Learn to code with Python and Raylib

release

Richard Smith

Aug 01, 2022

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 Raylib 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]
- No-one actually knows how best to teach coding, so there are a variety of approaches, but some studies have shown students enjoy doing purely syntactic practises, and those who do them go on to do better when learning semantics.
- Initially, 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. The more advanced chapters do use the piecemeal approach but always include a complete listing of the end result.
- 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 looser 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.

This is a practical approach, and so the programs are the core of it. In the first edition they were all of it. In this edition explanatory text has been added, but for those who don't have the patience to read, this is can be skipped. The code cannot.

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 is 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.

The third edition

The library used in this book has been switched from Pygame to Raylib. The initial motivation for this was to allow the addition of 3d graphics, which are the main new addition to the book.

But there are other advantages. It turns out that the 'zero boilerplate' approach, in the sense of making the program as short as it can possibly be, is not actually most conducive to learning. Sometimes being explicit rather than implicit is good.

So the Raylib programs are usually a couple of lines longer, but they expose the student to more of what is going on in the computer and I hope provide an easier transition into using a 'real' API, without being dependent on training wheels.

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 RLZero 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.

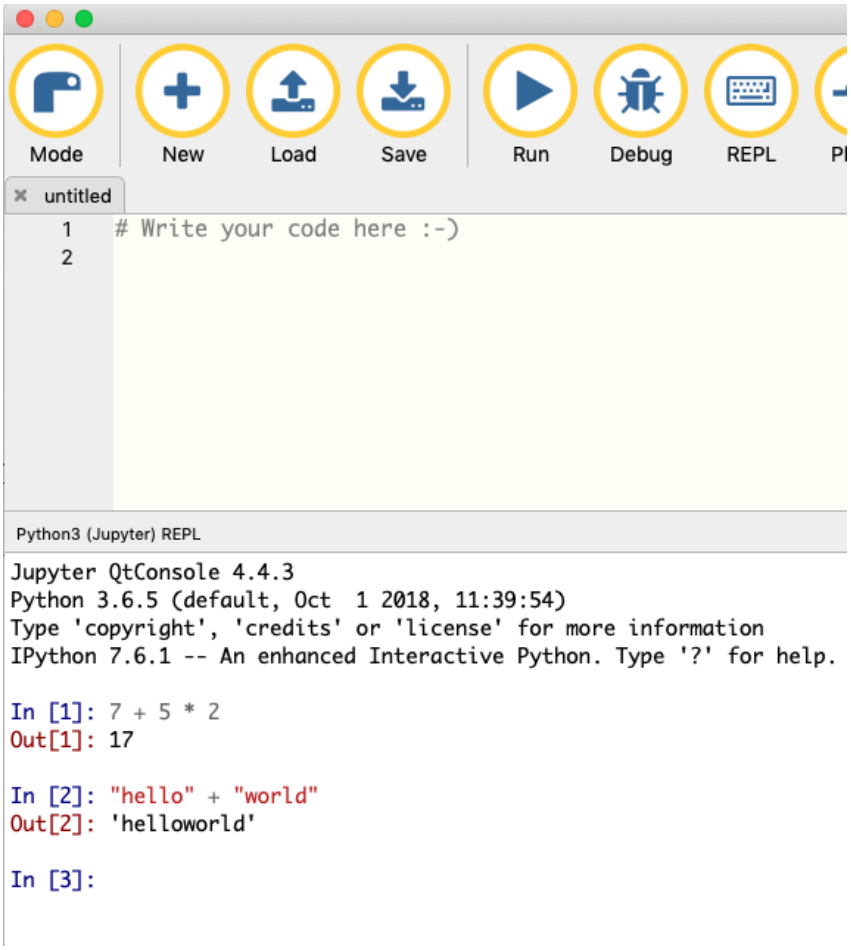


Fig. 2.1: The REPL

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 do you think Program 2.4 will print? Run it and see if your prediction is correct.

Exercise

Change the numbers in the program. Use a division / operation on the variable. 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(total)
```

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 sunny:
    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

A comparison operator takes two things (e.g. numbers, strings, variables), compares them, and then returns a *boolean* `True` or `False`.

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":
```

(continues on next page)

(continued from previous page)

```
    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

```
for a in range(0, 6):
    for b in range(0, 6):
        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

```
total=0
for x in range(1, 4):
    print("Customer", x, "how many cakes do you want?")
    cakes = int(input())
    total = total + cakes
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:
```

(continues on next page)

(continued from previous page)

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

2.10 Functions

You will meet specially named functions that are called by RLZero such as `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 to 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"
```

(continues on next page)

(continued from previous page)

```
print(message)

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

(continues on next page)

(continued from previous page)

```
        if i == 5:
            print("in if block")
        print("leaving for loop block")
    print("leaving function block")
print("not in any block")
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

```
a = 10
def my_function():
    global a
    a=20
my_function()
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
for name, age in friends.items():
    if age > 10 and age < 14:
```

(continues on next page)

(continued from previous page)

```
        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:
```

(continues on next page)

(continued from previous page)

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

Exercise

Can you spot and fix the bug in Program 2.21?

Program 2.22: More bugs

```
def myfunction:
    print "hello"

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

[2] https://codewith.mu/

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 `breaks` 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: Improved 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

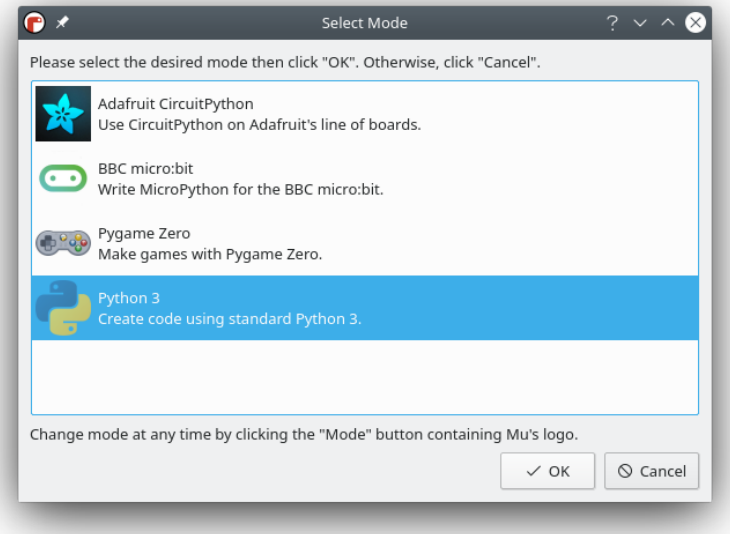
DRAWING GRAPHICS

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>=0.3<0.4
```

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

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

[3] https://www.raylib.com/
[4] https://electronstudio.github.io/rlzero/

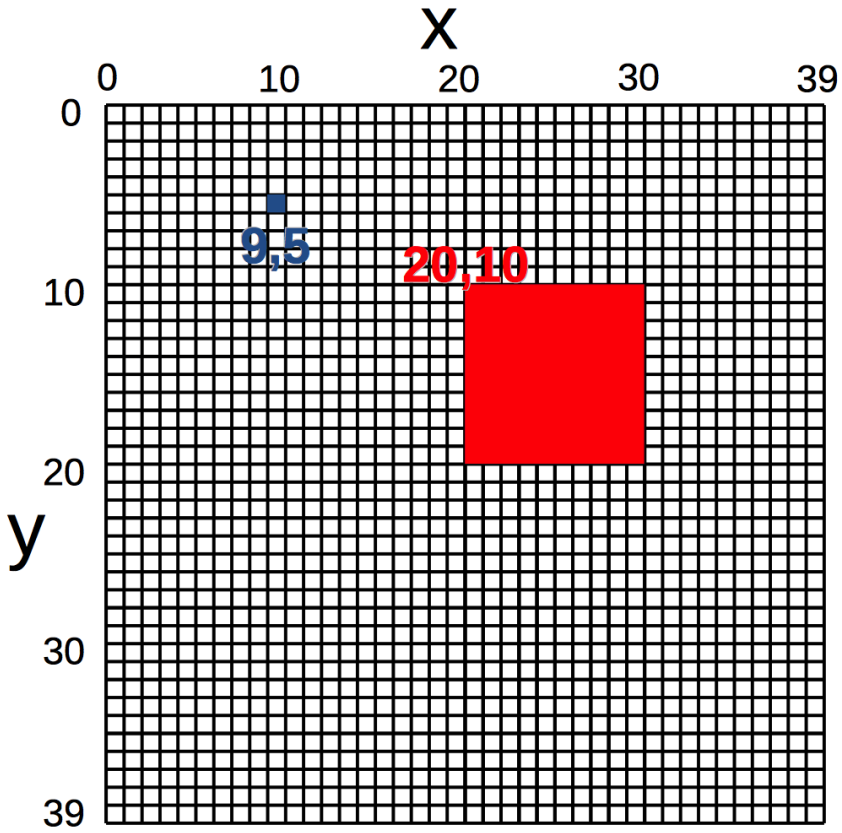
```
python3 -m pip install 'rlzero>=0.3<0.4'
```

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[5]- you don't have to stick to the RLZero functions.

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 will use them. (In maths this called a 'Cartesian coordinate system'.)

Now type in Program 4.1 and run it, to see a pixel. It's very small, so look carefully!

[5] https://electronstudio.github.io/raylib-python-cffi/pyray.html

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()
    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():
    draw_circle_lines(250, 250, 50, WHITE)
    draw_circle(250, 100, 50, RED)
    draw_line(150, 20, 150, 450, PURPLE)
    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

box = Rectangle(20, 20, 50, 50)


def draw():
    draw_rectangle_rec(box, RED)

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

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.

We are going to store the sprite in a variable so we can refer to it again when we want to draw it. We have called this variable *alan*. We could create more sprites using the same image, but we would have to use different variable names.

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

Program 4.4: Actor sprites

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

alan = Sprite('alien.png')
alan.x = 50
alan.y = 100


def draw():
    alan.draw()


def update():
    alan.x += 1
    if alan.x > WIDTH:
        alan.x = 0

run()

```

Exercise

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

[6] https://krita.org

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

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

background = Sprite('background.png')

def draw():
    background.draw()
    alan.draw()


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

run()

```

Exercise

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

4.8 Keyboard input

Alan moves when you press the cursor keys.

Program 4.6: Keyboard input

```
from rlzero import *

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

```

(continues on next page)

(continued from previous page)

```
6  def draw():
7      alan.draw()
8
9  def update():
10     if keyboard.right:
11         alan.x = alan.x + 2
12     elif keyboard.left:
13         alan.x = alan.x - 2
14
15 run()
```

Exercise

Make Alan move up and down as well as left and right.

Exercise

Use the more concise += operator to change the *alien.x* value (see Section 2.11).

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

ARCADE GAMES

5.1 Collisions

Arcade games need to know when one sprite has hit another sprite. Most of this code is copied from Program 4.3 and Program 4.6.

Program 5.1: Collisions

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

alan = Sprite('alien.png')
alan.pos = (400, 50)
box = Rectangle(20, 20, 100, 100)

def draw():
    draw_rectangle_rec(box, RED)
    alan.draw()

def update():
    if keyboard.right:
        alan.x = alan.x + 2
    elif keyboard.left:
        alan.x = alan.x - 2
    box.x = box.x + 2
    if box.x > WIDTH:
        box.x = 0
    if alan.colliderect(box):
        print("hit")


run()

```

Exercise

Add vertical movement (as you did in Exercise Program 4.6).

Advanced

Make the box chase the alien.

Advanced

Print number of times the box hits the alien (i.e. the score).

