Learn to code with Python and Raylib

release

Richard Smith

Sep 30, 2021

CONTENTS

Preface

For the instructor

This book contains all the example programs used in my CoderDojo class to teach Python programming. The primary goal of the class is to teach programming using action games to make learning more interesting. Some of the examples are entirely focused on introducing new language concepts or showing how the Pygame Zero API works, but most are a mixture of both.

The intention is that each program in the example chapters is *brief, complete* and *introduces only one or two new concepts*.

- The programs are short so that students can feasibly type them in during the class. Typing for themselves is very valuable for beginners, because it helps them learn to type, and to type precisely without mistakes (which show up as syntax errors). Even if it takes them a while, they should feel proud when they type a program correctly in the end![1]
- Each program is complete to avoid the confusion caused by worksheets that tell a student to add lines to a program piecemeal. If they make an error or omit a step they may never recover and not produce a working program. Also being *complete* and *separate* means the whole class can be told to skip to the same program and it won't matter so much that some of them didn't complete all the prior programs.
- Introducing ideas one or two at a time allows the students to learn mostly through *doing* and observing the output of each program rather than memorizing. To this end the *order* of the programs within each chapter has been carefully selected. The chapter ordering is more loose and it is possible to skip back and forth between chapters. If the text quizzes or the Python fundamentals are boring the students, skip ahead to the graphical games, then come back to the earlier examples when necessary.

Following each program are some ideas for how students can modify the program. Hopefully they will go further and modify these programs into their own unique games! The difficulty of these suggestions varies, to accommodate students of different age and ability.

This book may be suitable for self-teaching by a motivated learner, but does not attempt to be comprehensive or give detailed explanations because it is intended to be used in a class with an instructor who will fill in the gaps as needed by the students.

[1] For writing essays nowadays people can use voice input and touch-screen input, but for programming the majority of programmers still type on a keyboard. There's no sign of this changing in the immediate future, so practising keyboard skills now is still very important for the future. Students can practise at home with a program such as TuxType. That being said, many programs contain similar code, so it's also a useful skill for them be able to copy and paste from their other programs.

The second edition

I discovered two important things from feedback from the first edition:

1. It's not possible to learn coding with only a single lesson each week. Therefore if that is all we have, the time is better spent inspiring interest in the subject rather than memorizing syntax or wrapping their heads around abstract concepts. Those who are motivated to learn more in their own time will require more syntax and concepts, of course, and hence the Fundamentals chapter has been greatly expanded for their use.
2. Many students wanted to create larger games than the simple examples given in the first edition. It was always the intention that they absorb the knowledge from the examples and then go on to greater larger games of their own but some will require more hand-holding.

Therefore there are four new chapters of tutorial style examples. These are intended to be done in class, with the initial code file given to the students to save time. They are intended to create a sense of accomplishment as each subsequent modifcation improves the game, and also to allow lots of scope for student creativity and customisation.

As noted above, tutorials can become confusing, but I hope the class instructors can resolve any issues.

CHAPTER

ONE

OVERVIEW OF PYTHON

1.1 Comments

A computer program is intended to be understood by both humans and computers. However to make it easier for the humans, it can also contain comments written in English.

```
# A comment looks like this
```

Python ignores comments. They provide explanations for the human readers.

1.2 Literals

A Python program can contain any number and any *string* of text surrounded by quotes.

Examples:

5	1.23	“Hello”	‘Barry’

1.3 Keywords

Every computer language has a number of *keywords* that you will need to learn along with their meanings. Fortunately they look like English words and there are only a few of them in Python. You could tick them off as you meet them.

False	None	True	and	as	assert	async	await
break	class	continue	def	del	elif	else	except
finally	for	from	global	if	import	in	is
lambda	nonlocal	not	or	pass	raise	return	try
while	with	yield					

1.4 Built-ins

Python also comes with a large number of *functions*. The most common ones are built-in and always available, much like the keywords. Here is a list of them, just for the sake of completeness, *but you probably won't ever use them all*, and when you do use one you will probably look it up in the documentation. So you *don't need to remember these*.

abs	all	any	ascii	bin	bool
breakpoint	bytearray	bytes	callable	chr	classmethod
compile	complex	copyright	credits	delattr	dict
dir	divmod	enumerate	eval	exec	exit
filter	float	format	frozenset	getattr	globals
hasattr	hash	help	hex	id	input
int	isinstance	issubclass	iter	len	license
list	locals	map	max	memoryview	min
next	object	oct	open	ord	pow
print	property	quit	range	repr	reversed
round	set	setattr	slice	sorted	staticmethod
str	sum	super	tuple	type	vars
zip					

Once you understand all of these you effectively understand all of the Python language. By the end of this book you will be familiar with at least 20 keywords / functions which is enough to create a huge variety of programs.

1.5 Libraries

There are many more functions available (too many to list here), but not everyone will need them, so they are kept in libraries. Some libraries are supplied with Python. You can use their functions only after first *importing* the relevant library module. For example, if you want a random number, import the random library:

```
from random import randint
print(randint(0,10))
```

Other libraries are not supplied with Python and must be downloaded separately, such as the Minecraft, Pygame and Richlib libraries.

1.6 Names

You will see many words in a program that appear to be English words and yet they are not literals, keywords or library functions. These are names chosen by the programmer. For example, if the program needs to record a score and store it in a variable, the programmer might choose to give that variable the name `score`:

```
score = 1
print("Score: ", score)
```

Python has no understanding of what `score` means. It only cares that the same word is used every time. So a different programmer might decide to write the program like this:

```
points = 1
print("Score: ", points)
```

A programmer who doesn't like typing might use a shorter, less descriptive name:

```
p = 1
print("Score: ", p)
```

However the programmer must be consistent. This **would not work**:

```
points = 1
print("Score: ", score)
```

1.7 Whitespace

Python is unusual in that it cares about *whitespace*, i.e. what you get when you press the *tab* key or the *space* bar on the keyboard.

Python programs are arranged in blocks of lines. Every line in a block must have the same amount of whitespace preceding it - the *indentation*. See Program 2.19 for an example.

CHAPTER

TWO

PYTHON FUNDAMENTALS

There are some exercises here. Each exercise will ask you to write a program. The solution is often on the following page - do not turn the page until you have attempted your own solution! Save each program in a separate file.

2.1 The REPL

REPL stands for *Read Evaluate Print Loop*. In Mu you access it via the `REPL` button. It appears at the bottom of the window. It's a special mode in which you type an instruction to Python and Python executes it immediately (no need to click `RUN`) and displays the result (no need to type `print()`). It's useful for doing calculations and trying things out, but it won't save what you type, so you will only want to use it for very short programs.

