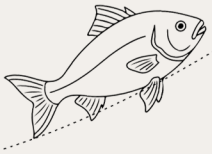
# Creative Python Programming with Strype

## Volume 1: Getting started with Python



Michael Kölling
Neil C.C. Brown

# Creative Python Programming with Strype

## Volume 1 – Getting started with Python

Michael Kölling and Neil C.C. Brown

Published using asciidoctor and asciidoctor-pdf.

Version 1.0
Last updated September 2026

# Contents

Preface . . . . . 5

A note for learners: How to use this book . . . . . 5

A note for teachers: Pedagogy . . . . . 6

Strype . . . . . 8

The book sequence . . . . . 9

Acknowledgements . . . . . 10

1 Getting to know Strype . . . . . 11

1.1 Getting started . . . . . 12

1.2 Running a program . . . . . 12

1.3 Editing a program . . . . . 13

1.4 Starting with graphics . . . . . 14

1.5 Importing the graphics library . . . . . 15

1.6 Setting the background . . . . . 15

1.7 Creating actor objects . . . . . 17

1.8 Working with images . . . . . 18

1.9 The world coordinate system . . . . . 20

1.10 Summary . . . . . 22

2 Animating graphics . . . . . 23

2.1 Assignment – a first look . . . . . 23

2.2 Actor movement . . . . . 25

2.3 Controlled movement . . . 27
2.4 Keyboard control . . . 29
2.5 Summary . . . 31
3 Working with functions . . . 35
3.1 Function calls . . . 35
3.2 Reading function definitions . . . 38
3.3 Optional parameters . . . 43
3.4 Data types: string, int and float . . . 46
3.5 Summary . . . 47
4 A closer look at our first Python statements . . . 49
4.1 Variables and assignment . . . 49
4.2 Objects . . . 51
4.3 Reading the library documentation . . . 53
4.4 Methods . . . 55
4.5 The "Fat Cat" example . . . 56
4.6 If-statements . . . 58
4.7 Summary . . . 60
5 Creating a game . . . 63
5.1 The plan . . . 63
5.2 Adding food . . . 63
5.3 Repetition – the for-loop . . . 64
5.4 Using the loop counter . . . 66
5.5 Creating random behaviour . . . 68
5.6 Importing from a library . . . 69

5.7 Random placement . . . 70
5.8 Summary . . . 71

6 Collision detection . . . 73

6.1 Detecting collisions: is_touching . . . 73
6.2 Tagging actors . . . 74
6.3 Return values . . . 74
6.4 Bounding boxes . . . 77
6.5 Adding a predator . . . 78
6.6 Summary . . . 80

7 Writing good code . . . 83

7.1 Analysing program structure . . . 84
7.2 Defining our own functions . . . 86
7.3 Readability and naming . . . 88
7.4 Abstraction . . . 91
7.5 Reuse . . . 93
7.6 Refactoring . . . 93
7.7 Summary . . . 94

8 Finishing our game . . . 97

8.1 Adding sound . . . 97
8.2 Game over . . . 100
8.3 Random turns . . . 100
8.4 Keeping score . . . 102
8.5 Other extensions . . . 102
8.6 Summary . . . 103

Appendix A: Frame-based editing – the basics . . . 105
A.1 Basic editing . . . 105
A.2 Convenience functions . . . 109
Appendix B: Working with Strype - Tips and tricks . . . 111
B.1 Display options . . . 111
B.2 Project and file storage . . . 114
B.3 Export / import . . . 115
B.4 Other . . . 117
Appendix C: Dealing with errors . . . 119
C.1 What to do when you get an error . . . 119
C.2 Deciphering errors: example 1 . . . 120
C.3 Deciphering errors: example 2 . . . 121
C.4 Deciphering errors: example 3 . . . 123
C.5 Welcome to the wonderful world of errors . . . 124
Appendix D: Python operators . . . 125
D.1 Arithmetic operators . . . 125
D.2 Boolean operators . . . 125
D.3 Comparison operators . . . 126
D.4 More operators . . . 126
Index . . . 127

# Preface

Welcome to Python programming with Strype! In this book, you will learn the fundamentals of programming in Python, using the Strype programming environment. The goals of this book are two-fold.

First, we want readers to learn fundamental programming principles. After working through this book, you will have acquired quite a thorough understanding of how programming works, how to structure programs, and what makes good programming practice. We will use Python as the language for doing so, and along the way you will learn a lot about Python programming. However, Python itself is not the main goal, and we do not attempt to teach all of Python's constructs. Rather, Python is a means to discuss programming principles more generally, and much of what you learn here will transfer directly to programming in other languages.

Second – and equally important – we want readers to experience the joy and thrill that programming can bring you. We are aware that readers who have not programmed before may frown at this statement, and question whether the words "programming" and "joy" can go together in the same sentence. But they really can!

This book emphasises creativity – the making of things. In each chapter, we will challenge you to add your own ideas, to extend what we show you, to create programs that are truly your own. There is real joy when a programmer sees one of their own creations working for the first time. Everyone who has programmed has experienced this, and we very much want you to experience this feeling as well.

## A note for learners: How to use this book

There are many ways to read this book. You could read it sequentially, from beginning to end, and everything in between. Or you can jump around, dip in and out, look forward, and come back when you are missing something.

The main thing is: be curious. With the sequence of the chapters in this book, we are suggesting what we think is a sensible order of activities, but we also realise that everyone's starting point is different. Concentrate on what interests you.

One thing, however, is crucial: Make sure to do the exercises and the practical work while you read this book. You will not get much out of the book without doing so. Programming is a contact sport: you will not become a programmer just by reading about it. This book tries to put you in a position where you can actively do some programming tasks, but it is only in the actual *doing* that you will learn.

You may wonder: Is it still worth going to the trouble of learning all this when AI is here and could do it for me? The answer is very definitely: Yes, it is still worth doing it! To work with AI properly, we need to understand, in detail, what the AI has written. And for that you need to be able to program. You will never become a good programmer – even with AI – if you cannot read and write the code yourself. So for the purpose of this book, we recommend that you do not use AI to write your code (although you may at times use it to explain the concepts). Only by reading, writing and struggling through yourself can you truly learn.

While you do the programming activities: be patient. The early chapters of the book start with easy tasks, but they get more difficult later on.

Spend a good amount of time with each topic. Extending the suggested projects will help you understand more. Add your own ideas, until you feel comfortable at each stage.

As you move through the book – whether it is in sequence or by jumping back and forth – pay attention to the concept summaries at the end of the chapters, and do not be afraid to go back to sections you have previously completed.

## A note for teachers: Pedagogy

Pedagogy is the term for the art of teaching. The authors of this book have many years of experience in teaching programming, and this experience has shaped the approach of this text. This book follows a carefully designed pedagogy based on worked examples, scaffolding, a spiral approach and working from the concrete to the abstract.

***Worked examples*** are used throughout, and all concepts are introduced and practiced via concrete projects. Emphasis is placed not only on the end product – the finished program – but also on the *process* of developing it.

Many textbooks follow a common pattern: A problem is presented, a solution is

shown, and then the constructs used in that solution are explained. This is a terrible approach. It creates the illusion that a good programmer goes in one step from reading the problem description to writing down a fully working solution. This is not how programming works.

A result of this approach often is that learners who struggle with finding the right solution, or with making it work quickly, think they are failing. They react by developing a self-image of "I am not a good programmer" and "programming is just not for me".

The fact that we all work in small steps, try things out, run into problems, backtrack, try something different, and slowly work our way to a solution seems to be one of the best kept secrets in programming.

This is a shame.

In this book, we show not only the product, but the *process* of programming. With each project, we go through several phases of extension and refinement. Learners experience the process, and can follow along. We use a process of ***scaffolding*** to provide meaningful context while keeping learners focussed on the topic at hand, and ***fading*** of the scaffold and guidance over time.

At the same time, the examples are selected to be open-ended and extendable, so that learners are encouraged to take each example further in a direction of their own interest. Each chapter includes suggestions to do more.

We also use a ***spiral approach***. When we first encounter a construct, we do not try to explain or understand everything about it. We learn what is necessary at that point, and then refine our understanding later. We come back to the concept in a different context, and then again later still – and in each iteration we deepen our understanding.

This is a more natural and useful approach to understanding programming. Firstly, many programming constructs are circularly dependent on each other, and there is no linear sequence in which everything can be explained only by backwards reference. But more importantly, when books try to explain everything about any given concept at once (think, for example: types) learners cannot usually distinguish what is needed at that moment ("what do I need to know right now?") from what is there only for completeness's sake. The important is buried in the detail, and learning is made harder.

The spiral approach serves to break this linear form, iterate over concepts several times, and lead to a deeper understanding.

## Strype

The other obvious difference of this book is the use of the ***Strype*** environment.

Strype is a browser-based educational Python development environment designed specifically for novice learners. It is free to use, and anyone can use it instantly – no creation of accounts is needed. Strype uses a *frame-based editor*: frame-based editing is an interaction style that combines the best of both worlds of block-based and text-based editing. It provides the discoverability and some of the error avoidance of blocks, but at the same time gives you the power and expressiveness of a full, professional language. Some of the friction of getting your programs to run is removed, and you achieve results more quickly and with less frustration. The use of Strype, and its integration in this book, are absolutely fundamental to our approach.

As we stated above, our firm goal is to make the first experience of programming engaging, fun and creative. One way to do so, for example, is the early use of graphics and animation. Yet animated graphics are much too complicated for novices to use in most Python environments.

Strype is designed to make this approach simple, painless and *possible*. We can create a graphic in a single line of code, and animate it in just a few lines more. The effect of program statements becomes immediately visible on screen in a way that no "Hello world" program can ever manage, and learners can observe their program execute.

The technology and the pedagogy used in this book were developed in tandem, with environment design strongly guided by our pedagogical principles. The result is a package of software, examples and explanation that is seamlessly integrated, and enables learners to achieve more satisfying results than they might with other systems.

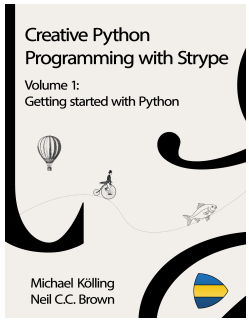
Overall, we strongly believe that this helps the most important aspect of all: motivation. We hope to spark our readers' curiosity, to leave you wanting to read on, to continue to learn, because you have caught the programming bug just like we have.

We hope that you will see not only how programming works, but also why you might be interested in it, and what it can do for you.

## The book sequence

This book is the first volume in a sequence of books, which take the reader from a complete novice, with no prior knowledge assumed, to quite sophisticated programs. You can work with this volume on its own – it forms a coherent unit – or you can go on to study the other volumes as well. Each volume is self-contained, but the sequence builds up to higher skill levels.

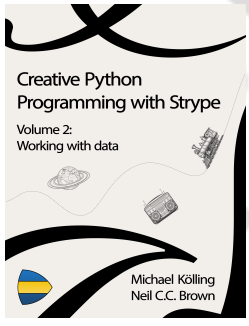
### Creative Python Programming with Strype




#### Volume 1 – Getting started with Python

This first volume is the one you are reading now. It takes you from the very beginning – knowing very little or nothing at all about programming – to the point of being able to write quite sophisticated programs. Part of this is learning how to create an interactive graphical game.

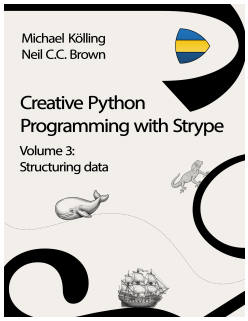
The activities in this volume are very carefully constructed and sequenced to teach you many general and fundamental programming concepts and constructs. Additional projects encourage you to try out what you learned and apply it in other contexts. At the end, you will have a good understanding of all of the important fundamental concepts of programming.




#### Volume 2 – Working with data

In this book, we start to work more seriously with large amounts of data. The reader learns, for example, how to write programs to analyse text, and we develop projects to analyse whole books to gain some insight about them.

In the process, we learn a lot about text processing, and about working with lists. In fact, with the exercises in this book you will become quite fluent in list processing, and understand how to use Python lists to solve all sorts of problems, from analysing data to manipulating sound.




**Volume 3 – Structuring data**

This volume builds on the previous one and dips into even more interesting projects. We learn how to use a greater variety of data structures, such as tuples and dictionaries in addition to lists, and how to combine these to model more interesting data. We do not only analyse data sources, but we also take a look at generating text, in a way a text prediction algorithm might do. In the process, we will even gain a basic understanding of the inner workings of large language models (LLMs) when we write a simple version of an algorithm that is at the core of these systems.

## Acknowledgements

We wish to thank Pierre Weill-Tessier for his work on the implementation of Strype. Pierre is, together with the authors of this book, a core member of the team that designed and implemented the Strype software. His work is present in many details of the system, and with that, he has contributed greatly to the outcome of this project.

We are also indebted to Pete Dring, who provided feedback on an early draft of this book. Pete's feedback has helped to improve this book in many places. His comments were thoughtful, detailed and specific, and the book is better because of them.

# Chapter 1 Getting to know Strype

**Topics:** getting started, running a program, the graphics system, first look at writing a program

**Constructs:** function call, parameter

This chapter will show you how to get started with Python programming in Strype. We will work on our first project and start to see what programming in Strype is like, and what we can do with it.

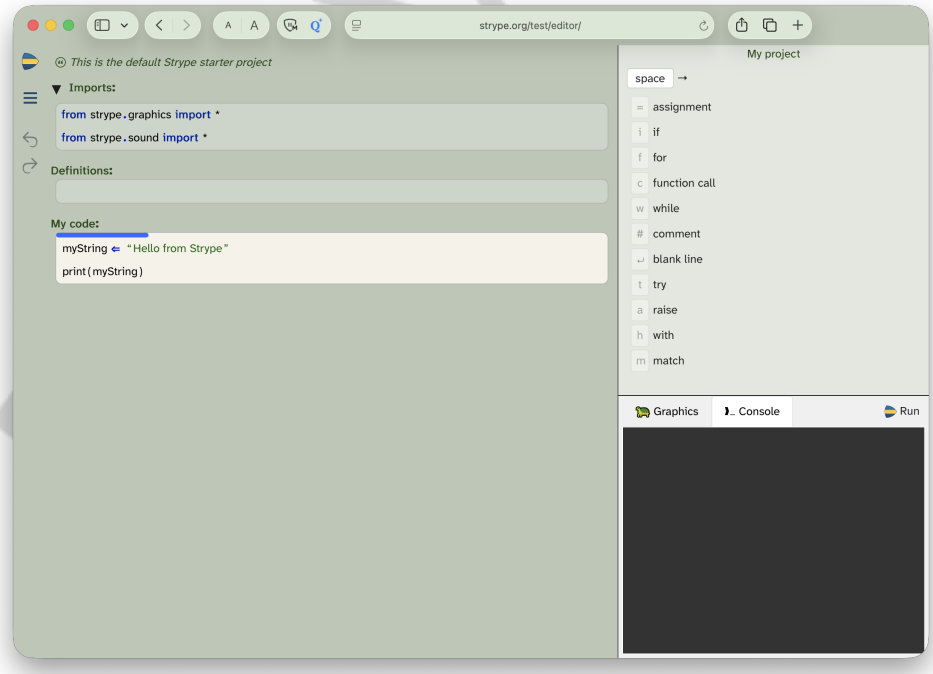



*Figure 1.1: The Strype editor window*

## 1.1 Getting started

Getting started with programming in Strype is easy: Point your web browser at strype.org, click on the *"Start coding now"* button, and you will see the Strype editor. Here, you can enter, edit and run Strype programs. Figure 1.1 shows a screenshot of the editor.

When you first open Strype, you will see a short program to get you started.

## 1.2 Running a program

The first thing to try out is to run the sample program. To do this, click the "Run" button on the right and observe what happens.

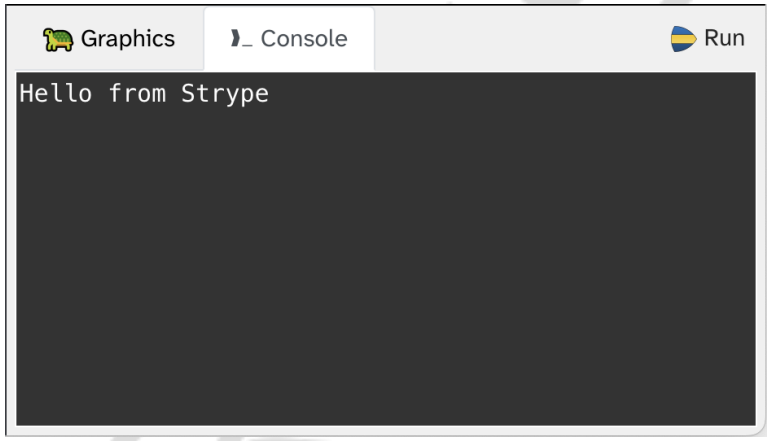



*Figure 1.2: Program output in the Console*

After clicking "Run", you should see the text "Hello from Strype" appear in the bottom right area of the window (Figure 1.2). This area is called the "Console", and it is where Strype will display text output. (You will note that there is also a "Graphics" tab next to the Console –- this is for graphical output, and we will use this shortly.)

Clicking the "Run" button executes the program shown in the editor and displays its results.

**Concept** Sometimes we will tell you "**run**" a program, or to "**execute**" a program. These mean the same thing — you do this by clicking the "Run" button.


There is a keyboard shortcut for running a program: You can type **Ctrl-Enter** (on Windows) or **Command-Enter** (on macOS) to run the current program.

## 1.3 Editing a program

You can also change (edit) the current program and then run it again. For example, you could change the text that is printed by the sample program to print something different. Try this now. (If you are not familiar with the basics of editing in Strype, you should read Appendix A: Frame-based editing – the basics first.)