5.2 Chase

Instead of moving constantly to the right we can make the movement conditional with an `if` statement so the box chases the alien. Most of this code is copied from Program 5.1. New lines are highlighted. We have also changed what happens when the box catches the alien: the program now exits and you must run it again to play again. This may not be what you want in your game!

Program 5.2: Alien chase

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

alan = Sprite("alien.png")
alan.pos = (400, 50)
box = Rectangle(20, 20, 100, 100)

def draw():
    draw_rectangle_rec(box, RED)
    alan.draw()

def update():
    if keyboard.right:
        alan.x = alan.x + 2
    elif keyboard.left:
        alan.x = alan.x - 2
    if box.x < alan.x:
        box.x = box.x + 1
    if box.x > alan.x:
        box.x = box.x - 1
    if alan.colliderect(box):
        exit()

run()
```

Exercise

Add vertical movement (as you did in previous exercise).

Advanced

Draw a new enemy image. Save it as *enemy.png* in your *mu_code/images* folder. Load it as an *Actor('enemy')* instead of the *Rect()*.

5.3 Powerup

Instead of an enemy the box here is a powerup that the player must collect. When he does it disappears and moves to a new location.

Program 5.3: Collect the powerups

```
from rlzero import *
import random

WIDTH = 500
HEIGHT = 500

alan = Sprite("alien.png")
alan.pos = (400, 50)
box = Rectangle(20, 20, 100, 100)
score = 0

def draw():
    draw_rectangle_rec(box, GREEN)
    alan.draw()

def update():
    global score
    if keyboard.right:
        alan.x = alan.x + 2
    elif keyboard.left:
        alan.x = alan.x - 2
    if alan.colliderect(box):
        box.x = random.randint(0, WIDTH)
        box.y = random.randint(0, HEIGHT)
        score = score + 1
        print("Score:", score)

run()
```

Exercise

Add vertical movement.

Exercise

Draw a new powerup image. Save it as `powerup.png` in the same folder as your program. Load it as a `Sprite("powerup.png")` instead of the `Rectangle()`.

Advanced

Combine this program with the enemy from Program Program 5.2 and the background from Program 4.5 and whatever else you want to make your own game.

5.4 Sounds

RLZero comes one sound effect: `eep.wav`. If you want more you will have to create them (or download them) yourself and save them in the same folder as your program.

This program plays a sound when you press space.

Program 5.4: Sound

```
from rlzero import *

my_sound = Sound("eep.wav")

def draw():
    clear_background(WHITE)

def update():
    if keyboard.space:
        my_sound.play()

run()
```

Exercise

Download a `.wav` audio file and play it in the program.

5.5 Simple animation

This program changes the image of the Sprite to create a simple animation when he is hit.

RLZero comes with the image file `alien_hurt.png`. If you want more you will have to create them (or download them) yourself and save them in the same folder as your program.

Most of this code is copied from Program 5.1

Program 5.5: Sound and animation upon collision

```
from rlzero import *

WIDTH = 500
HEIGHT = 500
alan = Sprite("alien.png")
alan.pos = (0, 50)
box = Rectangle(20, 20, 100, 100)
eep = Sound("eep")

def draw():
    draw_rectangle_rec(box, RED)
    alan.draw()

def update():
    if keyboard.right:
        alan.x = alan.x + 2
    elif keyboard.left:
        alan.x = alan.x - 2
    box.x = box.x + 2
    if box.x > WIDTH:
        box.x = 0
    if alan.colliderect(box):
        alan.image_file = "alien_hurt.png"
        eep.play()
    else:
        alan.image_file = "alien.png"

run()
```

Exercise

Record your own sound effect and add it to the game.

Advanced

Add more boxes or sprites that move in different ways for the player to avoid.

Advanced

Add a second alien controlled by different keys or gamepad for player 2.

5.6 Mouse clicks

This uses a *function call-back* for event-based input. It is similar to Program 5.5 but:

- The box has been removed.
- There is an `on_mouse_down()` special function that is called automatically when the player click the mouse.
- The score is displayed.

See Program 2.17 for more about functions.

Program 5.6: Getting input from mouse clicks

```
from rlzero import *

alan = Sprite("alien.png")
alan.pos = (0, 50)
eep = Sound("eep")

score = 0

def draw():
    alan.draw()
    draw_text("Score "+str(score), 0, 0, 20, WHITE)

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

def on_mouse_down(pos, button):
    global score
    if button == MOUSE_BUTTON_LEFT and alan.collidepoint(pos):
        alan.image = 'alien_hurt.png'
        eep.play()
        score = score + 1

run()

```

5.7 Mouse movement

Program 5.7: Getting input from mouse movement

```
from rlzero import *

# wiggle your mouse around the screen!

alan = Sprite("alien.png")

def draw():
    alan.draw()

def on_mouse_move(pos):
    alan.pos = pos

run()
```

Exercise

What happens if you change *on_mouse_move* to *on_mouse_down*?

Advanced

Make a game with one alien controlled by mouse and another controlled by keyboard

CHAPTER

SIX

IMPROVING YOUR GAMES

Here are some tips that will enable you to enhance your games.

6.1 Game controllers

You may call them *joysticks* or *controllers*, but Raylib calls them *gamepads*.

Some controllers have different inputs and some are not compatible at all so don't be surprised if this doesnt work properly! PS4 and Xbox One controllers connected by USB cable seems to work best.

Program 6.1: Joystick input

```
from rlzero import *

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

def draw():
    alan.draw()

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

run()

```

Optional, if you have a controller

Make the alien move up and down as well as left and right using the controller. Do the same for any other examples that use the keyboard!

6.2 Colours

Note: The word *colour* is incorrectly spelt *color* in American programs such as Raylib. We will try to use the correct spelling wherever we can but sometimes you will have to use *color*.

Instead of using ready made colours like `RED`, `YELLOW`, etc. you can create your own colour with a *tuple* of three numbers. The numbers must be between 0 - 255 and represent how much *red*, *green* and *blue* to mix together.

Program 6.2: RGB colours

```
from rlzero import *

# my_colour = Color(0,0,0,255) # makes black
# my_colour = Color(255,255,255,255) # makes white

red_amount = 0
green_amount = 0
blue_amount = 0
alpha_amount = 255

def draw():
    my_colour = Color(red_amount, green_amount, blue_amount, alpha_amount)
    clear_background(my_colour)

# This function makes the colour change every frame
# Remove it if you just want to see a static colour.
def update():
    global red_amount
    red_amount += 1
    red_amount = red_amount % 255

run()
```

Advanced

Change the green and blue amounts to make different colours.

Exercise

Make a gray colour.

Advanced

Make random colours. (see Program 3.4).

6.3 Using loops

The loops from Program 3.6 are useful for graphical games too! Here we draw red circles using a *for loop*.

We draw green circles using a *loop within another loop*.

Program 6.3: Loops are useful

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

def draw():
    for x in range(0, WIDTH, 40):
        draw_circle(x, 20, 20, RED)

    for x in range(0, WIDTH, 40):
        for y in range(60, HEIGHT, 40):
            draw_circle(x, y, 10, GREEN)

run()
```

Exercise

import random at the top of the program and then make the positions random, e.g:

```
x = random.randint(0, 100)
```

Advanced

Learn about RGB colour[7] and make random colours (see Program 6.2).

Advanced

Create a timer variable and change colours based on time (see Program 7.6)

[7] https://en.wikipedia.org/wiki/RGB_color_model

6.4 Fullscreen mode

Sometimes it is nice to play your game using the entire screen rather than in a window. This program show how to enter fullscreen and and how to quit.

Program 6.4: Fullscreen mode

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

alien = Sprite("alien")

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

def update():
    if keyboard.f1:
        toggle_fullscreen()
    if keyboard.enter:
        exit()

run()
```

Note: RLZero **automatically** uses the *F* key for fullscreen and *Escape* key to quit, so you don't actually need this code in your programs unless you want to use a different key.

6.5 Displaying the score

This game shows how you can keep the score in a variable and print it on to the game screen. You can display any other messages to the player the same way.

Program 6.5: Keeping score in a variable and displaying it

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

score = 0

def draw():
    clear()
    draw_text(f"Player 1 score: {score}", 0, 0, 20, RED)

# This is another special function that is called by RLzero automatically
```

(continues on next page)

(continued from previous page)

```
# each time a key is pressed. That way player cannot just hold down the key!

def on_key_pressed(key):
    global score
    if key == KEY_SPACE:
        score += 1

run()
```

Exercise

Make the score text larger.

Advanced

Add a second player who presses a different key and show their score too.

Exercise

Add text to one of your other games.

6.6 Animation

We did some animation ourselves in Program 5.5.

Animation is a RLZero class that makes it easier to show a sequence of different images on the same Sprite. You just need to give it the image file names and the number of frames-per-second you want and then call its update method.

Program 6.6: Animation

```
from rlzero import *

WIDTH = 500
HEIGHT = 500
alien = Sprite("alien.png")
animation = Animation(["alien.png","alien_hurt.png"], 5)
alien.pos = (0, 50)


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

def update():
```

(continues on next page)

(continued from previous page)

```
15      if keyboard.right:
16          alien.x = alien.x + 2
17      elif keyboard.left:
18          alien.x = alien.x - 2
19      animation.update(alien)
20
21  run()
```

Exercise

Draw three images of a man walking and save them as *man1.png*, *man2.png*, *man3.png*. Edit the program so it animates these images.

CHAPTER

SEVEN

MORE ADVANCED GAMES

These games demonstrate some essential building blocks you will need to make more advanced games of your own.

7.1 Lists

We introduced lists in Program 2.12. In this game, we create an empty list `[]` and use a loop to fill it with alien Actors.

We again use loops to draw all the aliens and move all the aliens in the list. When the mouse is clicked we add a new alien to the list.

Program 7.1: Lists are useful in games!

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

aliens = []
for i in range(0,10):
    aliens.append(Sprite('alien.png', (i*30, i*30)))

def draw():
    for alien in aliens:
        alien.draw()

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

def on_mouse_pressed(pos, button):
    aliens.append(Sprite('alien.png', pos))

run()
```

Advanced

Go back to a previous game (e.g. Program 5.1) and add a list of bullets that move up the screen. When the player presses the spacebar to shoot, add a new bullet to the list.

7.2 Timed callbacks

So far we have been using certain functions as *callbacks*: `update()`, `draw()` etc. These are called automatically for us by RLZero every frame. However, you can also create your own callbacks, and ask RLZero to call them back for you at a certain time.

Warning: Using a large number of callbacks can be confusing for the programmer. Especially you should avoid creating a callback from inside a callback function.

Program 7.2: Callbacks

```
from rlzero import *
import random

aliens = []

def add_alien():
    aliens.append(
        Sprite("alien", (random.randint(0,500), random.randint(0,500)))
    )

# call add_alien once, 0.5 seconds from now
schedule_once(add_alien, 0.5)

# call add_alien over and over again, ever 2 seconds
schedule_repeat(add_alien, 2.0)

def draw():
    for alien in aliens:
        alien.draw()

run()
```

Exercise

Make the aliens appear more often.

Advanced

Use `len(aliens)` to print how many aliens there are

Advanced

When there are too many aliens, stop adding them using this code:

```
schedule_cancel(add_alien)
```

7.3 Callbacks for animation

Instead of using the *Animation* class, in this program we define our own function for animation and then ask RLzero to *call it back* every 0.2 seconds. Most of this code is from Program 5.1.

Program 7.3: Animation via callbacks

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

alan = Sprite('alien.png')
alan.pos = (200, 200)

def draw():
    alan.draw()

def update():
    if keyboard.right:
        alan.x = alan.x + 2
    elif keyboard.left:
        alan.x = alan.x - 2
    if keyboard.space:
        schedule_cancel(animateAlien)

images = ["alien_hurt.png", "alien.png"]
image_counter = 0

def animateAlien():
    global image_counter
    alan.image_file = images[image_counter % len(images)]
    image_counter += 1

schedule_repeat(animateAlien, 0.2)

run()
```

Exercise

Make the alien animate more quickly.

