement to exit a for loop is one way to prevent the else statement from executing.

The following example slightly modifies the above example. It incorporates a break statement that is nested within an if statement. The loop will break if the current item is "Solid Snake":

```
1  foxhound = ["Decoy Octopus", "Psycho Mantis",
     ↪ "Revolver Ocelot", "Sniper Wolf", "Liquid
     ↪ Snake", "Solid Snake", "Vulcan Raven"]
2
3  for item in foxhound:
4      if item == "Solid Snake":
5          print "Solid Snake is longer a member of
               ↪ FOXHOUND"
6          break
7      else
8          print item
9  else
10     print "Those are the members of team FOXHOUND"
```

Can you work out what the output is?

[diagram including break statement]

Chapter 11

Advanced Topics in Python

This chapter covers advanced topics. In particular we'll cover list methods, tuples, list comprehension, list slicing and lambdas.

All of these topics require a thorough understanding of lists, so make sure you understand them.

11.1 Minecraft Exercises

Reflect a building stored in a list.

Slice a building.

. . . . . .

End of Exercises

. . . . . .

11.2 Iterating Over Data Structures

We have already covered methods of iterating lists and dictionaries using for loops. There are also other ways of iterating these data structures.

In this section we will cover techniques to iterate over dictionaries using functions. We will also be introduced to a third type of list, a tuple.

11.2.1 items()

When used with a dictionary, the items() method will return all of the key and value pairs of the dictionary. The items will be in no particular order and are returned as a list of tuples.

.....

.items

method

The items method returns a list of tuples containing every item stored in a dictionary.

Expression:

```
1 dictionaryName.items()
```

Statement:

```
1 print dictionary.items()
```

.....

We have items in our inventory. The code creates a dictionary to represent the inventory. Each item has a name, represented by the key, and a quantity. The items() method creates a new list of all the items in the inventory:

```
1 inventory = {'tranquilisers': 5, 'rations': 2,
    ↪ 'boxes': 1}
2 inventoryMenu = inventory.items()
```

This will set the inventoryMenu value to:

```
1 [(u'boxes', 1), (u'tranquilisers', 5),
    ↪ (u'rations', 2)]
```

You'll notice that there is a a list with three items in it. What you'll not recognise is that each item is a tuple, a special type of list, which we'll look at now.

11.2.2 Tuples

Tuples are a type of list that are immutable. Meaning they can't be changed. Like other lists they are a sequence of items of any variable type. Tuples

use () for their syntax.

. . . . . .

tuple

list

A tuple is type of list that can't be changed after it is created. It used regular brackets ( )

Expression:

```
1 (item, item)
```

Statement:

```
1 tuple = (1, 5, 7)
```

. . . . . .

Here's an example. Every day of every week we have a budget for food. As this will never change we can represent this a tuple:

```
1 budget = (5.17, 5.20, 4.56, 53.64, 9.58, 6.41,
    ↪ 2.20)
```

To write a tuple including a single value you must include a comma:

```
1 dailyBudget = (5.12,)
```

When defining a tuple the brackets are optional, so you can just define a tuple by placing commas between values.

```
1 budget = 5.17, 5.20, 4.56, 53.64, 9.58, 6.41, 2.20
```

To access values of tuples you use the [index] brackets notation that you would with regular lists. You can also use slicing as you would normally.

The main difference between lists and tuples is that are immutable: you cannot change the contents of a tuple. For example you cannot use append to the end of the tuple, delete items or update any values.

11.2.3 keys()

The keys() method will return all of the keys of a dictionary in no particular order. It will return the keys as a list.

Let's go back to our inventory example:

keys

method

The keys method returns a lists of all the keys stored in a dictionary.

Expression:

```
dictionary.keys()
```

Statement:

```
print dictionary.keys()
```

```
inventory = {'tranquilisers': 5, 'rations': 2,
    ↪ 'boxes': 1}
inventoryKeys = inventory.keys()
```

The value of the inventoryKeys variable will be:

```
['boxes', 'tranquilisers', 'rations']
```

Notice how the keys are not in any particular order.