You will note from this paragraph that Python (and other programming languages) use the term "printing" somewhat differently from how we normally use it. Here, it does not mean to print something on paper with an external printer, but simply to write something to the Console.

**Exercise 1.1** In the Strype editor, change the green text that reads "Hello from Strype" to say Hello, followed by your own name. Make sure to leave the double quotes around the text you wish to print out. When you have done this, run the program again.

**Exercise 1.2** Add a new function call frame at the end of the program. In it, call the print function and print out the words "How are you today?" You should now see two lines of text printed to the Console.


You can **add a frame** to the code by moving the **frame cursor** (the blue bar) to the intended location using the up and down cursor keys, and then typing **Space** followed by the **frame shortcut** for the desired frame. The shortcuts are shown in the top right.

Now that we know how to edit and run programs, we can, of course, write many

more programs that produce text output (and much more interesting programs too), and we will do so later in this book. First, however, let us look at another type of output: graphics.


If you are new to programming, you will be surprised how often it happens that we make errors in our programs. If you are not used to dealing with programming errors, you should read Appendix C: Dealing with errors first.

## 1.4 Starting with graphics

We will start our exploration of graphics programming in Strype with a very simple program that demonstrates some of the basic concepts.

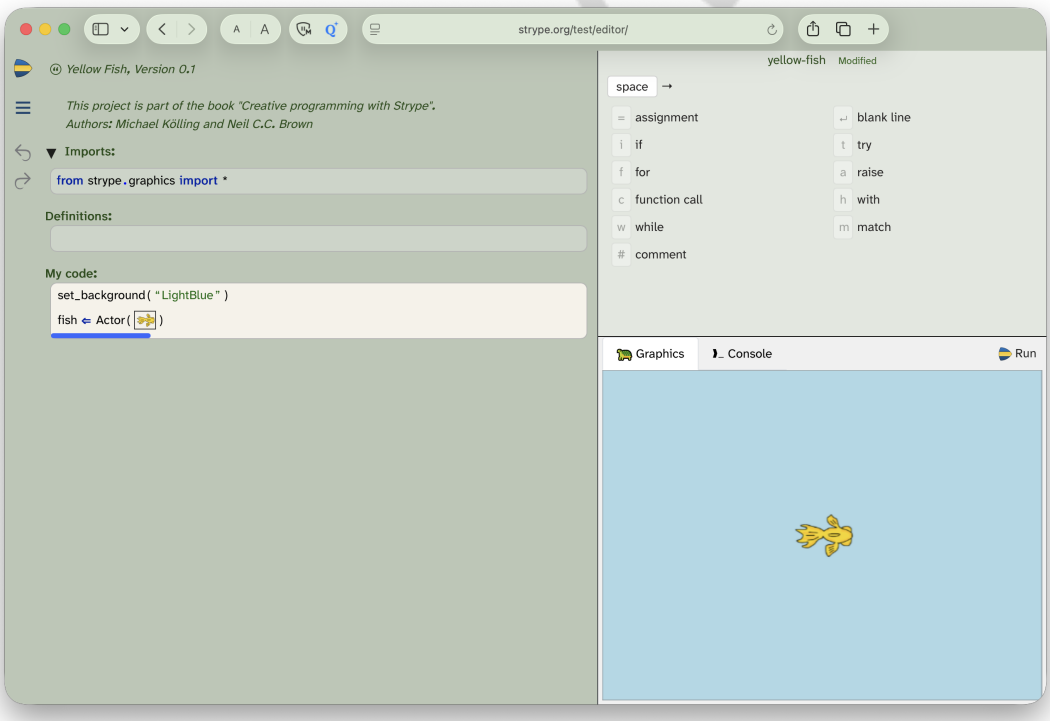



*Figure 1.3: The starting state of our 'yellow-fish' project*

Open the yellow-fish project from the Chapter 1 book projects. Execute the program.

You should see a result similar to the one shown in Figure 1.3.

Note **Opening book projects:** If you are reading this on screen, you should be able to click on the project name to load the project in Strype. If you are reading this on paper, open Strype, click the three-lines icon in the top left to slide out the menu, then click the "**Book...**" item to find and open the book projects; they are organised by volume and chapter.

## 1.5 Importing the graphics library

This yellow-fish program makes use of the Strype graphics library. We can see this near the top of the program, in the section labelled "**Imports**".

Python is an extendable language, which has some functionality built in, and makes use of "libraries" to add more functions as we need them. In this book, you will learn to use both standard Python functions, as well as commonly used libraries. What is more, you will learn how you can find out what functionality is available in a library, so that you can teach yourself to use it later.

Producing graphical output is an area that is not built in to the standard Python language, so we need to use a library to enable this functionality. This has been done already in the project by adding the following line in the "Imports" section:

```
from strype.graphics import *
```

The * symbol at the end of this line is a shortcut for "everything". So this line in effect tells the system to import everything from the Strype graphics library. "Importing" means to make it available for us to use in our program, so the effect is that we can now use all of the functionality of the graphics library in our program.

We will start by experimenting with some of the functionality that the library provides.

## 1.6 Setting the background

Let's have a look at the first line of code in our program, in the "My code" section:

```
set_background( “LightBlue” )
```

This kind of statement is called a "**function call**". A function call executes a function that is defined somewhere else (in this case: in our graphics library) and that performs some kind of action. Later on, we will need to work out how we can find out about all the functions that are available in the library, but for now we will just experiment with a few of them.

**Concept** A **function call** is a Python statement that executes a function that is defined elsewhere. It consists of the name of the function and a parameter list in parentheses (another word for round brackets, like the ones around this text).

In this case, "set_background" is the name of the function we are calling, and it is followed by a pair of parentheses. Between the parentheses is what is called a "parameter". A parameter provides some additional information to tell the function more precisely what to do. Here, we are calling the function set_background, whose job it is to set the background of the graphics world, and we are providing the parameter "LightBlue" to let it know what colour we would like it to use.

**Concept** A **parameter** provides additional detail information to a function. Parameters are written in parentheses, with multiple parameters separated by commas.

Let us experiment a bit with this function call to see how we can influence its behaviour.

**Exercise 1.3** Change the parameter of the set_background function call to "DarkRed". Make sure you keep the quotes around the name. Execute the program to see the effect.

**Exercise 1.4** The colour names you can use as a parameter are the standard colour names used in HTML for writing webpages. You could find a list of available colour names by doing a web search for "html colours". There is, however, an easier way to try out some other colours: Delete the colour name

and the quotes from your function call. You should then see a white slot that lets you enter a new parameter. With the cursor in this slot, click on the "Colour picker" option on the right. You will see a dialogue that lets you pick a colour. Choose a colour that you like.

Note **Types of brackets:** Python uses various different kinds of brackets in different places. The ones we have seen for functions above are called either "round brackets" or "parentheses": **( )**

Other bracket types Python uses are "square brackets", which look like this: **[ ]**, and "curly brackets", which are sometimes also called "braces": **{ }**

## 1.7 Creating actor objects

The next line of code looks like this:

```
fish ⇐ Actor( <image> )
```

There are several things happening in this one line of code, so we will investigate this in some detail.

In the Strype graphics system, you can add graphical elements to the graphics world by creating "Actor" objects. Actor objects (we may also call them just "actors") are created by inserting a function call frame into your code, and then writing the word "Actor" in place of the function name. Try this out now.

**Exercise 1.5** Insert a function call frame at the end of your code. To do this, move the frame cursor to the end of the code and then press space c (or click on the *function call* option in the frame palette). In place of the function name, write "Actor".

Once you have completed the exercise above, you will see the call to create an actor:

```
Actor( image )
```

The frame also tells you that a parameter is expected here. It shows the grey word "image" on a white background. This tells us that creating an actor requires an image as a parameter. If you were to execute your program now (try it!), you will see that an error is reported in this frame when we try to run the program without providing an image. So let's fix this, and add an image.

## 1.8 Working with images

You can add images to your program, by copying them from outside of Strype (for example, from the web, or from a file in your file system) and pasting them into Strype (Figure 1.4).

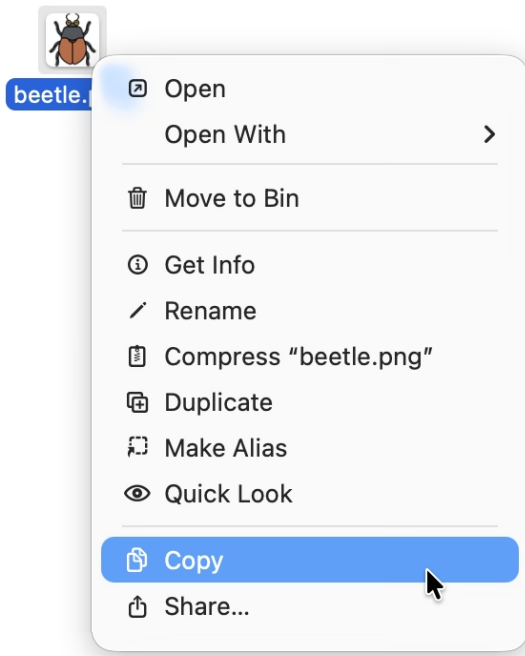



*Figure 1.4: Copying an image from the file system*

**Exercise 1.6** Add your own image to the actor object now. You can, for example, do a web search for "beetle icon", find an appropriate image, and copy it. (The image does not need to be an animal. It could be a car, or a spaceship, or anything else that moves.) To copy the image, you can either copy it straight out of your web browser, or save it to your disk first and then copy it from there. Figure 1.4 shows the context menu of a local file with the "Copy" operation

selected. This menu will look different on different operating systems, but you should be able to find the copy operation and use it. After copying the image, paste it into the Actor parameter.

Once you have pasted in your own image as the actor image, try running the program again. You should now see the new actor on screen. If you have left the line of code that creates the fish actor in place, you may also see the yellow fish. Both actors will, however, be drawn in the centre of the screen, so if your new actor is larger than the fish, its image may entirely cover the fish image.

It is quite possible that your new image does not look exactly as you would like it to. For this situation, Strype offers some simple image editing functions to adjust the image to make it more appropriate for our task. When you hover the mouse pointer over an image in the Strype code, you will see a preview of the image with an option to open a simple image edit function (Figure 1.5). Here, you can adjust the image for your project. If you need more sophisticated image editing functions, you can of course always use a separate image editing program before using the image.

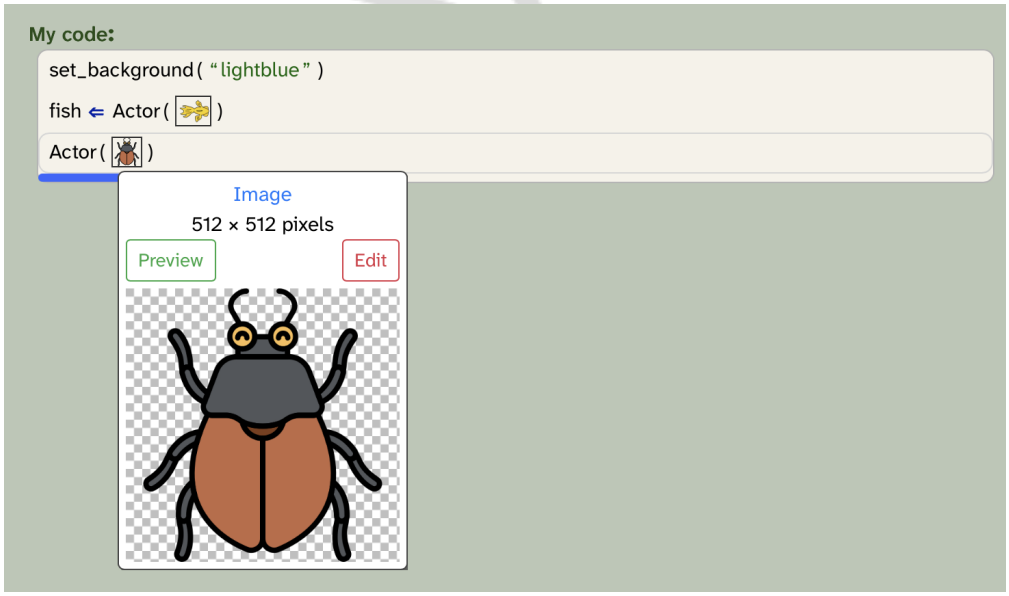
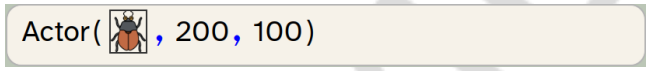



*Figure 1.5: Editing an image in Strype*


You almost always want actor images that have a transparent background, so that you can correctly see your world's background behind the actor. You should use images of type PNG or WEBP to do this. JPEG images do not support transparent backgrounds.

## 1.9 The world coordinate system

When we placed our second actor into the world, we saw that they are both drawn on top of each other in the centre of the graphics world. We can change this by assigning initial coordinates to a new actor:




The two additional parameters to the actor specify *x* and *y* coordinates for the positioning of the actor. You have probably encountered such coordinates in mathematics; *x* is the horizontal position and *y* is the vertical position. Here, the actor is placed at x-coordinate 200 and y-coordinate 100. The coordinate system of the graphics world reaches from -399 to 400 in the horizontal (x) dimension, and -299 to 300 in the vertical (y) dimension. It is shown in Figure 1.6.

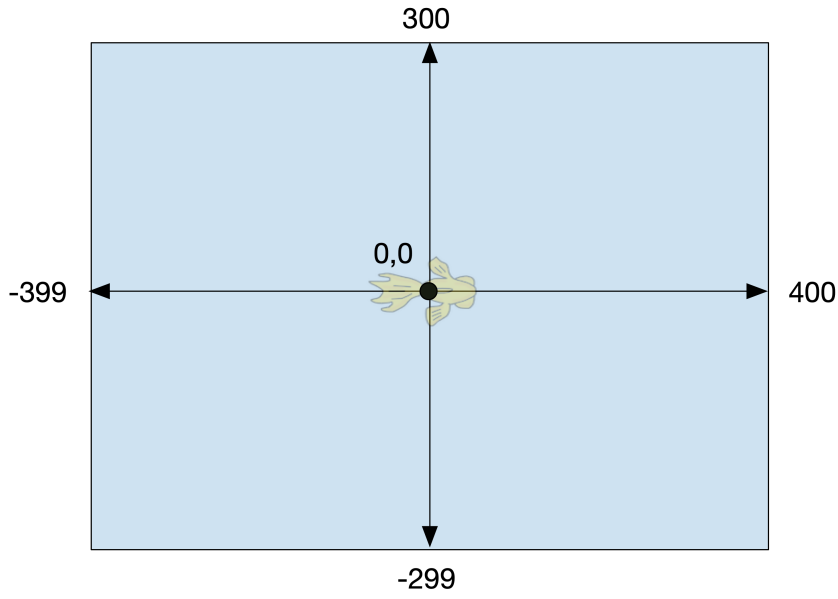



*Figure 1.6: The graphics world coordinate system*

Note **World coordinates:** You may wonder why the negative edge of the world is at -399 for the x-coordinate, and not at -400 (with a similar difference for the y-coordinate). Would it not seem more consistent to make it -400 to 400? The reason is that the size of the world is 800 x 600 pixels, which is a commonly used aspect ratio. With coordinates ranging from -400 and -300, the size would actually be 801 x 601, because zero is also included. The coordinates are as they are because of the fixed 800 x 600 world size.

**Exercise 1.7** Place your own actor into the world somewhere near the top left corner, using the coordinates as shown above.

We have now reached a point where we can create actors, give them images and place them into the world at a specific location. However, currently our actors do not actually *do* anything. In the next chapter, we will look at making our actors move, so that we can see more interesting things happen.

## 1.10 Summary

In this chapter, we have jumped straight into the start of our first Python program: showing a graphic. The intention is that we will develop this project into a simple animated game. So far, we only have the very beginning of the project implemented, but we will continue to work on this over the following chapters.

### Concept summary

- Sometimes we will tell you "**run**" a program, or to "**execute**" a program. These mean the same thing — you do this by clicking the "Run" button.
- A **function call** is a Python statement that executes a function that is defined elsewhere. It consists of the name of the function and a parameter list in parentheses (another word for round brackets, like the ones around this text).
- A **parameter** provides additional detail information to a function. Parameters are written in parentheses, with multiple parameters separated by commas.

# Chapter 2 Animating graphics

**Topics:** movement, animation speed, keyboard control

**Constructs:** assignment, variable, object, if-statement, while-loop

In this chapter, we will make our graphical character actually *do* something. It won't be very much at first, but at least we want to make it move across the screen. We continue straight where we left off in Chapter 1.

If you have done the exercises in Chapter 1, and you compare the two lines we currently have in our program to create actors, we can see that they look different:

```
fish ⇐ Actor(  )
Actor(  , 200, 100)
```

The difference is not only the additional coordinates that we have specified with our second actor, but also the arrow symbol (⇐) and name (fish) that we have used in the first line. This is called an *assignment*, and we will investigate this next.

## 2.1 Assignment – a first look

The purpose of an assignment is to keep a reference to our actor object, so that we can continue to work with it afterwards. When we create an actor on a line on its own (as in our second line above), the actor object is created, it is automatically placed into the world, and that is all that happens. After this statement, we do not have access to the actor anymore, and we cannot directly work with it further.

This is why we often use an *assignment* to assign the actor to a *variable*, which allows us to continue working with this actor. Let us investigate what this means.

We can enter an assignment frame into our code by typing the = key. When we enter the frame, it looks like this:

```
identifier ⇐ value
```

The effect of an assignment frame is that it creates a variable and stores a value into it. A variable is a small amount of storage that allows us to store some information, and to access and use it again later. Variables have names – this is how we refer to them – and store a value.