Advanced

Add another image to the list of images.

Advanced

Draw your own animation, e.g. a man walking left which plays when the left key is pressed

7.4 Simple physics

Here we draw a ball and move it, like we did in Program 4.3. But instead of moving it by a fixed number of pixels, we store the number of pixels to move in variables, `vx` and `vy`. These are *velocities*, i.e. speed in a fixed direction. `vx` is the speed in the horizontal direction and `vy` is the speed in the vertical direction. This allow us to change the velocity. Here we reverse the velocity when the ball hits the edge of the screen.

Program 7.4: Simple physics: velocity

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

ball = Rectangle(200, 400, 20, 20)
vx = 1
vy = 1

def draw():
    draw_rectangle_rec(ball, RED)

def update():
    global vx, vy
    ball.x += vx
    ball.y += vy
    if ball.x > WIDTH or ball.x < 0:
        vx = -vx
    if ball.y > HEIGHT or ball.y < 0:
        vy = -vy

run()
```

Advanced

Make the ball move faster by increasing its velocity each time it hits the sides.

7.5 Bat and ball game

Pong is the classic bat and ball game.

Program 7.5: Pong

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

ball = Rectangle(150, 400, 20, 20)
bat = Rectangle(200, 480, 60, 20)
vx = 4
vy = 4

def draw():
    draw_rectangle_rec(ball, RED)
    draw_rectangle_rec(bat, WHITE)

def update():
    global vx, vy
    ball.x += vx
    ball.y += vy
    if ball.x > WIDTH or ball.x < 0:
        vx = -vx
    if check_collision_recs(bat, ball) or ball.y < 0:
        vy = -vy
    if ball.y > HEIGHT:
        exit()
    if(keyboard.right):
        bat.x += 2
    elif(keyboard.left):
        bat.x -= 2

run()

```

Exercise

Make the ball move more quickly.

Advanced

Add another bat at the top of the screen for player 2.

Advanced

Add bricks (Rectangles) that disappear when the ball hits them.

7.6 Timer

The `update()` and `draw()` functions are called by RLZero many times every second. Each time `draw()` is called we say it draws one *frame*. The exact number of frames per second is called the *framerate* and it will vary from one computer to another. Therefore counting frames is not the most reliable way of keeping time.

Fortunately RLZero can tell us exactly how much many seconds have passed since the last frame in a parameter to our update function. We use this *delta time* to keep a timer.

Program 7.6: Timer

```
from rlzero import *

timer = 0

def update(dt):
    global timer
    timer += dt


def draw():
    draw_text(f"Time passed: {timer}", 0, 0, 20, RED)
    if timer > 5:
        draw_text("Time's up!", 50, 50, 40, WHITE)

run()
```

Exercise

Make the timer count down, not up.

Advanced

Add a timer to one of your other games.

Advanced

Add a timer to Program 7.1 that deletes one of the aliens when the timer runs out, then starts the timer again.

CHAPTER

EIGHT

TUTORIAL: CHASE GAME

In this chapter we will build a complete chase game together, step by step. The Python we will use is very simple: just conditionals and loops.

The techniques here should be familiar to you because we used them in Program 4.5, Program 4.6, Program 5.1 and Program 5.2

Now we will show you how to put them all together in one program.

8.1 Moving Actor over a background

We must create two image files for our game. You can use a program such as Krita[1] to draw them and save them in the `mu_code/images` folder (accessible with the `images` button in Mu). One is the player, called `player.png`. It should be small, about 64×64 pixels. Ideally it should have a transparent background.

The other is the background for the game itself. It can look like whatever you want, but it must be the same size as the game window, which will be 600×600 pixels.

[1] https://krita.org

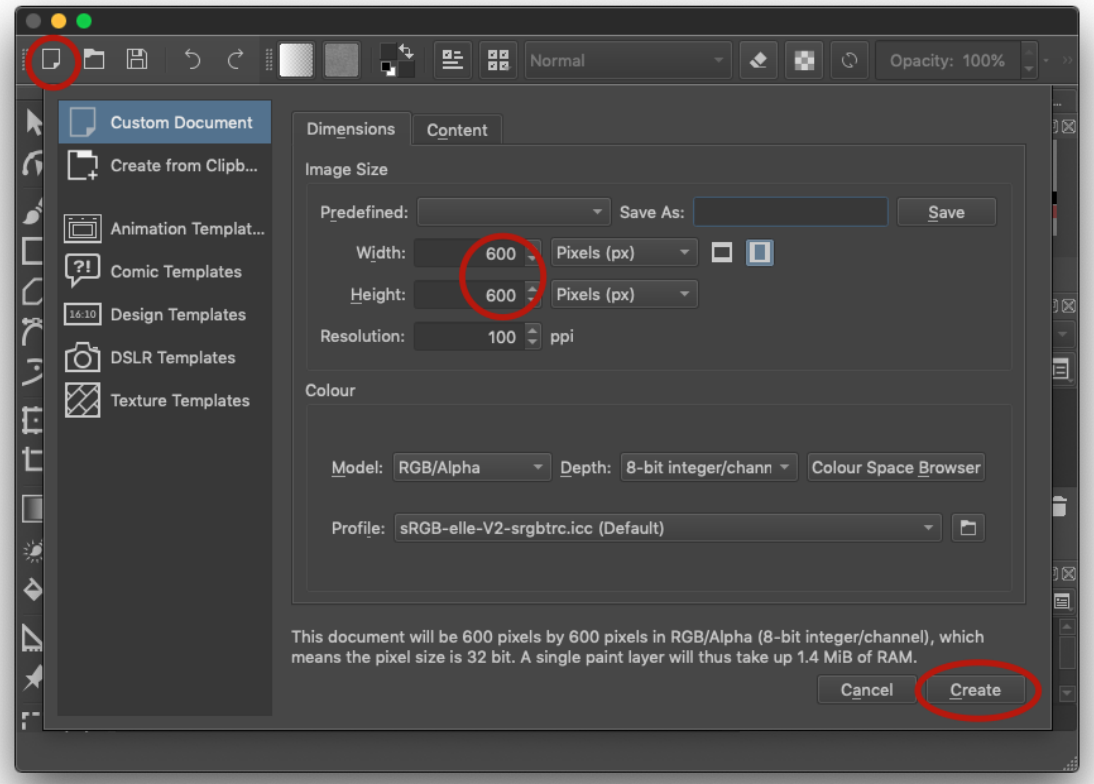


Fig. 8.1: New image in Krita, 600×600 pixels

Program 8.1: Chase game

```
WIDTH = 600
HEIGHT = 600

background = Actor("background")
player = Actor("player")
player.x = 200
player.y = 200

def draw():
    screen.clear()
    background.draw()
    player.draw()

def update():
    if keyboard.right:
        player.x = player.x + 4
    if keyboard.left:
        player.x = player.x - 4
    if keyboard.down:
        player.y = player.y + 4
```

(continues on next page)

(continued from previous page)

```
21      if keyboard.up:
22          player.y = player.y - 4
23
24
25
```

Exercise

Run the program and move the Actor around with the keys.

8.2 Screen wrap-around

One problem you will soon find with the program is that you can move off the edge of the screen and get lost. One way to solve this would be to stop movement at the screen edges. Instead we going to make the player teleport to the opposite edge when he leaves the screen. Add this code to the end of the program, and make sure it is indented so that it becomes part of the `update()` function.

```
if player.x > WIDTH:
    player.x = 0
if player.x < 0:
    player.x = WIDTH
if player.y < 0:
    player.y = HEIGHT
if player.y > HEIGHT:
    player.y = 0
```

Exercise

Change the code so that the player stops at the edges rather than wraps-around.

8.3 Enemy chases the player

Let's add an enemy to chase the player. At the top of the program, create a variable to store the enemy Actor:

```
enemy = Actor("alien")
```

At the end of the `draw()` function (but still indented so part of the function), draw the enemy. Remember there is only ever one single `draw()` function. No program has two. All drawing goes inside this function.

Here is the complete function including the new line at the end:

```
def draw():
    screen.clear()
    background.draw()
    player.draw()
    enemy.draw()
```

At the end of the `update()` function (but still indented so part of the function), add these lines to move the enemy:

```
if enemy.x < player.x:
    enemy.x = enemy.x + 1
if enemy.x > player.x:
    enemy.x = enemy.x - 1
if enemy.y < player.y:
    enemy.y = enemy.y + 1
if enemy.y > player.y:
    enemy.y = enemy.y - 1
if player.colliderect(enemy):
    exit()
```

Exercise

Run the program verify the enemy chases the player.

Exercise

Make the enemy faster so the game is more difficult.

Exercise

Create an image file *enemy.png* and save it in the *images* folder. Change the code so it loads *"enemy"* instead of *"alien"*.

8.4 Collecting items

Create a small image file `coin.png` and save it in the `images` folder. It should look like a coin or something else you would like to collect. We will also need a variable to store the *score*, i.e. number of coins collected.

```
coin = Actor("coin", pos=(300,300))
score = 0
```

At the end of the `draw()` function (but still indented so part of the function), draw the coin. Remember there is only ever one single `draw()` function. No program has two. All drawing goes inside this function.

Here is the complete function including the new line at the end:

```
def draw():
    screen.clear()
    background.draw()
    player.draw()
    enemy.draw()
    coin.draw()
```

At the end of the `update()` function (but still indented so part of the function), add these lines to move the coin when it is collected:

```
if coin.colliderect(player):
    coin.x = random.randint(0, WIDTH)
    coin.y = random.randint(0, HEIGHT)
    score = score + 1
    print("Score:", score)
```

Exercise

Run the program and collect a coin. What happens?

```
NameError: name 'random' is not defined
```

This happens because we are using a function `randint()` to get a random number. This function is not build-in to Python; it is part of the *random* library. So at the top of the program, add the first line:

```
import random
```

Exercise

Run the program again and collect a coin. Does it work now?

No!

```
UnboundLocalError: local variable 'score' referenced before assignment
```

You will get an error because `score` is a global variable and we are trying to modify it inside the `update()` function. Therefore at the **top** of the `update()` function, add this line to declare to Python our intention to modify a global variable:

```
global score
```

It must be the **first** line in the function and it must be **indented**. The lines surrounding it should look like this:

```
def update():
    global score
    if keyboard.right:
```

Exercise

Run the program again and verify it works!

8.5 Player 2

We can make any game into a two player game. At the top of the program, create a variable to store the Actor for the second player:

```
player2 = Actor("alien")
```

At the end of the `draw()` function (but still indented so part of the function), draw the enemy. Here is the complete function with the new line at the end:

```
def draw():
    screen.clear()
    background.draw()
    player.draw()
    enemy.draw()
    coin.draw()
    player2.draw()
```

At the end of the `update()` function (but still indented so part of the function), add these lines to move the second player:

```
if keyboard.d:
    player2.x = player2.x + 4
if keyboard.a:
    player2.x = player2.x - 4
if keyboard.s:
    player2.y = player2.y + 4
if keyboard.w:
    player2.y = player2.y - 4
if player.colliderect(player2):
    exit()
```

Advanced

Create a variable *score2* and store the score for player two, i.e. it goes up when he collides with a coin.