11.2.4 values()

The values() method returns all of the values in a dictionary as an unordered list.

Using our inventory example:

```
inventory = {'tranquilisers': 5, 'rations': 2,
    ↪ 'boxes': 1}
inventoryValues = inventory.values()
```

The value of the inventoryValues variable is:

```
[1, 5, 2]
```

Once again, notice how the list is in no particular order.

values

method

The values method returns a list of all of the values stored in a dictionary.

Expression:

```
1 dictionary.values()
```

Statement:

```
1 print dictionary.values()
```

11.3 List Comprehension

List comprehension can be summarised as a shorthand method of generating complex lists.

You have already been introduced to the range() function, which generates a list of integers based on the arguments you provide it. List comprehension enables you to make more complex lists based on conditions.

11.3.1 List Comprehension Syntax

List comprehension uses a combination of the for, in and if operators to create a list of values if the values meet certain conditions. It enables you to filter the values of one list using a condition and create a new list from the result.

For example you could create a list containing all of your party guest's that aren't called Jim:

```
1 party = ["Jim", "Jim", "Catherine", "Jim", "Anita"]
2 partyWithoutJim = [guest for guest in party if
    ↪ guest != "Jim"]
```

The `partyWithoutJim` would contain the value `["Catherine", "Anita"]`

The two variables either side of the for operator must have the same name. These variables can be called whatever you want. A list follows the in operator and a condition must follow the if operator.

list comprehension

expression

List comprehension is shorthand for filtering one list using a condition and generating a new list from the result.

Expression:

```
[variable for variable in list if condition]
```

Statement:

```
androidNumber = [1, 6, 2, 5, 8, 2]
newAndroids = [item for item in androidNumber if
    ↪ item > 3]
# newAndroids contains [6, 5, 8]
```

Only values in the list that meet the condition at the end of the statement will be added to the new list.

List comprehension is more flexible than the range() function as it can use comparators and boolean operators.

11.3.2 List Comprehension With Operators

List comprehension allows you to use operators on values of the new list as it is being created. For example we could add the string "Hi, " to all of the guests who are not called jim:

```
party = ["Jim", "Jim", "Catherine", "Jim", "Anita"]
partyWithoutJim = ["Hi, " + guest for guest in
    ↪ party if guest != "Jim"]
```

11.4 List Slicing

List slicing is used when we only want segment of a list. We have already met list slicing in a previous section. There are a couple more things to learn about list slicing.

11.4.1 Stride

So far we have only used list slicing where we want all of the values between one index in a list and another. We can also access values in gaps. For example you can access every other item or you could access every fifth item. We do this using strides.

List Slice Stride

operator

List slicing returns a segment of a list. Strides can be used to only return every 2nd, 3rd, 4th value and so on.

Expression:

```
variableName = list[startIndex : cutOffIndex :
    ↪ stride]
```

Statement:

```
patriots = ["Big Boss", "Dr. Clark", "Zero",
    ↪ "Donald Anderson"]
partOfThePatriots = patriots[1:4:2]
print partOfThePatriots
```

Strides state the increment between each item when we're slicing a list.

The start index is inclusive, it will be included in the list. The cut off index is exclusive, it will not be included in the list.

For example, we could take every third person from a list:

```
team = ["Terra", "Locke", "Edgar", "Sabin",
    ↪ "Shadow", "Cyan", "Gau", "Celes", "Setzer",
    ↪ "Mog"]
party = team[0:10:3]
```

11.4.2 Omitting Index Arguments

It is possible to omit any of the three indices when you use a list slice. For example, you can omit the start index like so:

```
party = [:10:3]
```

Python has defaults for each of the indices in a slice:

- The start index is default of 0, meaning that if you omit the start index the slice will start from the beginning of the list.
- The cut off index has a default of the last index in the list + 1. Omitting the cut off means that the slice always reaches the end of the list.
- The default stride is 1, so it will move through each item by default.

