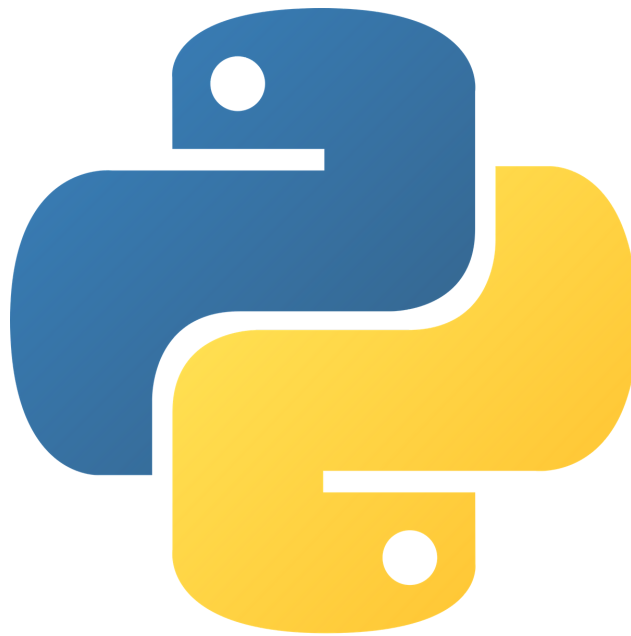




< Zim / Code >

"Bringing programming to the forefront of youth education and empowerment in Zimbabwe."

Introduction to Python



Student's Textbook



A comprehensive guide on the basics of programming, in Python.

by Mgcini Keith Phuthi & Alvin Chitena

©Zim Code 2017. All Rights Reserved.

Table of Contents

Thanks	4
The authors	4
Purpose of the textbook	5
Installing Python	6
Chapter 1 – Introduction to Python.....	7
The Computer Model.....	7
Why Python?.....	8
The IDE	8
Chapter 2 – Variables.....	12
Abstraction.....	12
Variables	12
Syntax and Semantics	14
Chapter 3 – Data Types.....	16
Integers	16
Floating Point Numbers	17
Type casting	17
Strings	18
Boolean types	20
Variables and Data Types.....	22
Chapter 4: Introduction to functions	24
Chapter 5: Flow Control	30
Chapter 6: Loops.....	36
While Loops.....	36
For Loops and iteration.....	39
Chapter 7: Data structures I (Lists, Tuples and Strings)	45
You already know a data structure!.....	45
Tuples.....	45
Lists	47
Chapter 8: More Functions and Variables	53
Scope.....	53
Recursion	60
Chapter 9: Algorithms and Pseudocode	63
Chapter 10: Working with APIs: Calico	69
Project Calico	69

Environment	69
A brief detour.....	70
Back to Calico	72
Animation.....	79
Chapter 11: More Examples	93
Easier Examples.....	93
Medium Examples.....	94
Difficult Examples	96

Who is ZimCode?

ZimCode is a non-profit organisation in Zimbabwe dedicated to giving accessible programming lessons to high-school students. It was founded in June 2016 by Alvin Chitena (CEO) a student of the Wesleyan University Class of 2019. For more information, visit www.zimcode.org.

The lessons and curriculum are developed and delivered by volunteers who are mostly university students in Zimbabwe, the United States and Ghana.

Thanks

ZimCode would like to thank the following organizations for their contributions in donations, grants, guidance and services to the success of the organization:

The Davis Projects for Peace

Wesleyan University

Ministry of Education Republic of Zimbabwe

Lead Us Today

SkyHub

Education Matters

The NUST American Space

The authors

This is a part of the complete book which is written by Mgcini K. Phuthi (MIT Class of 2019) with contributions from Alvin Chitena, Mthabisi Sibanda and Frederick Dandure. This book should not be distributed or sold by any party other than by ZimCode. If you have questions, email mgcini@zimcode.org.

**“Bringing programming to the forefront of
education and youth empowerment in
Zimbabwe”**

Purpose of the textbook

This text is designed to cater specifically to high school students in Zimbabwe and other African countries whose education system is based on the GCE A level curriculum. The need for a programming textbook was clear during ZimCode's inaugural programming classes. The idea is to provide a low barrier text in writing that is easy to understand for African high school students with very little assumptions made about the background of the student. A wide range of relatable examples are used throughout the text to provide analogies and to help students grasp the abstract concepts in computer science and programming. This is done with the hope that students may begin to appreciate how computer scientists and software engineers solve problems and how these skills can be applied in the real world.

In addition to learning the basics of coding, students will also inevitably learn and improve in other subject areas. The link between computer science and mathematics is undeniable and in fact fundamental. While it is easy to fall into the temptation of providing mostly computational and mathematical examples, we tried our best to include a wide range of examples appealing to all three major fields of science, commerce and humanities in every chapter.

It is important for the teacher to go through the guidelines provided in this teaching guide as they give tips on how to get the most out of the text and examples.

This course will give a 12-week introduction to programming in Python which is one of the most accessible languages to beginners due to its similarity to written English and white space delimiting. White space delimiting is the use of indentations and line breaks rather than brackets to group code blocks e.g. in a function/loop. Python is also a very powerful tool in computation (The numpy package is used in various fields for numerical computation) and web design (Instagram is written in Django/Flask, Python frameworks) with a very good support and development network. It is also easy to install and is often preinstalled on most operating systems. (Unfortunately not on Windows)

An important part of this text is that it should be fun and interactive to the students. It involves a lot of live demos. Examples are chosen from real world problems so that it becomes immediately apparent to the students how programming applies to real life.

Students will also learn a lot of broader concepts in programming such as abstraction, functional programming, object oriented programming* and optimization.

Installing Python

You can download and install Python™ from www.python.org/downloads. This textbook uses Python 3 (any version that starts with 3 will work the same).

If you are having difficulty, ask a tutor to help you.

Chapter 1 – Introduction to Python

Programming is writing down a set of instructions for the computer to follow. How do you give instructions to someone? You have to communicate using a language. There are many programming languages that exist, we are going to teach you Python. Python is called a high level interpreted language. We will explain what this means a bit later.

Computers “think” only in ones and zeroes yet they are capable of doing so many things. How is this possible? A computer is basically a calculator, all it actually does is calculations and from these calculations, you can do more complicated things. So if a computer only understands ones and zeroes, how do we communicate with it? Just like any other language, you have to translate it so that other people can understand, the ones and zeroes are translated to other languages until they are translated into a high level language. A high level language is a language that humans can read and understand easily.

When we say Python is an interpreted language, we mean that it follows instructions one by one in the order that you write them then if it gets to an error it crashes. If you give two instructions like:

- i) Go outside
- ii) Jump to the moon

Python will go outside and then it will realise that it can’t jump to the moon and give you an error.

The opposite of an interpreted language is a compiled language. A compiled language first checks to make sure there are no errors before it runs. So if you gave it the two instructions above it will not even go outside, it will immediately tell you that it cannot jump to the moon.

Unfortunately, computers are not very clever, in fact a computer is a very stupid machine that blindly follows any instructions you give it. It will always try to do whatever you tell it to do. It also follows its language very strictly, if you put a small (lower case) letter where there is supposed to be a capital (upper case) letter, it will be completely confused and give you an error. You will see this as we go along because you will definitely have lots of errors.

The Computer Model

A computer is a system made up of three things:

1. **Input** – this is the information that a computer gets from outside, it will never act on its own e.g. when you type you are giving an input of character. Things like keyboards, mice and touchscreens are called input devices
2. **Processing** – These are the instructions and calculations that the computer must follow to do something with the input. For all the devices you use, you are used to being involved with the input and output part. In programming, you get to directly deal with the processing part of a computer.
3. **Output** – This is what the computer gives out e.g. the picture on your screen, a sound or turning off the lights.

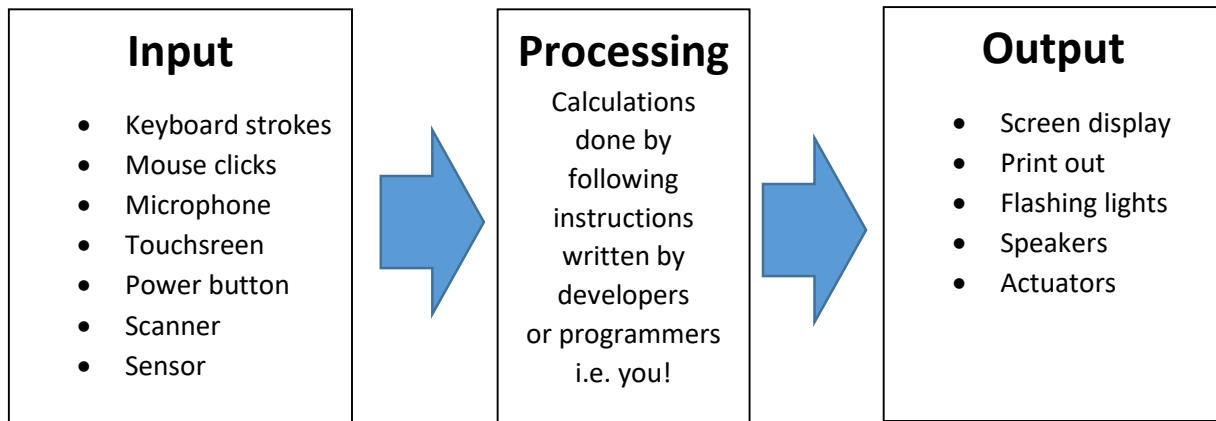


Figure 1

We choose to teach Python because it is a relatively simple yet extremely powerful programming language to understand. Some examples of apps/websites written in Python are Instagram, Pinterest, Parts of Google and most of YouTube and many more. The most important reason is that because it is simple, it will allow you to learn the core concepts in programming which allow you to learn other languages much faster.

The IDE

Let's write some code! After following the instructions on how to install Python 3, open the program IDLE that was installed with Python. IDLE is called an IDE which stands for Integrated Development Environment. It is the software that we will be using to write and run most of our code in this course. Your screen should look like this:



Figure 2

This is called the Python Shell, let's write our first line of code, inside the shell, type in the following then press enter.

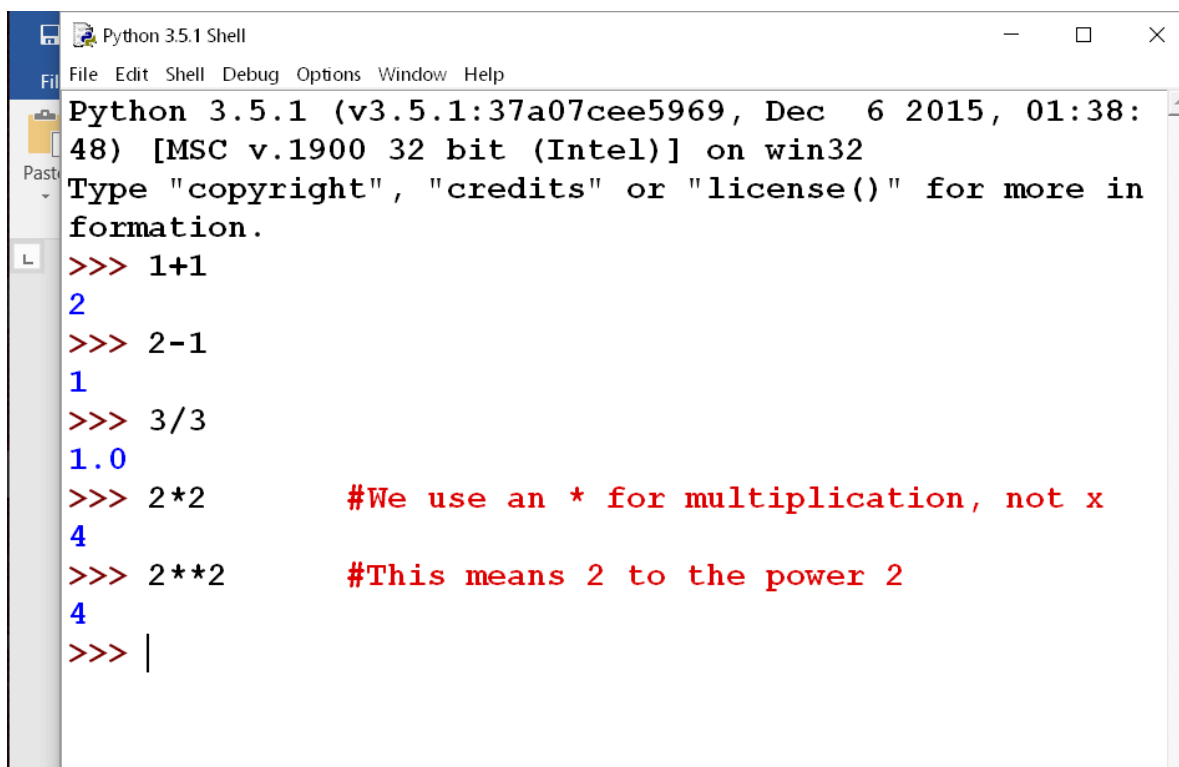

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:48) [MSC
v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>print('Hello World')
```

You have written your first line of code! It tells Python to output/print the words “Hello World”. Notice how you typed in an input, the computer processed that input and it gave you an output.

Exercise 1.1: Say my name

- a) Change the words inside the quotation marks so that it say “Hello” and followed by your name. Do not remove or add anything else! You have learned your first piece of code using the **print()** function.

Earlier we said a computer is a calculator, let’s use the computer as a calculator. Execute the following code and see if it gives you the right answer. Do not include the things that come after the “#” symbol.

A screenshot of a Python 3.5.1 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following content:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 6 2015, 01:38:
48) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more in
formation.
>>> 1+1
2
>>> 2-1
1
>>> 3/3
1.0
>>> 2*2          #We use an * for multiplication, not x
4
>>> 2**2         #This means 2 to the power 2
4
>>> |
```

Figure 3

Exercise 1.2: Back to first grade

- a) Use the symbols you used above to try and do more calculations

Now execute the following code:

```
>>> 3//2
>>> 3//3
>>> 3//4
>>> 3//7
```

What do you think the “//” symbol does? This is called floor division, it gives you the whole number from when you divide two numbers. E.g. 5//3 is 1 remainder 2 but the floor division operator only gives you the whole number part which is 1.

Now execute this code and try and figure out what it does.

```
>>> 3%2
>>> 3%3
>>> 3%4
>>> 3%7
```

The “%” symbol is called the “mod operator”, it gives you the remainder after division.

Let’s make our calculations a bit more complicated. The rules of BODMAS (Brackets, pOwers, Division, Multiplication, Addition, Subtraction) that you learned in primary school work here just like for your calculator. Decimals in Python use a full stop which is usually called a period (.), not a comma (,).

```
>>> (23//5)*7/(12%5)+5          #19.0
>>> ((67%5)*(-4*-0.5))**0.5    #2.0
```

Notice that even when you are multiplying things in brackets, you have to put the * operator!

```
>>> 2(2)          #Why is this an error?
>>> 2*(2)
```

Exercise 1.3: BODMAS

a) Do the following calculation by head and then confirm it on the shell

```
>>> 17//7 + 23%3 + 3**2
>>> (13+(34/2)*23%4-12)
```

Max/Min

We can also find the maximum and minimum of a group of numbers separated by commas

```
>>> max(1,2,3,4,5)
>>> min(1,2,3,4,5)
>>> max(24,3,7,-3)
>>> min(23,-4,0,89)
```

To round off numbers we use the **round()** function.

```
>>> round(9.4)
>>> round(9.7)
```

What if we want to use Python as a scientific calculator, to do this we have to borrow the tools from somewhere else using the following code:

```
>>> import math
```

This is called importing the math module, we will be importing many more modules so remember how it’s done.

To use the more advanced calculations, we always have to execute the code above. Let's see what we can do.

```
>>> math.sqrt(4)           #Find squareroot of 4
>>> math.sin(1)            #Find sin(1)
>>> math.pi                #The number  $\pi$  or pi
>>> math.cos(math.pi)      #Find cos( $\pi$ ) or cosine of pi
>>> math.log10(100)        #Find log(100) to base 10
```

Remember that this will not work if you do not import the math module!

Exercise 1.4: Scientific calculator

- Close IDLE and open it again.
- Calculate the area of a circle of radius 2 using **math.pi**
- Copy the answer and round it off to the nearest unit using **round()**

So far you have been typing in commands one by one and executing them, but what if you are writing some really complicated code like a music player. It would be disaster if it executed your code right after you've typed it, it's like writing a composition with an examiner who marks after every sentence. You want to first write out parts of your code, test them and then write more and if you need to change anything, you can go back.

In Python, you can write code using a script which allows you to write a lot of code at once. In IDLE, go to File > New or press Ctrl + N. You should see a blank screen. Type in the "Hello World" code from before and then go to Run > Run Module or press F5. It should give the same output as before. Now let's write some more complex code.

```
name = input("What is your name?: ")
print("Hi", name)
```

This code asks you to type in your name and then when you press enter, it greets you. When you run a script, it never prints anything unless you use the **print()** function unlike the shell where if you type in a calculation, it immediately gives you an output.

Ex 1.5: So many questions

- Change the words in green (the ones inside the quotation marks only so that it asks you different things and gives you different answer e.g. asking for your age then make it print "You are 17". Only edit the text in green

Chapter 2 – Variables

Abstraction

So far we have just been considering things that an ordinary calculator can do with just a few simple additions. The first and probably the most powerful tool in programming is abstraction. Abstraction is the use of an idea rather than an event. This is best shown through examples.

Example 2.1 – Using a formula

If I were to tell you that to calculate the distance travelled by a car I did the following calculation:

$$\text{distance} = 2 * 3 = 6$$

Is it possible for you to know how I did it? What is 2? What is 3? What units did I use? Is it 6 metres per second or 6 kilometres per minute? Clearly this is not useful if I want to learn how to calculate distance by myself. **How do we know how to calculate distance? How do we calculate distance?**

The answer is that we know the formula:

$$\text{distance} = \text{speed} * \text{time}$$

Now we can calculate any distance given any speed and any time. This is what we mean by abstraction. Rather than using an event (i.e. exact numbers) we use an idea (i.e. a formula).

Supplementary Exercise 2.1 (Science) – Making a complicated calculation easy

Imagine that you are the assistant to a crazy physicist named Larmour who needs to know the power radiated from a point (The point is that this is a complicated calculation, you do not even need to know what it means). Larmour says he wrote his formula on the board and needs the answer to finish making a machine that gets rid of all homework so you happily help him. The formula is:

$$P = \frac{e^2 a^2}{6\pi\epsilon_0 c^3}.$$

He also wrote what number each of the symbols is:

$$e = 1.6 \times 10^{-19}$$

$$a = 1$$

$$\epsilon_0 = 8.85418782 \times 10^{-12}$$

$$c = 3 \times 10^8$$

- i) What is P? The point is that you can calculate a very complicated thing just because you know the formula or because Larmour “abstracted” the idea.
- ii) Suppose Larmour says he made a mistake and a is actually 2, how do you use your answer from i) to fix this easily?

Variables

Let’s go back to the example with distance where we have:

$$\text{distance} = \text{speed} * \text{time}$$

We have multiplied before so we know what the “*” does. There are two new things i.e. the three words and the “=” sign. The three words are called variables, which has the same meaning as in mathematics. Variables are objects that hold something e.g. “time” holds the number for a time, “speed” the number for a speed etc. **When calculating distance, what do you know first, the things on the right or the things on the left?** We know the speed and time first so we first multiply the numbers on the right and then the “=” sign puts them in the “distance” variable. How do we know it is a variable? Because it has a name. Anything that has a name in Python is called an object and most objects are variables.

Exercise 2.1 – Declaring variables

In the shell, execute (type it then press enter) the following code:

```
1 >>> x = 3
2 >>> print(x)
3 >>> y = 4
4 >>> print(y)
5 >>> print(x+y)
6 >>> x = y
7 >>> print(y)
8 >>> name = "Python"           #You can put your own name here
9 >>> print(name)
```

Notice how the letter “x” now stands for 3 and “y” stands for 4 and you can do the things you can do to numbers to x and y because x and y actually represent the numbers in the calculation.

Demonstration 2.1: Let’s be variables!