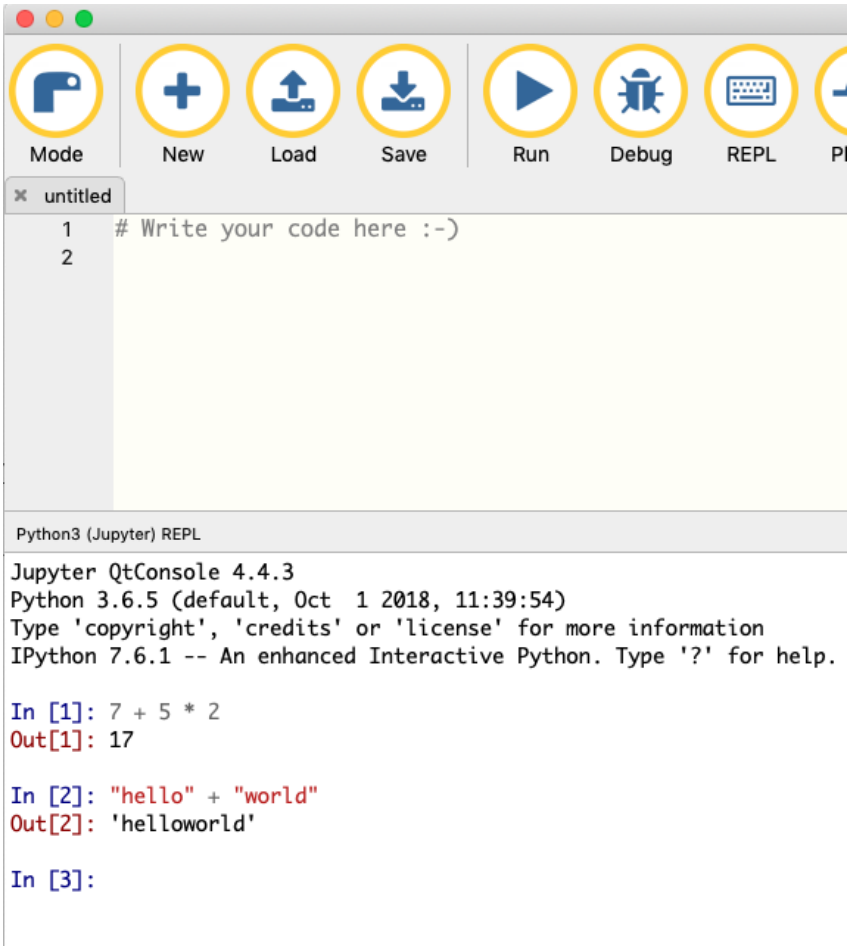


Fig. 2.1: The REPL

2.2 Arithmetic operators

Python understands several operators from maths. You can use them in your programs, or just enter these examples at the REPL to use Python as a calculator, as in the screenshot above.

Operator	Symbol	Example	Result
Addition	+	20 + 10	30
Subtraction	-	20 - 10	10
Multiplication	*	20 * 10	200
Division	/	20 / 10	2

There are some more advanced operators in Program 2.18.

2.3 Variables

A *variable* is a place in the computer's memory where data is stored. You can name a variable whatever you like; you should try to make the name descriptive. There are many *types* of variable but Python sets the type for us automatically when we store data in the variable. (Unlike in many other languages, we do not need to specify the type.) The types we will see most often are whole numbers (*integers*) and *strings* of text.

We create a variable and assign a value to it using the = operator. Note this is different from the == operator which is used for comparisons.

We use the `print()` function to print the value of our variables. It will print any type of data (numbers, strings, both literals and variables) provided each item is separated with a comma (`,`).

Program 2.1: Variable assignment

```
my_number = 7
my_string = "hello"
print(my_string, my_number)
```

We can use a variable anywhere we would use a literal number or string. The value of the variable will be retrieved from the computer's memory and substituted for the variable in any expression.

Program 2.2: Adding two variables together

```
apples = 27
pears = 33
fruits = apples + pears
print("Number of fruits:", fruits)
```

Exercise

Copy Program 2.2, but also add 17 bananas to the calculation of fruits.

We can store a new value in the same variable. The old value will be forgotten.

Program 2.3: Overwriting a variable with a new value

```
apples = 27
apples = 40
print("Number of apples:", apples)
```

Question

What do you think Program 2.3 will print? If you aren't sure, type it in.

More usefully, we can take the old value, modify it, then store it back in the same

variable.

Program 2.4: Modifying a variable

```
x = 5
x = x * 10
x = x + 7
print(x)
```

Exercise

What will Program 2.4 print? Change the numbers in the program. Use a division / operation. Then ask your friend to predict what the new program will print. Was he right?

You will often see this used for counting:

Program 2.5: Counting

```
total = 0
total = total + 1
total = total + 1
total = total + 1
print(x)
```

Question

What is the total count of Program 2.5 ?

See Program 2.18 for a quicker way of writing this.

2.4 Input

Program 2.2 is not very useful if the number of apples changes. This would require the *programmer* to change the program. We can improve it by allowing the *user* of the program to change the numbers. The `input()` function allows the user to type a string which can be different every time the program is run.

```
my_string = input()
print(my_string)
```

Sometimes we want the user to type in a number rather than a string. We can combine the `int()` function with the `input()` function to convert the string to a number.

Program 2.6: Getting input from user

```
print("Enter a number")
my_number = int(input())
print("Double your number is", my_number * 2)
```

Exercise

Copy Program 2.2 but use *input()* to ask the user to enter the number of apples and pears.

2.5 Booleans

A *boolean* is another type of variable that is not a string or a number. It can have only two possible values: `True` or `False`. In some languages and in electronics you may see these represented as `0` and `1`.

Booleans are used by keywords such as `if` and `while`. In an `if` statement, the indented code block is only run if the boolean is `True`.

```
sunny = True
if a:
    print("Let's go to the park")
```

You could write it like this:

```
sunny = True
if sunny==True:
    print("Let's go to the park")
```

but that would be redundant because `if` always tests if the boolean is `True`.

If the boolean is not true, and if you write an `else` clause, the indented code block under `else` is run instead.

```
sunny = False
if sunny:
    print("Let's go to the park")
else:
    print("We must stay at home")
```

2.6 Comparison operators

Comparison operators take two numbers, strings or other variables, compare them, and then return a *boolean* `True` or `False` from them.