**Concept** A **variable** is a small amount of storage space that allows us to store a value and retrieve and use it again later. Variables have names, which are used to refer to them.

We can see that the assignment frame has two slots which we must fill in. The slot on the left lets us provide a name for the variable. We can make up this name, and we usually choose a word that describes what we will store in this variable.

The right hand side of the assignment has a slot labelled "value". Here, we can specify what we want to store in the variable.

Some examples of assignments to variables are:

```
age ⇐ 17
name ⇐ "Leslie"
fish ⇐ Actor( <image> )
```

We can see from these examples that we can use variables to store different kinds of thing. We can store numbers, or we can store text (words, or whole sentences), or we can store actor objects. We will see later that we can also store all sorts of other things – we will get to that in due course.

**Concept** An **assignment** stores a value into a variable.

We can then use the value stored in that variable just by using the variable's name. For example

```
print(age)
```

will print out the contents of the 'age' variable (which will be 17 after the assignment which we made above). Compare that with the statement

```
print( "age" )
```

This statement would print out the word "age". Note the difference: the quotation marks around the word "age". If we print "age" with quotation marks, we are printing the word "age". If we print age (without quotation marks), we are referring to our variable named age, so we are printing out the value stored in this variable: 17.

Now that we have seen the first basics of variables, let us get back to our yellow-fish project.

## 2.2 Actor movement

We can now properly interpret the two lines of code we have seen before:

```
fish ⇐ Actor(  )
Actor(  , 200, 100)
```

The first line creates an actor with a fish image and assigns it to a variable named fish, while the second line creates an actor and does nothing further with it.

We want to do further actions with our actor, so we need to assign it to a variable. But we do not need two actors at the moment, so we can delete one of the actors again. For this book, we will continue to use the fish actor, but you are welcome to use your own actor for the remainder of this project while you read on.

**Exercise 2.1** Change your code again so that only one actor is created and assigned to a variable. The actor should use your own selected image. Rename the variable so that it correctly describes the type of actor that you have chosen.

(For example, if your image shows a beetle, you might like to name the variable `beetle`, if it is a car, name it `car`. Whenever we use our `fish` variable in the remainder of this chapter, you should then use your own variable.)

Now that we have our actor, and we have stored it in a variable, we can make it move:

```
fish.move(8)
```

Try this out with the following exercises.

**Exercise 2.2** Add the line of code which makes the actor move (shown above) to your own code, after the actor has been created. You can do this by adding a function call frame, typing `fish.move` as the function name, and adding the parameter. Run your program. What do you observe?

**Exercise 2.3** Change the parameter from 8 to 300. Run your program again. What do you observe?

After the first exercise, you will probably notice that the fish does not appear to move. This is because the parameter is in *pixels*. Eight pixels is a very small distance on screen, so when you ran your program, the fish was shown and then – very quickly – moved eight pixels to the right. But this distance is so small, and the movement so quick, that it is hard to notice it at all.

In the second exercise, the distance we have used is larger, so we can see that the fish is now at a different place. But again, the movement is so quick that we do not really see the fish move. If we want to turn this project into a game, we need our fish to move more slowly.

Before we go on to do this, a little more terminology: A function that is called on an object, such as the `move` function above, is called a *method*. Objects have a fixed set of methods, and we shall see over time what kinds of methods our actors have, and what we can do with them.

Methods are called by specifying the object, a dot, and the method name and parameters.

**Concept** A function that is called on an object is called a **method**. Methods are called by specifying the object we want to call, a dot, and the method name and parameters.

## 2.3 Controlled movement

We have seen in the previous exercises that calling the move method moves the actor very quickly. If we want to have slow, visible movement, we need to move the actor repeatedly, in small steps. We can do this by placing the move method call into a *loop*:

```
fish ⇐ Actor(  )
while True :
    fish.move(8)
    pace(30)
```

**Exercise 2.4** Add a while loop to your program to create the code shown above. Also add the call to the pace function as shown above. Run your program. What does it do now?

**STRYPE TIP** If you have a statement in your code, and you want to place it into a loop, you have two options: You can either insert the loop frame into your code and then drag your statement into the loop body with the mouse. Or you can select your initial statement (use shift-arrow-down) and then insert the while loop – this will surround the statement with the loop.

Let us look at this code a bit more closely. The *while* statement which we have just inserted is a *loop statement*: it repeats the statements inside it over and over again. Since the fish.move method call is inside the loop, it will be executed over and over, many times. Thus, we move the fish eight pixels to the right, and then move it eight

pixels again, and so on.

If we did this without the pace function in our loop, this would still be so quick that we would not see the fish move slowly. (Computers are fast – try it out!) The pace function, however, has the effect of slowing down the loop. To be precise: the parameter of the pace function specifies how quickly the loop should run. A parameter of 30 means that the loop will run 30 times per second. This is a good speed to actually see the movement of the fish as a smooth movement across the screen.

Note that because we now have a program that runs forever, when we want to edit it again, we need to manually stop the execution. You will see that the Run button becomes a Stop button; click the button while it says Stop in order to stop the execution and return to editing the program.

**Exercise 2.5** Experiment with different parameter values for the pace function. Also change the parameter to the move function to different values. You will see that you can use either one to change the speed of movement of the fish on screen. What is the difference between changing one or the other value?

The loop we have used here is called a *while loop*, because it starts with the keyword while. It has the ability to repeat a set of statements several times.

In the slot after the word while, we can specify how long we want to loop to run for. Using the word True here has the effect that the loop will run forever (or at least until we stop the program). We will see a bit later how we can use more sophisticated loop conditions to write loops that run a specified number of times.

Running our loop forever has the effect that the fish keeps moving until it hits the edge of the world. In Strype, actors cannot leave the world, so the fish gets stuck at the edge – every further attempt to move in that direction has no effect, but the program will keep running until we stop it.

**Exercise 2.6** Actor objects also have a method called turn, which also takes one parameter. The parameter is a number, and specifies the angle to turn in degrees. Try out this method: replace the fish.move method call with a call to fish.turn, and run your program. What do you observe?

**Exercise 2.7** Experiment with different parameter values for the `turn` method. What value gives you a nice, slow turning motion?

**Exercise 2.8** Can you make the fish turn the other way?

**Exercise 2.9** Try inserting the call to the `move` method back into the loop, so that you have a call to the `move` method and a call to the `turn` method, following each other. What does the fish do now? Try to predict this before trying it out.

**Exercise 2.10** What would you have to change to make the fish swim in a larger circle? Try it.

The program that we have written so far is available in the book projects as yellow-fish-v2. If you have written your own program while reading this, keep using your own program. If you want to compare your program with ours, you can look at this version, or use this one as a starting point for the next exercises.

## 2.4 Keyboard control

The next obvious step towards our goal to turn this into a game is to give us control: We would like to control the fish with our keyboard, so that it can become our player character. That is what we will do next.

Our plan is to use the left and right arrow keys to control the fish: If the left arrow key is pressed, we want the fish to turn left, and if the right arrow key is pressed, it should turn right.

Luckily, Python has a statement to express exactly this: an *if-statement*.

Concept An **if-statement** is an instruction that allows us to execute a set of statements only if a certain condition is true.


You can enter an if-statement in two ways: You can press space and then i (the shortcut is shown top-right), as we have done before. Alternatively, you can type it: typing if followed by space at the blue frame cursor will create an if-frame. Either way works. This works for other frames as well.

Modify your program so that the loop looks like this:

```
while True :
    fish.move(8)
    if key_pressed( "left" ) :
        fish.turn(5)
    pace(30)
```

In this code, we have added an if-statement into the loop, and using this if-statement, we make the fish turn only if the left arrow key is pressed down.

When you insert a frame for an if-statement into your code, it looks like this:

```
if condition :
```

We can see that this frame has two slots: a text slot for the condition and a frame slot for the statements that should be executed if this condition is true. This frame slot is also called the *body* of the if-statement.


An **if-statement** has two slots: the **condition** and the **body**.

In our loop, we use a function call as the condition: key_pressed. This function takes as its parameter the name of the key we want to check, and returns true if this key is currently pressed down. If it is true, then it executes its body to make the fish turn.

If the condition is false, then the body is not executed, and the fish does not turn.

**Exercise 2.11** Insert this code in your own program. Test it. This should work – if you see any errors, fix them.

All keys on the keyboard have names that we can use with the `key_pressed` function. The name of the character keys is just their character (so the a-key is called `"a"`, and so on) and other key names include `"left"`, `"right"`, `"up"` and `"down"` for the arrow keys, and `"enter"`, `"tab"` and `"backspace"` for some of the others.

Using this, we can now add a second if-statement to make the fish turn right when the right arrow key is pressed.

**Exercise 2.12** Add a second if-statement to your code, immediately following the first one. Use this statement to make the fish turn right when the right arrow key is being pressed. Test your program.

You can find this version of the program in the book projects as yellow-fish-v3.

## 2.5 Summary

We have now managed quite a big step forward in our goal to turn this project into a game: we can control a game character with our keyboard.

Along the way, we have seen quite a number of very useful and important Python programming constructs: function calls, parameters, variables, assignments, objects, methods, if-statements and a while loop.

That is a long list!

We have had a first contact with each of these constructs, and we can roughly understand what they do, but there is more to learn about each of them. So we will now – in the next chapter – take a step back and have a closer look at each of these constructs to deepen our understanding, before we get back to completing our yellow-fish game.

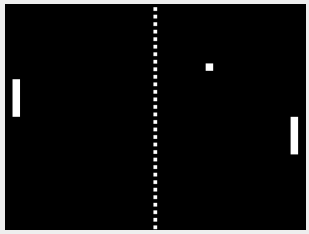
## Concept summary

- A **variable** is a small amount of storage space that allows us to store a value and retrieve and use it again later. Variables have names, which are used to refer to them.
- An **assignment** stores a value into a variable.
- A function that is called on an object is called a **method**. Methods are called by specifying the object we want to call, a dot, and the method name and parameters.
- An **if-statement** is an instruction that allows us to execute a set of statements only if a certain condition is true.
- An **if-statement** has two slots: the **condition** and the **body**.

## Project ideas

*At the end of some chapters, we will suggest additional projects which you may like to try out at this stage. These projects will not be discussed in detail, and it is up to you to study them, decide what you would like to do with them, and take them forward. They are designed to give you some ideas what you can do with the constructs you know now. They can generally be approached with the knowledge you have at this stage in the book (and sometimes require looking something up on your own). You should take these projects as suggestions and ideas what you might do – in either a new project, or to add to an existing project of yours – not as rigid specifications.*

**Pong**



Pong is a very old-school, classic arcade game. If you have not seen it before, you can find out about it by doing a web search for "pong game".

You can find a starter project in the book projects under the name pong-classic. This shows you how to do two-dimensional movement (up/down only), by directly setting the x/y-coordinates of an Actor, instead of using the move method. You can take it from here.

# Chapter 3 Working with functions

**Topics:** calling functions, passing parameters, reading function definitions

**Constructs:** function call, function definition, parameter (formal and actual, optional), data type (str, int, float), comment

In the previous chapter, we have jumped straight into the development of our first project – an animated game. Along the way, we encountered a series of Python statements. We shall now look at these statements in more detail and experiment with them a bit more, before we go on with completing our game.

Of the statements we have seen so far, the function call is the most fundamental: it allows us to use many different bits of functionality that Python provides, and it is essential for any kind of programming. In this chapter, we will gain a fuller understanding of function calls – how to use them, and how to find out what functions do.

## 3.1 Function calls

We have seen function calls a couple of times now. For example, we have seen the call

```
set_background("LightBlue")
```

and the call

```
pace(30)
```

In each case, we call, or *invoke*, an existing function to get it to do some task for us (setting the background colour, in this case, or setting the pace of the loop.)

In general, a function call always has the format

```
function-name ( parameters )
```

The function name specifies the name of the function we wish to call, and the parameters provide some more information to execute the call. Both functions above have one parameter, but a function can have more than one parameter, or none at all, indicating that no further information is required. If a function does not have any parameters, then we still write the parentheses. We just write nothing in between. For example

```
stop( )
```

calls a function called "stop" that has no parameters. It is the presence of the round brackets (also known as parentheses) that identifies this statement as a function call.

To practice with function calls, we will now use another project: *fireworks*.

**Exercise 3.1** Open the fireworks project from the Chapter 3 book projects. Run the project. What do you observe?

If you have completed the exercise above, you will have seen that this project includes some code, but does very little when we run it. In fact, you have to look very carefully to notice any effect at all: It sets the background of the graphics world to a dark blue colour (which is not so different from the default black), and does nothing else.

This project lets us create a fireworks display. It offers us functions to place some firework rockets into the world, and then ignite them to set them off. At the moment, nothing much happens because the project does not place any fireworks into the world.

Let us look at the program code that is executed when we run this project. It is the code in the "My code" section of your project:

**My code:**

```
prepare( "MidnightBlue" )


# Place your firework rockets here


ignite( )
```

We can see two method calls and one comment. The method calls, in this order, prepare the project to be ready (including setting of the background colour) and ignite the fireworks. In the middle, we see a comment that tells us where in our code we should place our firework rockets.

A comment always starts with the symbol #, and continues with some text. The comment is ignored by the Python system; it is here only for a human reader to give us a hint. We can insert a comment into our program by adding a comment frame (the quickest way to do this is to just type #).

**Concept** A **comment** is a frame that shows text for the human reader(s) of the program. It is ignored by the Python system.

We would now like to start creating a fireworks display by placing some rockets into our world, and then igniting them.

**Exercise 3.2** Delete the comment frame from your program. In its place, insert a function call frame for a function named *place_rocket*. As a parameter, you can either use a colour name, like this:

```
place_rocket( "magenta" )
```

Alternatively, with the cursor in the parameter slot, click on the "Colour picker" option on the right (or type space c) to bring up a colour picker. This lets you choose a colour interactively and insert it into your function call:

```
place_rocket( )
```

Run your program to see what it looks like.

**STRYPE TIP** A quick way to insert a function call frame is to type Ctrl-space when the frame cursor is focussed. This will insert a function call frame and bring up the code completion dialogue at the same time.

If all went well, you should now see a single fireworks rocket being fired into the night sky. The program, as it is now, prepares some internals to get the fireworks ready, places one rocket into the world, and then ignites it.

This is nice enough so far, and we could now go on to make our fireworks more interesting by adding more rockets. But we are running into a fundamental question: How do we actually know what methods are available? What are their names, what do they do, and how do we call them?

## 3.2 Reading function definitions

The three functions we are calling here, prepare(), place_rocket() and ignite(), are all defined further up in our own project. We can see the definition of these functions in the "Definitions" section:

```
def prepare (color ) :
def interpolate (start, end, d ) :
def make_fades (original ) :
def make_particle (sx, sy, ex, ey, ssize, esize, n, makes_trail, explodes, images, delay=0 ) :
def make_particle_image (color ) :
def ignite ( ) :
def place_rocket (color, target_x=0, target_y=0, delay=0 ) :
```

This list shows us what functions we have available to call in our project, what their names are (in bold), and what parameters they expect (in the parentheses). They all start with the word *def*, which is short for *define* and is how Python indicates a function definition.

We can, for example, see that there is a function called *ignite*. The fact that it shows a pair of parentheses after the method name, with nothing in between, tells us that this method expects no parameters. This matches the method call in our code section, where we invoke this method without passing a parameter, like this:

```
ignite( )
```

If we wanted to find out more about what this method does, we can *unfold* its definition. This is done by clicking on the small circle outline at the far right of the function definition frame. Once we do so, the function comment is folded out, and we can see some information about this function:

```
def ignite ( ) :
    " Ignite all the rockets that have previously been placed.
```

**STRYPE TIP**

You can **fold** and **unfold** function definitions by clicking on the small circle outline in the top right corner of the function definition frame.

Let us now look at another function: *prepare()*. Its definition looks like this:

```
def prepare (color ) :
```

Again, the def keyword tells us that we are looking at the definition of a function, the next word is the name of the function ("prepare"), followed by a pair of parentheses. This time, however, there is a word within the parentheses (color). This tells us that this function expects one parameter when it is invoked, and this parameter is called "color".

As before, we can unfold the function definition to see a function comment that gives us some more information:

```
def prepare (color ) :
    """ Prepare the scene for our fireworks. This must be called before
    rockets can be placed into the world.

    Parameters
    ----------
    color: str
        The background color of the world. This can be a color name
    (such as "magenta"), or a web color hex value (eg "#1A2BFF")
```

The comment first tells us what this function does, and then, crucially, gives us more information about the expected parameter. Function comments are often formatted like this: They include general information about the purpose of the function, and then some information about each parameter (there may be more than one).

The parameter information starts with the line

```
color : str
```

This line tells us the *type* of the parameter.

The type of a parameter tells us what kind of information is expected here. Python uses several different data types to distinguish different kinds of data. These include, for example, text, whole numbers, decimal numbers or boolean (true/false) values. We will explain these types more, later in this chapter.

Internally, each type of data is handled differently, so the Python system will always keep track of what type of data it is currently working with, and we often have to be aware of this type.

**Note** **Colours in Strype** You may wonder why we were able to insert a colour directly from the colour picker when the parameter type is expected to be a string. The reason is that colours are internally always saved as strings. Each colour has a *hex code* – a string representation encoding the colour value. It may look like this: "#39efe7". You can see this code in the colour picker. If you insert a string with a hex code into Strype, it shows you the colour for convenience, but internally the data is stored as a string. Thus, colours in Strype are actually strings.

**Concept** All data in Python has a **type**. This distinguishes different kinds of data, such as *text*, *numbers* and *boolean values*.

In our example above, it shows the name of the parameter (color), a colon, and then the term "str". "str" is short for "string", and it means that this variable expects a segment of text (such as some characters, a word or a sentence).