This is a good chance to learn everyone’s name. Everyone in the class will be a variable called by their name, there will be someone who is an operator and the teacher will be the interpreter (Python). Each person will get a piece of paper (this is the “memory”) with a number on it whenever they are ‘declared’ as a variable. Here are the rules:

- 1) Whenever the interpreter assigns a value to a variable, it will say **name = number** and then it will give someone with the name **name** a fresh piece of paper with a number **number** written on it. If **name** was already carrying a number, they throw the old number away and get the new number.
- 2) Whenever the interpreter says `print(name)`, the person with the name **name** should shout the number they are carrying. If they are not carrying a number they should say ‘**name** not defined’.
- 3) Each person can have a maximum of one number.
- 4) Whenever the interpreter says something like “`print(x+y)`”, the calculator checks the person called x and checks the person called y and shouts the answer

Shout at a few instructions declaring people as variables, printing variables, exchanging variables and adding/subtracting variables and assigning them to new ones etc. Have fun!

Exercise 2.2: Python Siri

- a) Now that we can code, we want to make our own version of iPhone’s Siri. Write a script that asks for the user’s name using `input()` and then asks for the age and then prints out a greeting that says “Hi, [name], you are [age] years old” where [name] is replaced by the person’s name and

[age] is replaced by the person's age. Think about what variables you can use and how you would name them

- b) Write a script that exchanges the values of two variables **x** and **y**. Think about the demonstration you saw/did and how you would do it in the demonstration and do it in code using only three lines. (Hint: You can use as many variables as you want)

Rules for naming variables

You may have noticed how much freedom we have when naming variables, this allows to be able to make smart decisions about how we name them. If you have a long script with lots of variables, you want to be able to know what that variable is easily instead of all your variables being x or y. Here are some rules for naming variables, the first 5 rules are **not** optional, you will get errors if you do them wrong

1. Never use python keywords (e.g. print, def, import etc.) Basically, if the word changes colour when you type it, do not use it as a variable.
2. Do not use the name of a module you have imported (e.g. math), if you import something, its name has been taken, you should not use it for a variable
3. Do not use any punctuation except underscores (_ or Shift + -) in your variable names
4. Do not use spaces in your variables, if you want to have a space, use an underscore (_) e.g. **two_words**.
5. Your variable name should never start with a number (e.g. 1st_variable)
6. Give your variables names that anyone can understand (e.g. name, age, number_of_cats)
7. If your variable is going to be changing, use only lower case letters e.g. **cost_of_food**.
8. If your variable is just a number that you input once at the beginning of the document use all upper case letters e.g. **NUM_WHEELS_ON_CAR**.

Python keywords and Comments

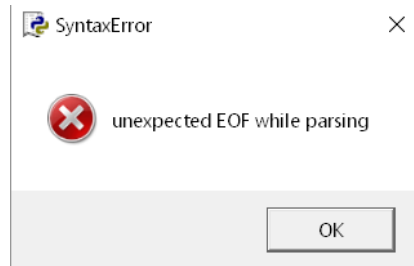
Python has special words that mean special things, you can always see a keyword by the way it changes colour when you type it in e.g. **if**, **import**, **elif**, **print** etc. These should never be used as variable names.

We say Python is human readable but sometimes our code gets complicated and someone who did not write the code might want to understand what it does. The person who writes the code can write comments to explain what is going on. Comments always start with the pound sign (aka hashtag) symbol (#). Anything written on the same line in red after the symbol is ignored by Python, so you can write anything you want. Usually they are used to explain steps. You are encouraged to always write your own comments to explain what you are doing, even to yourself. Here is an example where comments are used:

```
x = 1      #Declare a variable x as 1
x += 1     #Add 1 to x
print(x)   #Print the new x
```

Syntax and Semantics

So you have been typing code and you have probably run into errors already, for now all the errors you have come across are “syntax” errors. Syntax has to do with rules of a language, things like grammar, spelling and punctuation are the syntax of the English language. Similarly in Python, every bit of code that you type must follow the syntax of Python otherwise you will get a syntax error like the one below when you forget to close the bracket.



Another type of error is what is called a semantic error, this is an error where your code runs but it does not do what you want it to, for example if you want to add two numbers but Python subtracts them instead. These errors are very dangerous because Python will not tell you that your code is wrong, so you have to test your code and if it has problems, you “debug” it. Debugging means fixing code that does not work properly.

Toolbox 2.1

1. Variables
2. Comments
3. Coding vocabulary (syntax, semantics, debugging)

Chapter 3 – Data Types

Something that you may have learned in school is something like “data is information”. This is basically true and there are lots of different things you can do with data. For example, you can write words with letters, you can add together numbers, etc. A computer only understands ones and zeroes; this is how it stores all its data but this is useless to us because we cannot read it. Instead we want to store numbers like 7 or words like “Python”.

We now have a problem because we have many different data types and we want to do different things (operations) on them. For example, try the following code:

```
>>> 5 + 'Python'
```

This is your typical example of trying to add oranges to donkeys. It does not and should not work. This is because “5” and “Python” are different data types. **Why do we need different data types?** Different data types serve different purposes and thus have different operations.

Integers

What is an integer? You might remember what these are from form 1/2 mathematics. Integers are whole numbers with a + or – sign (e.g. 1, 5, -7, 345, 0, -111111). **What examples can you give?** Python refers to an integer in short as **int**. Let’s do some operations on integers, you have done some of these before:

```
1 >>> 2 + 345
2 >>> 34 - 54
3 >>> 56//3
4 >>> type(12)                                #Check the type of some data
5 >>> x = 15
6 >>> type(x)
```

The sixth and fourth line uses a **type()** function to check the type of some kind of data. We will go into more detail about functions in the next chapter. This works the same way as the **input** and **print** functions we used earlier. Notice that if you put something inside the brackets, it tells you what type it is.

Toolbox 3.1: Integers

1. Mathematical operations (+, -, *, /, //, %, ** etc.)
2. **type()**

Exercise 3.1: Spot the integer

Which of the following numbers are integers? Even if you think the answers are obvious, use the **type()** function in the shell to check.

- i) 2 ii) 7*9 iii) 8/3-2 iv) 12*4/(3-2)+5 v) 34//7

Supplementary Exercise 3.1: How to change an integer

Look at Toolbox 2.1. Which of those operations can give you an answer which is not an integer. What kind of numbers do they give you?

Floating Point Numbers

So we have looked at integers but is it possible to calculations with just integers? No. We need to have some way to represent fractions and decimals like $\frac{1}{2}$ or 0.3 or π . This is done using what are called floating point numbers. Python refers to a floating point number as a **float**. They are always represented as decimals, there is no built in way to have an output as $\frac{1}{2}$, it will always be **0.5**. The decimal point is always the fullstop/period. Let's do some examples:

```
1 >>> 0.4 + 0.4
2 >>> -2/3                                #See footnote1
3 >>> x = 0.5
4 >>> x * 4                                #Line 4
5 >>> type(0.3)
6 >>> type(0)
7 >>> type(0.0)
8 >>> type(2/3)
```

Notice a few very important things:

- Line 2 and 8: Dividing two integers and getting a fraction gives you a float.
- Line 4: Calculations involving a float always gives the answer as a float
- Line 7: Even if it's an integer to us, if it has a period(.), it will always be considered a float

Type casting

Try the following code:

```
1 >>> int(2.3)
2 >>> int(37.42)
3 >>> float(3)
```

Notice how we can change integers to floats and floats to integers using the **float()** and **int()** functions. This is very useful if you want to be sure your answer is of a certain type.²

Exercise 3.2: Do integers float?

- a) What type are the following numbers, you can use the **type()** function:

i) 45 ii) 45/2 iii) `math.pi * 4 ** 2` iv) `math.sqrt(4)`

Don't forget to import math first!

- b) i) Declare a variable **x** as your favourite integer.
ii) Check the type of **x**.
iii) Declare a variable **y** as **x+1/3.0**. What is its type?
iv) Convert **x** into a float and check its type.

¹ In older versions of Python, this would give you zero because they swapped division (/) and floor division (//)

² In fact, most other programming languages do not allow you to declare a variable without specifying the exact type it is going to be. Python allows your variable to store anything, this is called "dynamic typing".

Lesson 4: Teacher's notes

This lesson involves a lot of typing as well as a lot more teacher demonstration. For the first section on strings, having them write notes on the common functions and string methods will be helpful.

When it comes to indexing and boolean operators, the teacher can go through a lot of examples on a display, the ones chosen here try to capture all the ideas but the teacher can come up with their own and quiz the class on what answers they expect. It should reach a point where they are easily capable of figuring indexing/booleans out very quickly.

Strings

A string is the data type used by Python to represent text/characters. A character is simply any letter of the alphabet or any symbol or any punctuation including a space. This has to be different from variables. Strings are represented by putting them in quotation marks e.g. 'Hello World' or "Python". Strings in Python are represented as **str**. Let's practice making strings:

```
>>> 'This is a string'
>>> "This is also a string"
>>> This is not a string           #Why is it not a string?
>>> '3'
>>> "ERROR!                       #Why is this an error?
>>> ' '
>>> '~'
>>> my_str = "Hello World"
>>> print(my_str)
```

There are a lot of things we can do with strings, a lot of which we will just mention because you can do all of them in the same way. We will go down the list of the most important things we can do.

Concatenation

Concatenation is when you join two or more strings. This is done using a + in exactly the same way you would add integers but you have to be careful that you are adding two strings together, not a string and some other data type. Let's do some examples:

```
>>> 'Hello' + ' ' + 'World'
>>> '3 + 5 = ' + 8           #Why is this an error?
>>> first_str = 'Python is'
>>> second_str = 'AWESOME!'
>>> print(first_str, second_str)
```

Notice that a space is also a character. When using the print function, you can put all the strings that you want to concatenate inside the parentheses (round brackets) and it will automatically concatenate the strings and add spaces between them.

Functions on strings

len() – This returns the number of characters in a string as an integer

str() – This is used to type cast a different type into a string

```

>>> len('Hello')
>>> s = 'We are learning data types'
>>> len(s)
>>> e = 2.71
>>> type(x)
>>> str(e)
>>> type(e)

```

Indexing

This is a very important concept in programming used to give the position of something inside a large ordered data structure i.e. something that is organized beforehand. An index is the position of an object among other objects e.g. the position of the character 'h' in the string 'hello world'. Counting in Python always starts from 0, this is often very annoying for beginners then you will meet languages that start from 1 and you will be even more annoyed. Consider the string 'hello':

Character	H	e	L	l	o
Position	0	1	2	3	4

In our code, we use square brackets [] (no space) to show that we are indexing as shown below:

```

>>> 'Hello'[0]
>>> 'Hello'[1]
>>> s = 'Python'
>>> s[3]
>>> s[6]           #Why is this an error?
>>> s[-1]          #You can start counting backwards from -1
>>> s[-2]
>>> s[len(s)]      #Why is this an error?
>>> s[len(s) - 1]

```

We can do a few more complicated things like indexing a group of characters at once using a ":" (colon) as shown below.

```

>>> w = 'strings'
>>> w[0:2]          #Gives the characters from position 0 to 1
>>> w[:2]           #If you leave out the zero it starts from zero
>>> w[2:4]
>>> w[3:-1]
>>> w[4:7]          #Why is this an error?
>>> w[:]            #this is how you copy the entire string3

```

If you add a second colon, you can change the order by using a '-1' to reverse the order.

³ Copying is different from assigning a new variable to the old one because the new value stored in a new place in memory.

```
>>> t = 'abcde'
>>> t[:2:-1]
>>> r = 'racecar'
>>> r[2::-1]
>>> r[::-1]           #This is called a palindrome, it looks the
                        same even if you reverse it
```

String methods

These are functions that allow you to make a modified copy of a string. This is always just a copy, if you want to use it, you have to make sure you store it in a new variable or you replace the old variable. Notice how a method is used, you give your string, add a “.” then put the name of the function (no spaces). Never forget the parentheses! (brackets). We will show you how to use a few then give a list and what they do in the toolbox.

```
>>> 'Hello'.upper()    #Give me text to capital letters(upper case)
>>> s = 'ELEPHANT'
>>> s.lower()          #Give me text in lower case (small letters)
>>> print(s)           #s is still the same
>>> s = s.lower()      #s is now updated so that it is in lower case
>>> s.islower()        #Returns True if all the letters are lower
                        case, False otherwise
>>> s.index('e')       #Gives first position of 'e'
>>> s.index('h')
>>> s.count('e')       #Gives the number of occurrences of 'e'
```

To get a full list of all the methods you can use on strings you can go to the python documentation. You can go to the shell and execute **help(str)**. If you find it difficult to read, in IDLE you can go to Help > Python Docs > The Python Standard Library > Built-in Types > Text Sequence Type –str > String Methods. I encourage you to look at the documentation.

Toolbox 3.2: Strings

1. Concatenation
2. Indexing
3. **len()**
4. **str()**
5. If you have declared a string in a variable called **s**, you can use these methods:
 - a. **s.upper()**
 - b. **s.lower()**
 - c. **s.isupper()**
 - d. **s.islower()**
 - e. **s.index(character)**
 - f. **s.count(character)**

Boolean types

To help you make decisions you often need one of two answers, “yes” or “no”. This allows you to know when to do something and when not to do something. Similarly, a computer must have something that works the same to know if the answer to some question is “yes” or “no”. **What are other pairs of words that mean “yes” or “no”?** Python has a special data type that only takes 2 values, **True** or **False**. A boolean type can never be any other value, only those two. Notice that in Python, the first letter of **True** and **False** must be in upper case (capital letters).

```
>>> True
>>> False
>>> type(True)
>>> type(true)                #Why is this an error?
```

Boolean types are usually an answer to a question for example; Do you want to learn how to code? Yes! So a computer needs a way to ask questions and find answers on its own otherwise it can't do smart things. To ask questions a computer uses certain operators listed below:

Toolbox 3.3: Comparison Operators

> greater than	!= is not equal to
< less than	>= is greater than or equal to
== is equal to	<= is less than or equal to

Let's use these below:

```
>>> 3 > 5
>>> 3.5 < 5
>>> 3 == 3
>>> 3 > 'seven'                #Why is this an error?
>>> 2.7 < 1
>>> 'h' == 'H'                 #Why do you get the answer that you get here?
>>> 2 >= 2
>>> 2 != 2
>>> 5 > max(4,5,6)
>>> 5 <= min(7,8,9)
>>> 'yes' == 'yes'
>>> True == True
>>> False == True
>>> 2 >= 3
>>> import math
>>> math.pi <= 2
>>> math.pi == 3.14159326
>>> 2.0 == 2.0
>>> 17 != 'seventeen'
```

Exercise 3.3: True or False?

Write down the answers to these comparison tests and then check if you were correct using the Python Shell.

i) `789 > 789.1` ii) `3 != min(3,4)` iii) `int(3.4) == 3` iv) `(9//4) >= (12%5)` v) `str(3) != 'three'`

Toolbox 3.4 Logical operators and chaining comparison tests

Let's add a few more things to our toolbox. What if we want to check many things for example, I will only go to school if all the following things are True:

and: If I have my books **and** if it is not raining – if I don't have my books I won't go even if it's not raining, if it's not raining I won't go even if I have my books.

or: If I have my books **or** if it is not raining – if I have my books I will go even if it is raining. If it is not raining I will go even if I do not have my books.

in: Used to check if a shorter string is in a longer string⁴

not: This takes the opposite of whatever you put in front of it e.g. not 1>3 is True

Let's do some examples:

```
>>> 2 < 3 and 3 < 4
>>> 3 == 4 or 3 == 3
>>> 'h' in 'hello'
>>> 2 != 7 and 12 <= 45
>>> 2 < (0 or 3)
>>> not True
>>> (3 < 4 and int('three') == 3) or 12 == 7
>>> 2 or 3                                     #Why does this not make sense?
>>> 3 or 2                                     #Why does this not make sense?
```

Exercise 3.4: Is it and and or or and or or?

- a) Evaluate the following tests manually and then confirm them in a shell
 - i) 4 != (3 or 4)
 - ii) '3' == str(3.0)
 - iii) 347 > 5*45
 - iv) 5 > len('Python')
 - v) 'te' in 'tea'
- b) The function **random.random()** gives a random float between 0 and 1. Simply use it the same way you would use max/min but make sure the brackets (parentheses) are empty.
 - i) **Import** the library **random**. The same way we did with **math** in Chapter 2
 - ii) Test the function **random.random()** by typing it into the shell
 - iii) Declare three variables **x**, **y** and **z** as random numbers between 0 and 1
 - iv) Use the comparison operators (>, < and ==) to find out which one is the biggest and smallest number.
 - v) Confirm your answer using the **max/min** function

Variables and Data Types

We now understand what are called the four primitive data types in Python. These are the most basic ways to store data in a high level programming language. So far we have not been strict about what our variables store, we have just been assigning values to variables but as a program becomes more and more complex we have to know which data types we store in our variables.

Consider the following script and explain why it does not work:

```
x = 3
y = '3'
print(x + y)                                     #Why is this an error?
```

This is going to be the source of many bugs in your code so it's important to remember to check your data types. The most useful tool in your scripts is to use the **print()** and **type()** functions as in the example:

```
X = 74
Y = '22'

print(type(Y))                                   #Tells you what type Y is
```

You can even do this in the shell after running because Python stores all the declared variables before the error.

⁴ This more generally checks if an object is inside a data structure. We will see this in chapter 8

A common source of this error is when using the **input()** function. The input function always returns (gives you) a string so if you want to do calculations with the input value you have to type cast it into a number. Write the following code to understand the source of the error.

Wrong Code:

```
x = input('Enter a number: ')
y = 5

print(x*y)
```

Correct code:

```
x = input('Enter a number: ')
x = int(x)                                #convert the str to an int
y = 5

print(x*y)
```

Exercise 3.5: Hosting a party

- a) You are planning to host a coding party and you want to sell the tickets for \$10 each. Because you are so excited, you want to calculate how much money you are going to make based on how many people are going to come to your party. Write a script that does the following things:
 - i) Ask the user how many people are coming to the party and store it in a variable (Remember to name this variable so that it is easy to understand what it stands for and what type it should be).
 - ii) Declare a variable called **price** and set it to 10
 - iii) Make Python print out how much money you are going to make, make sure there is some text so that you understand what the output is, not just a number.
- b) You make a killing from your first coding party and decide this is what you want to do for a living. So now you want to improve your program so that it works for all the different prices. Change one line in your code so that it asks the user how much he is going to charge for the party. This is a beautiful example of abstraction, because we used a variable to abstract a calculation for a specific situation, we only have to change one line in our code to make it work for all situations.
- c) [Supplementary] You want to make your program a bit nicer. Implement (add) as many of the following features as you can:
 - i) Make it greet the user first e.g. with “Hello, Party Host”
 - ii) Make it ask how many parties there are going to be and add another output that tells you how much you are going to make from all the parties.

Chapter 4: Introduction to functions

We have been using functions all along, for example, the `print()` function. **What other functions have we used? How do you know it's a function?- parantheses (round brackets).** In this chapter, we will try and explain what a function is and how we can write our own. A function is basically some code that was written before so that we can use it over and over again. The print function is actually more than 100 lines of code. Imagine having to write out all that code every time you wanted to print some output! Instead, all we have to do is call the print function and it will do the job.

To help make a clearer picture, let's go back to Ordinary Level mathematics where you have defined functions before. Defining a function is writing down the instructions that must be carried out and giving them a name. For example:

$$f(x) = 2 * x + 1$$

Here we have defined a function called **f** which takes in a number (parameter) called **x** and gives you the answer to **2*x + 1**. Now we can use this function on any number by replacing x with that number. For example:

$$f(1) = 3$$

$$f(0) = 1$$

$$f(2) = 5 \text{ etc.}$$

We can even have a function called **add(x,y)** which adds two numbers **x** and **y** by defining:

$$\text{add}(x, y) = x + y$$

That way, **add(1, 2) = 3**

Every function (in mathematics or programming) has 3 parts:

- **Parameters (input)** – These are the things inside the parantheses (brackets). They are the things that a function needs to do its job. For example, if you say `add()` with nothing inside the parantheses, the function does not know what to do and will 'give an error'. Knowing the data type of the parameter is very important because most functions only work for one data type.
- **Function code (processing)** – These are the instructions that the function must follow, these can take any form, it can be any of the coding we have been doing before. In the above example, the function code is **2*x + 1** or **x + y**.
- **Return Value (output)** – This is the final answer after all the calculations are done, for example the return value for **f(1)** is 3. Again knowing the return value's type is important in case we want to use it somewhere else

It's not a coincidence that functions have all three parts of our basic computer model (input, processing and output)

Exercise 4.1: Defining a function

a) Define a mathematical function called **mult** which multiplies three numbers **x**, **y**, and **z**.