Operator	Symbol
Equal	==
Not equal	!=
Less than	<
Less than or equal	<=
Greater than	>
Greater than or equal	>=

Program 2.7: Comparisons: greater than, lesser than, equal to

```
if 7 < 9:
    print("7 is less than 9")

a = 10
b = 5

if a == b:
    print("a is equal to b")

if a < b:
    print("a is less than b")

if a > b:
    print("a is greater than b")
```

2.7 Boolean logic

The `and`, `or` and `not` operators operate on booleans and return new boolean values.

Program 2.8: Boolean operators

```
a = True
b = False

if a:
    print("a is true")

if a and b:
    print("a and b are both true")

if a or b:
    print("either a or b is true")
```

Change the values of *a* and *b* in Program 2.8 and see what output is printed by different combinations of *True* and *False*.

2.7.1 Or

Only people older than 12 or taller than 150cm are allowed to ride the rollercoaster. This program checks whether people are allowed to ride.

```
print("How old are you?")
age = int(input())
print("How tall are you?")
height = int(input())
if age > 12:
    print("You can ride")
elif height > 150:
    print("You can ride")
else:
    print("YOU MAY NOT RIDE, GO AWAY!")
```

Boolean operators combine two truth values together. The `or` operator is `True` if either of its operands is true. Try this example:

```
a = True
b = False
print(a or b)
```

Exercise

Use the *or* operator to make the rollercoaster program shorter by combining the two tests into one test.

A possible solution:

```
print("How old are you?")
age = int(input())
print("How tall are you?")
height = int(input())
if age > 12 or height > 150:
    print("You can ride")
else:
    print("YOU MAY NOT RIDE, GO AWAY!")
```

2.7.2 And

The `and` operator is `True` if both of its operands is true. Try this example:

```
a = True
b = False
print(a and b)
```

Exercise

The rollercoaster is only allowed to run on days when the temperature is less than 30 degrees. Extend the program to ask the temperature and use the *and* operator to only allow riding when less than 30 degrees.

A possible solution:

```
print("How old are you?")
age = int(input())
print("How tall are you?")
height = int(input())
print("What is the temperature?")
temp = int(input())
if (age > 12 or height > 150) and temp < 30:
    print("You can ride")
else:
    print("YOU MAY NOT RIDE, GO AWAY!")
```

Note that we have put brackets around the `or` expression. This ensures it is calculated first and the result of that calculation is then used in the `and` expression. This is the same way you use the BODMAS rule to decide the order of operations in maths.

2.7.3 Not

The `not` operator is `True` if its operand is `False`. If its operand is `False` then it is `True`. Try this example:

```
a = True
b = False
print(not a)
print(not b)
```

We can get a user input and convert it to a boolean like this:

```
print("Is it raining? Y/N")
if input() == "Y":
    raining = True
else:
    raining = False
```

Exercise

Change the program so that you can only ride the rollercoaster if it is not raining.

Possible solution:

```
print("Is it raining? Y/N")
if input() == "Y":
    raining = True
else:
    raining = False
print("How old are you?")
age = int(input())
print("How tall are you?")
height = int(input())
print("What is the temperature?")
temp = int(input())
if (age > 12 or height > 150) and temp < 30 and not raining:
    print("You can ride")
else:
    print("YOU MAY NOT RIDE, GO AWAY!")
```

2.8 For loops

A `for` loop repeats a block of code a number of times. A variable is created which we can use to find the current number within the loop. Here the variable is called `x` but you can name it whatever you like. Run this program:

```
for x in range(0, 11):
    print(x)
```

You can also change the *step* of the loop. Run this program:

```
for x in range(0, 11, 2):
    print(x)
```

2.8.1 Nested loops

It is often useful to put one loop inside another loop.

Program 2.9: Nested for loop

```
1 for a in range(0, 6):
2     for b in range(0, 6):
3         print(a, "times", b, "is", a * b)
```

Exercise

Write a program which prints out the 12 times table.

2.8.2 Incrementing a variable in a loop

A baker has three customers. He asks them each how many cakes they want so he knows how many he must bake. He writes this program.

```
total = 0
print("Customer", 1, "how many cakes do you want?")
cakes = int(input())
total = total + cakes
print("Customer", 2, "how many cakes do you want?")
cakes = int(input())
total = total + cakes
print("Customer", 3, "how many cakes do you want?")
cakes = int(input())
total = total + cakes
print("I will bake", total, "cakes!")
```

Exercise

This program is longer than it needs to be. Write your own program that does the same thing using a *for* loop. It should be only 6 (or fewer) lines long.

Program 2.10: Possible solution to baker program exercise

```
1 total=0
2 for x in range(1, 4):
3     print("Customer", x, "how many cakes do you want?")
4     cakes = int(input())
5     total = total + cakes
6 print("I will bake", total, "cakes!")
```

Exercise

The baker gets a fourth customer. Change Program 2.10 so it works for 4 customers.

Exercise

The baker has a different number of customers every day. Change the program so it asks how many customers there are. Store the number typed by the user in a variable

called *c*. Change the loop so it works for *c* customers rather than 4 customers.

Program 2.11: Possible solution to variable number of customers exercise

```
print("How many customers are there today?")
c = int(input())
total=0
for x in range(1, c+1):
    print("Customer", x, "how many cakes do you want?")
    cakes = int(input())
    total = total + cakes
print("I will bake", total, "cakes!")
```

Exercise

If a customer orders 12 cakes, he gets an extra cake for free. Use an *if* statement to check *cakes* > *12*. If so, add one more cake.

2.9 Array lists

Variables can be stored together in a *list*. Most languages call this an *array* so try to remember that word also.[1]

[1] There are other kinds of list that are not arrays but this need not concern the beginner.

Program 2.12: Array lists

```
# a is a list of integers

a = [74, 53, 21]

# b is a list of strings

b = ["hello", "goodbye"]

# You can take a single element from the list.
print(a[2])

# You can use a for loop to print every element.
for x in a:
    print(x)
```

2.9.1 Looping over lists

Rather than the user typing in data, your program might be supplied with data in a list. Here is a list of prices - a shopping list. Note we don't use a currency symbol except when we print the price.

```
prices = [3.49, 9.99, 2.50, 20.00]
for x in range(0, 4):
    print("item costs £", prices[x])
```

In this program `x` is used an *index* for the array. Note that indices begin at `0` rather than `1`. If the array contains 4 elements then the final element will have index `3`, nor `4`.