This is useful in many circumstances. If you want slice from the start and to the end of a list, while using every other item:

```
alternateItems = [::2]
```

11.4.3 Reversing a List

The stride of a slice can either be positive or negative. As we have seen a positive stride will work through in ascending index order.

A negative stride will work through a list in descending index order. In other words, it will do it backwards.

For example, we can reverse the order of a list of team members:

```
team = ["Terra", "Locke", "Edgar", "Sabin",
    ↪ "Shadow", "Cyan", "Gau", "Celes", "Setzer",
    ↪ "Mog"]
reversedList = team[::-1]
```

You can of course change the stride length like you would with a positive stride. So if we wanted the list reversed with every other team member:

```
team = ["Terra", "Locke", "Edgar", "Sabin",
    ↪ "Shadow", "Cyan", "Gau", "Celes", "Setzer",
    ↪ "Mog"]
reversedList = team[::-2]
```

11.5 Lambdas

Python has a special feature, which allows the programmer to treat certain functions as if they are variables. These special functions are called lambdas.

Lambdas are shorthand functions that can be treated as arguments by other functions.

11.5.1 Lambda Syntax

Lambdas take one or more arguments and return a value based on an expression.

.....

lambda

operator

The lambda operator creates an anonymous function which can be used as an argument. An argument is taken, which is ran through an expression. The result of the expression is then returned.

Expression:

```
1 lambda arguments: expression
```

Statement:

```
1 chats = 6
2 loveChats = lambda chats: chats > 6
3 #value of False
```

.....

lambda argument: expression

For example, we could return a boolean based on a condition. Say we want to determine whether a team has more than five members:

```
1 team = ["Terra", "Locke", "Edgar", "Sabin",
    ↪ "Shadow", "Cyan", "Gau", "Celes", "Setzer",
    ↪ "Mog"]
2 largeTeam = lambda team: len(team) > 5
```

Or we could use an operator on the argument. Let's keep it simple and square the value:

```
1 cats = 5
2 moreCats = lambda cats: cats ** 2
```

You may have noticed that you can achieve these results without the use of a lambda. Well done. You may have also noticed that these lambdas don't really do anything useful at the moment. These are just basic examples so that you become familiar with syntax, their true use will become clear when we use them as arguments in other functions.

Lambdas are effectively really short functions that have no name. As they are nameless they are known as anonymous functions.

The major difference between lambdas and regular functions is that lambdas are not reusable, whereas regular functions are. This is because lambdas have no name to to identify them.

11.5.2 filter()

The filter() function uses a lambda as an argument to filter a list. The lambda is used to determine whether each item in the list meets its conditions.

.....

filter()

function

The filter function fliters a list based on a lambda. The lambda provides a condition for the filter and must return a boolean value.

Expression:

```
1 filter(lambda argument: expression, list)
```

Statement:

```
1 party = ["Jim", "Jim", "Catherine", "Jim", "Anita"]
2 partyWithoutJim = filter(lambda name: name !=
    ↪ "Jim", party)
```

.....

For example you could use the filter function to remove two characters from our team:

```
1 team = [”Terra”, ”Locke”, ”Edgar”, ”Sabin”,
    ↪ ”Shadow”, ”Cyan”, ”Gau”, ”Celes”, ”Setzer”,
    ↪ ”Mog”]
2 newTeam = filter(lambda name: name != ”Terra” and
    ↪ name != ”Gau”, team)
```

You’ll see that our newTeam variable contains all of the items in the team variable, except ”Terra” and ”Gau”.

Chapter 12

Binary and Bitwise Operators

This chapter hasn't been written yet.

Chapter 13

Classes

[THIS CHAPTER IS NOT COMPLETE]

Reusability is a very important aspect of coding. It saves time and effort. Object Oriented Programming is an advanced and effective way to make code reusable.

Object oriented programming is a way of grouping functions and values together to create things called classes. Each class can be used to create something called an object, which shares the same groups of variables and functions as the class. Many objects can be made from the same class. This makes the grouped variables and functions of the class reusable.

Objects are data structures that contain functions and variables. When a function is part of a class it is called a method and a variable that is part of a class is called an attribute.