We will now learn how to define functions in Python. Functions generally take the following form:


```
def function_name(parameters):  
    #Function code/statements  
    return value
```

All these things in bold must **always** be there when defining a function. Here is a checklist for writing functions:

- **def** is a Python keyword like **import**, it tells Python that you are about to define a function. As with other keywords it should never be used as a variable name.
- function_name can be replaced by anything that you want to name your function. The rules for naming a function are the same as for variables but we never really use upper case letters as this might confuse some readers.
- The parentheses **()** must always be included, this is where we put the parameters, which are usually variables. If the function does not need any parameter, we leave them empty but they should always be there.
- The colon **:** is very important and should never be left out. A lot of beginners forget the colon and this causes lots of errors. You have been warned. There are many more situations where we will use a colon. A colon means that the next line should be indented.
- Indentation – The function code and return statement should always be indented from the line of the def keyword. Indentation is just a blank space made using **one** TAB button on the keyboard. This is the most common source of error from beginners especially when they start writing longer code. We encourage you to do the indentation exercises.
- For the function code, we can write any instructions that we want e.g. **print('Hello')** etc.
- **return** is another keyword but it is optional, we include it when we want the function to give us back something like an answer to a calculation or some string etc.

Let us write the **add(x, y)** function in a Python script:

```
def add(x, y):  
    answer = x + y  
    return answer
```

Go through the checklist above and make sure everything is included. To use our function, we have to “call” it, just writing down the function will not make it do anything. This is how we call a function, below the above line add the following line:

```
def add(x, y):  
    answer = x + y  
    return answer  
  
print(add(1,1))
```

Notice that when we are done defining the function, our next line of code is not indented from the **def** line, this is how Python knows you have finished defining your function. What happens here is that the function add is called with parameters 1 and 1 and then it adds them and returns the answer which is 2 and then it prints the answer. **Why does the following code not print anything?**

```
def add(x,y):  
    return x + y  
  
add(1,1)
```

Exercise 4.1 – f of x

a) Why does the following code give an error?

```
def f(x)  
    return x + 3  
  
print(f(3))
```

b) Why does the following code give an error?

```
def g(x):  
    return x + 3  
  
print(f(3))
```

c) Why does the following code not print anything to output?

```
def h(x):  
    return 2*x + 3
```

d) Write the following function in Python and check that it works: $i(x) = 9x + x^2$

e) Write a function that subtracts x from y (finds $y - x$) and returns the answer

Let's write a few more functions as practice to show that functions are not mathematical only, they can do anything that can be written in code. Below is code that takes a string called **name** as a parameter and then whatever the string is, it makes it all lower case and then capitalizes the first letter. This might be useful for example if you want Python to write someone's name properly. First let's do this without a function:

```
name = "ben"  
name = name.lower()  
name = name.capitalize()  
print(name)
```

Run the above script to see what it does. Change the name e.g. put upper case letters or something crazy to see that it always gives you the string with only the first letter capitalized.

Imagine if you had to do this 10 times in your code i.e. you had to declare 10 variables for the 10 names you wanted to capitalize and you had to write those 4 lines of code 10 times. It would be messy and boring. Here is a golden rule in programming that is written in bold because it is very important:

Never write the same code more than once

If you see yourself doing this, it's a sign that you should use a function. Let's do it below:

```
def cap_name(name):
    name = name.lower()
    name = name.capitalize()
    print(name)

cap_name("ben")
```

Now we have solved the problem and we never need to solve it again, all we need to do is to call the **cap_name()** function.

Let's do a few more examples just to get used to defining functions. The description of what the function does is written in green inside the three quotation marks ("). You **do not** need to type this for your function to work, it is called a docstring and when you get used to defining functions you should always write your own, so that people reading your code can understand what it does.

Example 1

```
def division(x,y):
    """
    Takes a number x and divides it by then it prints the
    result of floor division and the remainder
    x -> number
    y -> number
    returns None #This means there is no return statement
    """
    quotient = x // y
    rem = x % y
    print(quotient, "remainder", rem)

division(14,5)    #This is to test the function, you can put
                  #any numbers, numbers only.
```

Example 2

```
def circle_area(radius):
    """
    Calculates the area of a circle with a radius given by
    the variable "radius"
    radius -> number
    returns -> float
    """
    import math
    area = math.pi * (radius ** 2)
    return area

print(circle_area(3))    #Check that this is the area of a circle
                        #with radius 3
```

Example 3

```
def is_even(x):
    """
    Checks to see if the number x is even. This is the same as
    checking if it is divisible by 2
    x -> number
    returns bool
    """
    return (x%2) == 0

print(is_even(2))
print(is_even(3))
```

There is something important you should notice, this is the difference between the **print()** function and the **return** statement. The **print()** function inside your function just prints things to output, it does not store them anywhere and they cannot be used for calculations or for some other purpose. The return value is a value that we can do anything for example if the area of a circle is 5, we can do whatever we want with it. We can store it in a variable called **area** or multiply it by 7 or we can do what we did in the last example and print it. It is more useful to use return statements and then print outside the function and it is encouraged that you do this unless we say otherwise.

This is the last chapter in the first section to do with abstraction and data types. The two ways we have learned how to abstract is through the use of variables where make formulas so that we can do a calculation for different values easily. The second form of abstraction is the use of functions which allow us too write a set of instructions that work for one input to work over and over again for many inputs without us having to rewrite all the code all the time.

Ex 4.2: That was not my inden(ta)tion

- a) All of the following code snippets have one problem with syntax (in this case, indentation), find it.

i) `

```
def say_hi():
    answer = "Hi"
    return answer

print(say_hi())
```

ii)

```
def my_upper(text):
    return text.upper()
    print(my_upper("Hi"))
```

iii)

```
def f(x,y):  
    def g(z):  
        return z + 1  
  
    return x * g(y)  
  
print(h(2,3))
```

- b) Explain what each of the above functions does and then fix them and try them out.
- c) Define a function called `hello_world` that prints out the string "Hello World". (Does it need a parameter?)
- d) Define a function that prints out any string it gets as a parameter in capital letters and returns `None`
- e) Define a function that checks if a parameter `x` is divisible by 6.
- f) Define a function that calculates the volume of a square of any side.
- g) Define a function that calculates the area of a rectangle (How many parameters do you need?)

Chapter 5: Flow Control

For computers to be more useful than an ordinary calculator, they should be able to 'make decisions'. This means that given many options, they can choose the proper one and do the right thing. Consider how you make decisions every day, here are some examples of how you might think:

Q: **If** I am hungry

A: Eat

Q: **If** it is 6AM and if I have to go to school

A: Wake up

Q: Or **else** (it is no 6AM or I do not have school)

A: Continue sleeping

Q: **If** I am bored

A: Watch TV

Q: or **else if** someone is watching the TV

A: Read a book

Q: or **else**

A: Do my homework

This is usually what happens in your mind. You ask yourself questions and then based on whether the answer is yes or no, you do something. Python can do the same, **how does Python ask questions and what are the possible answers?** This is where booleans are extremely useful if you were wondering. In the above examples, the tests (questions) are asked in a specific way using three methods; **if**, **else if** and **else**. You are encouraged to ask yourself questions in this way to get the hang of it, it also helps you translate your ideas into code.

To make decisions, Python uses what are called if statements which have the following syntax:

```
if conditional_is_True:
    #Follow these instructions
```

Notice the colon at the end of the line which must always be there, followed by an indentation of the instructions. We replace conditional_is_True with a boolean statement such as those that we used when we were learning about booleans or a function that returns True or False. We then replace the comment with the actions we want to take if and only if the boolean is True. e.g. for our previous example:

```
if i_am_hungry():
    eat()
```

We have replaced the actions with functions. Let's do some example scripts:

```
if 2>1:
    print("2 is greater than 1")
#What is the boolean value?
#Must be indented
```

Of course this is not useful since 2 is always greater than 1 so it will always print the statement. Let's say we want to test if a random number **x** is greater than 5 and print a statement if it is. This number can be anything. **(If it can be anything what do we use to represent it?)** Here is the script that checks.

```
x = 10

if x > 5:
    print(x, "is greater than 5")

print("Done!")
```

We chose the `x = 10` randomly, we could have chosen any number we want to test. **Why do I have to declare `x` as something for this code to work?** If `x` is nothing, how do I compare it to 5? `x` has to be a number so that I can compare it to 5 so we have to say `x = something`. Let's make this code more useful so that it asks the user for a number and tells the user if the number is greater than 5.

```
x = input("Enter a number: ")
x = float(x)                                #Why is this line important?(Ch. 3)

if x > 5:
    print(x, "is greater than 5")

print("Done!")
```

Test this with different numbers and see what it does.

If-Else Diagrams

These are diagrams which show the order in which instructions are going to be run. They are very useful when you start using flow control. You always start at the same point and end at the same point but what happens is determined by the conditionals. Each path is called a branch of the if statement. Anything under True is indented, everything under False or with nothing starts on the same line as the if statement. You can use this to help you know when to indent.

Start

Finish

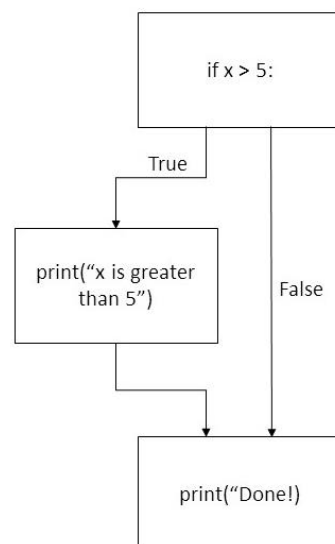


Figure 4

You will notice that our first example does not print anything if the number is less than 5 or equal to 5. We want to account for these cases as well. Let's start with the case where `x == 5`. We will now use an **elif** (pronounced else if) statement. These statements mean exactly what they mean in English. An **elif** has the following syntax:

```
if conditional_is_True:
    #Do these instructions and ignore all the following elifs
elif other_conditional_is_True:
    #Do these instructions and ignore all the following elifs
```

Here are the important things about elif statements:

- i. An **elif** statement always comes after an **if** statement, you will never start with an **elif** statement. This makes sense if you think about it in English grammar.
- ii. The **elif** statement starts on the same line as the if statement. If this does not happen you will get an error.
- iii. You can add as many elif statements as you like e.g. if there are a thousand options, you can include them all.

Let's improve our previous example:

```
x = input("Enter a number: ")
x = float(x)

if x > 5:
    print(x, "is greater than 5")
elif x == 5:
    print(x, "is equal to 5")

print("Done!")
```

Try this out and see what it does, especially with 5 and with numbers greater than 5. It's important to understand how this works. Each **if** or **elif** statement is called a branch because these are all the different branches you code can follow. See Figure 5-1 for a drawing of this. The code first tests if x is greater than 5, if this is True, it prints x is greater than 5 and then ignores the elif statement and goes to the next block of code on the same line as the if statement (i.e. **print("Done!")**). If x is not greater than 5 (the first branch test is False) it then goes to the second branch and tests if x is equal to 5. If x is equal to 5, it prints x is equal to 5 then goes to the last line. If x is not equal to 5, it ignores the elif instructions and just goes to print "Done!". Here is the if-else diagram for the above script.

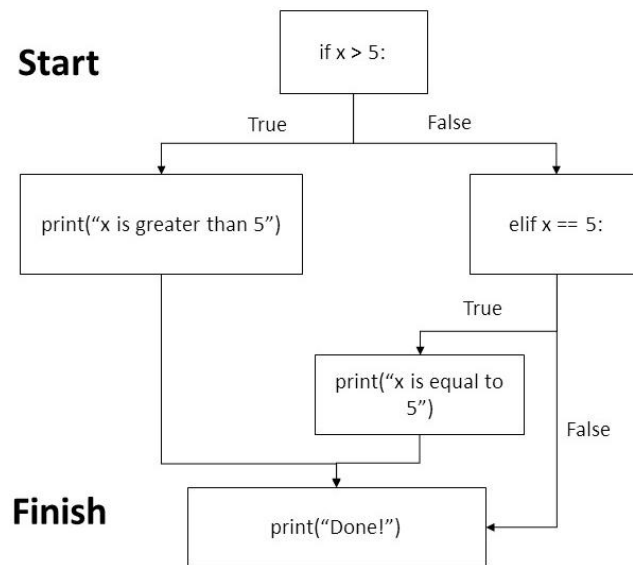


Figure 5

Now let's deal with the last case where x is less than 5. Here we can use another elif statement the same way we for the last case or because we know that this is the only other option, we can use an else statement. Else statements take the following form:

```

if conditional_is_True:
    #Do these instructions and ignore all the other elifs and else
elif other_conditional_is_True:
    #Do these instructions and ignore all the other elifs and else
else:
    #Do these instructions
  
```

Things to remember about else statements:

- i. They always come after if statements or elif statements. Never put them before an elif statement
- ii. You do not have to put a conditional for an else statement because it catches all other cases

Applying this to our example:

```

x = input("Enter a number: ")
x = float(x)

if x > 5:
    print(x, "is greater than 5")
elif x == 5:
    print(x, "is equal to 5")
else:
    print(x, "is less than 5")

print("Done!")
  
```

Here is the if-else diagram:

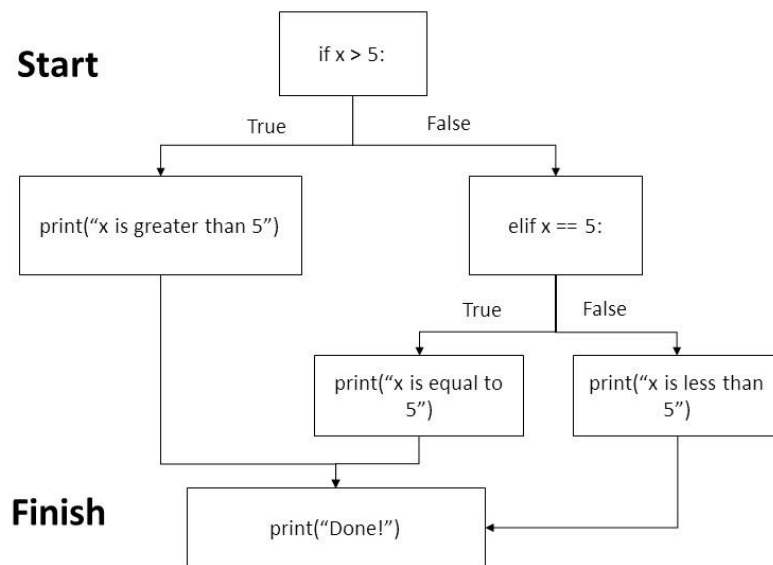


Figure 6

To finish this example, we will do one more thing which is very good to do when dealing with some input (e.g. user input) and returning an output. We will put this all into a function that checks if a number is greater than 5 and call it **compare_with_5(x)**. Remember that function code must be indented from the line with **def** so we have to indent everything from the if statement to the end as shown below. We also changed the names of some of the variables to avoid confusion:

```

user_input = input("Enter a number: ")
user_input = float(user_input)

def compare_with_5(x):
    '''
    Compares x with 5
    x -> number
    returns None
    '''
    if x > 5:
        print(x, "is greater than 5")
    elif x == 5:
        print(x, "is equal to 5")
    else:
        print(x, "is less than 5")

    print("Done!")

compare_with_5(user_input)
  
```

Let's do one more example. **Need second short example**

Toolbox 5.1: if statements

- i) **if** statement
- ii) **if** and **elif** statements

- iii) **if** and **else** statements
- iv) **if**, **elif** and **else** statements

Exercise 5.1: If this then that

- a) The following function checks if a number is divisible by 2. If a number called **n** is divisible by 5, this means that **n % 5** is zero. The function returns True if the number is divisible by 5 otherwise it returns False. Find the 2 errors in indentation. Type it out with you correction and see if it works.

```
def is_divisible_by_5(n):
    '''
    Checks if n is divisible by 5
    n -> int
    returns bool
    '''
    if n % 5 == 0:
        return True
    else:
        return False
```

- b) The following script checks asks the user to input their name. If their name is less than 5 characters (e.g. Ben), it prints "Your name is short". If your name is between 5 and 8 characters (e.g. Thomas), it prints "Your name is okay". If your name is longer than 8 characters (e.g. Uvuvuvwevwevwe Onyetenyevwe Ugwemubem Ossas) it prints "You have a long name. Find the errors in indentation and fix them in your own code. Try out this code for yourself

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

if len(name) <= 5:                    #Remember len()? (Chap. 3)
    print("You have a short name")
elif len(name) <= 8 and len(name) > 5:
    print("Your name is average length")
else:
    print("You have a long name")
```

- c) Draw if-else diagrams for the a) and b)
- d) Define a function called **is_even(n)** that checks if an integer **n** is even and returns a boolean if True/False
- e) Write a script that asks the user for a word and then checks if the first letter of that word is in upper/lower case and prints a response. (Hint: Remember indexing (Chap. 3) and you might find the string method **str.is_upper()** useful)

Supplementary exercise 5.1

- a) You want to help out your school because it takes a long time for teachers to mark so you want to write a script that helps them mark faster which automatically gives you a letter grade for a mark. Define a function that called **find_grade(x)** that returns a letter grade (e.g. 'A', 'B', 'C', 'D', 'E', 'U') as a string if **x** is between certain values, use the values your school uses. (Hint: Draw an if-else diagram first)
- b) Different classes use different cutoffs for As and Bs etc. how would you change your function so that it is easier to set what an A or a B is (Hint: Abstraction)

Chapter 6: Loops

While Loops

Flow control gives Python the ability to make decisions. Now we need some way for Python to know how many times it should do something. Imagine that there is something you have to keep doing until something changes as in the examples below.

While I am alive, breathe

While the teacher is talking, listen

While schools are open, go to school

Here we are using what is called a **while** loop. It basically means we want to do something over and over again until something changes or in Python language, conditional becomes False. The syntax for a while loop in Python is given below:

```
while conditional_is_True:
    #Do this
```

As usual we have a colon at the end of the line with the while statement meaning we should indent the next line from the **while** keyword for all the code that is within the while loop. To add code outside the while loop, we go back to the line with the **while** statement.

Let's do an example and then explain how it works step by step.

```
i = 0

while i < 3:
    print(i)
    i += 1
```

Here is a workflow for how the above code works:

- Declare **i** = 0
- Check if **i** is less than 3, it is so move to the code block in the while loop
- print(0) then add 1 to **i** so **i** is now 1
- Go back and check if **i** is less than 3, it is so move to the code block in the while loop
- print(1) then add 1 to **i** so **i** is now 2
- Go back and check if **i** is less than 3, it is so move to the code block in the while loop
- print(2) then add 1 to **i** so **i** is now 3
- Go back and check if **i** is less than 3, it is not so break out of while loop and execute any code outside the loop

An important thing to notice is that we declared **i** outside the while loop. This should always be the case if you are incrementing/decreasing a variable in a while loop otherwise **i** will not be defined when it runs its first conditional test.

A while loop is best used when you do not know how many times you are going to do something just like the scenarios given at the beginning of the chapter. To give a code example, we are going to make a small game where the computer generates a random number between 1 and 100 and the

user has to guess it. We will build each part step by step starting with the code that will test if a number is correct. This is the same idea as the one we used in the last chapter.

```
import random
number = random.randint(0,100)
guess = input("Guess the number:" )
guess = int(guess)

if guess == number:
    print("You guessed correctly!")
```

The first two lines are used to produce a random integer. The first line imports the **random** module, we have imported before. The second line uses the **random.randint(a,b)** function of the random module which produces a random integer between **a** and **b**. Don't forget that we need to type cast the input to an integer as on line 4! Execute this code and see what it does.

You may have realised that the above code is pretty bad. You only get one chance and you have to keep running the code. Let's implement a while loop which will make this game much more playable.

```
import random
number = random.randint(0,100)
guess = input("Guess the number:" )
guess = int(guess)

while guess != number:
    guess = input("Guess the number:" )
    guess = int(guess)

print("You guessed correctly!")
```

Now this allows you to guess multiple times until you get the correct answer but if you've played you'll realise that this game is actually very hard. It would be better if there were hints to tell you if your guess is too high or low. While we could use the previous code, we will choose to do this in a way that is good practice, and makes more sense when using while loops and it also shows you other ways in which we can use while loops.