However, `for` can directly give you all the array values without the need for an index or to specify the size of the range:

Program 2.13: A shopping list

```
prices = [3.49, 9.99, 2.50, 20.00]
for price in prices:
    print("item costs £", price)
```

Exercise

Change the Program 2.13 so that it prints the total price of all the items added together.

Program 2.14: Possible way of calculating the total cost of shopping list

```
prices = [3.49, 9.99, 2.50, 20.00]
total = 0
for price in prices:
    print("item costs £", price)
    total = total + price
print("shopping total", total)
```

There is a problem with solution, can you see what it is when you run it?

The problem is that we are using *floating point* numbers for the prices and floating point maths in the computer is not entirely accurate, so the answer will be very slightly wrong. One way to fix this is to round the result to two decimal places using the `round()` function:

```
print("shopping total", round(total,2))
```

This works for a short list, but if the list was millions of items long it might not give the right result. Can you think of a better way?

Instead of storing the number of pounds, store the the number of pennies. Britain no longer has a half-penny, so the numbers will always be whole numbers - *integers* - and no floating points will be needed for the addition.

Program 2.15: Better way of calculating the total cost of shopping list

```
prices = [349, 999, 250, 2000]
total = 0
for price in prices:
    print("item costs £", price/100)
    total = total + price
print("shopping total", total/100)
```

Exercise

Conditional discount. Any item that costs more than £10 will be discounted by 20 percent. Use an *if* statement to check if the price is more than 1000 pennies. If it is, multiply the price by 0.8 to reduce it before you add it to the total.

Program 2.16: Possible way of discounting shopping list

```
prices = [349, 999, 250, 2000]
total = 0
for price in prices:
    print("item costs £", price/100)
    if price > 1000:
        price = price * 0.8
        print("  item discounted to", price/100)
```

(continues on next page)

(continued from previous page)

```
    total = total + price
print("shopping total", total/100)
```

2.10 Functions

You may have seen specially named functions that are called by Pygame: `draw()` and `update()`. However, you can define a function named whatever you like and call it yourself.

Functions are useful for many reasons. The simplest is that they make your program look more organized. They also enable you re-use code without needing to copy it and risk making mistakes. When your programs get longer they enable you to create *abstractions* so you only have to think about what function you want to call and don't need to remember the details of the code inside the function.

Program 2.17: Functions

```


def my_func():
    print("This is my function")
    print("Imagine there was lots of code here"
          " that you didnt want to type 3 times")


my_func()
my_func()
my_func()
```

2.11 Shortcuts

Here are quicker ways of doing basic things. You may have noticed some of these being used already.

Program 2.18: Shortcuts

```
# f is an easy way to insert variables into strings
score = 56
name = "Richard"
message = f"{name} scored {score} points"
print(message)

# += is an easy way to increase the value of a variable
score = score + 10  # hard way
score += 10         # easy way
```

(continues on next page)

(continued from previous page)

```
print(score)

# double / means whole number division, no decimals
x = 76 // 10
# MODULO is the percent sign %. It means do division and take the remainder.
remainder = 76 % 10
print(f"76 divided by 10 is {x} and the remainder is {remainder}")

WIDTH = 500
a = 502
b = 502
# Modulo is often used as a shortcut to reset a number back
# to zero if it gets too big.  So instead of:
if a > WIDTH:
    a = a - WIDTH
# You could simply do:
b = b % WIDTH
print(a, b)

# input() takes a string argument which it prints out.
# Instead of:
print("Enter a number")
num = input()
# You can have a single line:
num = input("Enter a number")
```

2.12 Indentation

Code is arranged in *blocks*. For example, a *function* consists of a one line declaration followed by a block of several lines of code. Similarly, all the lines of a loop form one block. A *conditional* has a block of code following the `if` statement (and optionally blocks after the `elif` and `else`.)

Many languages use `{}` or `()` to delimit a block. However Python is unusual: each block begins with `:` and then all the lines of the block are *indented* by the same amount of whitespace (tabs or spaces). The block ends when the indentation ends.

Blocks can be *nested* inside other blocks.

Program 2.19: Can you predict what this program will print?

```
def test():
    print("entering function block")
    for i in range(0,10):
        print("entering for loop block")
        if i == 5:
            print("in if block")
        print("leaving for loop block")
```

(continues on next page)

(continued from previous page)

```
8       print("leaving function block")
9   print("not in any block")
10  test()
```

2.13 Global variables

A variable defined inside a function has *local scope*: it cannot be used outside of the function. If you want to use the same variable in different functions then you must define it outside the functions, in the *global scope*. However, if you attempt to modify the value of the global variable inside a function you will get an error, or - even worse - you will create a local variable with the same name as the global variable and your changes to the global variable will be silently lost.

You must explicitly tell Python that you want to use a global variable with the `global` keyword.

Program 2.20: Try removing line 3 and see what happens

```
1  a = 10
2  def my_function():
3      global a
4      a=20
5  my_function()
6  print(a)
```

2.14 Dictionaries

A dictionary (called a *HashMap* in some languages) stores pairs of values. You can use the first value to look-up the second, just like how you look-up a word in a dictionary to find its meaning. Here is a dictionary of the ages of my friends:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
print("What is your name?")
name = input()
age = friends[name]
print("Your age is", age)
```

Exercise

Change the program so it contains 5 of your friends' ages.

2.14.1 Counting

Here is a loop that prints out all the ages:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
for name, age in friends.items():
    print(name, "is age", age)
```

Exercise

Can you add an *if* statement to only print the ages of friends older than 10?

Possible solution:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
for name, age in friends.items():
    if age > 10:
        print(name, "is age", age)
```

Exercise

Now add a *count* variable that counts how many of the friends are older than 10. Print the number at the end.

Possible solution:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
count = 0
for name, age in friends.items():
    if age > 10:
        count = count + 1
print("friends older than 10:",count)
```

2.14.2 Combining tests

Exercise

Use the *and* operator together with the < and > operators to only count friends between the ages of 11 to 13.

Possible solution:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
count = 0
```

(continues on next page)

(continued from previous page)

```
for name, age in friends.items():
    if age > 10 and age < 14:
        count = count + 1
print("friends age 11 to 13 :",count)
```

2.14.3 Finding

We make a variable `oldest` that will contain the oldest age in the list.

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
oldest = 0
for name, age in friends.items():
    if age > oldest:
        oldest = age
print("oldest age", oldest)
```

Exercise

Make a variable *youngest* that will contain the youngest age in the list. Print the youngest at the end.