8.6 Showing the score on the screen

At the end of the `draw()` function (but still indented so part of the function), draw a title on the screen:

```
screen.draw.text("My game", (200,0), color='red')
```

The `draw.text()` function is not like `print()` - it can only print strings of text, not numbers. Therefore we must convert the score into a string. Add these lines to the end of the `draw()` function:

```
score_string = str(score)
screen.draw.text(score_string, (0,0), color='green')
```

Exercise

Change the colour of the text.

Advanced

Display the word "Score: " before the score.

Advanced

When the *score* reaches 10, show a message on the screen to congratulate the player

8.7 Timer

Add a variable at the top of the program (but preferably after any `import` statements) to store the number of seconds of time remaining in the game:

```
time = 20
```

Pygame Zero calls our `update()` function many times per second. We can ask it to tell us how much time has passed by adding a *parameter* to the function, `delta`. We then subtract this from the remaining time. Modify `update()` so the first lines look like this:

```
def update(delta):
    global score, time
    time = time - delta
    if time <= 0:
        exit()
```

We can also display the time on the screen. At the end of the `draw()` function (but still indented so part of the function) add these lines:

```
time_string = str(time)
screen.draw.text(time_string, (50,0), color='green')
```

Exercise

Run the program. Could the displayed time be improved?

We don't need to see the decimal places in the time. Modify those lines to use the `round()` function, like this:

```
time_string = str(round(time))
screen.draw.text(time_string, (50,0), color='green')
```

8.8 Finished game

Here is the finished game with all the changes included:

Program 8.2: Finished chase game

```
from rlzero import *

import random


WIDTH = 600
HEIGHT = 600
DATA_DIR = "."

background = Sprite("background")
player = Sprite("player")
player.x = 200
player.y = 200
player.rotation_angle = 0
player.scale = 2

enemy = Sprite("alien")
player2 = Sprite("player")
coin = Sprite("alien", pos=(300,300))
score = 0
time = 20


def draw2d():
    clear()
    background.draw()
    player.draw()
    enemy.draw()
    player2.draw()
```

(continues on next page)

(continued from previous page)

```
    coin.draw()
    draw_text("My game", 200, 0, 20, RED)
    score_string = str(score)
    draw_text(score_string, 0,0, 20, GREEN)
    time_string = str(round(time))
    draw_text(time_string, 50,0, 20, GREEN)

def update(delta):
    global score, time
    time = time - delta
    if time <= 0:
        print("time out")
        exit()
    if keyboard.right:
        player.x = player.x + 4
    if keyboard.left:
        player.x = player.x - 4
    if keyboard.down:
        player.y = player.y + 4
    if keyboard.up:
        player.y = player.y - 4

    if player.x > WIDTH:
        player.x = 0
    if player.x < 0:
        player.x = WIDTH
    if player.y < 0:
        player.y = HEIGHT
    if player.y > HEIGHT:
        player.y = 0

    if enemy.x < player.x:
        enemy.x = enemy.x + 1
    if enemy.x > player.x:
        enemy.x = enemy.x - 1
    if enemy.y < player.y:
        enemy.y = enemy.y + 1
    if enemy.y > player.y:
        enemy.y = enemy.y - 1
    if player.colliderect(enemy):
        print("player 1 hit enemy")
        exit()

    if keyboard.d:
        player2.x = player2.x + 4
    if keyboard.a:
        player2.x = player2.x - 4
    if keyboard.s:
        player2.y = player2.y + 4
```

(continues on next page)

(continued from previous page)

```
    if keyboard.w:
        player2.y = player2.y - 4
    if player.colliderect(player2):
        print("player1 hit player 2")
        exit()

    if coin.colliderect(player):
        coin.x = random.randint(0, WIDTH)
        coin.y = random.randint(0, HEIGHT)
        score = score + 1
        print("Score:", score)


run()
```

8.9 Ideas for extension

There are many things you could add to this game.

- Add more enemies.
- Give the player three lives.
- Add music and sound effects.
- Create a powerup that makes the player move faster.
- Make the enemies more intelligent.
- Allow the player to kill the enemies.

CHAPTER

NINE

TUTORIAL: MAZE GAME

In this chapter we will build a maze game together, step by step. The Python we will use is quite simple: mostly just conditionals and loops. The technique of creating a tilemap is common in games and after seeing it here you should be able to incorporate it into your own projects.

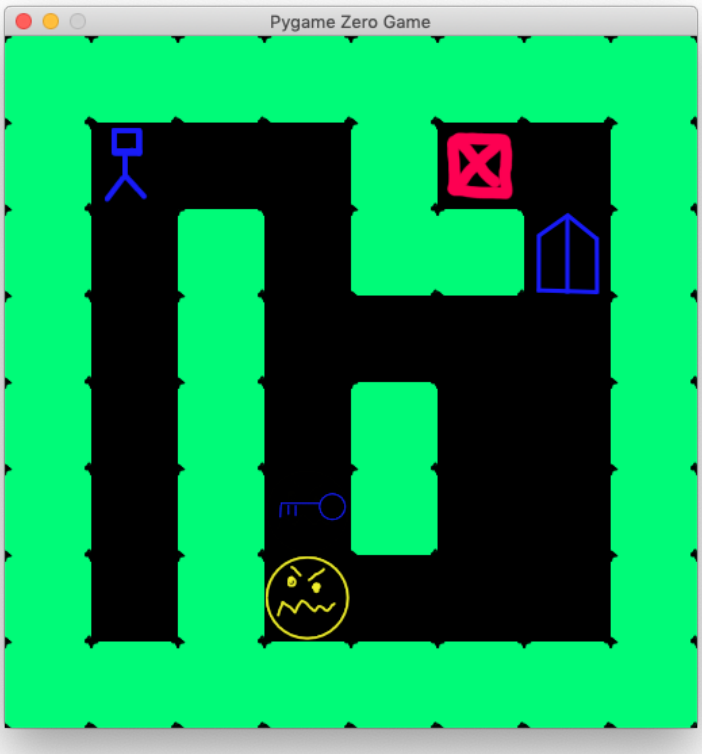


Fig. 9.1: Maze game

9.1 Tilemap

A tilemap uses a small number of images (the tiles) and draws them many times to build a much larger game level (the map). This saves you from creating a lot of artwork and makes it very easy to change the design of the level on a whim. Here we create a maze level.

We must create three image files for the tiles: `empty.png`, `wall.png` and `goal.png` and save them in the `mu_code/images` folder (accessible with the `images` button in Mu).

They must each be 64×64 pixels. For the player we will use the built in `alien` image.

Program 9.1: Drawing a tilemap

```
TILE_SIZE = 64
WIDTH = TILE_SIZE * 8
HEIGHT = TILE_SIZE * 8

tiles = ['empty', 'wall', 'goal']

maze = [
    [1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 2, 0, 1],
    [1, 0, 1, 0, 1, 1, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1]
]

player = Actor("alien", anchor=(0, 0), pos=(1 * TILE_SIZE, 1 * TILE_SIZE))

def draw():
    screen.clear()
    for row in range(len(maze)):
        for column in range(len(maze[row])):
            x = column * TILE_SIZE
            y = row * TILE_SIZE
            tile = tiles[maze[row][column]]
            screen.blit(tile, (x, y))
    player.draw()
```

The filenames of the tile images are stored in a *list*, `tiles`. The level design is stored in a list of lists, more commonly called a *two dimensional array*. There are 8 rows and 8 columns in the array. If you change the size of the array you will need to change the `WIDTH` and `HEIGHT` values too. The numbers in the `maze` array refers to elements in the `tiles` array. So `0` means `empty` and `1` means `wall`, etc.

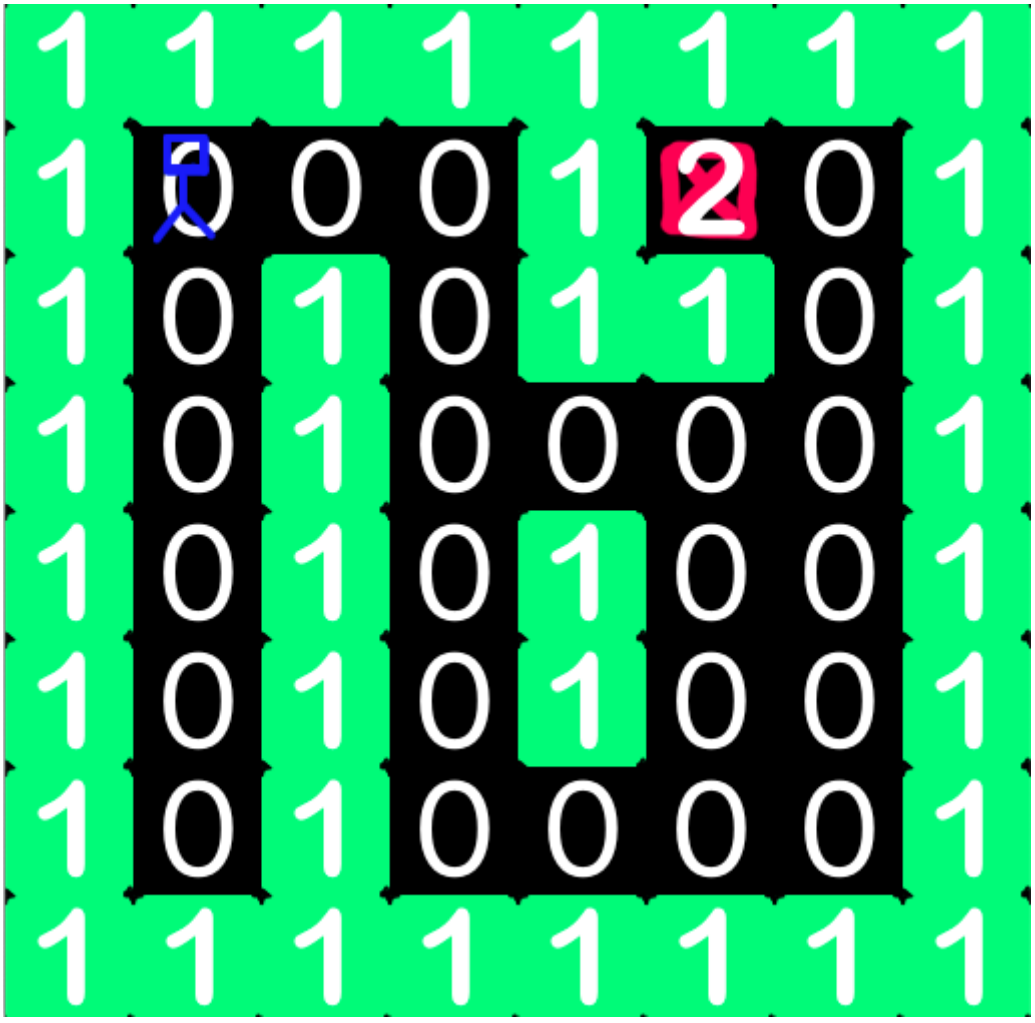


Fig. 9.2: Tile grid

To draw the maze we use a `for` loop within another `for` loop. The outer loop iterates over the rows and the inner loop iterates over the columns, i.e. the elements of the row.

Exercise

Create your own images *empty.png*, *wall.png* and *goal.png* and run the program.

Exercise

The player appears at the top left of the maze at row 1 column 1. Change the *pos* parameter so he appears at the bottom instead.

Exercise

Change the design of the maze by changing the numbers in the *maze* array.

Advanced

Make the maze bigger.