```
import random
number = random.randint(0,100)

while True:
    guess = input("Guess the number:" )
    guess = int(guess)

    if guess == number:
        print("You guessed correctly!")
        break
    elif guess < number:
        print("Your guess is too low.")
    else:
        print("Your guess is too high")
```

This is now much easier to play. Here the conditional for the while loop is True, this means the while loop will be executed forever as the conditional is always True unless we have a **break** statement inside the while loop. This is another Python keyword that is the same as making the while loop False and “breaking” out of it. This is a very useful keyword when working with while loops.

There are a couple of reasons why I claim this is better code. For one notice that we are not writing the code to get input from the user twice. Secondly the three different cases are clear from the use of the if statements. Thirdly the use of the break statement means with only one line of code we can control the while loop.

Let’s implement a few more features to our code, for example let’s make it tell us how many guesses it took to get the right answer. Also we will put all this code into a function as usual and include comments.

```
def guessing_game():
    '''
    This is a number guessing game where the user guesses a random
    number between 0 and 100
    '''

    import random
    number = random.randint(0,100)    #Get a random integer
    num_guesses = 0

    while True:
        guess = input("Guess the number:" )
        guess = int(guess)
        num_guesses += 1

        if guess == number:
            print("You guessed correctly!")
            print("It took you", num_guesses, "guesses")
            break
        elif guess < number:
            print("Your guess is too low.")
        else:
            print("Your guess is too high")
```

Play this game a few times and challenge your friends to see who can guess the number with the fewest guesses. How can you play this game so that you make sure that you always get the answer in less than 8 guesses?

Exercise: While I have time

- Write a script that prints the words “Disaster!” while $1 < 17$. What happens and why? (To make it stop, press Ctrl + C in the shell.
- Write a script that prints the number x while $x < 10$ where x is initially 1. Remember where you have to declare your variables and how you make sure that one is added everytime the loop runs.
- Write a script that asks the user if they want to stop running the script, if they want it to stop, they type in yes and it stops otherwise if they type in something else, it just keeps asking. e.g.

```
>>> Do you want to stop?: no
>>> Do you want to stop?: hai
>>> Do you want to stop?: kwete
>>> Do you want to stop?: okay
>>> Do you want to stop?: yes
>>>
```

For Loops and iteration

We use while loops when we want to do something until a condition becomes False, in fact that's the only thing we need to loop but sometimes a while loop makes things complicated and in larger programs it is slow and inefficient. One important situation is when you know exactly how many times you want to do something to everything in a certain number of things, to make this clear consider the following real life examples:

- for every homework I have out of many homeworks, I will do it
- for every item in my bag, I will take it out
- for every chance I get out of many chances, I will try and score
- for every Saturday of the week, I will sleep late

Notice how we have phrased the situations, we always do an action (e.g. take it out) to one thing (item) out of many things (Items in the bag). This is called iteration, iteration is doing the same thing over and over, in this case we are doing the same thing over and over to all the things inside an "iterable" (something that contains many things).

Exercise 6.: Iterate on iteration

Pick an object around you (anything! a person, a bag, a table) and iterate any action (reasonable action) you want over it, try and phrase it the way the examples above are phrased.

Below is the syntax for a **for** loop:

```
for item in many_items:
    #Do something
```

for and **in** are Python keywords, the line with a for loop ends with a colon so the next line is indented and to write code outside the loop, we go back to the line with **for** (you should be understanding this by now!)

item is a variable that you do not declare beforehand, that first line is the declaration of the variable item. If you want to do something to the items in many_items, this is the variable you use to refer to the item.

many_items is an iterable which can be stored in a variable declared before the for loop, the only iterable you have learned so far are strings, strings are iterables because they are made up of many letters/characters. Integers, floats and booleans are not iterables!

Below is an example which might give you a better idea:

```
for char in "hello":
    print(char)
```

Notice that it takes every character in the string “hello” and prints it i.e. for every character in the string, print the character on a new line.

Exercise 6. : Everyone up goes down, everyone down goes up

- b) Define a function that takes a string as input and prints every letter of the string in upper case on a new line. e.g. if the input is ‘Atm’, the output should be:

```
>>>
'A'
'T'
'M'
>>>
```

- c) Modify your code from a) so that the function changes the case of the letter before it prints i.e. if it is lowercase, make it upper case, if it is upper case, make it lower case e.g. for ‘Atm’

```
>>>
'a'
'T'
'M'
>>>
```

Now we can iterate over strings, let’s introduce to another iterable called a range. A range gives you asset of numbers that follow a pattern e.g. the range from 0 to 5 with a spacing of 1 gives me the following numbers: [0, 1, 2, 3, 4]. Notice that the last number of the specified range “5” is not included. We create a range using the inbuilt function **range()**. The range I just gave is created by calling **range(5)** The syntax of range is given below

```
range(start, stop, step)
```

start -> the first number of the range (float but if stop is not #given it must be an integer)

#stop -> the number after the last one in the range (float, optional)

#step -> the spacing between numbers in the range (float, optional)

More examples of range are given below:

Function Call	Range
range(4)	0,1,2,3
range(2,9)	2,3,4,5,6,7,8
range(2,10,2)	2,4,6,8
range(0.4,1.8,0.2)	0.6,0.8,1.0,1.2,1.4,1.6
range(0,50000,10)	0,10,20,30,.....,49980,49990
range(-2*math.pi,2*math.pi,math.pi)	-2π,- π,0, π

When we use it in a for loop we do the following:

```
for i in range(4):
    print(i)
```

You may have noticed that the for loop prints all the numbers in the range we specify. we can use these numbers stored in the variable **i** however we want. The following example adds all the

numbers from 1 to 5. Work it out by hand by first writing out the range like in the table and adding all the numbers together.

```
sum = 0

for i in range(6):
    sum += i

print(sum)
```

Notice that we have to declare the variable `sum` outside the for loop. **Why? Try declaring `sum` inside the for loop and see what happens.**

This example adds all the even numbers from 10 to 100:

```
sum = 0

for i in range(10,102,2):
    sum += i

print(sum)
```

We can use string indices (plural for index) to iterate over a string, this is actually more common in other languages, Python is nice in that we can iterate over a string without using an index. The following script makes a new string with word “loops” from `my_str`.

```
my_str = 'for loops are cool'
loops_str = ''                #Try without this line and see what
                               #happens

for i in range(4,8):          #Think about why we use this range
    loops_str += my_str(i)

print(loops_str)
```

Exercise 6. : Loop for loops

- Print the number 34, 34.5, 35, 35.5,40 using a for loop
- Print the letters of the string 'I heart loops' one by one without indexing
- Repeat b) using indexing. Print just the word loops from the string in b) using indexing. Remember the function **length()**

Supplementary Exercise(Mathematics): Arithmetic and Geometric Series

a) An arithmetic series is a sequence of numbers starting from a with a spacing of d e.g. if a is 1 and d is 3, the series is 1,4,7,10,....., the formula for the n th term is given by:

$$T_n = a + (n - 1)d$$

The sum of the series(S_N) to the n th term is the result when you add all the first n numbers of the sequence so:

$$S_N = \sum_{n=1}^N T_n$$

for the example I gave, the sum up to the 5th term is:

$$S_5 = \sum_{n=1}^5 T_n = 1 + 4 + 7 + 10 + 13 = 35$$

Write a script that finds the sum of an arithmetic series with $a = 12$, $N = 100$ and $d = 3$ using a for loop, to check your answer use the formula for the sum up to N which is

$$S_N = \frac{N}{2}(2a + (N - 1)d)$$

b) A geometric series on the other hand is a sequence made by multiplying the first number of the series a by a constant r $n-1$ times, so:

$$U_n = a * r^{n-1}$$

e.g if a is 1 and r is 2, the geometric series is 1,2,4,8,16,..... The sum of this series is

$$S_N = \sum_{n=1}^N U_n = a * r^1 + a * r^2 + a * r^3 + \dots + a * r^{N-1}$$

so for our example:

$$S_4 = \sum_{n=1}^4 U_n = 2^0 + 2^1 + 2^2 + 2^3 = 15$$

Write a script that finds the sum (S_N) up to the 5th, 10th, 50th, 1000th and 10000th term of the series with $a = 1$ and $r = 0.5$. Try it with any r less than 1, what do you notice about the sums when N is very large. Use the following formula to check your answers:

$$S_N = \frac{(1 - r^N)}{1 - r}$$

Imagine if you had to do the sum by hand!

An important use of a for loop is for counting the number of times to do something, for example, if I want to print "Hello" 5 times, I can use the following code:

```
for i in range(5):  
    print("Hello")
```

We will now do an interesting example where we will draw shapes using the computer using a module called turtle graphics which comes with Python 3.5. This package is not builtin so we will have to use the **import** keyword The package has the following functions, taken from the documentation, which is important to learn how to read!

```
turtle.forward(distance)
turtle.fd(distance)
```

Parameters: **distance** – a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
turtle.back(distance)
turtle.bk(distance)
turtle.backward(distance)
```

Parameters: **distance** – a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
turtle.right(angle)
turtle.rt(angle)
```

Parameters: **angle** – a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

If you've never seen documentation before, you should be amazed at how straight forward and clear good documentation is to read. These are the three functions we will use for now, how would you use these to draw a square?

Here is some code that will do it:

```
import turtle

turtle.fd(100)
turtle.right(90)
turtle.fd(100)
turtle.right(90)
turtle.fd(100)
turtle.right(90)
turtle.fd(100)
turtle.right(90)
```

If you remember our golden rule when we were talking about functions, you will realise that this is terrible coding, we're typing the same code over and over again! A cleverer way to do this is to use a for loop. Whenever you do the same action over and over again for a certain number of times (that you know), you are iterating and you MUST use a for loop. Here is a better script:

```
import turtle

for i in range(4):
    turtle.fd(100)
    turtle.left(90)
```

Exercise 6. : How turtles get in shape

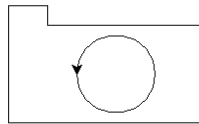
- a) Use the turtle module to draw a rectangle of sides 200x300.

- b) Use the turtle module and a for loop to draw an equilateral triangle and a hexagon of side 250. **What are the exterior angles of an equilateral triangle and a hexagon?**
- c) Given that **turtle.right()** is a function, can you guess another function that probably exists? You can type in:

```
>>> import turtle
>>> help(turtle.FUNCTION)
```

where **FUNCTION** is replaced by what you think is the function name to see if it exists and what it does. It turns out this works a lot of the time! You can guess the name of a function that probably exists, and google it or use the help function to see how it works.

- d) (Supplementary) Use **help()** or Go to the **turtle** documentation (In IDLE, go to help>Python Docs >Search then type in turtle and choose the first option on the list) and lookup how to use the **turtle.circle()** and **turtle.setpos()**, **turtle.penup()**, **turtle.pendown()** from the list, if you have internet you can google the functions directly) . Draw the camera below, choose your own measurements (Hint: The canvas (white space) is like a cartesian plan with an origin (0,0) at the centre)



Chapter 7: Data structures I (Lists, Tuples and Strings)

So far we have been dealing with data types and variables that store one object e.g. `x` can store an integer or a string etc. **What if I wanted to store 1000 numbers, would it make sense that I make 1000 variables?** Obviously not! It would be nice if I could collect all these single objects into one object containing all of them like your bag keeps your books, a car park keeps cars or a building keeps people, furniture etc. **Give at least 10 real world examples of things that contain other things (data structures). Is an empty bag one or should it have something?** Is an empty bag a data These objects that store other objects are called data structures. Data structures are Python objects that store other objects in a predictable way. It is important that they are stored predictably, otherwise there is no way to get information out of the data structure. This is like having a car park with no lines or rules, if cars can park anywhere, they can block each other and then some cannot get out. Another example is a bag, you can know there are things in the bag and that the bag is closed by a zip. If you try and open the bag with a bhobhojane (spanner) will you be able to?

Data structures in Python can store objects of different data types or things of the same type. Do you think it is easier to store things of one type in one data structure or to store many things of different types in one data structure?

You already know a data structure!

I claim you are already familiar with a data structure. Think of an object/data type that you have used that contains many smaller objects. The tools we used with this data structure are the same tools we will use when dealing with other data structures so it is worth revising them. Visit toolbox 3.2 and revise indexing and using string methods.

Exercise 7.1: What method do you use to index?

Declare a string called **mystr** and store the string 'I am a data structure' and do the following:

- Find the index of the letter d and store it in a variable
- Find the length of the string
- Use the variable from a) to print the substring (words) 'data structure' using `my_str`
- Use a for loop and indexing to print all the letters in odd positions of the string 'hahahahaha' i.e. print all the 'a's. Why are the 'a's and not the 'h's in odd positions?

Tuples

The first data structure we will consider is the tuple. A tuple is an ordered collection objects that cannot be changed once they are stored in the tuple (They are called immutable; we will explain this later). Each object inside a data structure is called an element and each element has an index, this is what we mean by ordered. Element number 0 is always element number 0 and the last element is always the last element. The syntax for creating a tuple with 3 elements is as follows:

```
>>> (object_1, object_2, object_3)
```

and to store it in variable:

```
>>> my_tuple = (object_1, object_2, object_3)
```

Python knows that an object is a tuple if it is in parantheses (This is very important!). Each element is separated by a an ordinary comma not a period. Let's practice indexing using tuples in the shell:

```
>>> t = (1,2,3,4)
>>> t[0]
>>> t[1:4]
>>> t2 = (1.2,'hey',5.6,-1.6,'eita')
>>> t2[1]
```

Toolbox 6. Data structure functions

The following are functions we can use with most data structures. Some can't be used on some data structures if it does not make sense.

- i) **length(structure)** – returns the number of elements in a data structure, not the index of the last item!
- ii) **sum(structure)** – returns the sum of all the elements in a data structure, can you do it with a string?
- iii) **print(structure)** – prints the data structure
- iv) **type(structure)** – returns the type of the structure
- v) **max()/min()** – returns the maximum or minimum value of a tuple

Let's practice:

```
>>> t3 = (12,45,13,87,94)
>>> sum(t3)
>>> print(t3)
>>> length(t3)
>>> t4 = ('eish')
>>> sum(s)
>>> t5 = ()
>>> t_copy = t3[:] #Copies the whole tuple
>>> length(t5)
>>> type(t5)
```

Each data structure also has a set of methods associated with it, remember how methods are called from the string methods examples?

Toolbox 7.1 Tuple methods

If the tuple is called **t**

- i) **t.index(object)** – gives the first index of the object given by **object**
- ii) **t.count(object)** – counts the number of objects that are the same as **object**

```
>>> t = (0,1,0,0,0,1,1,0,1,1)
>>> t.count(1)
>>> t.index(0)
```

Tuples, like strings are another example of an iterable, this means we can use a for loop to do something to all the elements in a tuple. Here are two examples explained in the comments.

```
#This script prints all the elements of the tuple t on different
#lines
```

```
t = (1,5.4,7,'g','h',True)
for item in t:
    print(item)
```

```
#This function prints all the numbers in the tuple t that are
#greater than 7
```

```
t = (13,54,2,9,-5,13)

def is_greater_than7(t):
    for number in t:
        if number>7:
            print(number)
```

```
#The function makes a sentence out of the elements of the tuple t
#that are strings only i.e. in this case
```

```
t = ('In',1,True,'life','noone','lives',4,'forever')

def is_string(x):
    """
    Returns true if the object x is a string, False otherwise
    """
    if type(x) == str:
        return True
    else:
        return False

my_str = ''
for item in t:
    #Go through all elements of the tuple
    if is_string(item):
        #Check if the element is a string
        my_str += (' ' + item)
        #Concatenate it to sentence

print(my_str)
```

Exercise 7.2

- Write a function called **has_string(t)** that returns **True** if there is a string inside a tuple t
- Write a function that prints all the even numbers inside a tuple of numbers on different lines
- Write a function that counts all the even numbers of a tuple

Lists

Lists are data structures that are similar to tuples but we can do a lot more with them and we will use them most of the time. Their syntax is as follows:

```
>>> [object_1, object_2, object_3]
```

and to store it in variable:

```
>>> my_list = [object_1, object_2, object_3]
```

Notice that the only difference is that lists use square brackets, this is important! Don't forget it. All the functions, methods and indexing we can with tuples can be done with lists

```
>>> l = (12,45,13,87,94)
>>> l[0]
>>> l[::-1]
>>> l[3,:]
>>> sum(l)
>>> print(l)
>>> length(l)
>>> l3 = ()
>>> print(l3)
>>> type(l)
```

Lists have another feature which is different from tuples, we can change the elements of a list without making a new list. We cannot do this with a tuple. Because of this, lists are called mutable and tuples and strings are called immutable. An example is the easiest way to see this.

```
>>> l = [13,345,8,6,863]
>>> l[2]
>>> l[2] = -0.0001          #Change the 2nd element of l into -0.0001
>>> l[2]
>>> t = (2)
>>> t[0] = 4                #Why is this an error?
```

This is why we usually use lists instead of tuples, because we can change the elements of a list one by one.

Let's see how we can use this property of lists to solve a problem. Let's say we have a form where people put in their names and all the names are added to a list whenever they are entered. Unfortunately, some people capitalize the first letter and some people do not, we can change our list so that all the names have the first letter capitalized. Because we want to capitalize each name out of a list of many names, we use a for loop (Remember this? Whenever you want to do something to each thing among many things, always think of a for loop)

```
name_list = ['Taku','sipho','senzeni','rudo']

for i in len(name_list):
    name_list[i] = name_list[i].capitalize()

print(name_list)
```

Concatenation

We can also concatenate (join) 2 lists to give a longer list just like with strings. This adds the second list to the end of the first one. We cannot do this with tuples, remember that once we have a tuple we cannot change it.

```
>>> [1,2] + [3,4]
>>> s = 'Pyt'
>>> t = ('h','o','n')
>>> l = list(s)+list(t)
```


Toolbox 7.2: List functions

Now let's look at the functions that we can use with lists, here is a list:

- **len(l)** – give the length of a list
- **sum(l)** – add up all the elements of a list
- **random.choice(l)** – gives a random element of a list (like closing your eyes and picking a ball from a bag).
- **list(l)** – this type casts any other iterable like a tuple or string into a list and returns it, the original is kept (see example below)
- **delimiter.join(l)** – if a list is made of strings only, this joins all the strings in the list into one string separated by the string stored in the variable delimiter. So delimiter (it can be called anything else but that is its actual name) has to be defined first before using this.

Here are examples where these are used:

```
>>> l = [1, 5, 8, 9, 3]
>>> length(l)
>>> sum(l)
>>> random.choice(l)
>>> random.choice(l)
>>> s = 'ABCDE'
>>> l2 = list(s)
>>> print(l2)
>>> delimiter.join(l2)           #Why is this an error?
>>> delimiter = '-'
>>> delimiter.join(l2)
```

Toolbox 7.3 List Methods

Now we can go into list methods, there are many more list methods than tuple methods, let's just go down the list of methods this book will use and use them in later examples.

- **l.append(object)** – adds the object to the end of the list l
- **l.extend(list)** – Does the same as concatenation (if concatenation does not work, use this. The reason why is because extend does not care about scope but that's explained in chapter 8)
- **l.insert(i, object)** – put an object before the index given by i. Like putting a book on a shelf at a certain position
- **l.index(i)** – you should know what this does by now...
- **list.pop(i)** – remove and return the object in the position i, like taking a book off the shelf, it no longer exists on the shelf but you have it in your hand and can do whatever you want to it but of course you have to store it in a variable.
- **list.remove(i)** – remove the object at position i and throw it away.
- **l.count(object)** – sound familiar?
- **l.sort()** – sort the elements of the list l from smallest to biggest, it does not return a sorted list, it sorts l.

Now let's practice!