Possible solution:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
oldest = 0
youngest = 100
for name, age in friends.items():
    if age > oldest:
        oldest = age
    if age < youngest:
        youngest = age
print("oldest age", oldest)
print("youngest age", youngest)
```

2.14.4 Finding names

Exercise

As well as the ages, print the names of the youngest and oldest friends.

Possible solution:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
oldest = 0
youngest = 100
for name, age in friends.items():
    if age > oldest:
        oldest = age
        oldname = name
    if age < youngest:
        youngest = age
        youngname = name
print("oldest friend", oldname)
print("youngest friend", youngname)
```

2.14.5 Find the average

Exercise

Create a *total* variable. Add each age to the total. At the end, calculate the average by dividing the total by the number of friends.

Possible solution:

```
friends = {'richard': 96, 'john': 12, 'paul': 8}
total = 0
for name, age in friends.items():
    total = total + age
average = total / 3
print("average age is ", average)
```

2.15 Bugs

Fixing bugs can feel frustrating but all programmers must wrestle with them. The simplest (but still annoying!) are *syntax errors*. The computer is not very intelligent and is unable to guess what you mean, so you must type the programs in this book **exactly** as they appear. A single wrong letter or space will prevent them from working.

A particular issue in Python is that *indentation* must be correct.

Program 2.21: Buggy program

```
1
2 x = 10
3  y = 11
4 z = 12
5 print(x,y,z)
```

Exercise

Can you spot and fix the bug in Program 2.21?

Program 2.22: More bugs

```
1
2 def myfunction:
3     print "hello"
4
5 myfunction()
```

Exercise

Program 2.22 has two bugs to fix.

CHAPTER

THREE

TEXT-BASED QUIZ GAMES

These programs can be entered using any text editor, but I suggest using the Mu editor[2] because it comes with Python, Pygame Zero and other libraries all pre-installed in one easy download.

3.1 Hello, world

The traditional first program used to make sure Python is working and that we can run programs.

If using the Mu editor:

1. Click the mode button and make sure the mode is set to `Python3`.
2. Type in the program.
3. Click `Save` and enter a name for the program.
4. Click `Run`.

Program 3.1: Hello, world

```
print("Hello world")

# This line is a comment, you dont have to type these!
```

3.2 Getting input from the keyboard

This program will pause and wait for you to enter some text with the keyboard, followed by the return key. The text you enter is stored in a variable, `x`.

Program 3.2: Getting input from the keyboard

```

print("Enter your name:")
```

(continues on next page)

[2] https://codewith.mu/

(continued from previous page)

```
x = input()
print("Hello", x)
if x == "richard":
    print("That is a very cool name")
```

Exercise

Add some names of your friends and display a different message for each friend.

3.3 Making decisions: if, elif, else

This is how to add another name to Program 3.2

Program 3.3: Decisions: if, elif, else

```
print("Enter your name:")
x = input()
print("Hello", x)
if x == "richard":
    print("That is a very cool name")
elif x == "nick":
    print("That is a rubbish name")
else:
    print("I do not know your name", x)
```

Program 3.3 is very similar to Program 3.2. The new lines have been highlighted. You can either modify Program 3.2, or else create a new file and use copy and paste to copy the code from the old program into the new.

3.4 A random maths question

Program 3.4: A random maths question

```
import random

n = random.randint(0, 10)

print("What is", n, "plus 7?")
g = int(input())  # Why do we use int() here?
if g == n + 7:
    print("Correct")
else:
    print("Wrong")
```

Exercise

Add some more questions, e.g.

- Instead of 7, use another random number.
- Use a bigger random number.
- Multiply (use *), divide (use /) or subtract (use -) numbers.

Exercise

Print how many questions the player got correct at the end.

3.5 Keeping score

We create a `score` variable to record how many questions the player answered correctly.

Program 3.5: Keeping score

```
score = 0

print("What is 1+1 ?")
g = int(input())
if g == 2:
    print("Correct")
    score = score + 1

print("What is 35-25 ?")
g = int(input())
if g == 10:
    print("Correct")
    score = score + 1

print("Your score:", score)
```

3.6 Guessing game with a loop

This `while` loop goes round and round forever ... or until the player gets a correct answer, and then it ``break``s out of the loop. Note that everything in the loop is indented.

Program 3.6: Guessing game with a loop

```
import random

n = random.randint(0, 10)

while True:
    print("I am thinking of a number, can you guess what it is?")
    g = int(input())
    if g == n:
        break
    else:
        print("Wrong")
print("Correct!")
```

Exercise

Give a hint to the player when they are wrong. Was their guess too high or too low?

Exercise

Print how many guesses they took to get it right at the end.

3.7 Improved guessing game

Program 3.6 with a hint whether the guess is greater or lesser than the answer.

Program 3.7: GImproved guessing game

```
import random

n = random.randint(0, 100)
guesses = 0

while True:
    guesses = guesses + 1
    print("I am thinking of a number, can you guess what it is?")
    g = int(input())
    if g == n:
        break
    elif g < n:
        print("Too low")
    elif g > n:
        print("Too high")
print("Correct! You took", guesses, "guesses.")
```

CHAPTER

FOUR

DRAWING GRAPHICS

To create graphics for our games we will use Raylib[3] along with my RLZero library[4]. You will find the documentation on the websites useful!

4.1 Installing the graphics library

Installing a Python package varies depending on what Python editor or IDE you are using. Here is how you do it if you are using the Mu editor.

Run *Mu*. Click **Mode** and select **Python3**. Then click *the small gadget icon* in the bottom right hand corner of the window. Click **third party packages**. Type

```
rlzero
```

into the box. Click **OK**. The library will download.

If you are not using Mu you can install from the command line like this:

```
pip3 install rlzero
```

4.2 RLZero and Raylib

Raylib is a graphics library. RLZero adds some extra functions to Raylib to make it easier to use. Once you have installed RLZero you are free to use all the functions of Raylib - you don't have to stick to the RLZero features.