9.2 Moving the player

Add this code to the *end* of the program:

```
def on_key_down(key):
    row = int(player.y / TILE_SIZE)
    column = int(player.x / TILE_SIZE)
    if key == keys.UP:
        row = row - 1
    if key == keys.DOWN:
        row = row + 1
    if key == keys.LEFT:
        column = column - 1
    if key == keys.RIGHT:
        column = column + 1
    player.x = column * TILE_SIZE
    player.y = row * TILE_SIZE
```

This function will be called automatically by Pygame, like `draw()` and `update()`. However `on_key_down()` is not called every frame; it is only called when the player presses a key. The key that was pressed is passed to the function in the `key` *parameter*.

Exercise

Run the program and move the player. Are there any problems with the movement?

9.3 Restricting where the player can move

Delete the last two lines of the program and replace them with this modified version:

```
tile = tiles[maze[row][column]]
if tile == 'empty':
    player.x = column * TILE_SIZE
    player.y = row * TILE_SIZE
elif tile == 'goal':
    print("Well done")
    exit()
```

Exercise

Run the program and check that the player now *only* moves if the tile is empty.

Exercise

Check that the game ends when the player reaches the goal.

9.4 Animate the movement of the player

First, the alien Actor is bit too big. Draw a new image of size 64×64 pixels and save it as `player.png` in the `images` folder. In the program, change the line:

```
player = Actor("alien", anchor=(0, 0), pos=(1 * TILE_SIZE, 1 * TILE_SIZE))
```

to

```
player = Actor("player", anchor=(0, 0), pos=(1 * TILE_SIZE, 1 * TILE_SIZE))
```

Next, the movement of the Actor is sudden and jerky. Luckily Pygame includes a function to do smooth movement for us automatically. Find these lines of the program:

```
if tile == 'empty':
      player.x = column * TILE_SIZE
      player.y = row * TILE_SIZE
```

replace them with:

```
if tile == 'empty':
    x = column * TILE_SIZE
    y = row * TILE_SIZE
    animate(player, duration=0.1, pos=(x, y))
```

Exercise

Verify the player image has changed and moves smoothly.

9.5 Create an enemy

We will create a simple enemy that moves up and down. Add this code near the top just *above* the `draw()` function.

```
enemy = Actor("enemy", anchor=(0, 0), pos=(3 * TILE_SIZE, 6 * TILE_SIZE))
enemy.yv = -1
```

To make the enemy visible, add this line at the end of the `draw()` function, after the player is drawn:

```
enemy.draw()
```

`enemy.yv` is the velocity in the y-axis direction (up and down). Add these lines to end of the program (still inside the `on_key_down()` function) to make the enemy move and reverse velocity when it hits a wall.

```
# enemy movement
row = int(enemy.y / TILE_SIZE)
column = int(enemy.x / TILE_SIZE)
row = row + enemy.yv
tile = tiles[maze[row][column]]
if not tile == 'wall':
    x = column * TILE_SIZE
    y = row * TILE_SIZE
    animate(enemy, duration=0.1, pos=(x, y))
else:
    enemy.yv = enemy.yv * -1
if enemy.colliderect(player):
    print("You died")
    exit()
```

Exercise

Verify that the enemy moves up and down and kills the player.

Advanced

Make another enemy that moves horizontally (left and right).

Advanced

The collision detection is quite lenient (i.e. buggy) because it only tests for collisions between the enemy and player when a key is pressed. Define a new function called *update()* and move the collisions detection there so that is called every frame.

9.6 A locked door and a key

We will add two new tiles to the game. Draw images `door.png` and `key.png` and save them in `images` folder.

Find the `tiles` list near the top and **change** it to include the new images, and modify the `maze` with some number 3s and 4s where you want to new tiles to appear. Mine looks like this:

```
tiles = ['empty', 'wall', 'goal', 'door', 'key']
unlock = 0

maze = [
    [1, 1, 1, 1, 1, 1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
    [1, 0, 0, 0, 1, 2, 0, 1],
    [1, 0, 1, 0, 1, 1, 3, 1],
    [1, 0, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 4, 1, 0, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1]
]
```

At the top of the program, create a new variable to store the number of keys the player is carrying:

```
unlock = 0
```

Find the `if` statement where we test for `goal`:

```
if tile == 'goal':
     print("Well done")
     exit()
```

Modify it like this to also test for the key and door tiles. Since we are modifying a *global* variable inside a function we must declare it.

```
global unlock
if tile == 'goal':
    print("Well done")
    exit()
elif tile == 'key':
    unlock = unlock + 1
    maze[row][column] = 0 # 0 is 'empty' tile
elif tile == 'door' and unlock > 0:
    unlock = unlock - 1
    maze[row][column] = 0 # 0 is 'empty' tile
```

9.7 Finished game

Here is the finished game with all the changes included:

Program 9.2: Finished maze game

```
1  TILE_SIZE = 64
2  WIDTH = TILE_SIZE * 8
3  HEIGHT = TILE_SIZE * 8
4
5  tiles = ['empty', 'wall', 'goal', 'door', 'key']
6  unlock = 0
7
8  maze = [
9      [1, 1, 1, 1, 1, 1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
    [1, 0, 0, 0, 1, 2, 0, 1],
    [1, 0, 1, 0, 1, 1, 3, 1],
    [1, 0, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 4, 1, 0, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1]
]

player = Actor("player", anchor=(0, 0), pos=(1 * TILE_SIZE, 1 * TILE_SIZE))
enemy = Actor("enemy", anchor=(0, 0), pos=(3 * TILE_SIZE, 6 * TILE_SIZE))
enemy.yv = -1

def draw():
    screen.clear()
    for row in range(len(maze)):
        for column in range(len(maze[row])):
            x = column * TILE_SIZE
            y = row * TILE_SIZE
            tile = tiles[maze[row][column]]
            screen.blit(tile, (x, y))
    player.draw()
    enemy.draw()

def on_key_down(key):
    # player movement
    row = int(player.y / TILE_SIZE)
    column = int(player.x / TILE_SIZE)
    if key == keys.UP:
        row = row - 1
    if key == keys.DOWN:
        row = row + 1
    if key == keys.LEFT:
        column = column - 1
    if key == keys.RIGHT:
        column = column + 1
    tile = tiles[maze[row][column]]
    if tile == 'empty':
        x = column * TILE_SIZE
        y = row * TILE_SIZE
        animate(player, duration=0.1, pos=(x, y))
    global unlock
    if tile == 'goal':
        print("Well done")
        exit()
    elif tile == 'key':
        unlock = unlock + 1
        maze[row][column] = 0 # 0 is 'empty' tile
    elif tile == 'door' and unlock > 0:
```

(continues on next page)

(continued from previous page)

```
        unlock = unlock - 1
        maze[row][column] = 0 # 0 is 'empty' tile

    # enemy movement
    row = int(enemy.y / TILE_SIZE)
    column = int(enemy.x / TILE_SIZE)
    row = row + enemy.yv
    tile = tiles[maze[row][column]]
    if not tile == 'wall':
        x = column * TILE_SIZE
        y = row * TILE_SIZE
        animate(enemy, duration=0.1, pos=(x, y))
    else:
        enemy.yv = enemy.yv * -1
    if enemy.colliderect(player):
        print("You died")
        exit()
```

9.8 Ideas for extension

However that is not the end! There are many things you could add to this game.

- Show the player score.
- Coins that the player collects to increase score.
- Trap tiles that are difficult to see and kill the player.
- Treasure chest that is unlocked with the key and increases score.
- Instead of ending the game, give the player 3 lives.
- Add more types of tile to the map: water, rock, brick, etc.
- Change the player image depending on the direction they are moving.

CHAPTER

TEN

TUTORIAL: SHOOTING GAME

In this chapter we will build a shooting game together, step by step. The Python we will use is: conditionals, loops, lists and functions.

10.1 Step 1: Decide what Sprites you will need

Our game will need these Sprites, so **we must create images for all of them and save them** as `.png` files in the program folder.

variable name	image file name	image size
player	player.png	64x64
background	background.png	600x800
enemies (list)	enemy.png	64x64
bullets (list)	bullet.png	16x16
bombs (list)	bomb.png	16x16

The `player` and `background` variables will contain Sprites. The others are lists which we initialize to the empty list `[]`. Sprites will be appended to the lists later.

Program 10.1: Shooter game part 1 of 4

```
from rlzero import *
import random

WIDTH = 600
HEIGHT = 800
MAX_BULLETS = 3

level = 1
lives = 3
score = 0

background = Sprite("background.png")
player = Sprite("player.png", (200, 730))
enemies = []
bullets = []
bombs = []
```

10.2 Step 2: Draw your Sprites

Every RLZero game needs a `draw()` function, and it should draw all the Sprites we created above.

Program 10.2: Shooter game part 2 of 4

```
def draw():
    background.draw()
    player.draw()
    for enemy in enemies:
        enemy.draw()
    for bullet in bullets:
        bullet.draw()
    for bomb in bombs:
        bomb.draw()
    draw_gui()
```

10.3 Step 3: Move your Sprites

Every RLZero game needs an `update()` function to move the Sprites, check for collisions, etc.

Program 10.3: Shooter game part 3 of 4

```
def update(delta):
    move_player()
    move_bullets()
    move_enemies()
    create_bombs()
    move_bombs()
    check_for_end_of_level()
```

10.4 Step 4: Define your functions

Python cannot call a function that has not yet been defined. Therefore we must at least provide empty, dummy versions of our functions that don't do anything so we can fill them in later. However Python cannot define a completely empty function - it must contain at least one line. Therefore we use the `pass` keyword to create a line that doesn't do anything.

Program 10.4: Shooter game part 4 of 4

```

def move_player():
    pass

def move_enemies():
    pass

def move_bullets():
```

(continues on next page)

(continued from previous page)

```
    pass

def create_bombs():
    pass

def move_bombs():
    pass

def check_for_end_of_level():
    pass

def draw_gui():
    pass

run()
```

Exercise

Create the png image files (`player.png, background.png, bullet.png, bomb.png, enemy.png`). Type in program Program 10.1, Program 10.2 and Program 10.3 and Program 10.4 into a single file. Verify the game now runs and you can see the player at the bottom of the screen. (He can't move yet.)