```

>>> l = [4,2,2]
>>> l.append(5)
>>> print(l)
>>> l.insert(1,3)
>>> print(l)
>>> l.count(7)           #Why is this an error?
>>> l.count(2)
>>> unwanted_index = l.index(2)
>>> l.remove(unwanted_index)
>>> l2 = ["let's", 'go!', 1] #See what we did to use ' in our string?
>>> num1 = l2.pop(2)
>>> l.append(num1)
>>> print(l)
>>> l.sort()
>>> print(l)
>>> l.extend(l2)
>>> print(l)
>>> l = l[::-1]          #Remember that this reverses the list
>>> l.extend(l2)

```

It's at this point where an entire world is opened up to anyone who codes. You may have been coming up with ideas for things you want to code but haven't been able to figure out how. Having the ability to store and manage data adds power to all the things you have learned. Let us now consider script examples in using data structures.

Example: The party theme is lists

(I suggest you look back at Exercise 3.5 first) You are back to hosting parties and you have come up with a new idea. You want to offer discounts for people who have attended at least 3 parties so that you encourage them to come again. We are going to write a script that stores the names of all the people who attend each party in a list. Each party will have its own list and there will be a larger list that contains all the parties somewhere else.

Getting attendance

When people get to the door, the bouncer asks for their id and types it in so that he records that they attended. When the bouncer is done he will type in a code (\$money\$) so that he returns all the ids of the people who attended.

```

def get_attendance():
    '''
    This function gets all the names of the people
    who attended one party
    returns -> list of ints
    '''
    attendance = []    #list to people who attended

    while True:        #Keep asking for names
        id_num = int(input("Enter the member id of partygoer: "))
        if id_num == '$money$':    #Party is over, get all the ids
            return attendance
        else:
            attendance.append(id_num) #Add a person's name to the
                                     #list

#Testing your code
print(get_attendance())

```

Who gets a discount

We were able to store all the ids of the people somewhere(we won't explain how) now we have a list for each party. For simplicity we will make 3 lists and put numbers so that we can test each case:

- Someone has not attended 3 parties
- Someone has attended 3 parties and needs a discount
- Lots of people have attended 3 parties
- Someone who has attended more than 3 parties already got a discount so don't give them another one

```

party1 = [1,2] #1 and 2 attended
party2 = [1,3] #1 and 3 attended
party3 = [1,3] #1 and 3 attended, 1 needs a discount
all_parties = [party1, party2, party3]

def find_discount(all_parties):
    '''
    This function checks all the parties(lists) inside
    all_parties(list) and returns all the people who have
    attended discount_num (3 in this case) times
    all_parties -> list
    returns -> list
    '''
    all_attendance = [] #List to store all ids that attend
    discounted_ids = [] #List to store ids that get discounts

    for party in all_parties:
        #Put all the ids of people who attended in one list
        all_attendance = all_attendance + party

    #Checkpoint: print(all_attendance)

    #Get the newest id number
    max_id = max(all_attendance)

    #Checkpoint: print(max_id)

    for id_num in range(max_id+1): #Why the +1?
        if all_attendance.count(id_num) == 3:
            discounted_ids.append(id_num)

    return discounted_ids

#Tests
print(find_discount(all_parties))
#Change the ids in the party1, party2 and party 3 and make sure
#your function works for all cases listed above.

```

Chapter 8: More Functions and Variables

By now you should be able to define functions and use them to solve simple problems. Now we are going to go even deeper into using the power of functions and learn more things we can do with functions to solve more problems.

Scope

Suppose we are in a building where sound cannot pass through the walls, so if I am in the next room and I shout your name, you cannot hear me. We say you are not in my scope meaning if I call your name, you cannot hear me so you will not do anything.

When you define a variable/function in python, it is declared in a scope. This means that only commands that are in the same scope as the variable can use the variable/function/module. Try the following script:

```
print(x)
```

This gives an error because x is not defined, so the print function does not know what to print. Now try the following:

```
x = 2
print(x)
```

x is declared in the scope of the script so print(x) knows that it must print 2. Now try the following script:

```
def f():
    x = 2

f()          #Remember we have to call the function otherwise it never
            #runs the function code!
print(x)
```

This gives an error because x is not defined in the scope of the script aka the global scope. Whenever a function is called, it follows all the function code and then throws everything away except the return value. So if you declare 100 variables in a function but only return 1, all of the other variables are forgotten until the function is called again.

Here are some of the things that can define a scope/frame:

- i. function
- ii. Global (I like to call it script scope)
- iii. folder on a computer and Python modules

A nice way to visualize scopes is using scope/frame diagrams. Frame means the same as scope. These images are taken from pythontutor.com. If you can, you should visit the site and it will help you understand scope better.

The screenshot shows a Python 3.6 IDE with the following code:

```

1 x = 2
2 print(x)

```

The code is executed, and the output is 2. The Frames pane shows the Global frame with the variable x set to 2.

The circle on the left has the code and the circle on the right has all the scopes, in this case they're called frames. This is our first example, you can see that x was declared as 2 in the global frame. Let's consider a second example:

The screenshot shows a Python 3.6 IDE with the following code:

```

1 x = 2          #Declare x
2
3 def f():       #Define f
4     y = 2      #Declare y inside f()
5
6 f()            #Call f()
7
8 print(x)
9 print(y)

```

The code is executed, and the output is 2. The Frames pane shows the Global frame with the variable x set to 2. The function f() is also shown in the Frames pane, with its own scope containing the variable y set to 2. The Return value is None.

Notice that f is also in the Global frame, As we said before, a function name is just like a variable name. A variable name stores numbers, strings etc. and a function name (called a handle) refers to some function code. so x and f are in the global scope and y is in the scope of f() **ONLY** while f is being called, otherwise it does not exist.

We can also have scopes inside scopes and so on but we always have to be careful about which scopes the variables/functions we want to use are defined. The examples in this lesson will explain these things in more detail.

Exercise 8.1: Scope this variable ekse

Find the scope of the following variable/functions: var1, func1, func2, var2, var3. Simple say if it is in the global scope or inside the scope of a function (give the name of the function).

```

var1 = 'I am var1'

def func1():

    var2 = 102

    def func2(var3):
        return var3

    func2(var2)

    return 5

print(func1())

#Remember we always have to call the function, otherwise nothing
#happens!

```

Importing and Scope

If there are variables/functions that exist somewhere outside our script, we can import them into our script. Functions like `print()`, `input()`, `list()` etc. are functions that are always in the scope of a Python script but a function like `random.choice()` has to be imported so that it is available to Python, these are also built-in functions but we need to import them to use them within a scope.

```

import random          #Bring all the variables and functions in the
                        #random module into the scope

random_num = random.choice([1,2,3,4,5])    #call random.choice()
print(random_num)

```

What if you wrote a function a long time ago and want to use it? You can also import that as long as it is in the **same folder**. Write the following functions and save it as **names.py**.

```

def get_my_name():
    """
    Asks the user for their name and returns their name
    returns -> str
    """
    return input("What is your name?")

def print_my_name(name):
    """
    Prints name
    name -> str
    returns None
    """
    print(name)

```

After saving the script, make a new script in the same folder and use the following code to import your function.

```
import names

name = names.get_my_name()
print(name)
```

Now try this code:

```
import names

name = get_my_name()
print(name)
```

This gives an error because `names` is in the scope, not `get_my_name()`. Be careful! To call a function inside another module after importing it the way we did, we should use the period (fullstop) to show that the function is inside the imported module called `names`.

We can actually put the function `get_my_name()` into the scope so that we don't have to call it the long way using `names.get_my_name()`. The **from** keyword allows us to do that. Make a new script and type in the following code:

```
from names import print_my_name      #Don't put the parantheses
print_my_name("Python")
```

This puts the function `print_my_name` into the scope. If we want to put all the functions inside the scope we can use an asterisk (*) like below

```
from names import *      #Import all the functions and variables in
                        #names.py
```

Toolbox 8.1: Importing keywords

import – used to import a built-in module or a python file in the same folder

from module import object – Used to import an object from a module or file in the same folder

Let's use our new tools to define a simple function that calculates the area of a circle and use it in another script. Save the following script with the function as `circle_tools.py`

```
from math import pi

def circle_area(radius):
    '''
    This function finds the area of a circle
    radius -> number
    returns number
    '''
    return pi*radius**2
```

After saving the script, go to the shell and type in the following:

```
>>> from circle_tools import circle_area
>>> circle_area(4)
>>> help(circle_area)
```

Look at what we did in the last line, we used the `help` function and it gave us the docstring for the function. This is why you should always write docstrings for functions, that way you can understand them without even looking at the code.

Exercise 8.2: We import more than we export

- Import the `sin()` function from the `math` object and use it to calculate `sin(1)` in both a script and the shell.
- Import the `random` module and use the `help` function to read the docstring for the `random.randint()` function.

It is good practice to always do all your importing at the top of your code, anything you import in the global scope will be in the scope of every function in your script. It is also important to never use the same names for variables as for modules that you import e.g. don't import `math` and then declare `math = 2`. This will cause errors.

Example: Mini-scrabble

Let's do an example where scope might be a problem. We want to make a word game where the user is given 10 random letters and is asked to make a word out of them. The random characters will be stored in a list. We will generate a random set of 10 letters and print them for the user. The user will then type in a word and we have to check that the user used proper letters that are given in the list. For example, Python will print out:

```
Hi there! Your letters are:
a f j e j e i l g l
What is your word?:
```

The user will then type in a word and we call a function called `is_word_valid(word)` to test if the word is valid. If it is valid Python will print out the remaining characters e.g.

```
Hi there! Your letters are:
a f j e j e i l g l
What is your word?: jail
You didn't use f e j e g l!
>>>
```

The first thing we need to do is generate the random characters, to do this, we will import a variable that has all the lowercase characters from the `string` module called `string.ascii_lowercase` then use a `for` loop to add characters to a list.

```
import string
from random import choice

chars = string.ascii_lowercase #this is 'abcdefghijklmnopqrstuvwxyz'
NUM_LETTERS = 10               #Letters for the game
options = []                   #List to store 15 characters

for i in range(NUM_LETTERS):
    #Add a random choice from the chars string to the list
    options.append(choice(chars))

#Checkpoint: Check that this produces 15 characters using:
#print(options) #List of 10 characters
```

Now that we have the random letters we can add the wording. At the end of the previous script add the following function:

```
def get_word():
    """
    Gets a word from the user
    returns -> str
    """

    print("Hi there! Your letters are:")
    print(' '.join(options)) #Print the chars separated by space
    return input("What is your word?: ")

#Checkpoint: Check that this function works by calling it using
#print(get_word()) #Word that you typed
```

Here's something I have not mentioned because it is bad to get used to it. Notice that even though we did not declare **options** inside the **get_word()** function, we can still use it, in fact anything in the global scope can be access but **NOT** edited inside a function. I cannot append something to options inside the function. **Can you think of why this would be bad anyways?** Functions must be able to stand alone without needing things from outside but it's not always easy as we shall see.

Now we need to define a function that checks if the word is valid. It does this by checking each character one by one, if the character is in the options list, it removes it from the list so that if there are 2 of the same character in the word it counts them twice. If any character in the word is not in the options list that means the word is False. Add the following:

```
def is_word_valid(word, options):
    """
    Checks to see if word uses only characters from the list l
    """

    for char in word:

        if char in options:
            #Remove each letter so that it doesn't count twice
            options.pop(char)
        else:
            #If any character is wrong, return False
            return False

    return True

#Checkpoint: Check that the function works by using:
word = get_word()
print(is_word_valid(word, options)) #True or False
```

Here's the tricky part about this function, because we entered options as a parameter, this means that while the function is working, **options** is only in the scope of the function. No matter how much we chop and change it (No matter how much it screams in this function "room"), the **options** we defined outside the function is unchanged. It is good practice if you follow this rule: Everything that goes into a function must be a parameter, everything that goes out must be returned. So this is one way to change things from outside a function inside a function, then if we want to use it outside we can return it. **But we also want to return the boolean statement! How would we return both things if we wanted to?** We use a data structure, preferably a tuple. e.g.

```

return (True, options)

#or

return True, options #We don't have to put parantheses

```

Now we want to define a function that plays the game called `play_game()`. This function will call `get_word()` to get a word, then it will call `is_word_valid()` to check if the word is valid and finally it will print the remaining characters. The best way to do this is to have options be a parameter and remove all the characters in word like in `is_word_valid()` but to teach you something we will do it differently. We want to be able to change the actual options list that is in the global scope inside the function. To do this we will use the **global** keyword. This keyword puts a global variable inside the scope of the function.

```

def play_game():
    """
    Plays the game
    returns -> None
    """

    global options #Give access to the global variable options
    word = get_word() #Get a word
    if is_word_valid():
        for char in word:
            options.remove(char)

    print(options)

play_game() #Play the actual game

```

Now you can play the game. That was a lot of code! Here's the full script. Enjoy!

```

import string
from random import choice

chars = string.ascii_lowercase #this is 'abcdefghijklmnopqrstuvwxyz'
NUM_LETTERS = 10 #Letters for the game
options = [] #List to store 15 characters

for i in range(NUM_LETTERS):
    #Add a random choice from the chars string to the list
    options.append(choice(chars))

#Checkpoint: Check that this produces 15 characters using:
#print(options) #List of 10 characters

def get_word():
    """
    Gets a word from the user
    returns -> str
    """

    print("Hi there! Your letters are:")
    print(' '.join(options)) #Print the chars separated by space
    return input("What is your word?: ")

```

```

#Checkpoint: Check that this function works by calling it using
#print(get_word())      #Word that you typed

def is_word_valid(word, options):
    '''
    Checks to see if word uses only characters from the list l
    '''

    for char in word:

        if char in options:
            #Remove each letter so that it doesn't count twice
            options.pop(char)
        else:
            #If any character is wrong, return False
            return False

    return True

#Checkpoint: Check that the function works by using:
#word = get_word()
#print(is_word_valid(word, options))  #True or False

def play_game():
    '''
    Plays the game
    returns -> None
    '''

    global options    #Give access to the global variable options
    word = get_word()    #Get a word
    if is_word_valid():
        for char in word:
            options.remove(char)

    print(options)

play_game()          #Play the actual game

```

Recursion

We have said before that we use functions when we want to use a snippet of code (lines of code) in many places. Now we will learn a very powerful technique to solve very complicated problems easily. We will explain this with an example. If I were to ask you to add 1 to an integer, you would consider this very easy, you could always do it. What if I were to ask you to add 2 to an integer? This ofcourse is easy but it's a little harder than adding 1. If I were to ask you to add 234 to a number then you would have to do some thinking. But what does it mean to add 234? It means adding 1 over and over again until you have added 234. **Now let's suppose you only have the ability to add and subtract 1, how then would you add 234?** I would subtract 1 from 234 and try and add 233 but that number is still too big so I would subtract again and try and add 232. In fact, I would keep subtracting 1 until I have to add 1, then that's easy! Then I go back and keep adding 1s I tell I have added 234.

So here is the thought process when I try and add x to y:

```
#Try and add x to y
#If x is 1
#Add 1 because I know how to do it
#else x is not one so let me try and add x-1 to y then add 1 since x
is too big
```

Notice that what we are doing in the first line is the same thing that we are doing on the last line, it's simply changing x to x-1. This means that we can define a function that takes in a parameter x that tries to add x to y otherwise it tries to add x-1 and so on. Here is what our code looks like:

```
def rec_add(x,y):
    '''
    Adds x to y recursively
    x -> int
    y -> int
    returns int
    '''

    if x == 1:          #If x is 1
        return x+y      #Add 1 because we can do it

    else:
        return 1 + rec_add(x-1,y) #Try and add x-1 to y and add 1
```

Recursion means doing a simple operation over and over again to simplify a more complicated problem. The simplest case i.e. the case where $x = 1$ is called the base case, always goes in the if statement (or elif if there are many base cases as in Fibonacci below) and the recursive case always goes in the else branch. This example of course is not very useful because we can simply add two numbers but we will have examples where recursion is actually the simplest way.

Here's a little maths that might help you understand what we are doing if you are into maths. Let's say we wanted to add 4 to 5, we would write this as:

$$total = 4 + 5$$

but we can only add 1's so we simplify our problem by saying:

$$total = 1 + (3 + 5)$$

Then we recursively add 1 by continuing the process:

$$total = 1 + (1 + (1 + (1 + 5))))$$

but we can add 1 to five so now we can say:

$$total = 1 + (1 + (1 + (6))))$$

$$total = 1 + (1 + (7))$$

and so on until we get 9 as our answer.

Let's do another example. The factorial of an integer is defined such that"

$$n! = n(n-1)(n-2) \dots (2)(1)$$

so

$$4! = 4 * 3 * 2 * 1 = 24$$

The only thing we know is that $1! = 1$, this is the only factorial we know for sure so this is our base case. Any other recursive case where n is not 1 is $n*((n-1)!)$. Think about it. $4!$ is the same as $4*(3!)$ which is the same as $4*3*(2!)$ which is the same as $4*3*2*(1!)$ but $1!$ is our base case! Let's write the code.

```
def rec_factorial(n):
    """
    Finds n factorial recursively
    n -> int
    returns -> int
    """

    if n == 1:
        return 1    #We know the answer for 1 factorial

    else:
        return n*rec_factorial(n-1)    #Try and solve (n-1)!
```

How this works is almost like magic, we only need to know the answer for the simple case and now we have the answer for all cases. If you do maths you might have heard of something like this before (induction).

Exercise: Recursive recreation

- Rewrite the **rec_add(x,y)** function but this time, you only know how to add 0.5 to any number. (Hint: It's as easy as you think)
- Rewrite the **rec_add(x,y)** but you can only add 0.0000001. What do you notice about the time it takes to calculate the answer. Why?
- Rewrite the **rec_factorial(n)** function as **for_factorial(n)** and find the answer using a for loop. Find 100! using both **rec_factorial(n)** and **for_factorial(n)**. Which one is faster?

Supplementary Exercise (Mathematics): Fibonacci? Sounds expensive

- The Fibonacci series is famous in mathematics, it is a series of numbers where the n th term is a sum of the 2 previous terms starting from 0 and 1 (or sometimes 1 and 1):

$$F_n = F_{n-1} + F_{n-2}$$

so the series is: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Write a recursive function called **rec_fibb(n)** that finds the n th Fibonacci number. There are 2 base cases, we only know the first and second Fibonacci numbers the rest fall under a recursive case. (Hint: This is only 6 lines of function code, don't complicate the problem for yourself, simply write down the simplest cases and the recursive case. Nothing more)

Chapter 9: Algorithms and Pseudocode .

You may have noticed that our code is getting more and more complicated, both in terms of length and the number of concepts we are using in each line. When things get complicated, it's important to have a plan for how you will solve a problem that you can break down into simple steps that are easy to understand. We will use this chapter to go through an example that may seem complicated at first but we will break it down into simpler problems and solve them one by one.

Demonstration: All sorts of colours

Imagine you are given a box of coloured balls which are in the box. You are also given two boxes labelled 'RED' and 'BLUE' in which to put the balls of the right colour

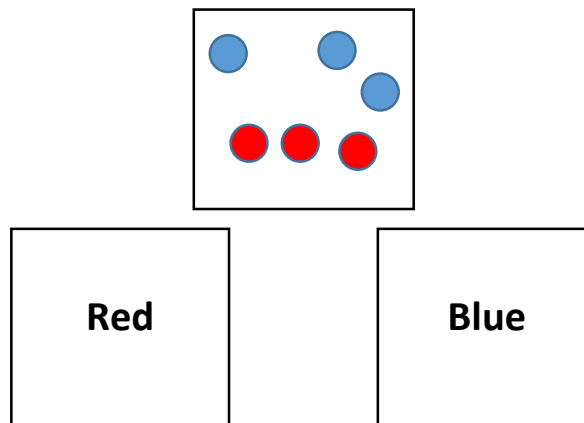


Figure 9-7

There are only a few things (action) you are capable of doing:

1. You can pick one ball from the box (**pick_ball()** returns ball)
2. You can see if a ball is red (**is_red(ball)** returns boolean)
3. You can see if it is blue (**is_blue(ball)** returns boolean)
4. You can put the ball in the RED box (**put_in_red(ball)** returns None)
5. You can put the ball in the blue box (**put_in_blue(ball)** returns None)
6. You can check if the box has a ball (**has_ball(box)** returns boolean)

How would you sort the balls? There are many ways to sort the balls, here's one way to do it:

1. While the box has balls,
2. take out one ball
3. Check its colour. If it is red,
4. put it in the red box
5. Else if it is blue,
6. put it in the blue box

We have given the method we are going to use to sort the balls, any method we choose is called an algorithm. There are many different algorithms we can use but some better than others this is what makes you good at coding, not how you type but how you make algorithms to solve problems.

There are many things that we need to think about when designing algorithms, things like:

- what variables are useful?
- where do we need flow control?

- what kinds of loops (for or while) we will we use?
- which functions we should define to make our code shorter?
- whether we can use advanced methods like recursion etc.

For this problem as we have presented it, try and answer these questions.

We will immediately comment that a for loop will not work, even if we are iterating over a list and we will explain why. For loops iterate over data structures using their indices. That means a for loops looks at object 1 then object 2 then 3 and so on until it gets to object N which is the last object. The for loop only checks the length of the list at the beginning so if there are 20 objects in the list before it started, it thinks there are 20 objects when it finishes but here we are removing balls so the list will become shorter than the indices which gives an error.

Demonstration: What are you hiding for?

Make a pile of five books. Each book, starting from the bottom of the pile will have an index,so the bottom book is book zero and the second book is 1 etc. Someone is going to play the role of a for loop that takes each book in the pile and puts them to the side, remember that a for loop takes the book in position 0 then in position 1 then 2 and so on. **What happens when you want book 4?**

The algorithm has many of the properties of what you would write in your code, we just have to be more specific because a computer is not smart enough to know when something is close to the correct thing or completely wrong. Your algorithm also has to have steps that you know the computer can do. If your algorithm involves a step where the computer eats food, you will never code it!

Let us say that each action corresponds to a Python function given in brackets that was defined by someone and given to us. For this purpose, it doesn't matter how it works, all we know is what it does. This is the power of functions, even if we have no idea how their function code works, we can use them. The return values and parameters of each function are also given.

This is the script, including the pseudocode, we would use:

while has_ball(MIXED):	#while the MIXED box has balls
ball = pick_ball()	#Take out a ball from MIXED box
if is_red(ball):	#If the ball is red
put_in_red(ball)	#Put it in the RED box
elif is_blue(ball):	#Else if the ball is blue
put_in_blue(ball)	#Put it in the BLUE box

Notice that we still have not yet written the exact Python Code that would solve this problem in real life but, we have a working plan (schematic) of what the actual function looks like. The writing down of solutions to problems in this summarized or English-like form is called Pseudocode. The actual steps within the Pseudocode that outlines the sequence of instructions and calculations is called the Algorithm.

Starting out this ways helps us visualize the logic behind a solution and check for its correctness without the presence of executable code. Now, let's try to write out the code for our problem.

Exercise 9.1: Balls in boxes

Suppose that the box containing all the balls is a list where each element represents one ball and name of the element corresponds to its colour, i.e.