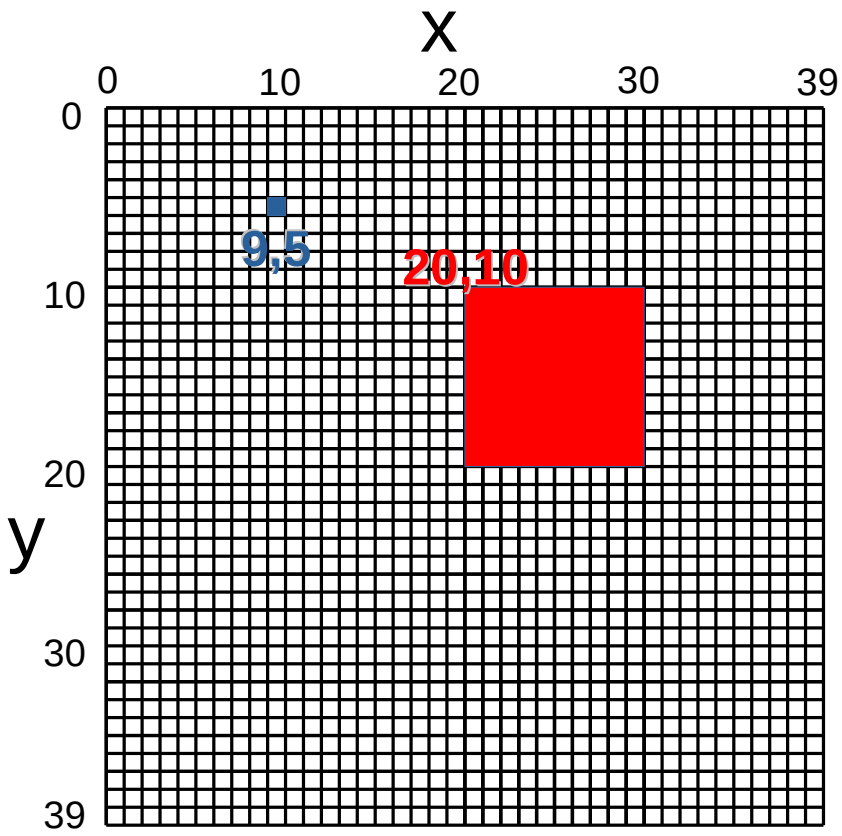
Specifically, whenever we use a function prefixed by `screen.` that is a Raylib function. You can use any function from the Raylib documentation[5] if you prefix it with `screen`.

The RLZero functions don't need a prefix.

[3] https://www.raylib.com/
[4] https://electronstudio.github.io/rlzero/
[5] https://electronstudio.github.io/raylib-python-cffi/pyray.html

4.3 Pixels

The smallest square that can be displayed on a monitor is called a *pixel*. This diagram shows a close-up view of a window that is 40 pixels wide and 40 pixels high. At normal size you will not see the grid lines.



We can refer to any pixel by giving two co-ordinates, *(x,y)* Make sure you understand co-ordinates before moving on because everything we do in Pygame Zero will use it. (In maths this called a 'Cartesian coordinate system').

Program 4.1: A pixel

```
from rlzero import *

WIDTH = 500  # What are these units? What if we change them?
HEIGHT = 500  # What if we delete this line?


def draw():
    clear()
    screen.draw_pixel(250, 250, RED)

run()
```

Exercise

Make this pixel blue.

4.4 Lines and circles

Program 4.2: Lines and circles

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

def draw():
    clear()
    screen.draw_circle_lines(250, 250, 50, WHITE)
    screen.draw_circle(250, 100, 50, RED)
    screen.draw_line(150, 20, 150, 450, PURPLE)
    screen.draw_line(150, 20, 350, 20, PURPLE)

run()
```

Exercise

Finish drawing this picture

Exercise

Draw your own picture.

4.5 Moving rectangles

To make things move we need to add the special `update()` function. We don't need to write our own loop because *RLZero calls this function for us in its own loop*, over and over, many times per second.

Program 4.3: Moving rectangles

```
from rlzero import *

WIDTH = 500
HEIGHT = 500
```

(continues on next page)

(continued from previous page)

```
5
6  box = screen.Rectangle(20, 20, 50, 50)
7
8  def draw():
9      clear()
10     screen.draw_rectangle_rec(box, RED)
11
12 def update():
13     box.x = box.x + 2
14     if box.x > WIDTH:
15         box.x = 0
16
17 run()
```

Exercise

Make the box move faster.

Exercise

Make the box move in different directions.

Exercise

Make two boxes with different colours.

4.6 Sprites

Sprites are very similar to boxes, but are loaded from a **png** or **jpg** image file. RLZero has one such image built-in, `alien.png`. If you want to use other images you must create them and place the files in the same directory as your program.

You could use Microsoft Paint which comes with Windows but I recommend you download and install Krita[6].

Program 4.4: Actor sprites

```
1  from rlzero import *
2
3  WIDTH = 500
4  HEIGHT = 500
5
```

(continues on next page)

[6] https://krita.org

(continued from previous page)

```
alien = Sprite('alien.png')
alien.x = 0
alien.y = 50


def draw():
    clear()
    alien.draw()


def update():
    alien.x += 2
    if alien.x > WIDTH:
        alien.x = 0

run()

```

Exercise

Draw or download your own image to use instead of alien.

4.7 Background image

We are going to add a background image to Program 4.4

You must create or download a picture to use a background. Save it as `background.png` in the folder with your program. It should be the same size as the window, 500×500 pixels and it must be in `.png` format.

Program 4.5: Background

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

alien = Sprite('alien.png')
alien.x = 0
alien.y = 50

background = Sprite('background')

def draw():
    clear()
    background.draw()
```

(continues on next page)

(continued from previous page)

```
    alien.draw()


def update():
    alien.x += 2
    if alien.x > WIDTH:
        alien.x = 0

run()

```

Exercise

Create a picture to use a background. Save it as *background.png*. Run the program.

4.8 Keyboard input

The alien moves when you press the cursor keys.

Program 4.6: Keyboard input

```
from rlzero import *

alien = Sprite('alien')
alien.pos = (0, 50)

def draw():
    clear()
    alien.draw()

def update():
    if keyboard.right:
        alien.x = alien.x + 2
    elif keyboard.left:
        alien.x = alien.x - 2

run()
```

Exercise

Make the alien move up and down as well as left and right.

Exercise

Use the more concise += operator to change the *alien.x* value (see Program 2.18).

Exercise

Use the *or* operator to allow WASD keys to move the alien in addition to the cursor keys (see Program 2.8).

CHAPTER

FIVE

TUTORIAL: FRACTALS

> In mathematics, a fractal is a subset of Euclidean space with a fractal dimension that strictly exceeds its topological dimension.
>
> —Wikipedia

> A fractal is a never-ending pattern. Fractals are infinitely complex patterns that are self-similar across different scales. They are created by repeating a simple process over and over in an ongoing feedback loop. Fractal patterns are extremely familiar, since nature is full of fractals. For instance: trees, rivers, coastlines, mountains, clouds, seashells, hurricanes, etc.
>
> —Fractal Foundation

Fractals are very interesting things if you're studying maths, but even if you are not, they are still just plain fun because:

- They make pretty pictures when drawn on the screen.
- They require very little code to draw.
- You can zoom in forever and always find new details, yet you don't have to write any extra code.