**Concept** The data type **string** is used to store text, such as a word or a sentence. It is often abbreviated to "**str**". When we specify string values, we always write them in quotes, "like this".

The parameter information in the comment of the prepare function tells us that we should supply a parameter that is a string, and then tells us a bit more about what this string should contain (a colour name or value). Thus, when we call the function, we supply a value for this parameter (in our case, we used the name "MidnightBlue").

When we write string values in Python, we must always enclose them in quotes. Python allows us to use either double quotes or single quotes:

```
print( “This is a string” )
print( ‘This is also a string’ )
```

In this book, we will usually use double quotes for strings.

When we write a function call, the call must match the function definition. If the function definition says that it expects one parameter of type string, then the function call must provide a parameter of type string. Let us look at this situation again in context. In Figure 3.1, we can see that the function definition states that the name of the function is prepare, and it expects a parameter named color. The parameter definition in the function definition is called the *formal parameter* – it tells us what is expected.

The function call then uses the function name to invoke the function, and it provides an *actual parameter* – that is, a value for the expected parameter.

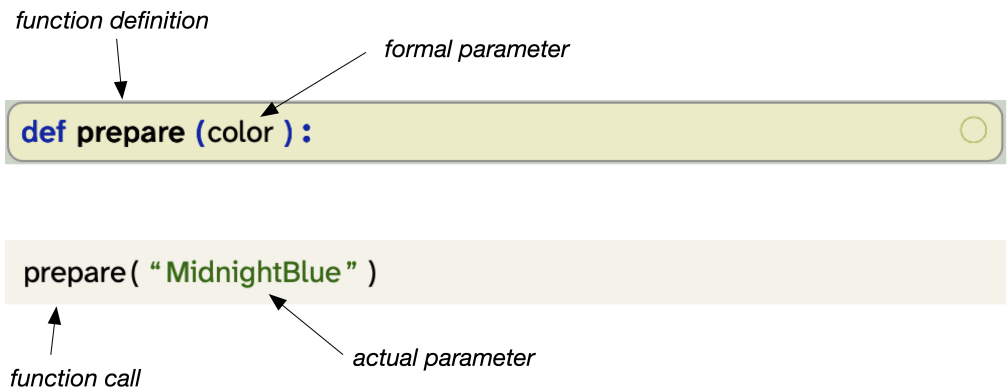



*Figure 3.1: A function definition and a matching function call*

**Concept** The number and type of the **actual parameters** in a **function call** must match the number and type of the **formal parameters** specified in a **function definition**.

**Exercise 3.3** Change the actual parameter in the call to the prepare function to another colour value. Run your program to see the different background colour.

**Exercise 3.4** Remove the quotes from the colour name in your function call. Run your program. What do you observe? (Afterwards, fix the problem again.)

**Note** We use the terms **formal parameter** for the declared parameter in the function definition, and **actual parameter** for the value passed in with the function call. In other books you will sometimes see the terms **parameter** and **argument** instead. In this usage, the argument is the actual parameter, and the formal parameter is just called parameter.

In this book, we prefer the formal/actual parameter terms, since the stand-alone term "parameter" can be ambiguous.

## 3.3 Optional parameters

Let us now look back at the call to the place_rocket function which we added in an exercise above.

**Exercise 3.5** Find the function definition for the place_rocket function which we used earlier. How many parameters does it have?

**Exercise 3.6** What do you notice about the parameter definitions that is different from what we have seen before?

**Exercise 3.7** What do you think the parameters are used for? Try to find out. (Remember that you can unfold the function definition.)

If you have completed the exercises above, you will have noticed that the number of parameters in the place_rocket function call does not match the number of formal parameters in its definition. So what is going on here?

To invoke the function, we have used the call

```
place_rocket( "magenta" )
```

Looking at the definition of the function, we can see that it is

```
def place_rocket (color, target_x=0, target_y=0, delay=0 ):
```

The new (and important) construct here is the =0 specification used for some of the formal parameters. This is called a *default value*. If a parameter has a default value, then we can leave out the actual parameter in our method call, and the default value will be used for the parameter in this case. Therefore, these two calls

```
place_rocket( "magenta" )
place_rocket( "magenta", 0, 0, 0)
```

achieve exactly the same thing, since the values used in the second call are exactly the same as the default values. We can, however, use values different from the defaults for the parameters. If we do, these values are used instead of the defaults.

**Concept** Function definitions may include **default values** for parameters. If they do, then we do not need to provide an actual parameter for this value; the default will be used instead.

The following calls are all valid:

```
place_rocket( "yellow", -200, 100)
place_rocket( "white", 280)
place_rocket( "magenta", 300, 200, 3)
```

Here you can see that we do not have to supply all of the values at the same time: We can override some of the defaults with different values, while leaving the later parameters to their defaults.

If we provide only some values for the parameters, but not all, the actual values are assigned to the parameters in the order of their definitions. So if we provide only one number after the colour name, it will be assigned to the target_x parameter, while the remaining two parameters use their defaults.

But what if we want to provide a value for the delay, and leave the target_x and target_y parameters to their defaults? In that case, we can specify the formal parameter name in the function call, like this:

```
place_rocket( "magenta", delay=2)
```

In this case, the two middle parameters use their default values, while the color and delay parameters receive their values from the function call.

You may have worked out by now that the two middle parameters (target_x and target_y) specify the location in the world area the rocket should aim at (using the coordinate system shown in Figure 1.6), while the delay parameter specifies a delay after igniting until the rocket should fire.

Experiment with this yourself.

**Exercise 3.8** Add some more rockets to your fireworks. Make them explode in different areas of the sky.

**Exercise 3.9** Add some rockets that use a delay after igniting, so that not all rockets fire at the same time. (The delay value is specified in seconds.)

**Exercise 3.10** Create a fireworks show that uses several rockets. It should use at least four different colours. It should also use symmetrical pairs of rockets, where two rockets of the same colour are shot to the right and left of centre, at the same angle from the vertical.

STRYPE TIP

**Freezing functions**: If you look carefully at the function definitions in the *fireworks* project, you see a little snowflake symbol towards the right of the function definition frame. This means that the function definition is ***frozen***. A function is usually frozen if it should not be changed as part of the work in this project. You can freeze or unfreeze a function using the frame's context menu (mouse right-click).

When a function is frozen, its implementation is hidden, and folding/unfolding alternates between showing the function comment, or showing only the function header. When it is unfrozen, the folding cycles through three states: header only, header with comment, full implementation.

If you provide projects for others (such as a teacher making starter projects for their

students), you can freeze a function as a signal that this function implementation should not need change, and that the user may not need to understand the implementation of this function. The function is here just to be called as it is.

If you are a learner working with a project, the frozen function tells you that you can use this function without being expected to understand how it works internally. (If you are curious, you can of course unfreeze a function and look at its implementation, but you should not need to.)

## 3.4 Data types: string, int and float

The first data type we have encountered above was `str`, which is short for "string". We have seen that the values for this type are snippets of text, written in quotes: `"This is a string"`.

If you paid attention when you did the exercises above, you may have noticed that the comment for the `place_rocket` function specified that the type of the last three parameters is `float`. "float" is short for "floating point number", and it means that the value for this parameter should be a number that may include a decimal point. `3.14` and `0.0001` and `42.0` are all floating point numbers. It is allowed to provide numbers without a decimal point: `128`, for example, will just be treated the same way as `128.0`.

There is a third data type that is worth mentioning at this point, because we will encounter it very soon: `int`. "int" is short for "integer", which means a whole number. If a parameter type is `int`, then we have to supply a number, but this number cannot have a decimal point. Only whole numbers are allowed, such as `1001`, `-42`, or `0`.

**Concept** The three most common data types are **str**, **int** and **float**.

**str** (string) is used to store text, **int** (integer) is used to store whole numbers, and **float** (floating point number) is used to store decimal numbers.

Some examples are shown in the table below.

| type name | full name | used for | examples |
|---|---|---|---|
| **str** | string | text | "hello"<br>"a"<br>"this is a sentence" |
| **int** | integer | whole numbers | 402<br>0<br>-9 |
| **float** | floating point | decimal numbers | 0.001<br>-1.0<br>7654321.00002 |

**Exercise 3.11** In your fireworks display, use floating point numbers (numbers with a decimal point) for some of your delay values. For example, delay a rocket by 4.75 seconds.

After doing these exercises, you might like to compare your project to the fireworks-finished example, which is an implementation of these exercises.

## 3.5 Summary

In this chapter, we have investigated and experimented with the most fundamental of Python constructs: the function call. We have learned how to read function headers to work out how we call a function. We have seen that functions can take parameters (which can include optional parameters), and that the types of the formal and actual parameters must match. You should now be able to work out how to correctly call a function when you see the function's header.

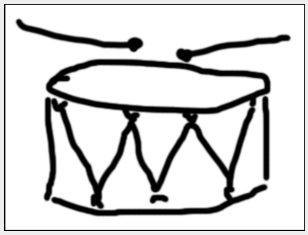
### Concept summary

- A **comment** is a frame that shows text for the human reader(s) of the program. It is ignored by the Python system.
- All data in Python has a **type**. This distinguishes different kinds of data, such as *text*, *numbers* and *boolean values*.

- The data type **string** is used to store text, such as a word or a sentence. It is often abbreviated to "**str**". When we specify string values, we always write them in quotes, "like this".
- The number and type of the **actual parameters** in a **function call** must match the number and type of the **formal parameters** specified in a **function definition**.
- Function definitions may include **default values** for parameters. If they do, then we do not need to provide an actual parameter for this value; the default will be used instead.
- The three most common data types are **str**, **int** and **float**. **str** (string) is used to store text, **int** (integer) is used to store whole numbers, and **float** (floating point number) is used to store decimal numbers.

## Project ideas

**Drum**



This is a very simple project that lets you bang a drum. Use your left/right cursor keys to play.

This project shows you some controlled movement (moving the drum sticks) and the use of functions to move either stick. It also shows you how to play a sound, which we will discuss in more detail later in this book. Maybe you can add another instrument?

# Chapter 4 A closer look at our first Python statements

**Topics:** storing values, conditional execution

**Constructs:** variable, assignment, object, method, if-statement

Before getting back to our yellow-fish game, it will help to also look at some other constructs a bit more closely. In our first two chapters we have already seen variables, assignments, objects, method calls and if-statements.

You will have got a sense of what these statements do, and what they are there for. You will not, however, understand fully how any of them work yet. Don't worry – that is perfectly okay. We will come back to talk more about each of these constructs and experiment with them, and you will gain a full understanding over time.

We will now go through this list and investigate each of these constructs a bit more, so that we can become comfortable with working with these language features before we start looking at new functionality.

## 4.1 Variables and assignment

Often, when we work with bits of data, it is not enough to just pass a value to a function. We often want to store a piece of data for use later on, or for doing further computations with it. This is when we need *variables*.

Concept We can use **variables** to store data for later use. Variables can be assigned a value, and later we can read that value again.

You have seen the first variable briefly at the start of Chapter 1, when we looked at the default program in the Strype editor. Let us briefly go back to this program and

look at it more closely.

**Exercise 4.1** From the Strype menu (click the three lines top-left in the Strype window), choose the "New project" option. This will reset the editor to the default two-line program that you started with at the beginning. (Don't forget to save your program first if you still have unsaved changes open from previous exercises.) Run this program. What does it do?

You will see that the program consists of just two lines of code:

```
myString ⇐ "Hello from Strype"
print(myString)
```

In this code segment, we can now recognise what the second line means: This is a function call to a function named print with one parameter. The print function is built into Python, and it prints its parameter to the Console (the text output area in the bottom right of the Strype window).

The first line is an *assignment statement*. We can add an assignment statement by inserting an assignment frame. A new assignment frame has two slots with an arrow symbol in the middle:

```
identifier ⇐ value
```

On the left side of the assignment, we can write the name of a variable. We can make up this name, and a variable with this name will be created. On the right side we write a value that we want to store in this variable. This can be any type of data that Python knows, including the three types we have seen before: strings, integers and floating point numbers.

In the example above we have stored a string into this variable ("Hello from Strype"), and we have given the variable a name that indicates that it stores a string (myString).

**Concept** An **assignment** statement stores a value into a variable. The variable name is written on the left, followed by an arrow symbol, and the value to store is written on the right.

**Exercise 4.2** Change the content of the string to a different string. Run your program.

**Exercise 4.3** Write quotes around the word myString in the print method call, so that the call looks like this: `print("myString")`. Will this work? Run your program. What do you observe? Explain what you see. (When you are finished, remove the quotes again.)

**Exercise 4.4** Change the value in the assignment (the right hand side) to a number. Run the program. Does this work?

**Exercise 4.5** Once you have changed the value on the right to a number, the name of the variable is misleading: It is called "myString", but it does not hold a string. Change the name of the variable so that it accurately describes what it stores.

**Exercise 4.6** Change the value on the right side of the assignment to `42 + 33`. What do you think this will do? Try it out, and explain what happened.

Variable names must not have any spaces in them. If you need to use multiple words (for example, *big fish*), there are two programming conventions to do this:

- Use underscores in place of spaces: *big_fish*
- Capitalise each new word: *bigFish*

In this book, we generally use the first format.

## 4.2 Objects

In the previous section, we have seen that we can store values, such as strings or numbers, into variables. Next, let us look at a special kind of data type to store and work with: *objects*. To investigate this topic, we will use a project called *knock-knock*.

**Exercise 4.7** Open the knock-knock example from the Chapter 4 projects. Run the project. Then read the code. Can you explain what the code does and how the project works?

**Exercise 4.8** Change the dialog between the lobster and the crab. For example, you might find another knock-knock joke on the internet and use that one instead.

**Exercise 4.9** Replace the images of the crab and lobster with different images.

This project makes use of an *Actor object*, which we first encountered in Chapter 1. The creation of this Actor object looks like this:

```
Actor(  , -220, 0)
```

We can see that it looks very similar to a function call: It has a name, followed by a parameter list in parentheses. Because these look so similar, Python uses a convention: function names start with a lowercase character, while object types start with an uppercase character. This bit of Python code creates an object of type Actor, using the parameters provided.

In our code, we can see that we then assign this object to a variable, so that we can use it later:

```
crab ⇐ Actor(  , -220, 0)
```

This shows us that variables can also be used to store objects as values.

Before we go on and examine the rest of the program, let us ask a question: How do we know what object types and what functions exist for us to use?

Looking at our *knock-knock* program, we can see that it uses objects of type `Actor`, and functions such as `set_background` and `pause`. What other objects and functions are available for us, and how do we find out?

## 4.3 Reading the library documentation

When working in Strype, we will often make use of objects and functions from the Strype graphics library. Over time, we will have to become familiar with most of the functions available in it. So how can we see what it offers?

The answer to this lies in the *library documentation*. This documentation lists all object types and functions, with all their parameters and information how to use them. In Strype, you can access this documentation by selecting "Library documentation..." from the Strype menu.

**Exercise 4.10** Select the "Library documentation..." option from the Strype main menu. Examine the webpage you see. Find the documentation for the `pause` function. How many parameters does it have? What are the types of its parameters? What do its parameters specify?

**Exercise 4.11** What are the names of the special keys that you can use with the `key_pressed` function?

**Exercise 4.12** How many parameters does the Actor's `move` method have?

**Exercise 4.13** In the documentation for the Actor's `move` method, what does it tell you about negative parameter values?

Learning to read this documentation will be very useful for our future programming tasks. It enables us to find out what we can do. Luckily, it is not very difficult.

The main points to understand are these:

- The heading for this documentation is *Strype API documentation*. "API" is short for "Application Programming Interface", which is a technical term for the functions available in a library.
- In the list on the left, we can see that this library contains three *modules*. They are called *strype.graphics*, *strype.sound* and *strype.builtins*. Each module provides functions for a specific purpose, and each can be imported into our programs separately. We will investigate the *strype.sound* module later. For now, let us concentrate on the *strype.graphics* module.

- We have mentioned before that we can distinguish functions and object types by their initials: Object types start with an uppercase letter, while functions start with a lowercase letter. Knowing this, we can see that this module contains three object types (*Actor*, *Color* and *Image*), and a list of about a dozen functions listed below them.
- Object types are also called *classes*. You can see this in the documentation when you, for example, click on Actor in the list on the left and then look at the details of the Actor definition on the right. Its first line starts with class Actor to tell you that this specifies a class – a type of an object. We will, from now on, also use the term "class" for types of objects.
- We can click on each of the functions in the list on the left to see a detailed description of the function, including its parameters.
- We can also see the *methods* of the classes. For example, we can see the move and turn methods of the Actor class, which we have used in Chapter 1.

**Concept** The type of an object is called a **class**. Classes define **methods** that can then be invoked on the objects of that class.

You will quickly get used to reading this type of documentation. Having it available will help us both understand the remaining parts of the *knock-knock* program, as well as enable us to find out what else we could do.

**Exercise 4.14** In the pace function, what is the default speed used when no actual parameter is provided?

**Exercise 4.15** What does the stop function do?

**Exercise 4.16** What is the difference between the say and the say_for methods of the Actor class?

We have now seen that functions and classes can be defined in two different places: They can come from a library (such as the set_background function we have used from the *strype.graphics* library) or they can be defined in our own project, as we have seen in the fireworks example.

## 4.4 Methods

In Chapter 1, we have briefly mentioned the difference between *functions* and *methods*. They are similar in that they both perform specific actions. Functions, however, perform a stand-alone action (such as `set_background("red")` or `ignite()`), while methods are actions that belong to a class and are performed by a specific object. We have seen for example, that we write

```
fish.move(6)
```

to make our fish object move. `move` is a method of the `Actor` class, as we have seen in the library documentation. There, methods are listed under the class, while functions are listed on their own at the end of the module.

**Exercise 4.17** The *knock-knock* project contains these two lines:

```
crab.say_for("Knock, knock", 2)
pause(2)
```

Which of these lines is a function call, and which is a method call? How can you tell?