This can be a difficult concept to understand, so let's relate it to something that is familiar.

You are a person. You have a number of methods: you can eat, breathe, sleep, count to ten and do whole load of other things. You also have a number of attributes: name, age, height, shoesize and so on.

Your friend Mary also has the same methods as you. She also has the same variables, although they contain different values.

In fact all people have these methods and attributes. We can therefore describe people as a class. Your friend Mary and you are both people, so we could say you are both objects in the person class.

13.1 Minecraft Exercises

No exercises yet.

. . . . . .

End of Exercises

. . . . . .

13.2 Basic Class Concepts

Without realising it you have already been using classes and objects. All data-types — strings, booleans, floats, integers, lists and so on — are all pre-built classes used in Python. What's even more useful is that you can also define your own classes.

. . . . . .

Attribute

variable

Variables that are associated with objects are called attributes.

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

. . . . . .

You have encountered methods many times before. Remember using dot notation like this:

```
1 print "FROG".lower()
```

... or this ...

Method

function

Functions that are part of an object are associated with an object are called methods.

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

```
1 inventory = {'tranquilisers': 5, 'rations': 2,
    ↪ 'boxes': 1}
2 inventoryMenu = inventory.items()
```

Well those functions are actually methods. They relate to the string and dictionary classes respectively. Every time you have used a method like this you have been using objects.

13.3 Creating a Class

Creating a new class is surprisingly easy.

You use the class keyword, the name that you want the class to be called and the class that your class inherits from.

```
1 class className (object):
2     #body of class
```

Don't worry about inheritance at the moment, we will cover it later. For now we'll just inherit from the object class.

For example we can create the agent class. The pass statement is used as a placeholder and it does nothing. We use it when we haven't written some

.....

class

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

.....

.....

pass

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

.....

code, but we don't want Python to throw an error.

```
1 class agent(object):
2     pass
```

13.3.1 __init__()

The __init__() method run when we create an object using a class. It initialises the object.

.....

__init__()
method

Expression:

```
"String".lower()
```

Statement:

```
lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
#OR
country = "BRAZIL"
country = country.lower() #value of "brazil"
```

.....

The method always takes one argument, self, which refers to the object you are creating. Including the __init__() method is necessary whenever you create a class.

```
class className(object):
    def __init__(self):
        #body of __init__()
```

For example:

```
class agent(object):
    def __init__(self):
        pass
```

13.3.2 Arguments with __init__()

It is possible to provide arguments to the class when you initialise it. you achieve this by placing them after the self argument in the __init__() method.

To set these arguments as attributes of the class we use the self argument with dot notation.

class className(object): def __init__(self, argument): self.argument = argument

For example:

```
1 class agent(object):
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
```

The agent class now has a name and age attribute.

13.4 Creating an Object

To create an object you use the class name with brackets like you would a function. className(arguments)

.....

object

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

.....

The number of arguments you provide when creating an object depends on the arguments of __init__(). You always ignore the self argument when counting the arguments. Self is automatically passed as an argument without the need to write it.

For example:

```
class agent(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

raiden = agent("Raiden", 26)
```

13.4.1 Accessing Attributes

To access the attributes of our object we use dot notation:

.....

attributes

Expression:

```
"String".lower()
```

Statement:

```
lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
#OR
country = "BRAZIL"
country = country.lower() #value of "brazil"
```

.....

object.attribute

For example, if we want to print Raiden's age from the above example:

```
print raiden.age
```

Member Variables and Functions

13.4.2 Class Scope

The scope of a variable refers to what parts of a program can use it.

You have come across variables that can be accessed by any part of the program. This type of attribute is called a global variable as it can be used anywhere in the program.

The other type of variable is only accessible by members of a class. These variables are known as local variables. [expand]

[Explain how to apply this]

Functions/methods can also be local or global.

13.4.3 Creating Methods

Classes can contain methods - functions which are associated with the class.

To create a method you include a function in the body of a class:

.....

method

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

.....

class className(object): def __init__(self): #body of __init__()

def methodName(self, arguments): #body of method

All methods in a class should have the self argument as their first argument. As with the __init__() method, when you call a method you do not need to include the self argument. It is sent automatically.