We are going to write a program to draw one fractal in particular, the Mandelbrot set.

5.1 Mandelbrot set

We are going to skip through the maths so we can get straight to coding, but if you would like to understand complex numbers, watch this video[7] or ask your maths teacher.

Every complex number is either *in* the Mandelbrot set, or *out* of the Mandelbrot set. To decide which it is, we do an iteration:

$$z_{n+1} = {z_n}^2 + c$$

If z gets bigger and bigger (tends towards infinity) then c is in the set. If it stays in the range of -2 to 2, then c is not in the set. However, we don't want to perform the

[7] https://youtu.be/NGMRB4O922I

iteration infinity times, so we are going to limit it to 80 iterations of the loop. If it's still in the range after that, we will return 80. If it goes outside the range, then we will return how many iterations it took.

(We could have simply returned True or False but we are going to use the number of iterations for something later.)

Here is our function. It *takes* a complex number c, and it *returns* the number of iterations (which will be 80 if c is in the Mandelbrot set.)

```
MAX_ITER = 80

def mandelbrot(c):
    z = 0
    n = 0
    while abs(z) <= 2 and n < MAX_ITER:
        z = z * z + c
        n += 1
    return n
```

Now we just need a function to draw the points on the screen. We have two `for` loops, one inside the other, which loop over every pixel on the screen. For each pixel, we convert the `x` and `y` coordinates into a complex number. We then send that number to the `mandlebrot()` function, and depending on what it returns we plot either a black pixel or a white pixel.

```
def draw2d():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH),
                        (IM_START + (y / HEIGHT) * IM_HEIGHT))
            m = mandelbrot(c)
            color = BLACK if m == MAX_ITER else WHITE
            screen.draw_pixel(x, y, color)
```

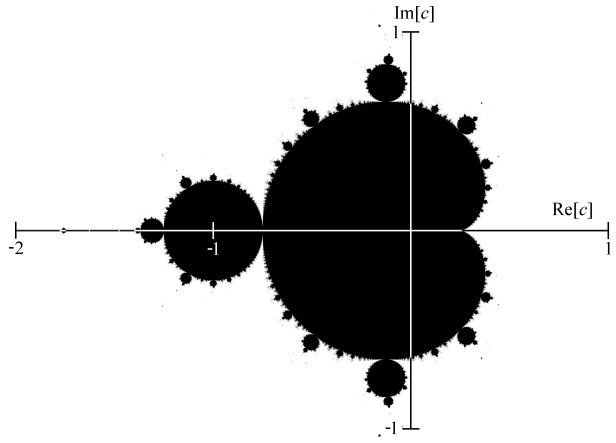


Fig. 5.1: Complex plain

A Mandlebrot is drawn in the *complex plain*. This means that the axes are not labelled

x and *y*. Instead we call the horizontal axis *RE* (for 'real') and the vertical axis *IM* (for 'imaginary'). These terms come from the maths of complex numbers. So we define some constants to specify the left, right, top and bottom limits of the graph:

```
RE_START = -2
RE_END = 1
IM_START = -1
IM_END = 1
RE_WIDTH = (RE_END - RE_START)
IM_HEIGHT = (IM_END - IM_START)
```

Here is the complete program to type in and run:

Program 5.1: Mandlebrot set

```
from rlzero import *

WIDTH = 700
HEIGHT = 400
RE_START = -2
RE_END = 1
IM_START = -1
IM_END = 1
RE_WIDTH = (RE_END - RE_START)
IM_HEIGHT = (IM_END - IM_START)
MAX_ITER = 80


def mandelbrot(c):
    z = 0
    n = 0
    while abs(z) <= 2 and n < MAX_ITER:
        z = z * z + c
        n += 1
    return n


def draw2d():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH),
                        (IM_START + (y / HEIGHT) * IM_HEIGHT))
            m = mandelbrot(c)
            color = BLACK if m == MAX_ITER else WHITE
            screen.draw_pixel(x, y, color)


run()
```

5.2 Shades of Grey

The reason we returned the number of iterations is that for a point *outside* of the Mandlebrot set, this number tells us how far outside it is. So instead of just plotting black and white for *in* or *out* we can plot shades of grey.

Here is the modified function that does this,

```
def draw2d():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH),
                        (IM_START + (y / HEIGHT) * IM_HEIGHT))
            m = mandelbrot(c)
            i = 255 - int(255 * m / MAX_ITER)
            color = (i, i, i, 255)
            screen.draw_pixel(x, y, color)
```

Modify your `draw2d()` function and run.

5.3 Colours

In the previous function we created an *RGB* `color` where the red, green and blue values were the all same, i.e. a shade of grey.

RGB is the most common colour space, but it is not only one. The *HSV* colour space is useful because one single value, the *hue*, can be changed to produce completely different colours.

```
def draw2d():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH),
                        (IM_START + (y / HEIGHT) * IM_HEIGHT))
            m = mandelbrot(c)
            hue = int(255 * m / MAX_ITER)
            saturation = 255
            value = 255 if m < MAX_ITER else 0
            color = screen.color_from_hsv(hue, saturation, value)
            screen.draw_pixel(x, y, color)
```

Modify your `draw2d()` function and run.

5.4 Zooming in

Lets introduce a new variable, *zoom*. We will multiply our co-ordinates by this factor to enable zooming.

We will also add an `update` function. This is called automatically and will handle input. You can now hold down the space and enter to zoom in and out, and hold the cursor keys to move around.

Program 5.2: Mandlebrot set with colour and zooming

```
from rlzero import *

WIDTH = 700
HEIGHT = 400
RE_START = -2
RE_END = 1
IM_START = -1
IM_END = 1
RE_WIDTH = (RE_END - RE_START)
IM_HEIGHT = (IM_END - IM_START)
MAX_ITER = 80

zoom = 1.0


def mandelbrot(c):
    z = 0
    n = 0
    while abs(z) <= 2 and n < MAX_ITER:
        z = z * z + c
        n += 1
    return n


def draw2d():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH) * zoom,
                        (IM_START + (y / HEIGHT) * IM_HEIGHT) * zoom)
            m = mandelbrot(c)
            hue = int(255 * m / MAX_ITER)
            saturation = 255
            value = 255 if m < MAX_ITER else 0
            color = screen.color_from_hsv(hue, saturation, value)
            screen.draw_pixel(x, y, color)


def update():
    global zoom, IM_START, RE_START
```

(continues on next page)

(continued from previous page)

```
    if keyboard.space:
        zoom *= 1.2
    elif keyboard.enter:
        zoom *= 0.8
    elif keyboard.up:
        IM_START -= 0.2
    elif keyboard.down:
        IM_START += 0.2
    elif keyboard.left:
        RE_START -= 0.2
    elif keyboard.right:
        RE_START += 0.2


run()
```

This is the complete program. Modify yours to match, or enter it again, and run it.