**Exercise 4.18** What does the `say_for` method do? What is its second parameter?

**Exercise 4.19** Make each of the actors (the crab and the lobster) turn a bit after every time they say something. The crab should turn left, and the lobster should turn right every time they speak.

*Figure 4.1: The fat cat*

## 4.5 The "Fat Cat" example

To continue experimenting with objects and their methods, we will use a different project: the *Fat Cat* project (Figure 4.1). This project defines its own class: Cat.

**Exercise 4.20** Open the fatcat example. Run it. What do you observe?

**Exercise 4.21** How many methods does the Cat class have?

**Exercise 4.22** How many parameters does the sleep method have?

**Exercise 4.23** Make the cat walk a bit further than it originally did.

**Exercise 4.24** Make the cat walk left instead of walking right.

**Exercise 4.25** Make the cat eat.

You can see that this project defines its own class called Cat. This class is also an actor, so it is also shown in the graphics world when we create an object of this class. We can also see that the Cat class defines a number of methods which we can call on the cat.

If you have paid attention, you will have noticed one oddity: the formal parameter called self as the first parameter of all the Cat's methods. This special parameter is used only in definitions of methods within classes, not for functions. It is needed in

the method body to implement the method, but it is not used when calling the method. So the Cat method

```
def move (self, distance ) :
    " Move forward ...
    ...
```

is called using the following method call:

```
cat.move(3)
```

In other words: This parameter is used only when defining methods, but when calling methods it is ignored. Since we are – at the moment – concerned only with calling methods, we can just ignore this parameter. (In the library documentation, this parameter is not listed at all, so we are implicitly ignoring it there as well.)


When looking at method definitions in the Strype editor, we see the implicit "self" parameter. We can, for now, ignore this parameter.

Let us now experiment a little more with our cat. Instead of calling just a single method, we can also call a sequence of methods. Try this with the following exercises.

**Exercise 4.26** Make the cat walk to the left, then make it eat.

**Exercise 4.27** Make the cat dance, then sleep.

**Exercise 4.28** Create a sequence of actions of your choice for the cat. The cat should do at least four different things.

**Note** If you look carefully, you can see another odd-looking construct: For classes, the formal parameters for the construction of an object of that class are not shown after the class name, but in a special method called __init__ (note the two underscores each side of init).

This, too, is specific to classes: If the class name is Cat, and the init method has the formal parameter list (self, x, y), then we create an object of this class like this:

```
cat ⇐ Cat(50, 60)
```

That is, to create the cat object, we use the name of the class and the parameter list from the init method, ignoring the formal parameter self. This, again, has to do with the way classes are implemented in Python. You will see this notation only when you look at a class in the editor. When you look at the class documentation (in the library documentation), the same class header would be shown as Cat (x, y).

## 4.6 If-statements

Looking at the Cat methods, we can see a number of methods starting with the prefix is_, for example is_hungry, is_tired, and so on. These methods return a value of either True or False, and we can use them to introduce conditional actions: make our cat do something only if a given condition is true.

In Chapter 1, we have already seen that we can use if-statements to call a method only under certain conditions. In this case, we can, for example, make the cat eat only if it is hungry:

```
if cat.is_hungry( ) :
    cat.eat( )
```

Try this yourself.

**Exercise 4.29** Make the cat eat if it is hungry. If it is not hungry, it should do nothing.

**Exercise 4.30** Change your program so that the cat dances if it is bored.

**Exercise 4.31** Change your program again to do the following: If the cat is tired, it sleeps, and then it shouts hooray. If it is not tired, it just shouts hooray. (For testing, call another method first to make the cat tired. How can you make the cat tired?)

**Exercise 4.32** How can you make the cat hungry? When will it be bored? Test it by writing a program that shows this.

**Exercise 4.33** Change your program to do the following: If the cat is alone, let it sleep. If it is not alone, make it shout hooray. Test this by creating a second cat at the beginning of your program. The second cat should be at a different location than the first one, and its variable should be called cat2.

**Exercise 4.34** Extend your program so that the second cat dances if is not alone.

If-statements can also have an *else-case*. An else-case is added to the if-statement by moving the frame cursor to the end of the body of the if-statement (the last line within the if-statement), and then inserting an else frame. The result is an extended if-statement that looks like this:

```
if cat.is_hungry( ) :
    cat.eat( )
else :
```

The if-statement now has two bodies: one for the if-case, and one for the else-case. The first one is executed when the condition is true, and the second one is executed if the condition is false. Thus, one or the other of the bodies will always be executed, but never both.

Concept An if-statement may have an **else-case**. The else case is optional. If it exists, it is executed when the condition is false. If the if-statement has no else case, nothing happens when the condition is false.

**Exercise 4.35** Rewrite your program to use an if-else-statement.

**Exercise 4.36** Make a program with three cats. Make one cat eat, the second one sleep, and the third one dance.

**Exercise 4.37** Invent your own routine for the cat (or two cats), and implement it.

You can find a version of the project that does some of these things in the book projects as fatcat-v2.

If you have done all the exercises in this chapter, you will have a good enough understanding of the Python constructs we have used thus far. We are now ready to get back to our game project and continue developing it into something more interesting.

## 4.7 Summary

In this chapter, we have investigated and experimented with some of the most important Python constructs. We have seen more detail about assignments, and also discussed how to use Actor objects, and how to read the library documentation to find out more detail about what we can do with them. The documentation will be important when we work with our projects; it shows us the methods we have available.

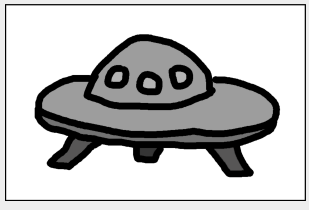
And finally, we had a closer look at if-statements to execute specific statements only if a certain condition is true. If-statements will be useful in just about every project we write from now on.

## Concept summary

- We can use **variables** to store data for later use. Variables can be assigned a value, and later we can read that value again.
- An **assignment** statement stores a value into a variable. The variable name is written on the left, followed by an arrow symbol, and the value to store is written on the right.
- The type of an object is called a **class**. Classes define **methods** that can then be invoked on the objects of that class.
- An if-statement may have an **else-case**. The else case is optional. If it exists, it is executed when the condition is false. If the if-statement has no else case, nothing happens when the condition is false.

## Project ideas

**Space**



The space project is a very simple example of fixed, four-directional movement. In the *yellow-fish* example, we were able to turn and move in any direction we chose. Here, we see an example where the game character (a space ship, in this case) can move up, down, left and right, but does so without turning. (It is an extension of the two-dimensional movement from the *pong* game earlier.) Maybe this is useful for your own game idea.

# Chapter 5 Creating a game

**Topics:** adding more actor types, repetition, random behaviour

**Constructs:** loop, for-loop, random numbers

We are now, finally, ready to come back to our yellow-fish project to start turning it into a playable game.

If you have completed all the tasks in Chapters 1 and 2, you can continue with your project from those chapters. If you would like a fresh start, you can use the yellow-fish-v3 project from Chapter 2 as your starting point.

## 5.1 The plan

To make this game even a little bit interesting, we need to have a task, a goal, and some danger. For our yellow fish, we have already seen how we can make the fish the player character by letting the player control the movement of the fish. The task will be to eat: We will introduce some shrimp to our game, which the fish can eat. The goal of the game then is to eat all the shrimp on screen.

To make this a challenge, we will also add a shark: the shark likes to eat fish, so we need to stay away from it, otherwise we will be eaten by the shark. The aim of the game then is to manage to eat all shrimp without being first eaten by the shark.

## 5.2 Adding food

Our first task is to add some food to the game: a shrimp in our case. We can start by adding one single shrimp. This should now be quite easy: We can add a line of code to add the shrimp immediately after the line that adds the fish, with a very similar structure. We just use a different name for the variable, different coordinates and a different image:

```
fish ⇐ Actor(  , 0, -200)
shrimp ⇐ Actor(  , 100, 50)
```

Of course, if you have changed your player character and image in this project to a different actor, you can choose a different kind of actor for the food as well. If your player is, for example, a turtle, then the food could be a lettuce. Or if your player character is a spaceship, then the second character could be an astronaut. (In that case, it is not "food", and the player is not "eating" the second character, but rather picking them up from space.) You get the idea: whatever your game scenario is, choose an appropriate image for your second character. Whenever we write "shrimp" in the remainder of this chapter, use your own character instead.

In case you would like to stick with our fish/shrimp theme, you can find a shrimp image in the Chapter 5 section of the book projects in Strype.

**Exercise 5.1** Add a shrimp to your project. Place it at a different location from the fish. Test to make sure that the fish and shrimp both appear on screen.

**Exercise 5.2** Add a second shrimp at another location. Then add a third one.

## 5.3 Repetition – the for-loop

Let us say we would like to have 15 shrimp in our game. We could now go ahead and duplicate the line that creates the shrimp another 14 times, each time changing the world coordinates to different values. This would work, but it would be tedious.

Computers are good at doing things repeatedly, but that does not mean that we should need to write the same statement repeatedly. We can just tell the computer to execute a statement 15 times. (This will become even more important later on when we will execute statements not 15 times, but thousands of times. In that case, it is really not practical to write all the statements out one by one.)

**Concept** A **loop** is a statement that can execute another statement (or a set of statements) repeatedly, potentially many times. The **loop body** contains the statements which we wish to repeat, and the **loop header** determines how often it should run.

The type of Python statement to do something repeatedly is called a *loop*. Python has several different kinds of loop statement, and the one we will use here is called a *for-loop*. It looks like this:

```
for count in range(0, 15) :
    shrimp ⇐ Actor(  , 100, 50)
```

**Exercise 5.3** Remove all but one of the statements that create the shrimp. Place the remaining one in a for-loop as shown here. Run your program. What do you observe?

**Exercise 5.4** How many shrimp are created by this loop? How many shrimp can you see on screen? Can you explain what you see?

The for-loop executes the statements within it repeatedly. (There can be more than one statement in the body.) How often it executes depends on the header of the loop. The loop we see above counts from 0 to 15, and for each count executes the loop body (the statement within the loop) once.

Let us discuss this in a bit more detail. The empty for-loop frame looks like this:

```
for identifier in list :
```

In the place of the expected variable, we can write the name of a variable we want to use for counting. We can make up the name for the variable ourselves, and a variable with that name will be created. In our example above, we have used `count` as the variable name, but we could have named the variable anything we like. (In programming, the variable name `i` is also often used as a loop counter.)

In the 'list' slot, a list of numbers is expected. Here, we use the range function: this function creates a list of numbers from a lower bound to an upper bound. In our case we have used 0 and 15, so we will get a list of numbers from 0 to 15. One important detail to be aware of is that the range *includes* the lower bound, but *excludes* the upper bound. So strictly speaking, the numbers we get are 0, 1, 2, ..., 14. This suits us well, because it means that we are getting 15 different numbers (0 to 14), and the loop will run 15 times.

When the loop executes, it first assigns the first number (0) to the loop variable count and then executes the loop body once. Then it assigns the second number (1) to count and executes the body again. Then again for the third number, and so forth.

Once it has done this for the last number, the loop body will have been executed 15 times. For each run of the loop body, the count variable has a different value, and we can use this value to our advantage to solve our next problem.

Concept A **for-loop** is a type of loop statement that uses a counter variable (called the **loop counter**) and a list. It executes the loop body once for each item in the list.

## 5.4 Using the loop counter

In the last exercise above, you will have noticed a problem: The code created 15 shrimp, but they were all placed at the same location. Because all shrimp were drawn exactly on top of each other, it made it look as if there was only one.

We can fix this by placing each shrimp at a different location. How do we do that if all 15 shrimp are created using the same statement?

The answer is not to use fixed values for the x- and y-coordinates, but to use variables instead.

As our first try, let us use the loop counter variable as the coordinates for both the x- and y-position:

```
for count in range(0, 15):
    shrimp <= Actor(  , count, count)
```

Using this code, the x/y coordinates will be 0,0 for the first shrimp, 1,1 for the second, and so on. This way, the shrimp are not all drawn at the same location.

**Exercise 5.5** Implement the loop as shown above, using the `count` variable. Run your program. What do you observe? Explain what you see.

**Exercise 5.6** To space the shrimp out more, multiply the `count` variables by 20 when you use them for the coordinates. That is: write `count * 20` as the actual parameter for the x- and y-coordinates.

**Exercise 5.7** Experiment with values other than 20 for the multiplication.

**Exercise 5.8** What effect does it have if you use different multipliers for x and y? What happens when you add or subtract a value from x or y?

**Exercise 5.9** Make the row of shrimp start from the left edge of the screen.

**Exercise 5.10** Arrange the shrimp in a straight line from the top left corner of the screen to the bottom right.

**Exercise 5.11** Python allows us to call the `range` function with only one parameter, for example `range (15)`. In this case, the parameter is the upper bound, and the lower bound is automatically set to zero. Change your program to use the one-parameter version of this function call.

In the exercises above, we have seen that we can use the `*` symbol for multiplication. Python has built-in operators for many frequently used mathematical operations; you can see a list of all available operators in Appendix D: Python operators.

For the purposes of our game, being able to place the shrimp at different locations is good, but placing them in such a regular pattern seems odd in our context. So let us make the next improvement: Let's place the shrimp at random locations.

## 5.5 Creating random behaviour

Often, we want an element of randomness in our programs. In our case, we would like to place our shrimp at random locations, but we may also want other characters to move randomly, or have interesting things happen at random times. Randomness makes our game less predictable, and thus more fun to play.

In computer programs, all random behaviour is based on random numbers. Python can create random numbers, and we can then write code to translate this into any kind of random behaviour we need. We will see an example here for the random placement of the shrimp, and another example later in this chapter, when we want to make our shark move randomly.

Python provides a function called randint to generate and return random integers (whole numbers). We can call this function to receive a random number and assign it to a variable. For example, we can write an assignment statement like this:

```
x ⇐ randint(5, 19)
```

The function call to randint returns a random number, and we assign this number to a variable called x.

The function takes two parameters, which are the lower bound and the upper bound of the random numbers we wish to receive. Both bounds are included in the possible range, so our example will produce numbers between 5 and 19, inclusive.

**Concept** Python provides functions to generate **random numbers**. The **randint** function provides a random integer from a given range.

**Exercise 5.12** Add the assignment statement with the randint function call to your own code, inside the for-loop. Run your program. What do you observe?

The exercise above shows us that we are not quite done yet: The code produces an error that tells us that randint is not defined. This is Python's way of telling us that it does not know the randint method.

The reason for this is that the randint method is defined in a *library*. Python provides so many functions for different purposes that it might get confusing to make all of them available all the time. Instead, they are arranged into libraries, which we can import when we need them. Each library contains a set of functions for a specific purpose. Python provides a set of libraries by default with the language – these are referred to as the *standard libraries*. They are always available in every Python system. Other libraries can be added with explicit statements.

To make our randint function work, we have to first import it from a library.

## 5.6 Importing from a library

When we import from a library, we have the option of importing all the definitions in that library, or we can import specific functions.

We have seen an example of the first option in Chapter 1, when we looked at the import of the Strype graphics library. We saw the following import statement in the *Imports* section of our program:

```
from strype.graphics import *
```

In this case, we used the asterisk (*) to specify what we wish to import, which is a symbol meaning "everything". Thus, we imported the entire Strype graphics library.

We can now add the following statement to our *Imports* section:

```
from random import randint
```

This statement shows us that Python has a standard library called random (which contains various functions dealing with random numbers), and from there we can import a function called randint. In this import statement, we import just a single function from the library.

As a general rule of thumb, we should import separate named functions whenever we need just one or two functions from a library, and we import everything when we need access to a large number of definitions from the same library.

**Exercise 5.13** Add the import statement for randint to the *Imports* section of

your program as shown above. Run your program. This should now work. If it does not, then you made an error that you need to fix. You will not see any effect of the random number yet, but you should not see an error either.

## 5.7 Random placement

It is now time to use our random numbers for random placement of our shrimp.

**Exercise 5.14** Add a second assignment of a random number to a variable. This time, call the variable `y`. Add this line directly below the assignment to `x`. Make sure both lines are inside your for-loop.

**Exercise 5.15** Use your variables `x` and `y` as the actual parameters for the x- and y-coordinates in the creation of your Actor.

**Exercise 5.16** Experiment with different values for the upper and lower bounds of your random numbers. Remember that the x-coordinates of the Strype world range from -399 to 400, and the y-coordinates from -299 to 300 (see Figure 1.6). What would be reasonable bounds for your random numbers?

**Exercise 5.17** Start and stop your program multiple times to convince yourself that the shrimp are placed in different random locations each time.

If you have successfully completed the exercises, you now have a version of your program that has a keyboard controlled player character and several actors of a second type at random locations. If you would like to compare your version to ours, you can find a version of the program in this state in the book projects as yellow-fish-v4.

In our version, we have made one additional change: instead of defining the variables `x` and `y` and assigning random numbers to them, we have written the call to the random number function directly into the parameter list of the Actor creation:

```
for count in range(0, 15) :
    shrimp ⇐ Actor( <image> , randint(-360, 360) , randint(-260, 260) )
```

This is an alternative way to achieve the same thing: we can write a function call that returns a value directly into a parameter list, and the result of the function will be used as the actual parameter.

Which version of the program you prefer is a matter of personal preference – both are good ways to achieve the same goal.

## 5.8 Summary

In this chapter, we have added an important step to our game: more characters for us to interact with.

We have seen how we can use a for-loop to create several actors without writing separate statements for each one. We have also seen that we can import functions from the random module to generate random numbers. These numbers allow us to implement random behaviour in our programs.