You must initialise the object before you can use any methods.

For example:

```
1 class agent(object):
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5     def train(self, trainingType):
6         print "%s is undertaking %s training" %
              ↪ (self.name, trainingType)
7
8 raiden = agent("Raiden", 26)
9 raiden.train("VR")
```

13.4.4 Multiple Objects

You can make several objects from the same class. This is achieved by creating two or more objects with different names from the same class constructor. For example:

```
1 class agent(object):
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5     def train(self, trainingType):
6         print "%s is undertaking %s training" %
              ↪ (self.name, trainingType)
7
8 raiden = agent("Raiden", 26)
9 snake = agent("Snake", 42)
```

We now have two objects, raiden and snake. Each has the same attributes, albeit with different values. The also have access to the same methods.

```
1 print raiden.name
2 print snake.name
3
4 raiden.train("VR")
5 snake.train("field")
```

13.5 Inheritance

[REWRITE INHERITANCE WITH SUBCLASS AND SUPERCLASS TO DESCRIBE INHERITANCE] Classes can share the same methods and attributes as other classes. This is called inheritance.

For example, robins are a type of bird. They share the same methods (flying, eating) as other birds and they have attributes like other birds (weight, wingspan). Therefore we can say that robins inherit from birds.

13.5.1 Inheriting a Class

When one class inherits from another it can use all of its methods and attributes. It can also add extra classes and attributes to itself without altering the original class.

.....

inherited class

Expression:

```
1 "String".lower()
```

Statement:

```
1 lowerCaseName = "Pratap Jefferson".lower()#value
    ↪ of "pratap jefferson"
2 #OR
3 country = "BRAZIL"
4 country = country.lower() #value of "brazil"
```

.....

To inherit from another class, the name of that class is put in the brackets when the new classs is defined:

class derivedClass(baseClass): #body of class

[does the derived class need an __init__()?]

Other than this you can treat the derived class in the same way as any other class. You can add methods and attributes.

For example, if we have one class... we can use all of its methods in a new class. We have also added another method, which the new class can use, but the original class cannot:

[example]

13.5.2 Overriding Methods and Attributes

It is possible for a derived class to redefine methods and attributes from its parents class.

For example, the vehicle class has a method to make it move. If you wanted to derive a boat class from the vehicle class you would want it to move over water. If you wanted to derive a helicopter class from the vehicle class it would move through the air. Therefore you would redefine the move method for each derived class.

To override a method you simply define a new method with the same name as the original within the derived class:

def className(arguments): #body of class

The number of arguments and their names can be different to that of the original class.

Here is an example of a class method being overridden:

[example]

13.5.3 Referencing Superclass Methods in a Subclass

[write this bit]

class DerivedClass(Base): def some_method(self): super(DerivedClass, self).meth()

Chapter 14

File Input and Output

Files are a major part of computing. They allow us to save data for long term storage and load data for immediate use.

So far your programs have been unable to store data. They've either had data hard-coded into the program or taken data from the command-line. This limits the potential of our programs as we can't store and retrieve large amounts of data.

In this chapter we'll cover file input and output with Python. Python has a set of in-built functions that make files easy to use in your programs.

14.1 Minecraft Exercises

Input a multi-dimensional array of blocks to a file

Save objects from one world and load them into another.

Output a multi-dimensional array of blocks from a file

. . . . . .

End of Exercises

. . . . . .

14.2 Introduction to File I/O

14.2.1 Opening a File

Opening a file is always the first thing we do when we're working with files. The open() is simple pretty to use. It takes two arguments, file location and permissions. As you would expect file location is the path to your file. Page [page number] shows you how to work out the path to a file.

open

function

Expression:

```
1 open("fileLocation", "permissions")
```

Statement:

```
1 shoppingList = open("./files/shopping.txt", "r")
```

The permissions argument states what the program can do with the file. There are four options for this:

- w: write only mode allows the program to write new data to the file, but it cannot read the contents of the file.
- r: read only mode allows the program to read the contents of the file, but it is not allowed to modify its contents.
- r+: read and write mode allows the program to read and modify the contents of the file.
- a: append mode writes new data to the end of the file.