```
all_balls = ["blue", "red", "red", "blue",  
            "red", "blue", "blue", "blue", "red", "red"]
```

and the boxes that should contain only red balls and only blue balls are also lists where each element represents one ball and the name of the element corresponds to the its colour, i.e.

```
red_balls = []      #red box  
blue_balls = []    #blue box
```

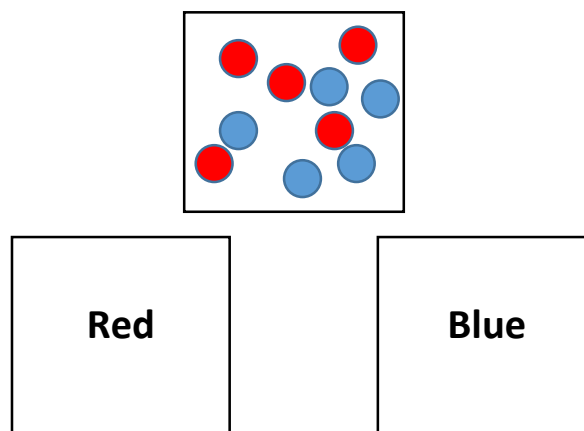


Figure 8

- Write a function `sort_balls` that takes a list `all_balls` containing all the coloured balls, sorts them into two lists according to their colour, i.e. `red_balls` being the list that takes only red balls and `blue_balls` being the list that takes only blue balls. Your function must return the two lists `red_balls` and `blue_balls` and at the end, `all_balls` must be empty because you moved the balls. (Hint: Use a while loop, a for loop will not work in this case. Why?)
- Add one line of code to your solution so that the functions prints the following before returning the two lists:

"I was given a box called `all_balls` which had __ balls inside it. I sorted the balls by colour. Now, `red_balls` box has __ balls inside it and the `blue_balls` box has __ balls inside it."

Exercise 9.1 Solution

We will start off by defining our function:

```
def sort_balls(all_balls):
    '''
    This function takes in a list all_balls with red and blue
    balls and sorts the red balls into a list called red_balls and
    the blue balls into a list called blue_balls
    all_balls -> list
    returns -> tuple of lists
    '''

    #Function code

    return red_balls, blue_balls    #This is another way of
                                    #writing a tuple, you can add
                                    #parantheses if you like
```

For the body, let's recall our pseudocode for this problem:

while has_ball(MIXED):	#while the MIXED box has balls
ball = pick_ball()	#Pick a ball from MIXED box
if is_red(ball):	#If the ball is red
put_in_red(ball)	#Put it in the RED box
elif is_blue(ball):	#If the ball is blue
put_in_blue(ball)	#Put it in the BLUE box

We can see that our solution will contain a while loop. We now need to “replace” the English-like language we have in the while loop to code that will run in Python.

1	while len(all_balls) != 0:	#while the MIXED box has balls
2	ball = all_balls.pop(0)	#Pick a ball from MIXED box
3	if ball == 'red':	#If the ball is red
4	red_balls.append(ball)	#Put it in the RED box
5	elif ball == 'blue':	#If the ball is blue
6	blue_balls.append(ball)	#Put it in the BLUE box

Line 1 -> if the mixed box is the Python list all_balls, what condition has to be True for it to have balls inside of it? It is clear that the length of the list must be more than 0.

Line 2 -> if the mixed box is the Python list all_balls, what does picking a ball inside of it mean? It is clear that we must take out the first element of the list because as long as the box is not empty, there must be a first ball. If we just indexed, we would have another copy of the ball left in all_balls but we want to take it out of all_balls

Line 3 -> if the ball we picked is a string element, how do we check if it is red? It is clear that we must compare it to the string “red”

Line 4 -> if the ball we picked is indeed red how do we put it inside the box of red balls? It is clear that we must add it to the list red_balls

Line 5 -> if the ball we picked is a string element and is not red, how do we check if it is blue? It is clear that we must compare it to the string “blue”

Note – We used an else statement instead of an elif statement because the question does not necessarily state that we will always have a box with only 2 colours. What would happen if you get a box with red, blue and green balls?

Line 6 -> if the ball we picked is a string element and is blue, how do we put it inside the box of blue balls? It is clear that we must add it to the list blue_balls :

Let us combine our replaced statements:

```
def sort_balls(all_balls):  
    while len(all_balls) != 0:  
        ball = all_balls.pop(0)  
        if ball == 'red':  
            red_balls.append(ball)  
        elif ball == 'blue':  
            blue_balls.append(ball)  
    return red_balls, blue_balls
```

Exercise 9.2: INTERNAL affairs

Write a function **counting(list_of_ints, n)** that takes in a list of integers called **list_of_ints** and an integer **n** and returns a **count** where **count** is the number of times that **n** appears in **list_of_ints**. Use the guideline below.

Specifications : lists_of_ints can be empty or non-empty

- Describe an algorithm for this program. You don't have to write code yet, just write down steps that can be taken to make the function work.
- Write the pseudocode for the program.
- Translate every individual line in your pseudocode into Python Code
- Reconcile your translated code into a functional program. Your function must be defined using the standards set out throughout this course.

Exercise 9.3: Are we sorted?

Write a function **is_sorted(list_of_ints)** that takes in a list of integers called **list_of_ints** and returns True if the list is sorted and False if the list is not sorted. A sorted list is a list with elements appearing in ascending order so [1,2,3] is sorted but [3,2,1] and [2,1,3] are not.

Specifications : lists_of_ints can be empty or non-empty.

- Describe an algorithm for this program. You don't have to write code yet, just write down steps that can be taken to make the function work.
- Write the pseudocode for the program.
- Translate every individual line in your pseudocode into Python Code
- Reconcile your translated code into a functional program. Your function must be defined using the standards set out throughout this course.

Exercise 9.4: All sorts of sorts

Imagine you have a list of integers, come up with 2 different algorithms to sort a list of integers in ascending order, they do not have to be good, just possible. For example, one way to sort is to shuffle the numbers check if they are sorted, if they are not shuffle again and check and keep doing

it until they are sorted. (This is called the bogo-sort, imagine if there are 20 integers or more!). You do not have to write code (unless you want to, you have been taught all you need!).

Chapter 10: Working with APIs: Calico

An Application Programming Interface (API) is a set of routine definitions, protocols, and tools for building software and applications. Programming is not supposed to be difficult. APIs make it a lot easier by providing us with a toolbox to things that programmers would want to do lots of times. You can think of an API like a toolbox with screwdrivers, hammers etc. If you want to unscrew a screw, you just use a screwdriver, you don't invent the screwdriver from scratch because people use them all the time even if some people use them on doors, some people on electronics etc. A good API makes it easier to develop a program by providing all the building blocks, which are then put together by the programmer to solve their specific problem.

An API uses regular programming languages (e.g. Python) but some also provide a development environment with useful functions that help us make games, applications, websites and applications.

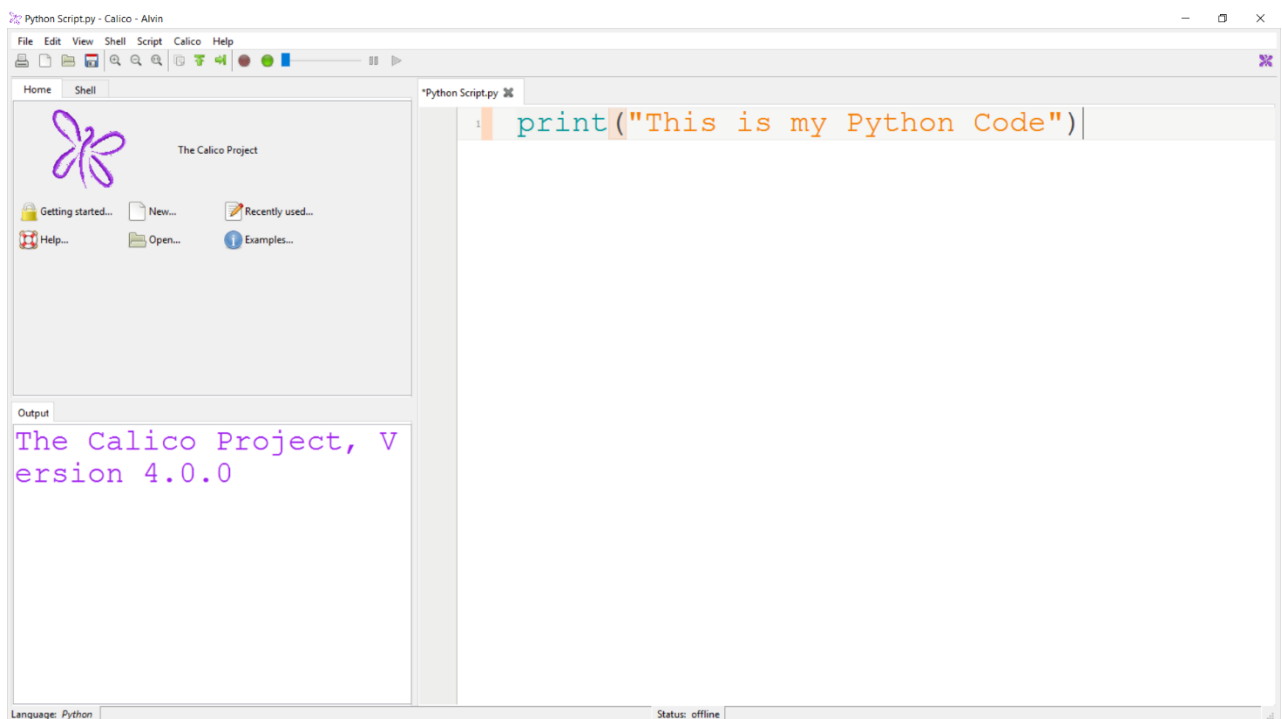
The API we will use for the rest of this chapter focuses on animation.

Project Calico

To install Calico, follow this link: http://wiki.roboteducation.org/Calico_Download

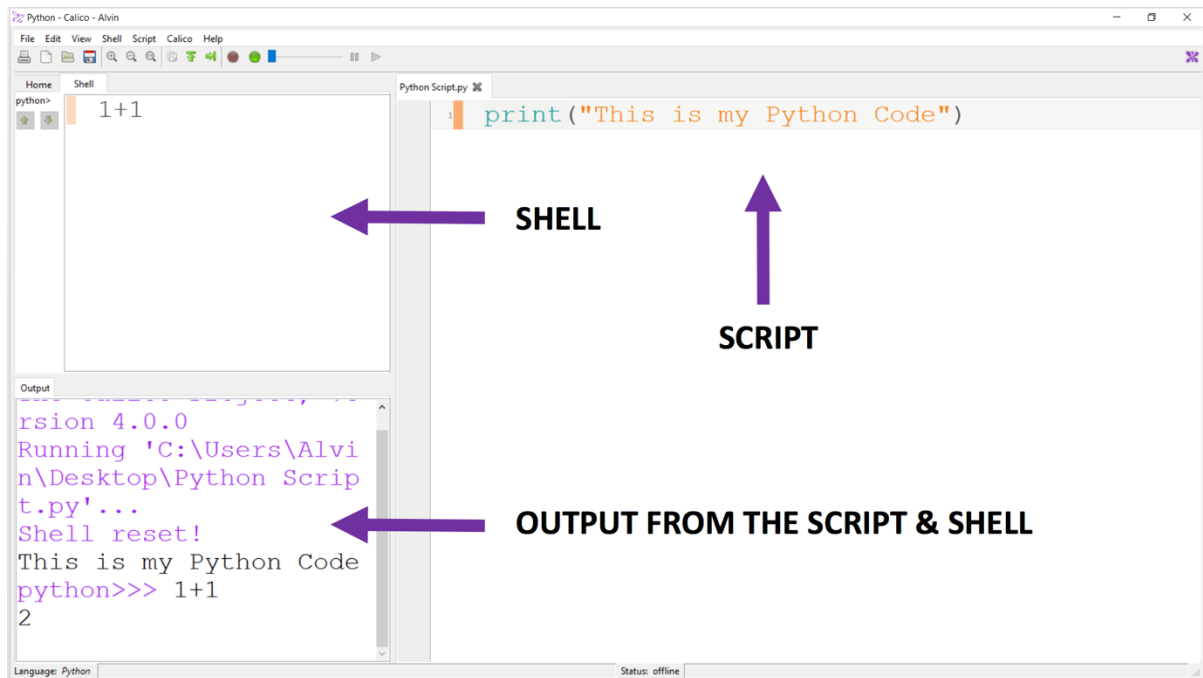
Interface

If you open the calico.bat file in the calico folder, this is what Calico looks like:



It supports many different languages, as you will see later, but we will continue using Python 3.

Environment



Calico has a script panel on the right side where the user writes and saves all their scripts. It has a separate shell where the user can write their Python commands. The output of both the script and the shell are displayed in the output. These work exactly the same way as in IDLE.

A brief detour

Cartesian planes

To use Calico to place objects at certain places on a screen, we first need to know how to use a Cartesian plane to map objects. This is something that you have seen in your maths classes when doing graphs but it's important to make sure you get it.

When a computer puts an object on a screen, it has to know where to put it i.e. it must have a coordinate to know where to put it. The first important thing to know is that the origin of coordinate systems for computers are the top left corner (not the bottom left like what you are used to) and the y-axis is flipped so the y values increase downwards like in the Figure 9 below. We have drawn some pictures to help you get used to the coordinate system.

The tiny circle labels the point (1,1). Points are used to reference specific spots on the screen.

If you want to draw a shape, you should always give the position of the shape from a reference point on the shape i.e. if I want to draw a circle on the screen, I have to say I want to draw a circle with a centre at (4,1). The centre of the circle is the reference point. But I also have to give the radius of the circle/ellipse, the radius in the x-direction is 1 and the radius in the y direction is also 1 for a circle, for an ellipse, the 2 can be different (major and minor axes). So to completely specify an ellipse, I need four parameters/numbers i.e. (x position of centre, y position of centre, x radius, y radius). I have drawn a circle with the parameters (4,1,1,1).

For a rectangle, the default reference point in Calico is the top left corner, so in the diagram I drew a rectangle at position (3,3) i.e. the top left corner is at (3,3). Then I also have to give the length of the rectangle in the x direction (width) and the length in the y direction (height). So the four parameters

are (x position of top left corner, y position of top left corner, width, height). Below I have drawn a rectangle with parameters (3,3,2).

A triangle is a bit annoying because we have to give 3 points with (x,y) for the 3 corners so we have 6 parameters (x1,y1,x2,y2,x3,y3). The triangle below has parameters (1,3,0,5,2,5).

In Calico the we will usually use numbers like 400, 200 etc. for coordinates and lengths, that doesn't change the approach.

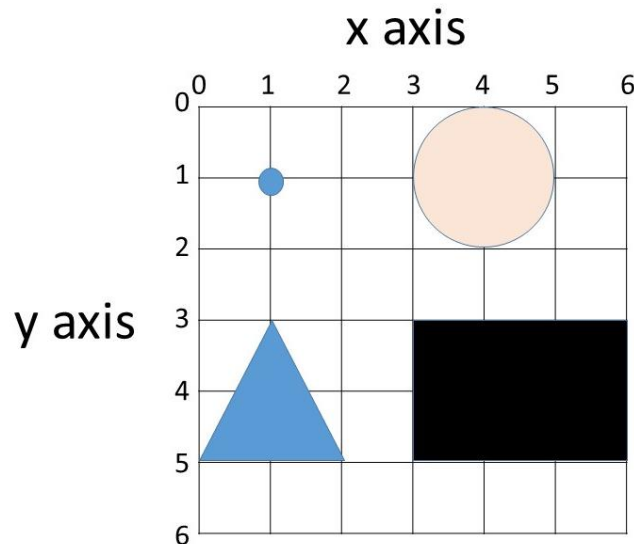


Figure 9

Exercise 10.1

a) Draw a grid like the one above but with your axes having divisions of 100, so on the x-axis you have 100,200,300,400 etc. and sketch the following:

- A point at the position (400,400)
- A rectangle with width 100 and height 400 at position (350,400)
- An ellipse centered at position (0,0) with x radius of 300 and y radius of 200. Would this fit on the screen?

b) If you wanted to draw a hexagon, how many parameters would you need? Which ones?

c) If you wanted to draw a parallelogram, how many parameters would you need? Which ones?

RGB Values

Colours in most computers and programming are represented using numbers rather than names. This is because when you say that something is red, it is not clear what kind of red it is, there are many things that you call red that are very different. We are going to use the RGB convention to represent colors⁵. Every color we can make will be represented by three parameters; (R, G, B) for how much Red, Green and Blue is in the colour. Each number goes from 0 to 255. It turns out that you can make every colour out of just those three "primary" colours. Below are some examples of RGB colour values for common colours.

Colour	(R, G, B)
Black	(0,0,0)

⁵ If you are interested you can look at other ways to represent colour online e.g. CMYK

White	(255,255,255)
Red	(255,0,0)
Green	(0,255,0)
Blue	(0,0,255)
Yellow	(255,255,0)

Back to Calico

Calico Environment Basics

Calico has a module called Processing that allows us to develop digital works of art, data visualizations, interactive applications and animations. You can import all the functions in this model using the following as usual.