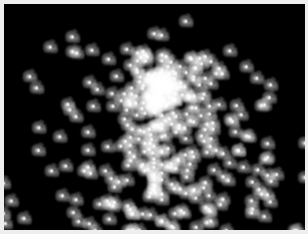
We are now ready to make our fish eat the shrimp – we will do that in the next chapter.

### Concept summary

- A **loop** is a statement that can execute a statement (or a set of statements) repeatedly, potentially many times. The **loop body** contains the statements which we wish to repeat, and the **loop header** determines how often it should run.
- A **for-loop** is a type of loop statement that uses a counter variable (called the **loop counter**) and a list. It executes the loop body once for each item in the list.
- Python provides functions to generate **random numbers**. The **randint** function provides a random integer from a given range.

## Project ideas

**Particles**



Here is a project that shows a special effect. It is called particles, and it shows a sparkler effect around the mouse position. It demonstrates a few useful things: it shows how to create lots of actors to create a visual effect (and make them disappear just as quickly), it shows how to implement a gravity effect (the sparkles fall down, and accelerate as they do), and it demonstrates how to use a Gaussian distribution fors random numbers (where more extreme numbers are less likely). It also shows how to read the mouse coordinates to use them in your program.

Altogether it might give you ideas for visual effects you may like to add to your own games.

# Chapter 6 Collision detection

**Topics:** interacting actors, collision detection

**Constructs:** bool type, return value

In the previous chapter, we have added the shrimp for us to eat. It is now time to investigate how we actually implement the eating. The idea is simple: we just want to remove the shrimp from the game when our fish moves over them.

To implement this, we need to learn about a concept in games: *collision detection*.

## 6.1 Detecting collisions: is_touching

*Collision detection* is the technical term for detecting when one actor (the fish) touches another actor (a shrimp).

**is_touching**(*actor_or_tag*)

Check if this actor is touching the another actor.

Two actors are deemed to be touching if the bounding rectangles of their images are overlapping (even if the actor is transparent at that point).

The parameter can be either a specific actor (of type `Actor`) or a tag. If a tag is used, this function will return True if any actor with this tag is touched.

| | |
|---|---|
| **Parameters:** | **actor_or_tag** (*Actor* / *Any*) – The actor (or tag of an actor) to check for overlap. |
| **Returns:** | True if this actor overlaps that actor (or an actor with the given tag), False if it does not. |
| **Return type:** | bool |

*Figure 6.1: The documentation of the* is_touching *method*

The Strype graphics library provides a method to help with this: the is_touching() method from the Actor class. Figure 6.1 shows the documentation of this method (you can also find this in the library documentation in Strype).

We have previously discussed how to read the header of the method: We know to use the method name to invoke this method, and to provide a matching number of parameters. Let us first take a closer look at the parameter of this method.

We can see that we can provide either an actor (using a variable that holds an actor object) or a "tag" as a parameter. A tag is a label that we can provide for actors to identify them later. In our case, we wish to check for collision not just with a single actor, but with any of the shrimps. To achieve this, we will tag all shrimp objects as "shrimp" and then check for that tag.

```
Actor(image, x=0, y=0, tag=None)
```

*Figure 6.2: The header of the Actor class*

## 6.2 Tagging actors

Figure 6.2 shows the header of the Actor constructor, and we can see there that it has an optional fourth parameter: the tag we have just mentioned.

We can use this to attach a tag to our shrimp objects to identify them later:

```
shrimp ⇐ Actor(  , x, y, "shrimp" )
```

The tag could be of any type, but it is most common to use a simple string to mark a set of actors so that we can easily recognise them later. In the method call to check whether we are touching a shrimp actor, we can then write:

```
fish.is_touching( "shrimp" )
```

and this call will return true only if we are touching an actor that was tagged with the string "shrimp". This would ensure, for example, that we are not eating other fish, should we decide later to introduce other fish into our game, or the shark once we add that. Note also that this is a method – called on an object – not a stand-alone function, so we call it on the fish object. This makes sense, since it is the fish which wants to check whether it is touching another actor.

## 6.3 Return values

We should now also pay closer attention to the *return value* of the method. We

already know that the documentation typically includes information about the general function of the method, and about its parameters. Here, we can see that the documentation also includes information telling us what the method returns to us.

**Concept** Functions and methods may return a value to the caller of the function. This is called the **return value**. Like all data in Python, the return value has a type. This is called the **return type** of the function.

We have already seen earlier that some methods and functions return a result to the caller of the function. It is time to see how we can find out about this from the documentation.

Figure 6.1 shows two bits of information about the return value, in its two last lines. It first tells us what it returns, and then it tells us the type of the return value. The first line tells us that the method will return the value `True` if this actor touches a specified other actor, and `False` if it does not. The type in this case is a data type we have not discussed before: `bool`.

The *boolean* type (in Python abbreviated to `bool`) is a type that can only hold two possible values: `True` or `False`.

In Chapter 3, we have already seen the types `str`, `int` and `float`. To this list, we should now add our boolean type:

| type name | full name | used for | examples |
|---|---|---|---|
| **bool** | boolean | true/false values | True<br>False |

**Concept** The **bool** type provides a type for data that can only hold two possible values: **True** or **False**. It is used, for example, as the expected type for the condition in an if-statement.

When a method returns a value, we typically do something with that value. One option is to store it in a variable:

```
found_shrimp ⇐ fish.is_touching( “shrimp” )
```

For method calls returning boolean values, we have already seen before that we can also use them directly in if-statements. Figure 6.3 illustrates this: The condition in an if-statement needs an expression that returns a boolean value (true or false), and our method call returns just such a value.

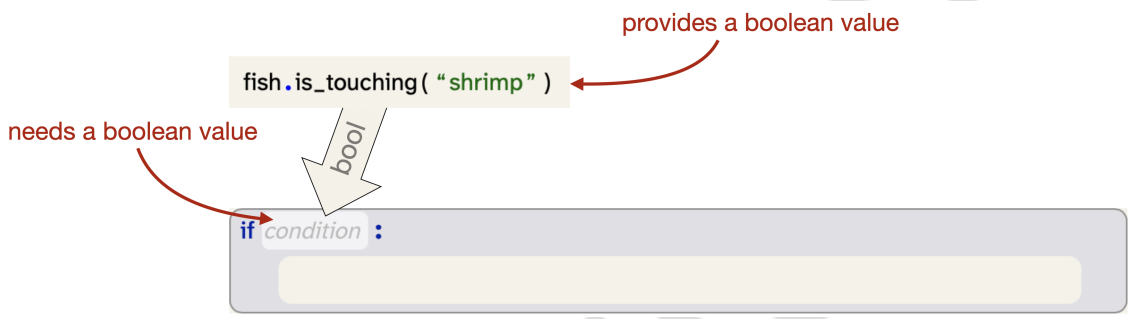



*Figure 6.3: Using a boolean function to provide a condition for an if-statement*

If you have done the exercises in Chapter 4, you have, in fact, made use of this before when you worked with the *Fat Cat* project. There, too, we used return values from method calls as conditions in our if-statements.

**Exercise 6.1** Look through the documentation for the Actor class. What other methods can you find that return boolean values?

**Exercise 6.2** What methods can you find in the Actor class that return int values?

**Exercise 6.3** In your program, add the "shrimp" tag to your shrimp actors. You do this by adding an additional parameter to the creation of the shrimp, as shown above.

**Exercise 6.4** Add an if-statement to your program to check whether the fish is touching a shrimp, as discussed. In the body of this if-statement, call the method fish.remove_touching("shrimp") to remove the shrimp if we have run into it. Where should this if-statement go?

**Exercise 6.5** Test your program. The fish should now remove the shrimp when it swims over it. If this is not happening in your program, go back and check

your code. Have you added the right tag to the shrimp? Have you used the same tag in your `is_touching` check?

**Exercise 6.6** In the exercise above, we have used the `remove_touching()` method. What does this method do? What does it do when the actor is not currently touching another actor? What does it do when it is touching two other actors at the same time?

**Exercise 6.7** If the fish touches two shrimp at exactly the same time, will they both be eaten? Explain your answer.

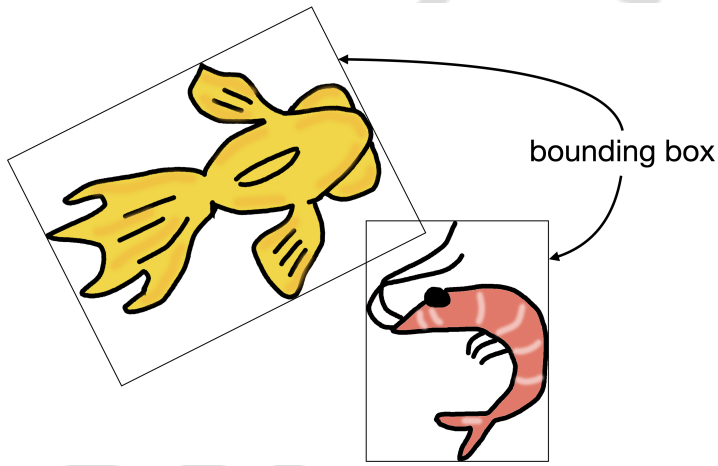



*Figure 6.4: Two images and their bounding boxes*

## 6.4 Bounding boxes

If you have watched closely when playing with your project, you may have noticed that often the shrimp disappear just before the fish seems to touch them. This has to do with how graphics are managed in many computer graphics systems (including Strype).

In Strype, even though images appear to have free form, all graphics drawn to the screen are, in fact, rectangular. Figure 6.4 illustrates this: it shows the *bounding box* of each image. The bounding box is the actual boundary of the graphic being painted. The image appears to be of arbitrary shape by using transparent pixels: the parts of the image where we do not want to show anything are transparent, making them essentially invisible. For the computer, they are, however, still part of the image, even though we cannot see them.

For the purposes of collision detection, two actors are "touching" when their images overlap. Since the invisible pixels are part of the image, this means that they are effectively touching when their bounding boxes overlap. The effect is that they are often technically "touching" even though the visible images do not touch each other.


When you make your own images for use with actors, make sure to crop any unnecessary transparent area around the visible part of the image. If an image has transparent padding, it increases the size of the (invisible) bounding box and makes collision detection less precise.

## 6.5 Adding a predator

Now, let us make the game more interesting by adding a predator that hunts the fish: a shark. An image for this is in the Chapter 6 section of the book projects, but as always, you are free to use your own actor type and image for this. For example, if your player actor is a spaceship, then the "predator" might be an asteroid which you have to avoid hitting.

The beginning of this is easy: at the beginning of the game, where we create the fish and shrimp, we now add a line to create a shark:

```
shark ⇐ Actor( , -200, 100)
```

Next, we add code into our main loop (after all the code dealing with our fish) to make the shark move:

```
shark.move(12)
```

It might be a good idea to make the shark move a bit faster than the fish to increase the challenge.

**Exercise 6.8** Make the additions shown here to your own program: Add the shark and make it move.

Of course, at the moment the shark moves only in a straight line, and it gets stuck at the right edge of the screen. To fix this, we want to make the shark turn when it reaches the edge of the screen. The Actor class has a method to check for this: the is_at_edge() method returns True if the actor is at the edge of the world.

**Exercise 6.9** Look up the is_at_edge() method in the documentation. What is its return type? What does it mean, precisely, to be "at the edge" of the world?

**Exercise 6.10** In your program, after the method call that makes the shark move, add an if-statement. Use the shark.is_at_edge() method for the condition of the if-statement, and make the shark turn 15 degrees if it is at the edge.

**Exercise 6.11** Is 15 degrees enough to turn away from the edge? Will the shark manage to turn enough to continue? Why not turn 180 degrees? Experiment with different degree values for the turn and see what the effect looks like. Explain what you see.

Next, we should make the shark eat the fish when they meet. So let us add some collision detection checking whether the shark touches the fish. This is now not too difficult, because it closely mirrors the code for the fish eating the shrimp:

```
if shark.is_touching("fish"):
    shark.remove_touching("fish")
```

We need to remember, of course, to tag the fish with the "fish" tag when we create it, so that it will be identified.

**Exercise 6.12** Add code to your main loop to make the shark eat the fish. Test your program to make sure this works.

We now have the beginnings of a simple game: We have a keyboard-controlled character that has a task and has to avoid being caught. (This version of the program is in the book projects as yellow-fish-v5.)

If you have programmed along with your own images and characters, your story might, of course, be different: you might have a game where you control a spaceship

that picks up astronauts, or you might be controlling a white blood cell that removes bacteria and tries to avoid a virus. You can see that very similar game playing code can be used to present different stories.

There are many ways in which we can now improve this game and make it more interesting, and we will suggest some shortly.

## 6.6 Summary

In this chapter, we have made good progress in developing our yellow fish game. In fact, we are getting close to having a playable game now.

We have investigated collision detection of actors, using the `is_touching()` method. This has allowed us to implement eating the shrimp, and the possibility of being eaten by the shark.

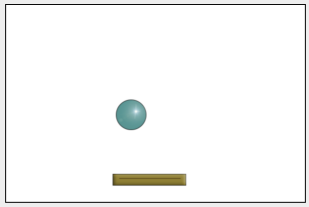
In the process of this, we have encountered return values and the `bool` type, and we have seen some more examples of using methods that return a boolean as a condition in an if-statement.

### Concept summary

- Functions and methods may return a value to the caller of the function. This is called the **return value**. Like all data in Python, the return value has a type. This is called the **return type** of the function.
- The **bool** type provides a type for data that can only hold two possible values: **True** or **False**. It is used, for example, as the expected type for the condition in an if-statement.

## Project ideas

**Breakout**



Breakout is another classic arcade game. (Again, search online for "breakout game" if you are not familiar with it.)

In this version, only the paddle and the ball have been implemented so far. To complete the game, you would have to add the rows of bricks at the top and implement the game logic. Of course, many visual and audio effects can be added.

# Chapter 7 Writing good code

**Topics:** program structure, readability, defining functions, naming

**Concepts:** abstraction, refactoring

The obvious next step now is to forge ahead and make our game more interesting. You can probably think of all sorts of things that you could now add to make your game more exciting. Every new idea results in new code, and our program will get longer and longer. That is normal. But we will slowly get into a problem: long programs are hard to read and difficult to understand.

Therefore, before we make further additions, we will discuss how we can manage this. By using a few techniques to maintain good code structure, we can avoid this problem, and ultimately achieve much more with our game.

So let us take a step back and reflect on our code first, and then go on to add more functionality in the next chapter.

```
set_background("LightBlue")

fish ⇐ Actor(  , 0, -200, "fish")
for count in range(0, 15):
    shrimp ⇐ Actor(  , randint(-360, 360), randint(-260, 260), "shrimp")
shark ⇐ Actor(  , -200, 100)

while True:
    fish.move(8)
    if key_pressed("left"):
        fish.turn(5)
    if key_pressed("right"):
        fish.turn(-5)
    if fish.is_touching("shrimp"):
        fish.remove_touching("shrimp")
    shark.move(12)
    if shark.is_at_edge():
        shark.turn(15)
    if shark.is_touching("fish"):
        shark.remove_touching("fish")
    pace(30)
```

*Figure 7.1: Our program so far: moving the fish and shark*

## 7.1 Analysing program structure

So far, our code is not too bad. Our program still spans only about a page (Figure 7.1), and with some effort and practice we can manage to read and understand it. Over time, however, this will get worse. Our program is still very simple, but as we add more improvements, its size will quickly grow. We will soon deal with programs that are not one page long, but dozens of pages. In professional systems, programs often span thousands or millions of lines of code.

To be able to manage longer programs, we need a way to structure our program text better. And while our program is still fairly short, it is never too early to start with this. Developing a good program structure is a habit we should get into from the start, because later on it will be essential. When we deal with larger projects later on, we would not be able to work with them without this practice.

So what does this mean exactly?

Currently, the program text does not really reflect the logical structure of the program very well. We have written the program as a single sequence of statements in the "My code" section of the editor. If another programmer now came to read our program for the first time, they would have few clues about which part of the code does what, and they will have to read the program line-by-line to find out. As our programs get longer, this will become more and more challenging.

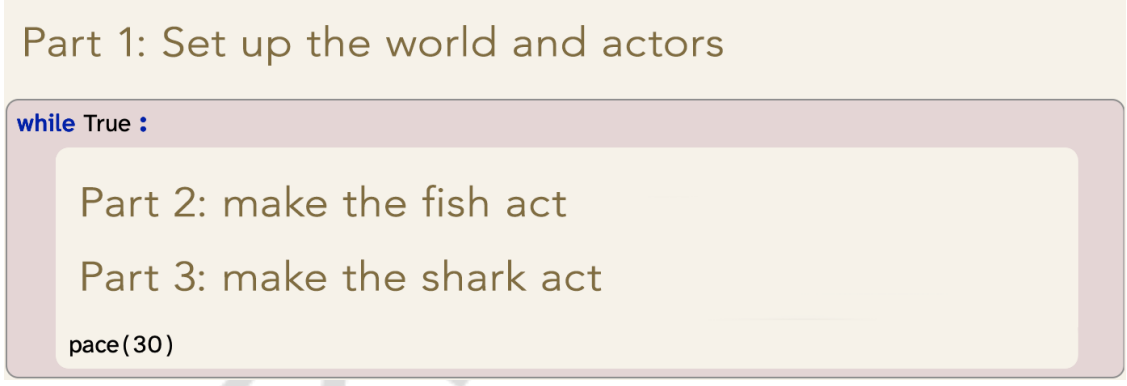



*Figure 7.2: The logical structure of our program*

If we analyse our program carefully, we can see that it consists of three distinct logical parts (Figure 7.2):

- First, we set up the world (set the background and create the actors)
- Then (in the loop), we make the fish act (move, turn and eat)
- And finally, we make the shark act (move, turn and eat)

We can improve our program by reflecting this logical structure in our program structure. To use this, we use *functions*.