10.5 Create enemies

Add this new function to the end of the program, and then call it immediately. It uses a loop within a loop to create enemy Sprites and put them in the `enemies` list. The reason we put this in a function is we will need to call it again at the start of each level.

```
def create_enemies():
    for x in range(0, 600, 60):
        for y in range(0, 200, 60):
            enemy = Sprite("enemy.png", (x, y))
            enemy.vx = level * 2
            enemies.append(enemy)


create_enemies()
```

10.6 Move the player

Replace the `move_player()` dummy function definition with this. Remember **there can only be one function with a given name**. *There cannot be two ``move_player()`` function definitions.*

```
def move_player():
    if keyboard.right:
        player.x = player.x + 5
    if keyboard.left:
        player.x = player.x - 5
    if player.x > WIDTH:
        player.x = WIDTH
    if player.x < 0:
        player.x = 0
```

10.7 Move the enemies

Replace the `move_enemies()` dummy function definition with this:

```
def move_enemies():
    global score
    for enemy in enemies:
        enemy.x = enemy.x + enemy.vx
        if enemy.x > WIDTH or enemy.x < 0:
            enemy.vx = -enemy.vx
            enemy.y += 60
        for bullet in bullets:
            if bullet.colliderect(enemy):
                enemies.remove(enemy)
                bullets.remove(bullet)
                score = score + 1
        if enemy.colliderect(player):
            exit()
```

10.8 Draw text on the screen

Replace the `draw_gui()` dummy function definition with this:

```
def draw_gui():
    draw_text("Level " + str(level), 0, 0, 20, RED)
    draw_text("Score " + str(score), 100, 0, 20, RED)
    draw_text("Lives " + str(lives), 200, 0, 20, RED)
```

10.9 Player bullets

Add this new function to the end of the program. Pygame will call it for us automatically when a key is pressed.

```
def on_key_pressed(key):
    if key == keys.space and len(bullets) < MAX_BULLETS:
        bullet = Sprite("bullet.png", pos=(player.x, player.y))
        bullets.append(bullet))
```

Replace the `move_bullets()` dummy function definition with this:

```
def move_bullets():
    for bullet in bullets:
        bullet.y = bullet.y - 6
        if bullet.y < 0:
            bullets.remove(bullet)
```

10.10 Enemy bombs

Replace the `create_bombs()` dummy function definition with this:

```
def create_bombs():
    if random.randint(0, 100 - level * 6) == 0:
        enemy = random.choice(enemies)
        bomb = Sprint("bomb.png", enemy.pos)
        bombs.append(bomb)
```

Replace the `move_bombs()` dummy function definition with this:

```
def move_bombs():
    global lives
    for bomb in bombs:
        bomb.y = bomb.y + 10
        if bomb.colliderect(player):
            bombs.remove(bomb)
            lives = lives - 1
            if lives == 0:
                exit()
```

10.11 Check for end of level

Replace the `check_for_end_of_level()` dummy function definition with this:

```
def check_for_end_of_level():
    global level
    if len(enemies) == 0:
        level = level + 1
        create_enemies()
```

10.12 Ideas for extension

- Draw an explosion image and create an explosion Sprite every time an alien dies.
- Make the explosion Sprite disappear after a short time.
- Draw several explosion images, put them in a list and make the Sprite animate displaying each of them in order.
- The player can only fire 3 bullets at a time. Create a powerup that allows him to fire additional bullets.
- Add music and sound effects.

CHAPTER

ELEVEN

TUTORIAL: RACE GAME

In this chapter we will build a racing game together, step by step. The Python we will use is: conditionals, loops, lists, functions and tuples. We will show use of velocity, high score and a title screen.

11.1 Basic game

Similar to the shooter game, we will begin with a complete program listing but with empty bodies for some of the functions that we will fill in later. (Python will not run a program with completely empty functions, so they just contain `pass` to indicate to Python they do nothing.)

Fig. 11.1: Race game

Like the shooter program, we begin we three things:

1. Definitions of global variables.
2. A `draw()` function.
3. An `update()` function.

These functions now check a boolean variable `playing`. If `False` then instead of drawing/updating the game we show the title screen.

The only really complicated part of this program is how we store the shape of the tunnel the player is racing down. `lines` is a list of *tuples*. A tuple is like a list but *cannot be modified* and can be *unpacked* into separate variables. Each tuple will represent one horizontal line of the screen. It will have three values, `x`, `x2` and `color`, representing the position of the left wall, the gap between the left wall and the right wall and the colour of the wall.

Program 11.1: Basic skeleton of race game

```
from rlzero import *
import random
import math

WIDTH = 600
HEIGHT = 800

player = Sprite("alien.png", (300, 750))
player.vx = 0   # horizontal velocity
player.vy = 1   # vertical velocity

lines = []          # list of tuples of horizontal lines of walls
wall_gradient = -3  # steepness of wall
left_wall_x = 200   # x-coordinate of wall
distance = 0        # how far player has travelled
time = 15           # time left until game ends
playing = False     # True when in game, False when on title screen
best_distance = 0   # remember the highest distance scored

def draw():
    if playing: # we are in game
        for i in range(0, len(lines)): # draw the walls
            x, x2, color = lines[i]
            draw_line(0, i, int(x), i, color)
            draw_line(int(x + x2), i, WIDTH, i, color)
        player.draw()
    else:   # we are on title screen
        draw_text("PRESS SPACE TO START", 150, 300, 30, GREEN)
        draw_text("BEST DISTANCE: "+str(int(best_distance / 10)),
                  170, 400,  30, GREEN)
    draw_text("SPEED: " + str(int(player.vy)),
              0, 0, 30, GREEN)
    draw_text("DISTANCE: " + str(int(distance / 10)),
              200, 0, 30, GREEN)
    draw_text("TIME: " + str(int(time)),
              480, 0, 30, GREEN)


def update(delta):
    global playing, distance, time
    if playing:
        wall_collisions()
        scroll_walls()
        generate_lines()
        player_input()
        timer(delta)
    elif keyboard.space:
        playing = True
```

(continues on next page)

(continued from previous page)

```
        distance = 0
        time = 10


def player_input():
    pass

def generate_lines():
    pass

generate_lines()

def scroll_walls():
    pass

def wall_collisions():
    pass

def timer(delta):
    pass

def on_mouse_move(pos):
    pass

run()
```

Exercise

Run the program. Verify it has a title screen and you can start the game and see the player. (That is all it will do until we fill in the remaining functions.)

11.2 Player input

Replace the definiton of `player_input()` with this:

```
def player_input():
    if keyboard.up:
        player.vy += 0.1
    if keyboard.down:
        player.vy -= 0.1
        if player.vy < 1:
            player.vy = 1
    if keyboard.right:
        player.vx += 0.4
    if keyboard.left:
        player.vx -= 0.4
    player.x += player.vx
```

Exercise

Run the program. Verify the player can move left and right and has momentum. Try adjusting the speed or making a limit so you can't go too fast.

11.3 Generate the walls

We already have code to draw the walls, but currently `lines` is empty so nothing gets drawn. Replace the function `generate_lines()` with this. Note that we immediately call `generate_lines()` after defining it to generate the walls for the start of the game.

```
def generate_lines():
    global wall_gradient, left_wall_x
    gap_width = 300 + math.sin(distance / 3000) * 100
    while len(lines) < HEIGHT:
        pretty_colour = (255, 0, 0)
        lines.insert(0, (left_wall_x, gap_width, pretty_colour))
        left_wall_x += wall_gradient
        if left_wall_x < 0:
            left_wall_x = 0
            wall_gradient = random.random() * 2 + 0.1
        elif left_wall_x + gap_width > WIDTH:
            left_wall_x = WIDTH - gap_width
            wall_gradient = -random.random() * 2 - 0.1

generate_lines()
```

Advanced

Run the program. Change the colour of the walls from red to green.

11.4 Make the walls colourful

Modify the line that sets the colour of the generated line to this:

```
pretty_colour = (255, min(left_wall_x, 255), min(time * 20, 255))
```

11.5 Scrolling

Modify the `scroll_walls()` function so it removes lines from the bottom of the screen according to the player's vertical velocity.

```
def scroll_walls():
    global distance
    for i in range(0, int(player.vy)):
        lines.pop()
        distance += 1
```

Exercise

Modify *scroll_walls()* as above and check that the player can now accelerate forward.

Advanced

Change the amount of the forward acceleration to make the game faster or slower.

11.6 Wall collisions

Currently the player can move through the walls - we don't want to allow this. Also we want the player to lose all their velocity each time they collide as a penalty.

```
def wall_collisions():
    a, b, c = lines[-1]
    if player.x < a:
        player.x += 5
        player.vx = player.vx * -0.5
        player.vy = 0
    if player.x + 50 > a + b:
        player.x -= 5
        player.vx = player.vx * -0.5
        player.vy = 0
```

Exercise

Modify *wall_collisions()* as above and check that the player now bounces off the walls.

Advanced

Make the collision more bouncy, i.e. the player bounces further when he hits the wall.

11.7 Timer

Currently the player has infinite time. We want decrease the `time` variable by how much time has passed and end the game when time runs out.

```
def timer(delta):
    global time, playing, best_distance
    time -= delta
    if time < 0:
        playing = False
        if distance > best_distance:
            best_distance = distance
```

Exercise

Modify the *timer()* function as above. Verify the game ends after 15 seconds.

Exercise

Make the game last for 30 seconds.

11.8 Mouse movement

The game is easier but perhaps more fun if you can play it with mouse. Pygame will call this function for us automatically.

```
def on_mouse_move(pos):
    x, y = pos
    player.x = x
    player.vy = (HEIGHT - y) / 20
```

Exercise

Modify the *on_mouse_move()* function as above. How does the player accelerate using the mouse?

11.9 Ideas for extension

- Draw a new image for the player. Make the Actor show a different image depending on if the player is steering left or right.
- Give the player a goal distance that must be reached. If the player reaches this distance he gets extra time added to allow him to continue.
- Add sound effects and music.
- If you have a larger screen, make the game window taller (and make sure the alien appears at the bottom still).

CHAPTER

TWELVE

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.

12.1 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[8] 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 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.)

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

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
            draw_pixel(x, y, color)
```

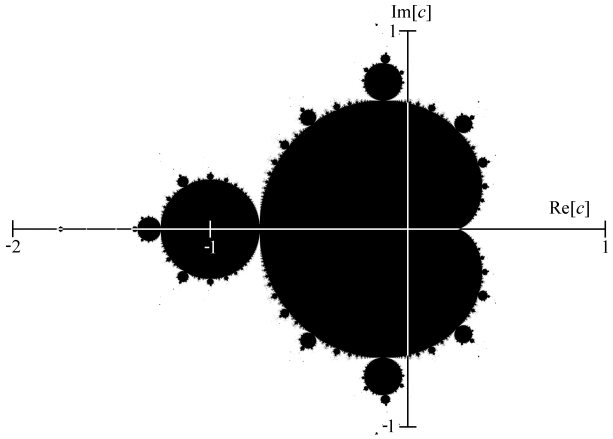


Fig. 12.1: Complex plain

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
```

(continues on next page)

(continued from previous page)

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

Here is the complete program to type in and run:

Program 12.1: 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()
```

12.2 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)
            draw_pixel(x, y, color)
```

Modify your `draw2d()` function and run.

12.3 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 = color_from_hsv(hue, saturation, value)
            draw_pixel(x, y, color)
```

Modify your `draw2d()` function and run.

12.4 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** keys to zoom in and out, and hold the **cursor** keys to move around.

Program 12.2: 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
    if keyboard.space:
```

(continues on next page)

(continued from previous page)

```
        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 will be slow and you may have to **hold keys down** for a long time before anything happens! We will try to improve this next.

12.5 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)
            image_draw_pixel(image, x, y, color)


image = 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 = load_texture_from_image(image)
    set_texture_filter(texture, 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():
    update_texture(texture, image.data)
    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()
```

Complete program:

Program 12.3: 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)
10 IM_HEIGHT = (IM_END - IM_START)
```

(continues on next page)

(continued from previous page)

```
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
        plot_image()
```

(continues on next page)

(continued from previous page)

```
    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()
```

12.6 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():
    update_texture(texture, image.data)
    draw_texture_ex(texture, (0,0), 0, SCALE, WHITE)
```

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

12.7 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.

CHAPTER

THIRTEEN

ADVANCED TOPICS

13.1 Instructor note

This introduces object oriented programming, but I wouldn't attempt this with young students since it requires abstract thinking.

13.2 Classes

You've already been using class types provided by Pygame Zero, e.g. Rect and Actor. But if we want to store velocity as in Program 7.4 we find these classes do not include *vx* and *vy* variables inside them by default. We have to remember to add a *vx* and *vy* every time we create an Actor.

So let's create our own class, called *Sprite*, that is the same as Actor but with these variables included.

Program 13.1: Classes

```
from rlzero import *


WIDTH = 500
HEIGHT = 500

class MySprite(Sprite):
    vx = 1
    vy = 1

ball = MySprite('alien.png')

def draw():
    clear()
    ball.draw()

def update():
    ball.x += ball.vx
    ball.y += ball.vy
    if ball.x > WIDTH or ball.x < 0:
        ball.vx = -ball.vx
    if ball.y > HEIGHT or ball.y < 0:
```

(continues on next page)

(continued from previous page)

```
        ball.vy = -ball.vy


run()
```

13.3 Methods

Classes can contain functions (called *methods*) as well as variables. Methods are the best place to modify the class's variables.