```
from Processing import *
```

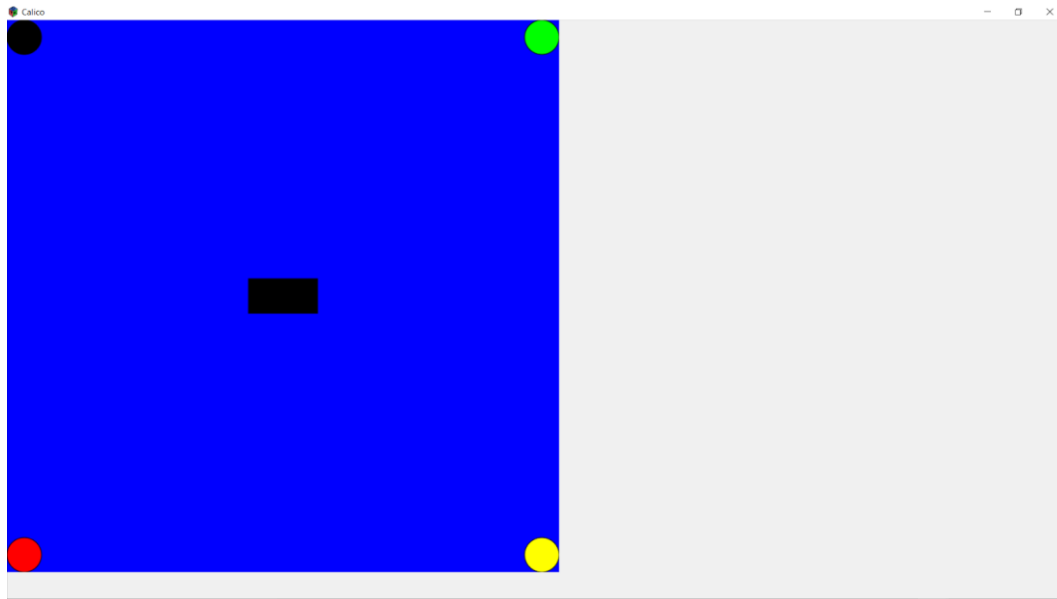
We can read the documentation for the Processing module to find out what functions are defined. Here is the documentation: http://wiki.roboteducation.org/Calico_Processing_Reference. In particular, look up the functions `window`, `background(red,green,blue)`, `fill(red,green,blue)`, `rect`, `ellipse`, `isMousePressed`, `isKeyPressed`, `onKeyPressed`, `rectMode`, `ellipseMode`, `stroke`, `strokeWeight`, `color`, `noStroke()`

Exercise 10.1

- Use the documentation to create a new window with dimensions 800x800 using **`window()`**.
- Run your script using the green circle button after saving the file with `.py` after the name of the file (**This is important, it must be saved as a `.py` file to be run**). Make sure you save and run the file after each step to make sure you're doing the right thing, otherwise you may accumulate errors.
- Change the background colour to blue using **`background()`**.
- Create a black rectangle with dimensions 50x100 right in the middle of the window using **`rect()`**.
- Create a circle of radius 50 at each corner of the window using **`ellipse()`**.
- Colour the circles black, red, green and yellow respectively using **`fill()`** before each line that draws a circle.
- Try to sketch this on a piece of paper first before moving on, to ensure that you understood the specification properly.

Notice that things are drawn in the same order as in your code, if you draw a rectangle and then a circle, the circle will be drawn on top of the rectangle. Also, if you change the background after you have drawn a rectangle, the rectangle will be covered.

The desired solution looks like:



Exercise 10.1 Solution

Our first step should always be to import whatever module we will use. In this case, that module is Processing

```
from Processing import *
```

Next, we want to create a new window with dimensions 800x800 and change its background to blue

```
window(800,800)
background(0,0,255)
```

Next, we want to create a black rectangle at the centre of the window, 50x100. Add:

```
fill(0,0,0)
rectMode("CENTER")
rect(400,400,100,50)
```

Next, we want to create our four circles using the colours specified. Add:

```
ellipseMode("CENTER")
ellipse(25,25,50,50)
fill(255,0,0)
ellipse(25,775,50,50)
fill(0,255,0)
ellipse(775,25,50,50)
fill(255,255,0)
ellipse(775,775,50,50)
```

Note that we have to change the **fill()** before creating a shape because the **shape()** functions will use, by default, the last know attributes of colour, stroke and background.

Exercise 10.2

Draw the flag of Zimbabwe in Calico using the Processing module. Ignore the bird.

Specifications:

- Make your window 1400x700
- The triangle is an isosceles triangle with inner tip at (350,700)
- For simplification, we will use the RGB colour scale.

Your solution should look something like this:

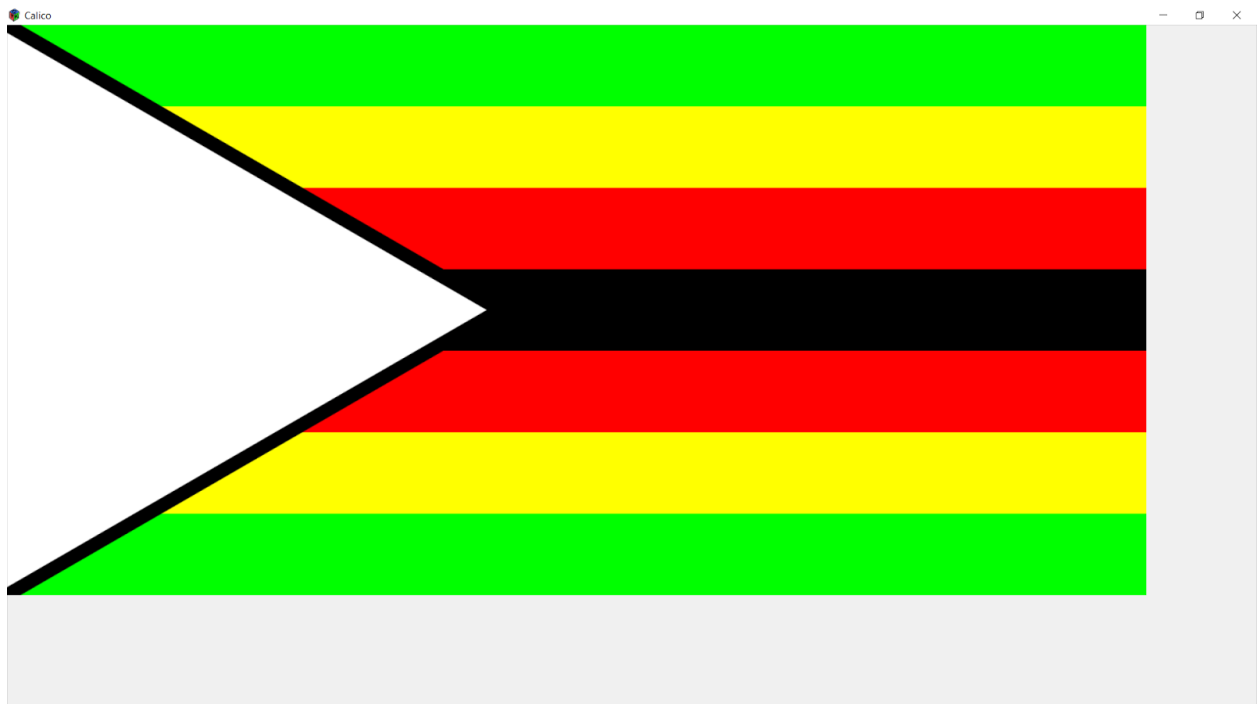


Figure 10

Hint: it is easy to go line by line and draw everything as you see it, but try to think of the most efficient way you can use the Processing module to draw this flag. without repeating the same steps over and over i.e. come up with an algorithm to draw the flag.

Write down/test your solution after each shape that you draw to avoid accumulating errors.

Exercise 10.2: Solution

This is a straightforward solution which uses a clever algorithm that draws big rectangles on top of one another rather than drawing strips one by one.

```
from Processing import *

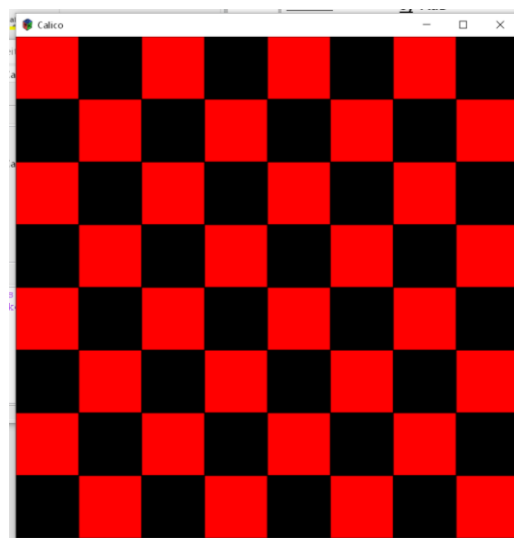
window(1400,700)
background(0,255,0)
fill(255,255,0)
rect(0,100,1400,500)
fill(255,0,0)
rect(0,200,1400,300)
fill(0,0,0)
rect(0,300,1400,100)
fill(255,255,255)
triangle(0,0,700,350,0,700)
```

Example: Chess board

We are now going to consider a nice example where we draw a chess/draught board using calico.

The specifications are as follows:

- The window should be 800x800
- There should be 8x8 squares that fill the whole window
- The squares should have alternating colours of red and black.
- Make the solution as abstract as possible i.e. I should only have to change one variable to change the number of boxes/the size of the window



What algorithms can you come up with to do this? Again as with most real problems, there are many ways you can do it. The bogus way is to draw all the squares one by one but then you have to draw 64 squares and this won't be abstract! Instead we can break our problem down and see what things we are repeating/what patterns we have.

The first step in all our Calico code is to import all the modules we need and define all our variables that follow specifications.

```

from Processing import *
#Problem specs stored in variables
WINDOW_SIZE = 800
NUM_TILES = 8
colour1 = color(255,0,0)          #red
colour2 = color(0,0,0)           #black

window(WINDOW_SIZE,WINDOW_SIZE)  #square window

```

I then run my code and make sure I get a square window. If not, I should debug. The next step is to come up with an algorithm to draw a grid of squares. If I were to draw the first 3 squares, I would do something like this:

```
rect(x,y,width,height)
```

I need four parameters, the width and the height are easier. If I have a window that has a size of WINDOW_SIZE and I want NUM_TILES to be the number of squares then each square has:

```

width = WINDOW_SIZE/NUM_TILES
height = width

```

Every square/tile will have the same size so these variables will not change unless WINDOW_SIZE or NUM_TILES changes but because we used a formula, we never have to worry about adjusting them. What about the parameters **x** and **y**? Let's consider the case where x is fixed. **For each x we will draw rectangles at the following positions so we need to produce the numbers 0,100,200,...,700 for each y (iteration).** We know how to make these numbers! **How?**

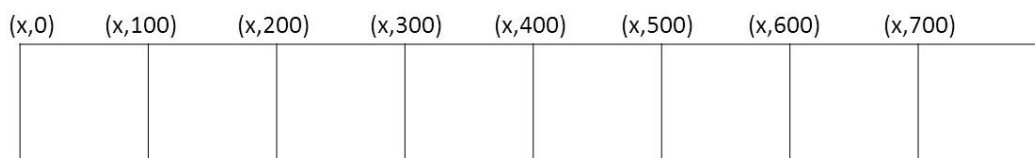


Figure 11

We will use a for loop to draw a row of squares:

```

for y in range(0,800,100):
    rect(x,y,width,height)

```

Now there's only one thing missing, we need x to also take the values 0,100,...,700 so we need another for loop but this will be a nested for loop since for each value of x, y must take all the possible values of y. Notice that you can see from the sentence in bold already that there had to be 2 for loops because there are two underlined "for" words, these are always hints when you think about solving problems. So we have:

```

for y in range(0,800,100):
    for x in range(0,100,800):
        rect(x,y,width,height)  #Don't forget indentation!

```

There is one more thing we need to change, we used numbers not stored in variables, and we used them more than once. That's bad! We can fix it by realizing that we have already defined what these numbers are. So we should have:

```
for y in range(0,WINDOW_SIZE,height):
    for x in range(0,WINDOW_SIZE,width):
        rect(x,y,width,height) #Don't forget indentation!
```

This should run and give you all the boxes. Notice how much shorter it is than using rect() 64 times and doing all the calculations by head! We can even change the number of boxes by changing NUM_TILES, or changing the WINDOW_SIZE. Try it

Now we can put it all together and test:

```
from Processing import *
#Problem specs stored in variables
WINDOW_SIZE = 800
NUM_TILES = 8
width = int(WINDOW_SIZE/NUM_TILES) #What happens if we don't type
                                   #cast?
height = width
colour1 = color(255,0,0)          #red
colour2 = color(0,0,0)            #black

window(WINDOW_SIZE,WINDOW_SIZE) #square window

for y in range(0,WINDOW_SIZE,height):
    for x in range(0,WINDOW_SIZE,width):
        rect(x,y,width,height) #Don't forget indentation!
```

Now we need to colour in the boxes. This requires us to use the function:

```
fill(colour1)
```

which needs only a colour as a parameter but the trick is that we have to change the colour after every square so we have to be switching the variable **colour1** between red and black. We can do this by using if statements (how?) or by remembering how to exchange to variables from exam 1 or using the tuple method:

```
colour1, colour2 = colour2,colour1    #switch the colours
```

This has to be done after every rectangle is drawn so it is inside the inner for loop. We should also change the colour when we change rows so we need to do it again inside the outer for loop. Getting your indentation correct is vital! If you still do not understand indentation, I suggest you revisit loops. The final solution then becomes:

```
from Processing import *
#Problem specs stored in variables
WINDOW_SIZE = 800
NUM_TILES = 8
width = int(WINDOW_SIZE/NUM_TILES)
height = width          #square tiles
colour1 = color(255,0,0) #red
colour2 = color(0,0,0)   #black

window(WINDOW_SIZE,WINDOW_SIZE) #square window

for y in range(0,WINDOW_SIZE,height):
    for x in range(0,WINDOW_SIZE,width):
        fill(colour1)
        rect(x,y,width,height) #Don't forget indentation!
        colour1, colour2 = colour2,colour1 #Switch colour
        colour1, colour2 = colour2,colour1 #Switch colour
```

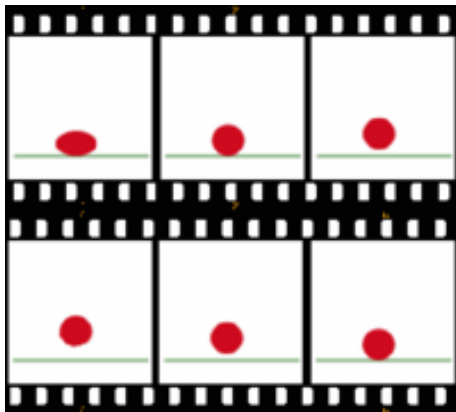
Isn't it cool? Play around with WINDOW_SIZE and NUM_TILES to make different boards.

Animation

Animation is the process of creating the illusion of motion using still images. For example, imagine that this was a video of a ball bouncing:



The motion you see is just a result of combining the following still images and showing them quickly, one after the other, like a slideshow.



If the images are shown to you fast enough, you will begin to perceive motion. This is how all the videos you watch work. When you watch a music video, what you are seeing is not actual motion, by intuitive definition. You are looking at photos of the musician that were taken milliseconds apart and are being shown in a very fast slideshow. The number of times the picture changes per second is called a framerate, measured in fps (frames per second). The more photos of the musician you have, the better the quality of perceived motion (This is what 60fps/30fps means if you're into gaming/phone specs).

Calico has built-in functions that allows us to animate objects. With this ability, we can animate objects and make them move according to a mouse click or a keyboard press. This is how video games work.

The structure of the code for animation looks like below, the things in bold are the things you change:

```

from Processing import *

#Declare global variables/variables that need initialization here

window(width,height) #We make the window outside the draw() function

def draw():          #Should never have parameters
    '''
        This function draws each frame, this is where we can add
        motion by changing the positions of things for each frame
    '''
    #Code that draws each frame

#These three lines must always be included when doing animation in
#Calico, they act like a while loop that calls draw() forever
frameRate(100)
onLoop += draw
loop()

```

For example, if I wanted to draw a rectangle on the screen I would do the following:

```

from Processing import *
window(800,800) #We make the window outside the draw() function

def draw():
    rect(350,0,100,100)

frameRate(100)
onLoop += draw
loop()

```

If I want to animate the box moving down the screen, what will be changing and how do I change it?

```

from Processing import *
window(800,800) #We make the window outside the draw() function

y = 0    #We have to initialize the variable outside the
          #loop/function, just like when we did while loops

def draw():
    background(0,0,0) #See what happens without this first
    global y          #This variable was declared outside scope of draw()
    rect(350,y,100,100)
    y += 1           #This is the velocity of the box in y direction
                    #This line moves the box down by 1px every frame

frameRate(100)
onLoop += draw
loop()

```

If the box is moving down the screen, this means the y value is increasing so we increase y by 1 each time draw is called. This is the speed of the box. Play around with the speed and the initial position of the box and see the difference.

Exercise 10.3: My Old Screensaver

You might (if you're old enough) remember those screensavers where a Windows logo bounces around the screen hitting the edges and bouncing off like a ball. We are going to animate that. The problem specifications are:

- Make a window that is 800x600
- Make a rectangle that is 100x100 that will bounce around the window that starts anywhere inside.
- The rectangle starts by moving 1px up and 1px right and keeps going that way until it hits a wall, if it hits the left/right (+/-) wall, the x speed changes sign, if it hits the right wall, the y speed changes sign
- Make sure the box never goes outside the window.

Extra points

Make the box change colour every time it hits a wall

Exercise 10.3: Solution

```
x = 0
xvel = 1      #We change initial values so we initialize outside
draw()
yvel = 1

def draw():
    background(0,0,0) #See what happens without this first
    global y, x, xvel, yvel #Very important!
    rect(x,y,100,100)
    y += yvel          #velocity in y direction
    x += xvel
    if x < 0 or x > 700: #If the box hits the wall
        xvel = -xvel    #Change direction by changing sign
    if y < 0 or y > 500:
        yvel = -yvel

frameRate(100)
onLoop += draw
loop()
```

Event raising and Handling Events in Calico

So far we have not been able to interact with any of our animations, it would be nice if we could make things happen by clicking/typing. Inputs from the user are also called events e.g. when you click the left mouse button or press the “U” key on the keyboard, that is an event. For the computer to do something when you click/press, it has to “listen” for events (using something called an event listener) i.e. it waits until an event occurs (we say the event is “raised”) and then calls a function called an event handler that does something. For example, when you type in IDLE, an event listener waits until it realizes you pressed the button “p”, when it knows you pressed the button, it raises an event and calls a function that gets the identity of the key pressed and then puts it on the screen for you to see.

From the Calico documentation, look at the following functions:

Useful functions for dealing with events
onKeyPressed() – Returns True if any keyboard button is pressed

key() – Returns a string containing which key button was pressed e.g. 'a' or 'space'
isMousePressed() – Returns True if mouse is pressed
mouseX() – Returns the x position of the mouse
mouseY() – Returns the y position of the mouse

Table 1

These functions are useful for getting information about whether events since they all return something useful. We will use them in the following example.

Let us do an example where we draw a circle when the user presses the mouse. We of course need to include the usual imports and definitions, the new code is in bold.

```
from Processing import *

window(800,800)

def draw():
    if isMousePressed():          #Check if mouse is pressed
        ellipse(mouseX(),mouseY(),100,100)

frameRate(100)
onLoop += draw
loop()
```

We can sometimes use the functions in Table 1 to handle events as we did in the previous example but the best way is to assign event handlers to events. An event handler is a function that is called when it has been assigned to a certain type of event e.g. a key press or a mouseclick. We will do the previous example again but using event handlers this time. Consider the following function (event handler) that sets the global position variables **x** and **y** to the position where the mouse was clicked.

```
def mouseclick():
    '''
    Event handler for mouse clicks that sets the global variables
    x and y to the current mouse position
    returns -> None
    '''
    global x,y
    x = mouseX()
    y = mouseY()
```

We want this function to be called whenever the mouse is clicked so we assign as an eventhandler for mouseclicks using the following code⁶:

```
onMouseClicked += mouseclick          #Without parantheses
```

The above two steps should be done in the global scope which is why **x** and **y** were made global variables. The full working code would be:

⁶ Assignment of event handlers differs depending on what API you use, some APIs would take the event handler as a parameter e.g. **onMouseClicked(mouseclick)**. It will always be in the documentation!