There are different circumstances in which you'd use each permission. For example, if you only want someone to view a file, but not be able to change it, you would use the read only permission. Alternatively if you want someone to add data to a file, but not be able to see the other data stored in the file, you would use the write only permission.

For example we want a file that we read, but can't write to:

```
1 secretFile = open("secretFile.txt", "r")
```

14.2.2 Writing and Closing a File

The write() method makes it easy to write to a file. You simply put the data you want written to the file as an argument in the write() method.

write

Expression:

```
objectName.write(dataToWrite)
```

Statement:

```
shoppingList = open("./files/shopping.txt", "r+")
shoppingList.write("Apples")
shoppingList.close()
```

You must first open the file using the open() function otherwise this will not work.

close

Expression:

```
objectName.close()
```

Statement:

```
shoppingList = open("./files/shopping.txt", "r+")
shoppingList.write("Apples")
shoppingList.close()
```

Once you have written all of the data to the file you must use the close() method. Unless you use the close() method after the write() method, the data will not be stored.

objectName.close()

For example, let's open a file and write a simple string to it:

```
1 secretFile = open("secretFile.txt", "r+")
2 secretFile.write("This is a secret file.")
3 secretFile.close()
```

14.2.3 Reading a File

You can also read the contents of a file. You may want to use the data in your program, modify the contents the send them back to the file or output the data for viewing. Whatever the reason, the read() method is the method used for reading files.

.....

read

Expression:

```
1 objectName.read()
```

Statement:

```
1 shoppingList = open("./files/shopping.txt", "r+")
2 print shoppingList.read()
```

.....

In order to use the file you must of course open it. It's also a very good idea to close the file once you're finished.

For example:

```
1 secretFile = open("secretFile.txt", "r")
2
3 print secretFile.read()
4
5 secretFile.close()
```

14.2.4 Reading a Line of a File

There are times when you will want to read the file one line at a time, and not all at once. The readline() method is used for this.

.....

readline

Expression:

```
1 objectName.readline()
```

Statement:

```
1 shoppingList = open("./files/shopping.txt", "r+")
2 print shoppingList.readline()
3 print shoppingList.readline()
4 print shoppingList.readline()
```

.....

objectName.readline()

Once again you must open the file before you use this method, and close it afterewards:

```
1 secretFile = open("secretFile.txt", "r")
2
3 print secretFile.readline()
4 print secretFile.readline()
5 print secretFile.readline()
6
7 secretFile.close()
```

The readline() method starts on the first line of your file. Each time the readline() method is used it will read the next line. Providing an integer argument to the readline() method states how many characters from the line you want to read. For example providing an argument of 5 will mean the method will only return the first five characters of the line.

14.3 The Buffer

When you write to a file Python does not immediately put the data into that file. Instead it stores the data in a buffer. This buffer temporarily stores all of the data your program has sent with the write() method. Python will not transfer the data from the buffer into the file until it is sure you have made all of the changes you need to. This is the reason we use the close method.

The close() method is one way to transfer data from Python's buffer into the file, there are other ways to transfer data from the buffer.

14.3.1 Automatically Closing a File

.....

with

Expression:

```
1 with open("file", "permissions") as variableName:
2     #body of with statement
```

Statement:

```
1 with open("./files/shopping.txt", "r") as
    ↪ shoppingList:
2     print shoppingList.readline()
3     print shoppingList.readline()
4     print shoppingList.readline()
5     print shoppingList.readline()
```

.....

Using a with...as statement it is possible to automatically close a file, without the need to use the .close() method.

For example:

```
1 with open("secretFile.txt", "w") as secretFile:
2     secretFile.write("Secrets and stuff")
```

Inside the body of the with statement, include all of the code you want to use with your file. Once the body of the with statement has finished the file will be closed automatically and all data from the buffer will be written.