Program 13.2: Class methods

```
from rlzero import *

WIDTH = 500
HEIGHT = 500

class MySprite(Sprite):
    vx = 1
    vy = 1

    def update(self):
        self.x += self.vx
        self.y += self.vy
        if self.x > WIDTH or self.x < 0:
            self.vx = -self.vx
        if self.y > HEIGHT or self.y < 0:
            self.vy = -self.vy

ball = MySprite("alien.png")

def draw():
    clear()
    ball.draw()

def update():
    ball.update()

run()
```

13.4 Joystick tester

This program demonstrates using joysticks and for loops, but is mainly included to help you test the input from your controllers.

(I don't suggest typing this one yourself.)

Program 13.3: Joystick tester

```
import pygame

def update():
    screen.clear()
    joystick_count = pygame.joystick.get_count()
    y = 0
    for i in range(joystick_count):
        joystick = pygame.joystick.Joystick(i)
        joystick.init()
        name = joystick.get_name()
        axes = joystick.get_numaxes()
        buttons = joystick.get_numbuttons()
        hats = joystick.get_numhats()
        screen.draw.text(
            "Joystick {} name: {} axes:{} buttons:{} hats:{}".format(
                i, name, axes, buttons, hats), (0, y))
        y += 14
        for i in range(axes):
            axis = joystick.get_axis(i)
            screen.draw.text("Axis {} value: {:>6.3f}".format(i, axis),
                             (20, y))
            y += 14
        for i in range(buttons):
            button = joystick.get_button(i)
            screen.draw.text("Button {:>2} value: {}".format(i, button),
                             (20, y))
            y += 14
        for i in range(hats):
            hat = joystick.get_hat(i)
            screen.draw.text("Hat {} value: {}".format(i, str(hat)),
                             (20, y))
            y += 14
```

13.5 Distributing your Pygame Zero games

This is often tricky to get working, but you can distribute your games to people who don't have Python or Mu installed. You can put them on a USB stick, or a website for people to download, or even on itch.io for people to buy.

1. Install the full version of python from www.python.org[9].
2. Edit your game source code (using Mu). We will assume your source is in a file `MY_GAME.py`. At the top of the file add the line:

   ```
   import pgzrun
   ```

   At the bottom of the file add the line:

   ```
   pgzrun.go()
   ```

   Save the file.

4. Open a command shell: Click start menu and type `cmd.exe`. You should see a prompt similar to this:

   ```
   C:\Users\YourName>
   ```

   This means you are in the user YourName home directory, with the `mu_code` sub-directory inside it. If you are in a different directory you will have to change it with the `cd` command.

5. Install Nuitka and PGZero. At the command prompt type:

   ```
   pip install nuitka pgzero
   ```

6. Create the executable. At the command prompt type this (all one long line):

   ```
   python -m nuitka --onefile --include-package-data=pgzero,pygame --include-
   ↪data-dir=mu_code=. --output-dir=Documents mu_code/MY_GAME.py
   ```

   Remember to replace `MY_GAME` with the actual name of the Python file. If Nuitka asks for confirmation, type `Yes` and press enter.

   This will generate a program called `MY_GAME.exe` in your `Documents` folder.

7. In Windows Explorer, double click the `MY_GAME.exe` icon in `Documents` to play your game. To distribute your game, copy this file.

[9] https://www.python.org/downloads/

CHAPTER

FOURTEEN

PYTHON IN MINECRAFT

Note: The Minecraft Python library is made by David Whale and Martin O'Hanlon. I highly recommend their book *Adventures in Minecraft* which contains a great deal more programs.

14.1 Setup

You will need to own Minecraft Java Edition[10] (not Bedrock edition).

If you already have a Mojang or Microsoft account, go to the Minecraft website and click login. If you haven't accessed it for years you may need to reset your password. If you already own Minecraft it will then tell you. If you don't have an account, create one, and buy Minecraft Java Edition.

14.1.1 Setup Java

You will need to download and install Java from adoptopenjdk.net[11].

I recommend the *latest* (currently **OpenJDK 16**) because it will be useful for other things, but if you have problems you can fall back to using **OpenJDK 8**. Choose **HotSpot** and click the button to download.

14.1.2 Setup Server

Download the *Adventures In Minecraft Starter Kit* from

https://adventuresinminecraft.github.io and unpack it in a folder on your desktop. There are videos on the site that explain how to set it up.

Use Notepad or SublimeText[12] to edit the file *Server/server.properties*. Change this setting

```
level-type=flat
```

to generate a flat world. (But it's up to you what sort of world you prefer!)

[10] https://www.minecraft.net/en-us/store/minecraft-java-edition
[11] https://adoptopenjdk.net
[12] https://www.sublimetext.com/

To run the server, double click the **StartServer** file. It will open a server console window, and ask you press space.

If you want to generate another world later you can change

```
level-name=world2
```

and then run the server again.

Once the server is running, to stop nighttime from happening I suggest you type this at the server console

```
gamerule doDaylightCycle false
time set day
```

If only we could do that in real life!

You must leave the server console window open at all times. When you want to quit, first type

```
stop
```

at the server console, so it saves your world.

Advanced Challenge

Setup a more modern Minecraft server, such as PaperMC[13].

Then try to get the Raspberryjuice plugin[14] to work with it so you can use it with Python. This is not easy; I couldn't do it!

14.1.3 Setup Minecraft

Run the Minecraft launcher you downloaded from minecraft.net. We will not be using the default which is the latest version of Minecraft. Instead we will be using **version 1.12**.

Run the **Minecraft Launcher**. Click **Installations** then **New Installation**, then select version **release 1.12**. Then click *Create*. Then click **Play** next to the new installation in the list.

Minecraft will run. Click **multiplayer**. Then **add server**. Enter the server address as

```
localhost
```

Click 'done'. Click on your server to connect to it.

[13] https://papermc.io/
[14] https://dev.bukkit.org/projects/raspberryjuice

14.1.4 Setup Mu

Mu is the Python editor we have already been using, so you probably already have it installed. However you need to make sure you have the latest version. You can download it from the links at the *top* of https://codewith.mu/en/download.

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

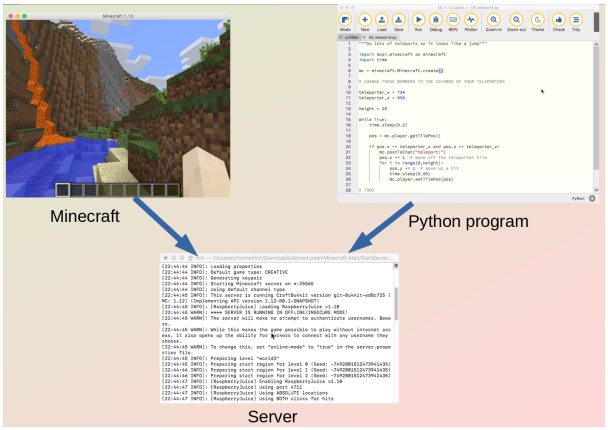
```
mcpi
```

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

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

```
pip3 install mcpi
```

14.1.5 Summary



You need to have the Minecraft server, Minecraft (the client) and Mu all running at the same time. It may be useful to arrange them in separate windows. Your Python program will talk to the server, and Minecraft will also talk to the server, allowing you to see the results of your program in Mincraft.

14.2 Hello Minecraft

This program tests you have a connection to the Minecraft server and displays a message on the client.

Program 14.1: Hello, Minecraft

```
from mcpi.minecraft import *
mc = Minecraft.create()
mc.postToChat("Hello Minecraft World")
```

14.3 Coordinates

This program gets the player's co-ordinates and prints them to the chat.

Program 14.2: Getting player coordinates

```
from mcpi.minecraft import *
import time

mc = Minecraft.create()

while True:     # loop will make sure your game runs forever
    time.sleep(1)
    pos = mc.player.getTilePos()
    print(pos)
    mc.postToChat(pos)
```

14.4 Changing the player's position

Find the coordinates of a location in your world, either by pressing F3 in the game, or running Program 14.2 Enter these coordinates in this program and run it to teleport to that location.

Program 14.3: Changing the player's position

```
from mcpi.minecraft import *

mc = Minecraft.create()

# change these to where you want to go
x = 10
y = 11
z = 12

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

14.5 Build a teleporter

Before you run this program, build two tiles in the game to be your teleporters and write down their co-ordinates.

Program 14.4: Teleporter

```
from mcpi.minecraft import *

mc = Minecraft.create()

# CHANGE THESE NUMBERS TO THE CO-ORDS OF YOUR TELEPORTERS

teleporter_x = 742
teleporter_z = 955

destination_x = 735
destination_z = 956

while True:
    # Get player position
    pos = mc.player.getTilePos()
    print(pos)

    # Check whether your player is standing on the teleport
    if pos.x == teleporter_x and pos.z == teleporter_z:
        mc.postToChat("teleport!")
        pos.x = destination_x
        pos.z = destination_z
        mc.player.setTilePos(pos)
```

Exercise

Add this line to the end of the program:

```
time.sleep(5)
```

Then add another line that teleports the player somewhere else.

14.6 Teleport player into the air

Program 14.5: Teleport player into the air

```
from mcpi.minecraft import *

mc = Minecraft.create()

# CHANGE THESE NUMBERS TO THE CO-ORDS OF YOUR TELEPORTERS

teleporter_x = 9
teleporter_z = 12

height = 30

while True:
    pos = mc.player.getTilePos()

    # Check whether your player is standing on the teleport
    if pos.x == teleporter_x and pos.z == teleporter_z:
        mc.postToChat("teleport!")
        pos.y += height  # up in the air!
        pos.x += 1  # move off the teleporter tile
        mc.player.setTilePos(pos)
```

14.7 Teleport jump

This program does a series of teleports in quick succession to give the effect of a jump.

Program 14.6: Teleport jump

```
from mcpi.minecraft import *
import time

mc = Minecraft.create()

# CHANGE THESE NUMBERS TO THE CO-ORDS OF YOUR TELEPORTERS

teleporter_x = 9
teleporter_z = 12

height = 20

while True:
    pos = mc.player.getTilePos()
```

(continues on next page)

(continued from previous page)

```
15
16      if pos.x == teleporter_x and pos.z == teleporter_z:
17          mc.postToChat("teleport!")
18          # move off the teleporter tile so we dont land on it again
19          pos.x += 1
20          for i in range(0, height):
21              pos.y += 1  # move up a bit
22              time.sleep(0.1) # short delay of 0.2 seconds
23              mc.player.setTilePos(pos)
```

Exercise

- Change the height of the jump.

Exercise

- Make the jump faster.

Exercise

- Move the player in X and Z directions as well as Y during the jump.

Advanced

Instead of checking if player is on a single teleporter tile, check if player is within a larger area. Use <, *and*, > operators.

14.8 Create a block

This program creates a block. Each type of block has it's own number, but if we import `mcpi.block` we can use names instead remembering numbers.

Program 14.7: Create a block

```
1  from mcpi.minecraft import *
2  from mcpi.block import *
3
4  mc = Minecraft.create()
5  pos = mc.player.getTilePos()
6  x = pos.x
7  y = pos.y
8  z = pos.z
```

(continues on next page)

(continued from previous page)