## 7.2 Defining our own functions

We have encountered functions before in the context of using ready-made functionality: We have, for example, called the set_background function to set the world background colour, or the randint function to obtain a random number. In both of these cases, the functions came from a library.

In the *fireworks* project in Chapter 3, we have also seen that functions can be defined in our own project. We called, for example, the place_rocket and ignite functions to invoke functionality defined in the "Definitions" section of our program.

We can now create our own functions and move some of our code into them. This way, we can put all code that serves one single logical purpose into one place and give it a logical name. We start with this by making a function for the fish actions.

**Exercise 7.1** Move the frame cursor into the "Definitions" section in your program and enter a frame for a function definition. Since the function we are about to create will be responsible for moving the fish, you can call it "move_fish". It does not need any parameters.

**Exercise 7.2** Move all the code that is responsible for the fish actions from the main loop into the move_fish() function. This includes the code to make the fish turn and eat shrimp. (Tip: Use Shift-Arrow down to select multiple frames, then use Cut (Ctrl-x) and Paste (Ctrl-v) to move them.)

**Exercise 7.3** In the place where the fish code had been (in the main loop), insert a call to the move_fish() function.

**Exercise 7.4** Do the same for the shark: Create a move_shark function, and move all the shark code into it, and call the function from the main loop. Test your program: If all went well, it should work just as it did before.

**Exercise 7.5** Create another function called "setup". Move all the code that sets up the world into this function. Then move the two frames that create the fish and the shark to the beginning of the "Definitions" section, *before* (outside) the first function. Again, test your program.


*Figure 7.3: Our program after restructuring with functions*

Figure 7.3 shows what our program now looks like. When the execution reaches the setup() function call, it jumps up to the setup function, executes its body, and then

returns to the function call and continues execution there. The same happens for the move_fish() and move_shark() calls.

One further change that we made here is that we have moved the creation of the fish and shark variables out of the function, and added them straight to the definitions section. (The definitions section can contain definitions of variables and functions, but no other code.)

The reason for this is that variables defined inside a function are, by default, only usable within that same function. We need, however, to use the fish and shark variables in the other functions as well, so they need to be defined outside the function to be visible to all functions. We will discuss this aspect of variables in more detail later.

Concept Sequences of code can be moved into **functions**, and the functions can be called at the place where we would like to execute that code. This makes our program more modular, and easier to read and maintain.

The new program does exactly the same thing it did before the reorganisation, but it has a nicer structure. So what exactly are the advantages of this new version?

## 7.3 Readability and naming

The more we work with programs, the more we will need to read programs. Sometimes beginning programmers think programming is all about writing code. That is certainly important, but it is by far not everything: *reading* code will be just as important – if not more important – as writing code.

As your programs become more interesting and larger, you will take more and more time to work on them. You will work with code that you have written months earlier, and you have to read it again to remind yourself what it does.

If you continue programming, you will start working with other programmers. You will need to read code that others have written, and you will need to write code for others to read. When you progress even further, you may contribute to an open source project, or you may find some publicly available code on the internet that you'd like to use and extend. In each case, your work will start by reading large

amounts of code that others have written before you can add your own.

In each of these cases, it is important that the code you are working with is *readable*. Being readable means that it is written in a way that helps a human reader to make sense of it and understand it. Not all code is easily readable, and there are big differences in how different programmers write code. Writing your code with readability in mind is one of the most important principles if you care about your program.

> Concept **Readability** is one of the most important aspects of a program. Programs can be written in many different ways, and this can affect greatly how easy or hard they are to understand. Always strive to make your programs as easy to read and understand as possible.

The most basic thing you can do to aid readability is to choose good names for your variables. From the beginning, we have named our variables so that they describe what they are used for. We used, for example "fish" for the variable holding the fish, and "shark" for the variable holding the shark.

This seems obvious in retrospect, but it is by no means automatic. We could have named our variables "a", "b" and "c" instead of "fish", "shrimp" and "shark", and the program would have worked just the same. In fact, we have seen many programs where programmers did just this: they used one-letter variable names, just to save a bit of typing, and the program is full of variables called things such as "n", "p1", or "c".

A programmer can get away with using lazy names for variables as long as the program is really small, and no other programmer needs to read it. But as soon as we want to work with others, or work with our own program for a longer time, we should take variable naming seriously. If you name your variables in a way that they describe well what is stored in them, the whole program will be much easier to read and understand. You should get into the habit of always thinking carefully about naming your variables.

So what has this discussion to do with our use of functions?

The answer is that using functions gives us a chance to attach a name to a section of

code to describe what it does. For example, we have taken all the lines of code that have to do with moving the fish, and put them into a function called "move_fish". The name of the function describes to the reader what happens in this function, and at the place in the code where it is actually needed, it now says "move_fish()" instead of listing a long sequence of statements. This is of great help to a reader of this program, as it gives them a very useful hint what the program does here, without cluttering the main loop.

The ability to attach a meaningful name to the block of code – in the form of a function name – is the first advantage of our restructure.

**Concept** Take the **naming** of your variables and functions seriously. Always try to use a descriptive name that gives a good hint what the variable is used for, or what the function does.

**Exercise 7.6** Look back through your program. Make a list of all names that you have used in the program. Think about each of them: are they well-chosen? Do they describe their purpose well?

**Note** A note about **names**: In Python, you are often expected to name things. For example, you can make up your own names for variables or for functions. Python has some rules about what these names must look like:

- You can use letters, digits and the underscore ( _ ) for names, but no other special characters.
- You cannot use spaces in a name.
- The first character cannot be a digit. (It has to be a letter or an underscore.)
- Names in Python are case-sensitive. This means that the names "number" and "Number" are different names.

The technical term for these names is "**identifier**". You may come across this term occasionally: When the system asks you for an "identifier", it is asking for a name in this format.

**Exercise 7.7** Which of the following names are valid Python identifiers, and which are not? Why?

```
sum
sum01
1sum
sum_
sum 01
sum%1
_
33
__33__
sum+
```


**Good programming practice: Choose names carefully**

Pay close attention to the names you give to variables and functions. Choose names that describe as well as you can what a variable is used for, or what a function does.

## 7.4 Abstraction

The second advantage of using functions in our program is *abstraction*. Abstraction is the technique of solving a sub-problem first, and then being able to ignore the details of the problem once it has been solved.

For example, if we now read our main loop, we can see that in each loop iteration it moves the fish, moves the shark, and maintains the pace of the loop. This is really easy to understand, and as long as we are happy with the behaviour of the fish movement, and we trust that it works as intended, we do not need to worry about the details of how it works any longer.

We have essentially treated the programming of the fish behaviour as a distinct sub-problem, which we solved by writing a function. When we need to invoke the fish behaviour, we can now call it just using its name, without needing to worry about

the details of how it works. We treat "fish_movement" as if it were a single task, and we free our minds to concentrate on other things. We "abstract from" the details of the task, and treat it as a single, easy thing.

As our programs become longer and the tasks we wish to implement more complex, this becomes really important. Soon we will write programs solving problems that are too big to hold all details in our head at the same time. We will solve those problems by dividing them into sub-problems, which are smaller and can be solved more easily. These sub-problems are often solved in functions. And once we have written enough functions solving enough sub-problems, the solving of the overall problem becomes easy and straight forward.

Abstraction is thus a technique that wraps a potentially complex task into one unit and gives it a name. It hides the complexity, and makes it easy for us to use it without needing to think about the complicated details.

Even more importantly: it allows us to share the work between different people. For example, one programmer, somewhere, once implemented the randint function from the random library, or the is_touching method from the graphics library. But since we are using abstraction, we do not need to understand or worry about how these functions work internally. We just use them as a single instruction that does what we need. We aim to achieve the same with our own functions.

To use a function without the need to study its body, we just need to understand how to call it and what it does. The one additional thing that is needed is a good function comment that gives a reader enough information to understand what a function does without the need to read the body. So far, we have neglected the comment in our own functions. It is time to fix this.

**Exercise 7.8** For each of the functions defined in your program, write a function comment. This is done in the area under the function header, marked with a small double-quote symbol. In the comment, describe what the function does.

The idea of abstraction is supported in Strype using the folding functionality of the function definition frame. Once the implementation of a function is finished, you can fold in the implementation to simplify the view of your program. Since you only need to know the header to call the function, this will be all you need to see for much

of the time. Only when you later want to modify the function do you need to fold it out again.


**Good programming practice: Always comment carefully**

You should always add a comment to every function you create. The comment should state what the function does, what parameters it expects, and what it returns. Another programmer should be able to call your function without having to look at the implementation; the comment should tell them all they need to know.

Within the code, use comments to add information when a segment of code is not obvious.

## 7.5 Reuse

Yet another advantage of using functions for part of our code is reuse: Once we have created a function for a subtask, we can call this function multiple times in our program, without the need to write out the whole code again. We will see more examples of this in later projects.

## 7.6 Refactoring

The technical term for what we have done when we move part of our code into functions is *refactoring*. Refactoring is an activity of improving the structure of our code without adding any new functionality. Typically, after a step of refactoring, the program behaves in exactly the same way as before, but the internal structure is better. The structure matters, because it typically makes it easier to understand, extend or modify the program.

Concept **Refactoring** is the restructuring of a program to improve the internal quality of the code. Refactoring maintains the functionality of the program: it will do exactly the same after the refactoring as it did before. The improved structure makes it easier to do further work with the program.

The program as it looks after the improvements described here is available in the book projects as yellow-fish-v6.


**Good programming practice: Use small functions**

You should always strive to break up your code into separate functions. Your "My code" section should never hold a long sequence of code. If it does, it is time to move some code into separate functions.

If functions get long, break them up as well. Each function should always be responsible for only one well-defined logical task. If it does more than that, create a new function.

## 7.7 Summary

In this chapter, we have not added new functionality. Instead, we have worked on improving the structure of our code. This is necessary, because adding more functionality would very quickly become very difficult otherwise.

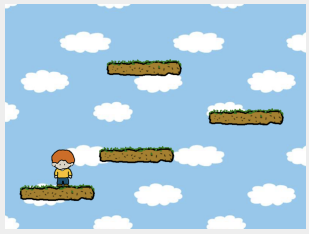
We have discussed *modularisation*, which is the structuring of your program into distinct logical units. We have seen that we can define our own functions, and how functions are used to create those units. Each function is given a name, and we have discussed the important aspect of how you should name functions and variables.

## Concept summary

- Sequences of code can be moved into **functions**, and the functions can be called at the place where we would like to execute that code. This makes our program more modular, and easier to read and maintain.
- **Readability** is one of the most important aspects of a program. Programs can be written in many different ways, and this can affect greatly how easy or hard they are to understand. Always strive to make your programs as easy to read and understand as possible.
- Take the **naming** of your variables and functions seriously. Always try to use a descriptive name that gives a good hint what the variable is used for, or what the function does.
- **Refactoring** is the restructuring of a program to improve the internal quality of the code. Refactoring maintains the functionality of the program: it will do exactly the same after the refactoring as it did before. The improved structure makes it easier to do further work with the program.

## Project ideas

**Platformer**



This project implements the beginnings of a classic platform jumper game. So far, nothing very interesting happens in the game, but it shows how you can implement the jumping from platform to platform (platform-v1).

There is also a second version of the same project (platform-v2), in which the game character is animated to show a walking effect.

Extending the game to make it interesting and playable will let you practice many of the concepts you have learned so far.

# Chapter 8 Finishing our game

**Topics:** using sound, making the game more engaging

**Constructs:** (no new constructs in this chapter)

In this last chapter, we discuss some ideas of functionality you could add to your game. These do not need to be done in this order, or indeed at all. You are very much invited to invent your own improvements and try to add those as well. You might like to treat the following sections as suggestions rather than as strict recipes you have to follow. If you want to compare your code to ours, you can look at yellow-fish-v7, which is the last of the yellow-fish projects we provide. It contains implementations of some of the functionality suggested here.

## 8.1 Adding sound

Currently, our game is silent. One obvious addition is to add sound effects. This is fairly straight forward:

- To use sound in our project, we need to use the strype.sound library. We can import the entire library, using the * symbol, as we have done with the strype.graphics library. Make sure this library is imported.
- We can either copy sound files from our file system into our program, or we can record sounds directly in Strype.
- We can then use the play method from the Sound class to play the sound file.

Let us start by recording our own sound – this is the easiest option. Strype has a built-in sound recorder that lets you produce your own sound effects easily. Insert an assignment frame and place the cursor into the value slot on the right-hand side:

In the help area at the right of the Strype window, you will see the 'Record sound' option. You can click on it, or use the space s key sequence to activate it. Try this out.

You will see first a sound recorder and, after recording your sound, a simple sound editor (Figure 8.1). In this editor, you can trim the sound (remove unwanted parts at the start and end) by using the red handles, and you can adjust the volume. Experiment with this.

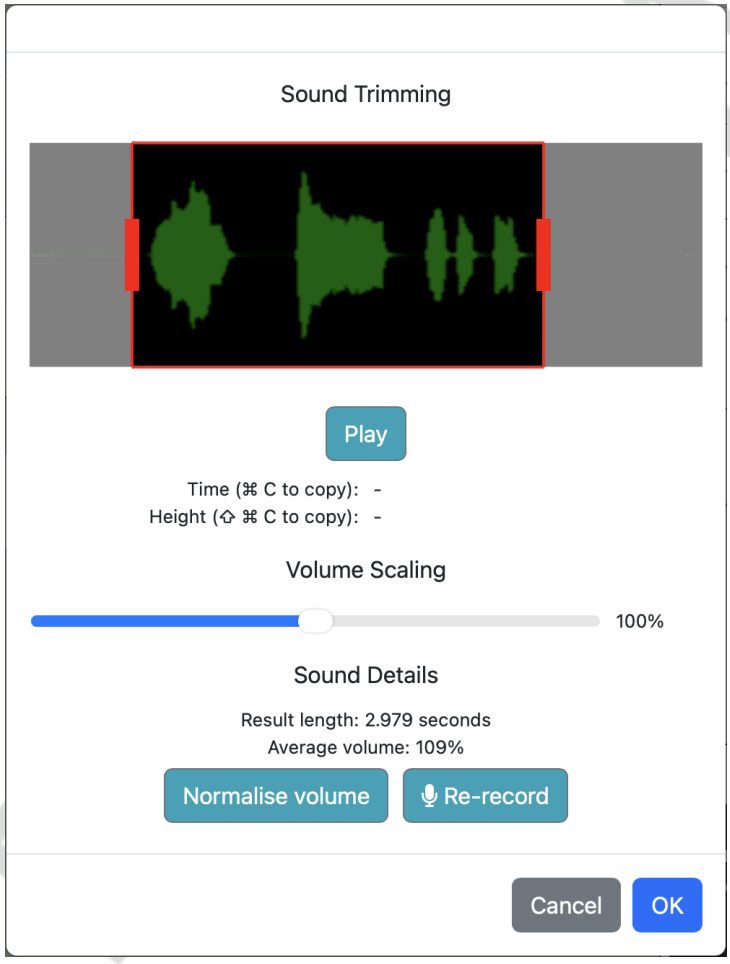



*Figure 8.1: Strype's built-in sound editor*

**Exercise 8.1** Find the place in your program where the fish eats a shrimp. Add an assignment frame there for a sound to be played at that point.

**Exercise 8.2** Place the cursor into the value slot of this assignment and activate the sound recorder to record and store a sound. This is the sound that should be

played when a shrimp is eaten. Change, trim, check, re-record the sound until you are happy with it. Give the variable an appropriate name.

Now that we have a sound recorded, we can play it:

```
my_sound ⇐ 
my_sound.play( )
```

**Exercise 8.3** Enter the code shown above into your own program. Try it out.

**Exercise 8.4** Also play a sound when the shark eats the fish.

If a sound is only used once in a program, we can also play it without assigning it to a variable. For this, we create a function call frame, record the sound directly into that frame, and then call play on the sound literal:

```
 .play( )
```

Either way is fine.

Another option is to find and copy sound effects from the internet. Some sounds are provided with the Strype projects, and you can find libraries of free sound effects by doing a web search. To use those sounds, download them to your own computer and then copy/paste them into your program, just like you did with images.

If you would like to explore this, do the following exercises.

**Exercise 8.5** Add ready-made sounds to your program. With the book projects in Strype you can find the sounds we have used for our own version of the game. You can copy them from there. Or you can find some more sounds in the Material section of the book website.

**Exercise 8.6** Search for free sound effects on the internet. Download one to your computer, copy it, and then paste it into your own program. Sound file formats that work include MP3 and WAV format files.

## 8.2 Game over

At the moment, the game just runs on when the fish is caught. Implement a proper end to your game.

**Exercise 8.7** Add a game-over message to your game. You could do that by creating an actor that has a "Game Over" image as its actor image, at the time when it should be shown. Or you could investigate using the `show_text` function from `strype.graphics`.

**Exercise 8.8** Stop the execution of the game when the game is over. (You can do this using the `stop()` function from the graphics library.)

**Exercise 8.9** As an advanced exercise, you could also end the game when the player has eaten all the shrimp. This requires using the `len()` function to check the length of a list, which we have not discussed yet, so do this exercise only if you feel adventurous and are happy to look ahead and find out about this on your own. The idea is to put up a "You win" sign (and maybe play a sound) when all the shrimp are gone. You can check this by using a call to `get_actors("shrimp")` to get a list of all shrimps, and then checking whether the length of this list is zero.

## 8.3 Random turns

Winning the game is still quite easy. This is partly because the shark swims in a very predictable way: when it is not at the edge of the world, it just swims straight. We can make it a bit more interesting if we make the shark swim more randomly.