14.3.2 Closed Attribute

You may need to check whether your file is open or closed after you have used some of the methods listed above.

.....

closed

attribute

Expression:

```
1 objectName.closed
```

Statement:

```
1 shoppingList = open("./files/shopping.txt", "r+")
2 shoppingList.write("Apples")
3 shoppingList.close()
4
5 print shoppingList.closed
```

.....

The .closed attribute of a file objects states whether a file is closed as a boolean.

When True is returned, the file is closed. False means the file is open.

For example:

```
1 if secretFile.closed == False:
2     secretFile.close
```

Chapter 15

Error Handling

This unit is not currently included at Codecademy. Error handling is extremely useful. The topics covered in this chapter will inclide:

- try-catch statement
- error handling lists (not in)

Appendices

Appendix A

Checklist of Topics Covered

Syntax

- Variables
- Data types
- Integers
- Floats
- Booleans
- Changing Variables
- Statements
- Whitespace and Tabs
- Single-line Comments
- Multi-line Comments

Maths Operations

- Expressions and Statements
- Maths Operators
- Addition
- Subtraction
- Multiplication
- Division
- Exponentials
- Modulo
- Operator Order
- Interchanging variables and values
- Shorthand Operators

String and console output

- Strings
- Substrings
- String Functions
  - len()
  - lower()
  - upper()

  - str()
- Print
- Concatenation
- Placeholders
- Console input
- Date and Time

Comparators and Control Flow

- Comparators
  - Equal to (==)
  - Not equal to (!=)
  - Less than (<)
  - Less than or equal to (<=)
  - Greater than (>)
  - Greater than or equal to (>=)
- Boolean Operators
- If statements
- Else statements
- Elif statements

Functions

- Creating and Calling Functions
- Returning a value
- Arguments
- Modules
- Importing modules
- Built-in Functions:
  - max
  - min
  - abs
  - type

Lists and Dictionaries

- Creating lists
- Accessing index positions
- Adding items
- List length
- Slicing
- Searching a list
- Inserting an item
- Removing items
- for loop
- Sorting a list
- Combining lists
- Defining a dictionary
- Changing/adding items in a dictionary
- Deleting items in a dictionary

Functions and Lists

- Lists as arguments
- Modifying every list item
- Range function
- Converting a list into a string
- Splitting a string into a list
- Multi-dimension lists
- Joining two lists
- Undefined number of lists

Loops

- While Loops
- Boolean Operators with While Loops
- Infinite Loops
- Break
- While/else
- For Loops
- Strings as lists
- Looping dictionaries
- Indexes and for loops
- Zipping two lists
- For/else loops
- For/else break

Advanced Topics

- Iterating data structures
- items
- tuples
- keys
- values
- List comprehension
- List slicing
- Stride
- Omitting slice index arguments
- Reversing a list
- Lambdas
- filter

Classes and Object Oriented Programming

- Creating classes
- __init__()
- Arguments
- Creating objects
- Accessing attributes
- Class scope

- Creating Methods
- Multiple Objects
- Inheritance
- Overriding methods and attributes
- Referencing superclass methods in a subclass

File Input and Output

- Opening a file
- Writing and closing a file
- Reading a file
- Reading a line
- Automatically closing a file
- Closed attribute

Exercise structures:

- Following instructions
- Reusing code
- Problem solving
- Flowchart

Transferable Programmer Competencies

- Problem decomposition - breaking down a problem into smaller manageable parts
- Debugging and grit - The process and perseverance to succeed when a program fails
- Systems thinking - understand how parts of a program relate to one another (functions and modularity)
- Communication - explain what the program does in laymen's terms
- Documentation - explain how the program works in technical terms
- Sharing Knowledge - Share solutions to problems
- Collaboration/Team work - Working with others to achieve a common objective
- Testing - Checking the quality, reliability and performance of a program
- Critical Analysis and Reflection - Constructively reviewing the work of others and their own work to explain strengths and suggest improvements
- Choice of technology - Choosing appropriate approaches and technologies to solve a problem

- Requirements analysis - Understanding and prioritising the different needs of users of a system