```
9  blocktype = 1
10 mc.setBlock(x, y, z, blocktype)
```

Exercise

Make the block appear a short distance from the player.

14.9 Types of block

AIR	BED	BEDROCK
BEDROCK_INVISIBLE	BOOKSHELF	BRICK_BLOCK
CACTUS	CHEST	CLAY
COAL_ORE	COBBLESTONE	COBWEB
CRAFTING_TABLE	DIAMOND_BLOCK	DIAMOND_ORE
DIRT	DOOR_IRON	DOOR_WOOD
FARMLAND	FENCE	FENCE_GATE
FIRE	FLOWER_CYAN	FLOWER_YELLOW
FURNACE_ACTIVE	FURNACE_INACTIVE	GLASS
GLASS_PANE	GLOWSTONE_BLOCK	GOLD_BLOCK
GOLD_ORE	GRASS	GRASS_TALL
GRAVEL	ICE	IRON_BLOCK
IRON_ORE	LADDER	
LAPIS_LAZULI_ORE	LAVA	LAVA_FLOWING
LAVA_STATIONARY	LEAVES	MELON
MOSS_STONE	MUSHROOM_BROWN	MUSHROOM_RED
OBSIDIAN	REDSTONE_ORE	SAND
SANDSTONE	SAPLING	SNOW
SNOW_BLOCK	STAIRS_COBBLESTONE	
STAIRS_WOOD	STONE	STONE_BRICK
STONE_SLAB	STONE_SLAB_DOUBLE	SUGAR_CANE
TNT	TORCH	WATER
WATER_FLOWING	WATER_STATIONARY	WOOD
WOOD_PLANKS	LAPIS_LAZULI_BLOCK	WOOL

14.10 Create a block inside a loop

This program creates a block over and over again in a loop. Move around to see it.

Program 14.8: Block loop

```
1 from mcpi.minecraft import *
2 from mcpi.block import *
3
4 mc = Minecraft.create()
```

(continues on next page)

(continued from previous page)

```
5
6  while True:
7      pos = mc.player.getTilePos()
8      x = pos.x
9      y = pos.y
10     z = pos.z
11     blocktype = WOOL
12     mc.setBlock(x, y, z, blocktype)
```

Exercise

Make the block appear one meter **below** the player's position.

Exercise

Change the block to something else, e.g. *ICE*

14.11 Create a tower of blocks

We will use a `for` loop to easily build a tower of blocks.

Program 14.9: Tower of blocks

```
1  from mcpi.minecraft import *
2
3  mc = Minecraft.create()
4  pos = mc.player.getTilePos()
5  x = pos.x + 3
6  y = pos.y
7  z = pos.z
8
9  for i in range(10):
10     mc.setBlock(x, y + i, z, 1)
```

Exercise

How high can you make the tower?

Exercise

Change the program to create *three* towers next to one another.

14.12 Clear space

The `setBlocks()` function lets us create a large cube of blocks. If we create blocks of type `AIR` this has the effect of removing all blocks! This is such a useful thing that we will need it in the future, therefore in this program we put it in its own *function*. Make sure to save the program as `clear_space.py` so you can `import` it into the next program.

Program 14.10: Clear space

```
from mcpi.minecraft import *
from mcpi.block import *

def clear_space(mc, size):
    pos = mc.player.getTilePos()
    mc.setBlocks(pos.x-size, pos.y, pos.z-size, pos.x+size, pos.y+size, pos.
↪z+size,
                 AIR)

mc = Minecraft.create()

clear_space(mc, 10)
```

14.13 Build a house

Make sure you have saved the previous Program 14.10 to the same directory before you run this program because we are going to `import` the function from `clear_space.py`. Save this program as `house.py`.

Program 14.11: A simple house

```
from mcpi.minecraft import *
from mcpi.block import *

# This MUST be the name you gave to your clear space program!
from clear_space import *

def make_house(mc, x, y, z, width, height, length):
    mc.setBlocks(x, y, z, x + width, y + height, z + length, STONE)

    # What happens if we make AIR inside the cube?
    mc.setBlocks(x + 1, y + 1, z + 1,
                 x + width - 2, y + height - 2, z + length - 2, AIR)

mc = Minecraft.create()
pos = mc.player.getPos()
x = pos.x
y = pos.y
z = pos.z

width = 10
```

(continues on next page)

(continued from previous page)

```
height = 50
length = 60

# Use the function from the other program
clear_space(mc, 10)
make_house(mc, x, y, z, width, height, length)
```

Exercise

Run the program and manually bash a hole in the wall to see what is inside and to give you a way to get into the building.

Exercise

Change the program so it *automatically* makes a hole for a door.

Exercise

Lower the floor in your house.

Exercise

Add some furniture, torches, windows.

Advanced

Make the windows get bigger if you increase the size of the house.

Exercise

Try filling a house with *LAVA*, or *WATER*, or *TNT* (Be careful with *TNT*, too much will crash your computer!)

14.14 Build a street of houses

Make sure you have saved the previous Program 14.11 to the same directory before you run this program because we are going to `import` the function from `house.py`.

Program 14.12: A street of houses

```
from mcpi.minecraft import *
from mcpi.block import *

# This MUST be the name you gave to your clear space program!
from clear_space import *
# This MUST be the name you gave to your house program!
from house import *

mc = Minecraft.create()
pos = mc.player.getTilePos()

x = pos.x
y = pos.y
z = pos.z

width = 10
height = 5
length = 6

clear_space(mc, 100)

for i in range(1, 100, 20):
    print(x+i, y, z)
    make_house(mc, x+i, y, z, width, height, length)
```

Exercise

How many houses are there? Make the street longer with more houses.

Exercise

Make the houses get taller as the street goes on.

Exercise

Add some towers to the street.

Advanced

Put a loop inside the loop to create multiple streets.

Advanced

Make some roads or fences.

Exercise

Make your houses out of TNT. Use flint tool on them.

14.15 Chat commands

This program can read chat messages posted by players. It builds a block next to any player who says "build". This is the first example that will work for more than one player.

Program 14.13: Chat commands

```
from mcpi.minecraft import *

mc = Minecraft.create()

while True:
    events = mc.events.pollChatPosts()
    for event in events:
        print(event)
        if event.message == "build":
            id = event.entityId
            pos = mc.entity.getTilePos(id)
            x = pos.x
            y = pos.y
            z = pos.z
            mc.setBlock(x, y, z, 1)
            mc.postToChat("done building!")
```

Advanced

Build a house around the player if the player says "house".

Advanced

Build a lava trap if the player says "trap".

Advanced

use *mc.getPlayerEntityId("fred")* to get the id of a certain player named Fred (or whatever your friend's player name is). Build something at the position of this player.

14.16 Turtle

This requires the **minecraftstuff** *package to work.* You can install it in Mu by clicking in the bottom right gadget and adding `minecraftstuff` to list of third party packages

In olden days at school we used robotic turtles to draw on paper. You may have done similar on the screen in Python or Scratch. This is the same but in Minecraft.

Program 14.14: Turtle

```
from mcpi.minecraft import *
from mcpi.block import *

from minecraftstuff import MinecraftTurtle

mc = Minecraft.create()
pos = mc.player.getTilePos()
pos.y += 1

turtle = MinecraftTurtle(mc, pos)

turtle.forward(5)
turtle.right(90)
turtle.forward(5)
turtle.right(90)
turtle.forward(5)
turtle.right(90)
turtle.forward(5)
```

Exercise

Draw a triangle, hexagon, etc.

Exercise

What do turtle.up(90) and turtle.down(90) do?

CHAPTER

FIFTEEN

REVISION NOTES

15.1 Truth tables

0 is `False`, 1 is `True`.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

A	NOT A
0	1
1	0

15.2 Data types

Type	Examples	Meaning
bool	True, False	Boolean
int	1, 5, 0, -3	Integer
float	1.0, 0.56, -17.1	Floating point, real
str	'abc', "hello", "10"	String
char	'a', 'z'	Character[1]
list	[1, 2, 3]	List
dict	{'a': 1}	Dictionary

noo

foo woo sdf

sdfs

Note: this is a note note foo

sdfsdf

Tip: this is a tip note foo

sdfsdf

And, by the way...

You can make up your own admonition too.

Optional Sidebar Title

Optional Sidebar Subtitle

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

[1] Character is not a type in Python, it is just a string of length 1.

CHAPTER

SIXTEEN

TUTORIAL: NETWORK CODING

16.1 Sockets

A socket connects a program on one computer to a program running on another computer. One program writes data into its socket. The data may be raw bytes, or any of the datatypes we have seen (e.g. integer, string, etc.) The data travels across the network. The program on the other computer reads the data from its socket.

Connecting a socket is a bit like making a phone call. The program that is waiting to receive the call is the *server*. The program that makes the call is the *client*. Once the call is connected they can both speak to each other.

If possible, you should run the server program on one computer and the client program on a different computer, both connected to the same network. If you don't have two computers you can run both programs on the same computer. Using Mu, you will have to load two copies of Mu in order to run two programs at the same time.

First find out the IP address of your server computer and write it down. The easiest way to open a command terminal and type a command. Possible commands:

- `ipconfig` (Windows)
- `ifconfig` (Mac)
- `ip address` (Linux)

The IP address will look similar to this: *192.168.0.105*

Enter and run the server program:

Program 16.1: Socket server

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(("0.0.0.0", 65439))
while True:
    sock.listen()
    connection, address = sock.accept()
    message = connection.makefile().readline()
    print("received: ", message, "from:", address)
```

After creating the socket, we *bind()* it to an *internet address*. **0.0.0.0** is a special address that means "any address belonging to this computer". We also give a *port number*, **65439**. This can

be (almost) any number, but it must match on the client and server.

Then we *listen()*, which pauses the program until a connection arrives. Then we read a line from the socket connection and print it out.

Enter and run the client program. We have put the address of the server in a variable. If your server is on a different computer, use the address of that computer. If it on the same computer, use the special address **127.0.0.1**.

Program 16.2: Socket client

```
import socket

server = ("127.0.0.1", 65439)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(server)
sock.send(b"hello")
sock.close()

```

Note that we send a string, but we prefix it with *b* to convert it into bytes. We close the socket after we are finished to end the connection.

You should see the string appear on the server program output.

Exercise

Send a different message from the client to the server.

Exercise

Make the client program ask the user to input a message, and then send it to the server. Make a loop so it keeps asking and sending.

16.2 Gopher

Gopher is a hypertext system, similar to the World Wide Web. However Gopher is older than the web, and its protocol is simpler, which makes it easier for us to implement. A *protocol* is just an agreement between the client and server on what sort of data they are allowed to send and receive.

We are going to write a Gopher client, connect it to a Gopher server, and see if we can figure out the protocol.

Program 16.3: Gopher client

```
import socket
server = ("gopher.floodgap.com", 70)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(server)
```

(continues on next page)

(continued from previous page)

```
sock.send(b"\r\n")
for line in sock.makefile().readlines():
    print(line)
sock.close()

```

You can change the server to connect to any gopher server, but we are using *gopher.floodgap.com* because it is popular.

When you run the program, it simply sends **"\r\n"** which means a return followed by a newline. It then prints the lines it gets back from the server. You should get some output that looks a bit like this:

```
iWelcome to Floodgap Systems' official gopher server.                error.host  ␣
↪   1

iFloodgap has served the gopher community since 1999         error.host      1

i(formerly gopher.ptloma.edu).               error.host      1
```

This is readable, but we can improve it. You might notice that every line starts with a special character, usually an **i** (meaning *info*). This isn't part of the text, so we will chop it off.

Python lists and strings allow us *slice* them up. For example, if you have a string

```
s = "Hello"
```

Then these some slices you could make:

```
s[0] == "H"
s[1] == "e"
s[0:4] == "Hell"
s[1:] == "ello"
s[:2] == "He"
```