```

from Processing import *

window(800,800)
x = 0
y = 0

#Define mouseclick handler function
def mouseclick():
    '''
    Event handler for mouse clicks that sets the global variables
    x and y to the current mouse position
    returns -> None
    '''
    global x,y
    x = mouseX()
    y = mouseY()

onMouseClicked += mouseclick           #Assign mouse click handler

def draw():
    ellipse(x,y,100,100)  #Notice that we only have to draw the
                        #ellipse inside the draw function now

frameRate(100)
onLoop += draw
loop()

```

The key here is the line in bold. There are many other events we could handle. All we have to do is replace **onMouseClicked**. In Toolbox 10.1 is a list of event listeners that can be used to assign event handlers.

Toolbox 10.1: Event listeners

Event Listeners	Event raised when:
onMousePressed	Mouse is pressed
onMouseClicked	Mouse is clicked once (usually similar to onMousePressed)
onMouseDragged	Mouse is held down and moved
onMouseReleased	Mouse was pressed and is then let go
onKeyPressed	Keyboard button is pressed
onKeyReleased	Keyboard button is released

Exercise 10.4: Can you handle this?

Consider the previous example where we drew circles at the locations where the mouse was clicked. By changing the line in bold, try out all the different event listeners e.g. if you want to use **onMouseDragged** instead, change only that line to:

```
onMouseDragged += mouseclick
```

Test each one to see the difference between dragging, releasing, key presses etc.

Here is an example that draws a rectangle if 'r' is pressed, draws a triangle if 'e' is pressed or erases everything by changing the background by pressing space. This example does not use event

handlers. Notice that the case of the character matters, 'r' is not the same as 'R' and 'space' is always written as 'space' i.e. they are strings.

```
from Processing import *

window(800,800)
rectMode('CENTER') #Draw rectangle from its center position

def draw():
    if isKeyPressed():
        if key() == 'r':
            rect(400,400,100,100)
        elif key() == 'e':
            ellipse(400,400,300,400)
        elif key() == 'space':
            background(255,255,255)

frameRate(100)
onLoop += draw
loop()
```

Notice that we erase everything by calling **background()**, this is useful if you want to refresh the page for an animation. Another thing is that we don't have an else branch because we don't want to do anything if any other button is pressed.

Here is an example where we make a box move around the screen by pressing the arrow buttons using event handlers.

```

from Processing import *

window(800,800)
x = 0          #Give the position of the rectangle
y = 0

def move_box():
    '''
    Event handler for direction key presses
    Returns -> None
    '''
    global x,y          #Remember to make them global!
    if key() == 'Up':
        y -= 10          #Going up means decreasing y
    elif key() == 'Down':
        y += 10          #Going down means increasing y
    elif key() == 'Left':
        x -= 10          #Going left means increasing x
    elif key() == 'Right':
        x += 10

onKeyPressed += move_box

def draw():
    background(255,255,255) #Clear screen, see what happens without
    fill(255,0,0)
    rect(x,y,100,100) #Draw the rectangle everytime the loop runs

frameRate(100)
onLoop += draw
loop()

```

Notice how the string for the direction buttons has upper case first characters (Important!!!)

Exercise 10.5: Boxed in with a good listener

Using guidelines from the examples above, write a program that allows a user to move a box in the calico but the box should never be allowed to leave the window (unlike before). You can test your code and make sure there are no errors from step 2 going onwards. Do not continue if there is an error!

1. Create an 800x800 window for animation (that means includes all the animation code). See below:

```

from Processing import *

window(800,800)

def draw():
    #Code that draws each frame

frameRate(100)
onLoop += draw
loop()

```

2. Declare the variables **x = 0**, **shape = False** and **y = 0** and add them to the global scope.

3. The loop should draw a circle every time when **shape** is True.
4. When the user presses 'c', **shape** should be changed to True (Hint: what event listener do you use?)
5. Make the circle move using the direction buttons on the keyboard
6. Clear the window every time the mouse is pressed. (Hint: what will stop the circle from being drawn all the time?)
7. Make sure the box/circle can never leave the screen (Hint: If x or y become too big or small, make sure the value of x is changed to the maximum value)

Things to be careful about: Indentation (Everything after 3 should be inside the draw() function), the spelling and case of the keys.

8. Try and do the above exercise using event handlers instead. Be careful about declaring global variables where you need them.

Exercise 10.5: Solution 1

```
from Processing import *

window(800,800)
x = 400
y = 400
shape = False

def draw():
    global x,y,shape          #Stop here for no. 2

    if shape:
        background(255,255,255)
        fill(0,255,0)
        ellipse(x,y,100,100)  #Stop here for no. 3

    if isKeyPressed():
        if key() == 'c':
            shape = True      #Stop here for no. 4
        elif key() == 'Up':
            y -= 10
        elif key() == 'Down':
            y += 10
        elif key() == 'Left':
            x -= 10
        elif key() == 'Right':
            x += 10            #Stop here for no. 5

    if isMousePressed():      #Stop here for no. 6
        background(255,255,255)
        shape = False

    if x > 750:
        x = 750
    if x < 50:
        x = 50
    if y < 50:
        y = 50
    if y > 750:
        y = 750              #Stop here for no. 7

frameRate(100)
onLoop += draw
loop()
```

Exercise 10.6: Solution 2 (Event handler method)

```
from Processing import *

window(800,800)
x = 400
y = 400
shape = False

def moveBox():
    global x,y, shape
    shape = True
    if key() == 'c':
        shape = True          #Stop here for no. 4
    elif key() == 'Up':
        y -= 10
    elif key() == 'Down':
        y += 10
    elif key() == 'Left':
        x -= 10
    elif key() == 'Right':
        x += 10              #Stop here for no. 5

    if x > 750:
        x = 750
    if x < 50:
        x = 50
    if y < 50:
        y = 50
    if y > 750:
        y = 750

    #Stop here for no. 7

    return None

def mousePress():
    global shape
    background(255,255,255)
    shape = False

onKeyPressed += moveBox
onMousePressed += mousePress

def draw():
    if shape:
        background(255,255,255)
        fill(0,255,0)
        ellipse(x,y,100,100)  #Stop here for no. 3

frameRate(100)
onLoop += draw
loop()
```


Project Example: Space Invaders

You might be familiar with games like space invaders/space impact etc. where you can have a spaceship or something that can shoot enemies that keep coming from the edge of the screen as you gain points. We are going to create a space invaders game. The aim of the game is to shoot enemies from our ship before they get to us. The game keeps going on until we lose. This will seem like a lot of words but don't panic! There is nothing we have not done before here and lots of hints.

Follow the instructions in order, your script should run without any errors after every number. Make sure you test it after every instruction. We will ask you to do some stuff yourself but other parts we will provide pseudocode for you to follow. **Do not copy the pseudocode! Replace it with code.** No function handlers are used in this project.

Variables are written in **bold**, functions and keywords are written in ***bold italics*** then code and pseudocode are written in Prestige Elite Standard font.

#MayTheCodeBeWithYou

Creating a window

1. Create a new Calico file and call it battlestar.py
2. Create a window that is 400X400 pixels

Making our amazing ship of destruction

Our ship will be a rectangle, because its position will be changing, we will keep its position in variables. By number 7 you must have a rectangle at the bottom of the screen.

3. Declare the following variables

```
shipPositionX = 150
shipPositionY = 350
```

4. Add the following lines at the end to create the loop that will animate

```
frameRate(100)
onLoop += draw
loop()
```

5. Define a function called ***draw()*** with no parameters
6. In this function at the top (before ***global***), make the background white. Add the line:

```
global shipPositionX, shipPositionY
```

7. In the ***draw()*** function, draw a rectangle(the ship) that is 100X30, it's position should be the variable **shipPositionX, shipPositionY**

Making our ship move

To make our ship move from left to right we will listen for key presses. By number 9 your ship should move from left to right

8. Now we want to handle key presses. Inside the **draw()** function after the line the line with “global shipPositionX, shipPositionY” follow this pseudocode and indentation:

```
#If a key is pressed (use isKeyPressed())
    #If key() is “Left”
        #Reduce shipPositionX by 10
    #elif key() is “Right”
        #Increase shipPositionX by 10
```

9. We want to stop the ship from escaping the screen. To do this we will make sure that **shipPositionX** never goes below 0 or above 300. After the code we just put, follow this pseudocode inside draw():

```
#if shipPosition > 300
    #shipPosition = 300
#if shipPosition < 0
    #shipPosition = 0
```

Shooting bullets

10. To make our ship capable of shooting bullets, we need to be able track the position of the bullet as well as track if we have shot a bullet already. Declare the following variables outside the **draw()** function

```
bullet = False      #This will be a Boolean type which is True if we have shot a
                    #bullet
bulletX = 0
bulletY = 0
```

11. Inside the **draw()** function at the top, add the following line:

```
global bullet, bulletX, bulletY
```

12. We want to make the ship shoot bullets when we press space. Add the following **elif** statement after (**#elif key() is “Right”**):

```
if isKeyPressed():
    .... (This is the code we had before)
```

```

elif key() == "space":
    bullet = True
    bulletX = shipPositionX + 50           #This is the
                                           #centre of the ship
    bulletY = shipPositionY

```

To shoot the bullet and make it move add this in the **draw()** function (same indentation as **global**):

```

#if bullet is True:
    #Draw a circle that has radius 20 at bulletX and bulletY
    #subtract 10 from bulletY    #This moves bullet up
    #if bulletY < 10 or when bullet reaches top
        #bullet = False

```

Making the enemy ships

Every hero needs an enemy. Our enemies will be boxes that fly in from the top of the window. By number 15, you should see an enemy box coming from the top.

13. We have to keep track of our enemies positions. Declare the following variables outside the **draw()** function(at the top)

```

enemyX = 0
enemyY = 0

```

14. Inside the **draw()** function add the following line:

```

global enemyX, enemyY

```

15. At the end of the **draw()** function but inside it, follow this pseudocode:

```

#draw a rectangle at enemyX, enemyY that is 100X50
#Add 1 to enemyY           #This moves the enemy ship
                           #by 1

```

Shooting the enemy ships

Our bullets are useless if they can't stop the enemy, let's make our bullets make our enemy disappear if the bullet is in the same position as the enemy. This means that if the bullet is in the enemy box, stop drawing the enemy.

16. Follow this pseudocode inside **draw()** :

```

#If bulletX is between enemyX and enemyX+100 (This means that we want to see if
bulletX is a number between the two, what do we use to make these numbers?)

```

```

#if bulletY is between enemyY and enemyY+50
    #Declare enemyX as a random integer between 0 and 300

```

```
#Declare enemyY as -50  
#Declare bullet as False
```

Let's have some fun

Well done! You have completed the basic game. Now we can play it or make it better! Make your shapes colourful, Add more shapes to your background e.g you can make it look like a road. This just shows how much you have achieved in 12 weeks.

Here are some extra things you can do:

1. Calculate points
2. Make a "You lose" sign
3. Ask the user for input if they want to play again in the shell
4. Play around with the speed of the bullets and the enemies

We can do many interesting things with Calico that can help us visualize what we are doing using a relatively simple API. If you can think of different things you can do with Calico, don't be afraid to try them. I hope you had fun!

Chapter 11: More Examples

Here are some examples that you should be able to do. In brackets, you are told after which chapter you should be able to do the exercise.

Easier Examples

Example 11.1. Multiplication Tables (Chapter 7)

Write a function **multTable(n)**, that writes the multiplication tables from 0 to n. An instance of execution is:

```
>>> multTable(10)
0  0  0  0  0  0  0  0  0  0  0
0  1  2  3  4  5  6  7  8  9 10
0  2  4  6  8 10 12 14 16 18 20
0  3  6  9 12 15 18 21 24 27 30
0  4  8 12 16 20 24 28 32 36 40
0  5 10 15 20 25 30 35 40 45 50
0  6 12 18 24 30 36 42 48 54 60
0  7 14 21 28 35 42 49 56 63 70
0  8 16 24 32 40 48 56 64 72 80
0  9 18 27 36 45 54 63 72 81 90
0 10 20 30 40 50 60 70 80 90 100
```

(Hint: Use a for loop in this exercise)

Example 11.2. Drawing Triangles (Chapter 7)

Write a function **triangleUp(n)**, with one argument n, that prints a triangle with a base and height of n in the following manner:

```
>>>triangleUp(4):
*
**
***
****
```

Next, write a function **triangleDown(n)**, with one argument n, that prints a triangle with a base and height of n in the following manner:

```
>>>triangleDown(5):
*****
****
***
**
*
```

Example 11.3: Grade function (Chapter 5)

Write a function **grade()** that takes in a user's percentage grade and returns a corresponding letter grade. The program should only work for inputs ranging from 0-100 inclusive. It should run continuously. Feel free to determine your own grading scale and include it in your project.

```
>>> What is the percentage?: 100
The student got an A
>>> What is the percentage?:
```

Example 11.4: Testing Palindromes (Chapter 4)

A palindrome is a word that doesn't change if you reverse all the characters that make it e.g. "racecar" in reverse is still "racecar" so it is a palindrome. Write a function **isPalindrome(s)**, that returns **True** if the string **s** is a palindrome, and **False** otherwise. For example:

```
>>>isPalindrome('abba')
True
>>>isPalindrome('neverodddoreven')
True
>>>isPalindrome('isthisapalindrom')
False
```

You will need to ensure that all string inputs are lowercase before operating on them. (Hint: How do you reverse all the items in a data structure?)

Medium Examples

Example 11.5: Making Change (Chapter 4)

Write a function **my_change(n)** that asks the user for an amount **n**, in dollars, and that outputs the breakdown of the amount in terms of banknotes and coins. Use the following money denominations -> 100, 50, 20, 10, 5, 1, 0.25, 0.10, 0.05 and 0.01. For instance:

```
>>>my_change(23.17)
1 twenties
0 tens
0 fives
3 ones
0 quarters
1 dimes
1 nickels
2 pennies
```

Example 11.6: Power without Power (Chapter 8)

Write a function **power(a,b)**, where **b** is a strictly positive integer, and that prints the result of computing **a** to the power **b**. Of course, the use of the power operator ****** is forbidden. You must use a loop/recursion instead. An example of this function would be:

```
>>> power(3,4)
3 to the power 4 is 81.0
```

Consider also the case when **b** is 0: in that case, **a** to the 0 needs to be 1, for any **a**, by definition.

Example 11.7: Binary to Decimal (Chapter 4)

We mentioned before that computers think in 1s and 0s i.e. binary. Binary is just another way to represent numbers. For example, the code 0 represents the number 0 and 1 represents 1 then 10 represents 2 etc.

The string '110' converts into binary in the following manner:

$$(0 * 2^0) + (1 * 2^1) + (1 * 2^2)$$

where 2^i denotes the i -th power of 2 (in Python, $2**i$). Notice that the string 110 can be seen in reverse order in the above expression (and is highlighted in red). Unwinding it means that '110' converts into:

$$(0 * 1) + (1 * 2) + (1 * 4) = 6$$

You can use a loop to write a short and straightforward solution for this problem. Also, beware of multiplying strings with numbers.

Write a function **decimal(s)**, that takes as input a string of 0s and 1s (binary number) and converts it into a corresponding decimal value. s is assumed to be a nonempty string. For instance:

```
>>>decimal('10')
2
>>>decimal('110')
6
>>>decimal('101010')
42
>>>decimal('10100111001')
1337
```

Example 11.8: Desired Daisy: Turtle (Chapter 6)

We would like to have “daisy” drawings like the following:

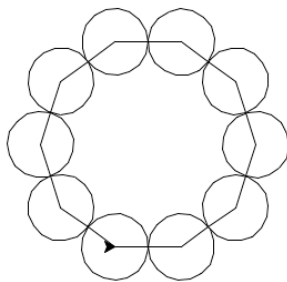


Figure 12

Use the turtle module to draw the pattern above. write a function **daisy()** to draw the pattern. Your function should ask the user for the number and size of the petals. Such a drawing has a very regular geometric pattern, and, once the radius of the petal and their number n are known, it goes as such:

You need to repeat the following n times:

- move forward distance **size**
- draw a circle of radius **size**

~Take care of your turtle being in the correct direction, before drawing the circle! You will have to turn it of 90° degrees. ~

- move forward a distance of **size**

~Put your turtle back in the correct direction after the previous step. ~

- turn left, with angle $360.0/n$. (The exterior angle of a regular polygon)

~Notice the use of the floating-point number 360.0, to avoid integer division and increase precision.

Remember that, at the very beginning of your file, you must include the following line: **import turtle**

Difficult Examples

Example 11.9: Compute your Age (Chapter 4)

Write a program **age()** that computes your age. It must first ask you the year, then the month and lastly the day you were born (all in numbers: 1-12 for the month and 1-31 for the day). It should then tell you how old you are in terms of years, months and days. If today (the day you check your age) is your birthday, the program should also wish you a happy birthday. You should use the built-in Python module **time** to extract the current date from the computer and use it in your calculations. You can google what functions it contains or use the help function.

Example 11.10: Guess my Number (Chapter 6)

Write a function **guess(max)** that works in the following way:

- The computer picks a **random number** between 0 and max (exclusive)
- The user has to try and guess the number that the computer chose.
- Every time a user enters their guess, one of the following will happen:-
 - Their guess is equal to the number the computer chose -> The user wins the game and it ends.
 - Their guess is less than the number the computer chose -> The computer tells the user that their number is less and lets the user guess again.
 - Their guess is more than the number the computer chose -> The computer tells the user that their number is more and lets the user guess again.

Technically, say you ran **guess(50)** and the computer chose 27, you could guess from 1 all the way up to 27 until you eventually win, which makes the game rather useless. To make the game challenging, we will **limit the number of guesses** that a user can make to be the least number of times required for you to guess a number between 0 and max (exclusive). In order for you to figure out the least number of attempts require for an input of max, you will need another function called **numberOfAttempts(max)** which works this way:

numberOfAttempts(0) is 1

numberOfAttempts(2) is 2

numberOfAttempts(7) is 4

numberOfAttempts(14) is 4

numberOfAttempts(30) is 5

numberOfAttempts(31) is 6

You first have to figure out the pattern before writing the function. If we run **guess(2017)**, the least number of tries required to guess the number chosen by the computer is **numberOfAttempts(2017)**. If the user uses up more than **numberOfAttempts(2017)** without winning the game, they lose!

Example 11.11: Iterative Dichotomy Search/Bisection Search (Chapter 9)

Dichotomy is a very efficient way to search for information in a list that is already sorted. Say that we have list $l = [1, 2, 3, 4, 5, 6, 7, 8, 9]$ and we are searching it for 3.

You can divide the list up in the following manner:

1	2	3	4	5	6	7	8	9
low				middle				high

Since the list is already sorted, we go to **middle** and see if it is equal to, greater than or less than 3. In this case, we know that it lies to the left of **middle** because the list is sorted, therefore we do the following:

1	2	3	4
low	middle		high

We repeat the process and 3 is to the right of **middle** and so we do the following:

3	4
low, middle	high

Now, **middle** is equal to three so we have found it.

Based on this explanation, write a function **search_dichotomy(l,e)** that takes as arguments, a sorted list of integers **l** and an integer **e** and searches for **e** inside the list **l**. If **e** is in **l**, the function should return the index of **e** in the list **l**. If **e** is not in **l**, it should return **False**. Assume that the list is sorted in ascending order and that the list is **n**.

Side note: This is actually a very important concept in computer science, the bisection search is the fastest guaranteed algorithm to search for an element inside a sorted data structure. As a professional, programmer, you might be dealing with a data structure that has billions of elements and then it becomes important how fast you can search for an element in that list because you want your code to run faster. We say the bisections search has the lowest complexity because it uses the fewest numbers of steps. This applies to other types of algorithms like sorting algorithms and copying algorithms.

Example 11.12: Space Invaders Advanced (Chapter 10)

Using Calico, make a 2D version of Space Invaders Advanced.

Specifications:

- Use a black 400x400 window.
- Your ship is a red rectangle measuring 100x30. It should be fixed at 50 above the bottom of the window.
- The enemy is a blue rectangle measuring 100x50. It should appear from random positions on the top of the window. They should disappear when shot by a bullet.
- The bullets are white and have a diameter of 20.
- Left and Right arrows move you left and right.
- Shoot is the space bar.
- If you lose, i.e. an enemy reaches the y coordinate of your ship, a message should display telling you that you have lost and the game restarts.
- You are free to build your solution off what we already did in class.

Your final solution should look like this:



Example 11.13: Going in circles

Use Calico to write a program that has a grid of $n \times n$ (n being a parameter set by you) circles arranged like a chessboard i.e., replace the chessboard squares with circles that just touch each other. Assign event handlers that change the colour of a circle to a random colour whenever it is clicked. Rather than saving the colours in variables and choosing one out them, make the inputs of the **color()** function random. (Hint: you don't have to redraw all the circles in every frame, you can just draw a circle on top of the circle that was clicked in a new colour)