Note: Zooming may be slow and you may have to hold for a long time. We will try to improve this next.

5.5 Performance

The `draw2d()` function is called automatically, up to 60 times per second, and every time it is called we plot a new fractal. This is very inefficient, because we are re-plotting the fractal even when it has not changed at all.

So, we will move the plotting code into a new function. Instead of plotting directly to the screen, we will create an `image` object and plot to this.

```
def plot_image():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH),
                        (IM_START + (y / HEIGHT) * IM_HEIGHT))
            m = mandelbrot(c)
            hue = int(255 * m / MAX_ITER)
            saturation = 255
            value = 255 if m < MAX_ITER else 0
            color = pyray.color_from_hsv(hue, saturation, value)
            screen.image_draw_pixel(image, x, y, color)


image = screen.gen_image_color(WIDTH, HEIGHT, GREEN)
plot_image()
```

We can't draw the `image` directly to the screen; it must be converted into a `texture` first. Unfortunately we can't create a `texture` too early in the program, because it requires the GPU to have been initialized. Therefore we do this in a special `init()` function which RLZero calls automatically after initialization.

```
def init():
    global texture
    texture = screen.load_texture_from_image(image)
    screen.set_texture_filter(texture, screen.TEXTURE_FILTER_BILINEAR)
```

Now our `draw2d()` function only has to update the `texture` based on the latest `image` and draw it to the screen, which is much faster than re-plotting the whole thing.

```
def draw2d():
    screen.update_texture(texture, image.data)
    screen.draw_texture_ex(texture, (0, 0), 0, 1, WHITE)
```

Finally, we must remember to re-plot the image if the user presses any keys:

```
def update():
    global zoom, IM_START, RE_START
    if keyboard.space:
        zoom = zoom * 1.2
        plot_image()
    elif keyboard.enter:
        zoom = zoom * 0.8
        plot_image()
    elif keyboard.up:
        IM_START -= 0.2
        plot_image()
    elif keyboard.down:
        IM_START += 0.2
        plot_image()
    elif keyboard.left:
        RE_START -= 0.2
        plot_image()
    elif keyboard.right:
        RE_START += 0.2
        plot_image()
```

Program 5.3: Mandlebrot set with improved performance

```
1 from rlzero import *
2
3 WIDTH = 700
4 HEIGHT = 400
5 RE_START = -2
6 RE_END = 1
7 IM_START = -1
8 IM_END = 1
9 RE_WIDTH = (RE_END - RE_START)
```

(continues on next page)

(continued from previous page)

```
IM_HEIGHT = (IM_END - IM_START)
MAX_ITER = 80

zoom = 1.0


def mandelbrot(c):
    z = 0
    n = 0
    while abs(z) <= 2 and n < MAX_ITER:
        z = z * z + c
        n += 1
    return n


def plot_image():
    for x in range(0, WIDTH):
        for y in range(0, HEIGHT):
            c = complex((RE_START + (x / WIDTH) * RE_WIDTH) * zoom,
                        (IM_START + (y / HEIGHT) * IM_HEIGHT) * zoom)
            m = mandelbrot(c)
            hue = int(255 * m / MAX_ITER)
            saturation = 255
            value = 255 if m < MAX_ITER else 0
            color = screen.color_from_hsv(hue, saturation, value)
            screen.image_draw_pixel(image, x, y, color)


image = screen.gen_image_color(WIDTH, HEIGHT, GREEN)
plot_image()


def init():
    global texture
    texture = screen.load_texture_from_image(image)
    screen.set_texture_filter(texture, screen.TEXTURE_FILTER_BILINEAR)


def draw2d():
    screen.update_texture(texture, image.data)
    screen.draw_texture_ex(texture, (0, 0), 0, 1, WHITE)

def update():
    global zoom, IM_START, RE_START
    if keyboard.space:
        zoom *= 1.2
        plot_image()
    elif keyboard.enter:
        zoom *= 0.8
```

(continues on next page)

(continued from previous page)

```
        plot_image()
    elif keyboard.up:
        IM_START -= 0.2
        plot_image()
    elif keyboard.down:
        IM_START += 0.2
        plot_image()
    elif keyboard.left:
        RE_START -= 0.2
        plot_image()
    elif keyboard.right:
        RE_START += 0.2
        plot_image()

run()
```

5.6 Quality setting

It would be nice if we could make the window bigger so we can see the fractal more easily, but depending on the speed of your computer you may already find even the small window is very slow.

A simple way of speeding it up is to plot the image at a lower resolution and scale it up to full size when we draw the texture to the screen.

Change the resolution at the top of the program, and add a `SCALE` variable:

```
WIDTH = 1920
HEIGHT = 1080
SCALE = 4
```

Change the first part of the `plot_image` function to use the `SCALE`:

```
def plot_image():
    for x in range(0, WIDTH//SCALE):
        for y in range(0, HEIGHT//SCALE):
            c = complex((RE_START + (x / (WIDTH//SCALE)) * RE_WIDTH) * zoom,
                        (IM_START + (y / (HEIGHT//SCALE)) * IM_HEIGHT) * zoom)
```

Change the `draw_2d()` function to use the `SCALE`:

```
def draw2d():
    screen.update_texture(texture, image.data)
    screen.draw_texture_ex(texture, (0,0), 0, SCALE, WHITE)
```

Run the program and experiment with different `SCALE` values.

5.7 Further improvements

If you have ever used a fractal viewer program before, you will probably notice that it is faster than the simple one we have written. How could we make ours faster?

- Divide the image up into parts and use multiple CPUs to plot the different parts of the image simultaneously.
- Save images once they are plotted so they don't need to be plotted again if the user returns to them.
- When scrolling, don't re-plot the whole image; just move the existing data and re-plot the now empty part.
- When zooming in, first scale-up the existing image to generate a low quality zoomed image, then plot a higher quality one later.
- Begin plotting with a low resolution image and then replace it with a higher resolution one later.