We can use randomness here in two ways: we can make the shark turn at random times, and turn by a random amount. Let's start with the second part of this: turning by a random amount. This is quite straight forward. We can use the shark's turn method, and then use a call to the random function for the actual parameter of the degree to turn. For example

```
shark.turn(randint(-45,45))
```

will make the shark turn a random angle, between -45 and 45 degrees.

**Exercise 8.10** Add code to your program to make the shark turn a random amount at each step.

When you test this version, you will notice that it makes the shark appear too nervous. The constant turning makes it flick back and forth – it does not look good. So our next step is to make it turn less often. Let's say, at every step we want a 10% chance of turning, and in 90% of the steps we just continue straight. How can we do this?

**Exercise 8.11** How can you use a random number to express a 10% chance? Describe at least two different ways.

One way to express a 10% chance is to generate a random number between 0 and 99, and then look at the number. There is an exactly 10% chance that this number is less than 10.

In Python, we can use the `<` operator to test whether one number is less than another one. For example `number < 20` returns `True` if the value in the variable `number` is less than 20. (Again, see Appendix D: Python operators for a full list of operators.)

This means that the expression `randint(0, 99) < 10` will be `True` exactly 10% of the time. By using this expression as the condition in an if-statement, we can make the turn happen in 10% of the steps:

```
if randint(0, 99) <10 :
    shark.turn(randint(-45, 45) )
```

**Exercise 8.12** Add this improvement to your code: make the shark turn with a 10% probability. Experiment with different values for the chance and the turn until you are happy with the movement.

## 8.4 Keeping score

Another possible addition is to keep a score. We could, for example, award 20 points for every shrimp eaten. Doing this involves creating a variable for the score, initialising it to zero, and then incrementing it by 20 every time a shrimp is eaten.

To start, you can create the variable and initialise it to zero:

```
score ⇐ 0
```

The type of the variable will be int (a whole number), and you can add to it every time points should be awarded:

```
score ⇐ score + 20
```

**Exercise 8.13** Add a variable for the score. Initialise it to zero.

**Exercise 8.14** Increment your score variable every time the fish eats a shrimp.

**Exercise 8.15** Show the score on screen by using the say_for method of the fish. Every time the score changes, get the fish to say the current score for one second.

The exercise suggested above shows your score only temporarily. An alternative would be to display the score permanently on screen. If you want to do this, look at the show_text() function in the graphics library – it lets you write something onto the world background.

## 8.5 Other extensions

Many more extensions to this game are possible. The following exercises suggest some ideas. We will not discuss these here, but leave it up to you to work these out. Some of them are advanced exercises that require you to look ahead and find out some things for yourself.

**Exercise 8.16** Make new shrimp appear, either every time a shrimp is eaten, or at random intervals.

**Exercise 8.17** Change the fish movement so that it moves forward only when the up-arrow key is pressed.

**Exercise 8.18** Make the game get more difficult over time by slowly increasing the shark's speed.

**Exercise 8.19** Turn the game into a two-player game by introducing a second fish.

There is much to discover and learn – and fun to be had – if you invent and implement your own ideas.

## 8.6 Summary

In this chapter we have suggested some more functionality which you can add to your game. One idea was to add sound, and we have discussed how to do this.

We did not introduce new concepts in this chapter. Instead, the ideas discussed here give you a chance to apply and practice everything you have learned before.

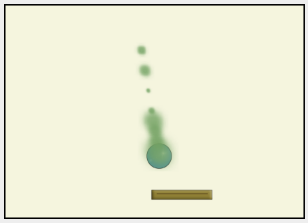
The programming concepts and practices you have learned in this book have put you in a position to progress well with further projects if you choose to do so. You should now be able to write a wide variety of programs, and to learn and study more advanced topics of programming to improve your skills further.

If you want to continue learning to program with Strype, you are now ready to move on to Creative Python Programming with Strype, Volume 2. In that book, you will learn how to write programs that process large amounts of data.

Wherever you choose to go next with your programming ideas, most of all: have fun.

## Project ideas

**Smoke**



In some games, it is not only the functionality that matters, but also the visuals: What does it look like? How nice are the graphics?

In addition to just making and choosing nice graphics for your projects, you can also implement visual effects. This project shows an example: it is called smoke, and it demonstrates a smoke effect. We start with the breakout project we have seen earlier, but now we add an effect of green smoke to the ball. It makes no difference to the game play, but it looks nice.

# Appendix A: Frame-based editing – the basics

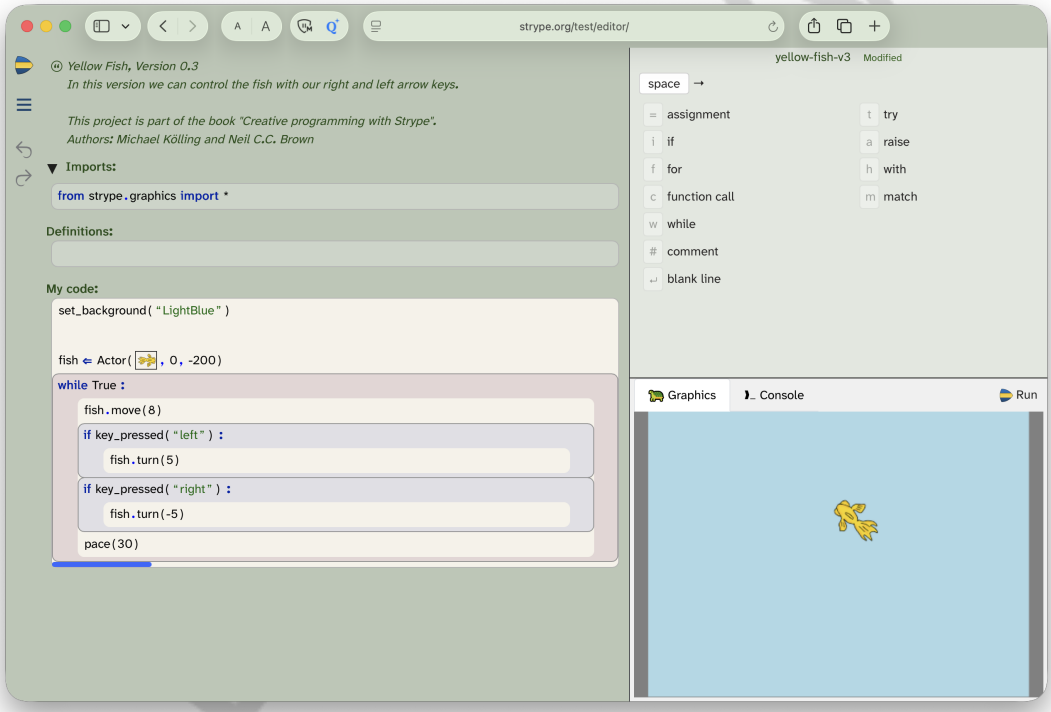



*Figure A.1: The Stryope editor interface*

## A.1 Basic editing

The Stryope editor allows the creation and manipulation of Python programs using the mouse or keyboard. All editing can be achieved with keyboard commands.

### Frames

In Stryope, program statements are represented by *frames* – coloured boxes which may be nested within each other (shown under the "My code" heading in Figure A.1). Each Python statement is represented by a frame.

Some frames have a distinctive background colour. Simple frames (such as assignments or function calls) have the same colour as the code background, and no visible outline. They are still frames, and can be manipulated like all other frames.

Frames have *slots* – areas that must be filled in to complete the frame. *Text slots* are used for expressions and accept textual input. *Frame slots* hold nested frames. In an if-statement, for example, the condition is held in a text slot, while the body is held in a frame slot.

### Inserting frames

Strype has a frame cursor (the blue bar in Figure A.1) that indicates the current editing position.

A panel in the top right of the main interface (the *frame palette*) shows all available frames which may be inserted at the current cursor position (see Figure A.1).

A frame can be inserted at the current position in one of three ways:

- You can click on the frame in the frame palette with the mouse.
- You can press Space or TAB to move focus to the frame palette and then press the shortcut key shown there. For example, pressing Space then w will insert a while-loop, or Space then c will insert a function call.
- You can just start typing a Python statement at the frame cursor. As soon as it is clear what kind of statement you want, Strype will convert your input into the appropriate frame. For example, if you type if (with a space character at the end), Strype will insert an if-frame. If you type =, you will get an assignment frame.

*Tip*: Several frames can be inserted without first typing the Space or TAB key. They are a comment, an assignment, and a blank line. This is because they can be immediately converted: when you insert them as text, it is immediately clear what type of frame you want, and you will instantly see the right frame.

### Cursor movement and selection

The frame cursor is moved using the cursor keys, which can also be used to select one or more frames.

| Key | Function |
|---|---|
| arrow-up / -down | Move the frame cursor one line up or down. |
| arrow-left / -right | Enter the next/previous text slot. |
| shift+arrow-left / -right | Select text in the current text slot. |
| option+arrow-up / -down | Move the frame cursor up/down at the current frame level (MacOS). |
| ctrl+arrow-up / -down | Move the frame cursor up/down at the current frame level (Windows). |
| shift+arrow-up / -down | Select the frame(s) above/below the cursor. Use repeatedly to select multiple frames. |

#### Surrounding frames

You can wrap existing frames in a surrounding control frame (such as a loop or conditional), by selecting the intended body frame(s), and then inserting the control frame.

For example, try selecting a few frames, and then press Space then i to insert an if-frame. This will place the selected frames inside an if-statement.

#### Surrounding text

Similar functionality exists in text slots. If you insert a symbol that is usually used as a pair, such as a bracket, parenthesis, or quote, the matching closing symbol is automatically added as well.

If you wish to add quotes or brackets to existing text, select the text to go inside the brackets first:

```
result ⇐ a + 1 * 3
```

and then insert the surrounding character:

```
result ⇐ (a + 1) * 3
```

This applies to all paired characters, including all quotes and bracket types.


*Figure A.2: The frame context menu*

### The context menu

Right-clicking a frame with the mouse displays its *context menu* (Figure A.2). The context menu offers some useful functions for frames.

Context menu functions can be applied to multiple frames at the same time by first creating a multi-frame selection, and then using the context menu.

Some context menu functions can be applied using keyboard shortcuts; these are shown in the context menu.

### Deleting frames

Frames can be deleted using the backspace and delete keys.

For a compound frame (such as an if-statement or a loop), the entire frame can be deleted (including children), or the outer frame can be deleted, leaving the body intact.

To delete the **entire frame**, place the cursor **after the frame** and use backspace.

To delete **just the outer frame**, place the cursor **after the frame header** and use backspace.

These functions can also be applied via the context menu.

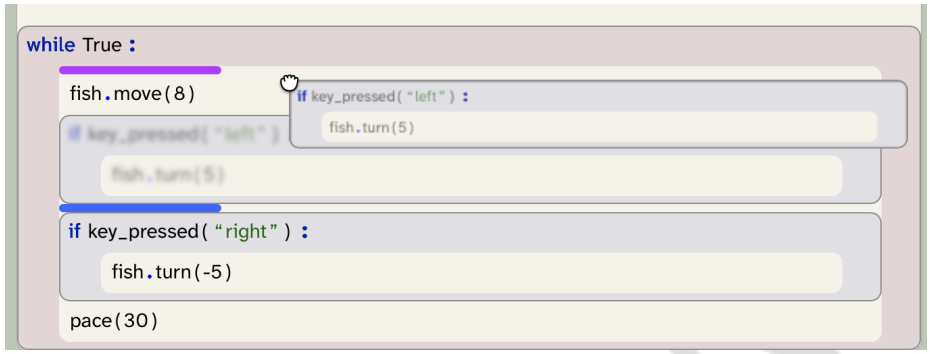



*Figure A.3: Dragging a frame*

### Moving frames

To move a frame, drag it with the mouse pointer to the desired location (Figure A.3). The purple frame cursor indicates the potential target location. Note that you will need to drag from an area with the frame background as dragging text will select it; we find it easiest to drag the right-hand end of the frames where often there is no text.

Frames can also be moved using the keyboard with frame selection, followed by cut/paste operations.

## A.2 Convenience functions

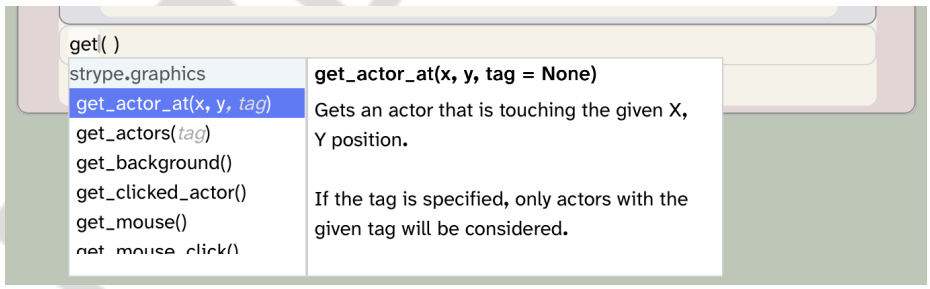



*Figure A.4: The auto-completion dialogue*

### Auto-completion

When inserting function calls or assignments, auto-completion can offer available identifiers. To activate auto-completion, use Ctrl-Space. This displays the auto-complete dialogue (Figure A.4); typing a prefix narrows the offered selection. Use the arrow keys and Enter to select an offered choice.

#### Instant auto-complete

If, with the frame cursor visible, you activate auto-completion (Ctrl-Space), a function call frame will be inserted and auto-complete activated in one action. There is no need to insert the function call frame separately first.

#### Auto-completion for strings

If you use auto-completion (Ctrl-Space) while the cursor is inside a string, the auto-completion dialogue will offer all file names for files available in Strype's built-in, read-only file system.

#### Disabling frames

Frames can be temporarily disabled without a need to delete them. Disabling is used in Strype instead of "commenting out" code. Disabled frames are shown with a blurred appearance.

Frames can be disabled using the context menu, or the keyboard shortcut shown in the menu.

If multiple frames are selected when the Disable keyboard shortcut is used, the state of each frame is toggled. This can be used to alternate between two different test statements: If one test statement is enabled and the other is disabled, select both frames and invoke the 'Disable' toggle. This will alternately activate each statement.

# Appendix B: Working with Strype - Tips and tricks

## B.1 Display options

### Pane layout

Strype offers four different layout options. These will determine how the text console and graphics world are displayed in the interface.

By default, the graphics and text output areas are displayed in a tabbed pane, allowing viewing of one of them at a time. Using the other options, these two output areas may be displayed at the same time.

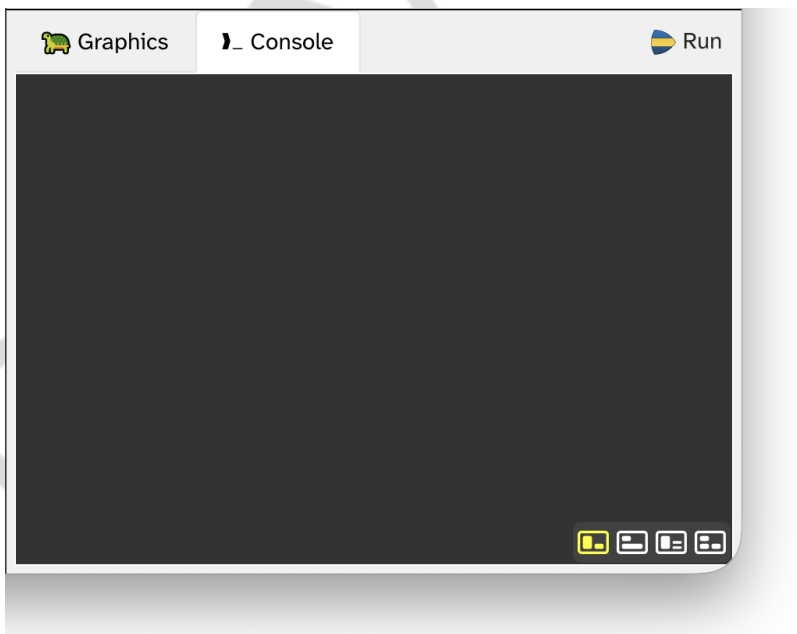



*Figure B.1: The layout selection controls*

The options can be selected using the icons in the bottom right corner of the main Strype window (Figure B.1). Note that the icons are only visible when hovering over the output area.

The size of the output areas can be adjusted by dragging the horizontal or vertical divider line around the console (Figure B.2).

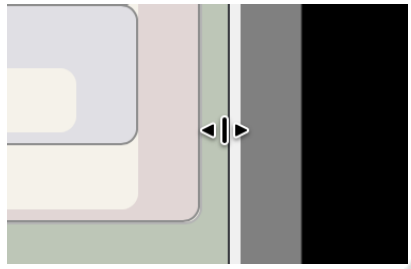


*Figure B.2: Adjusting the console size*

The interface layout is saved with the project. When saving and sharing a project, the interface will be restored to the same arrangement as it was when the project was saved.

### Folding functions and classes

Functions may be folded in to streamline how they are displayed in the editor. This is achieved by using the folding control in the top right of the function frame (Figure B.3). Note that this control is only visible when the mouse hovers over the frame, or when the function is folded.

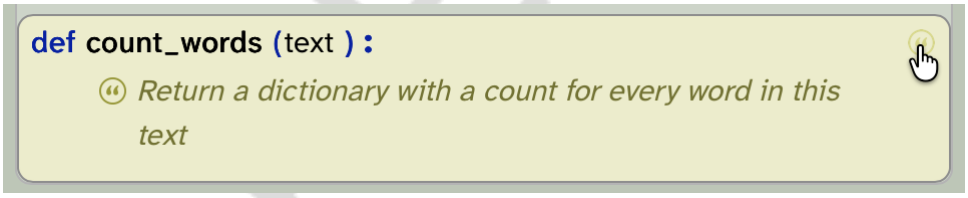



*Figure B.3: Folding functions*

Folding a function cycles through three states:

1. All code
2. Header only
3. Header and comment

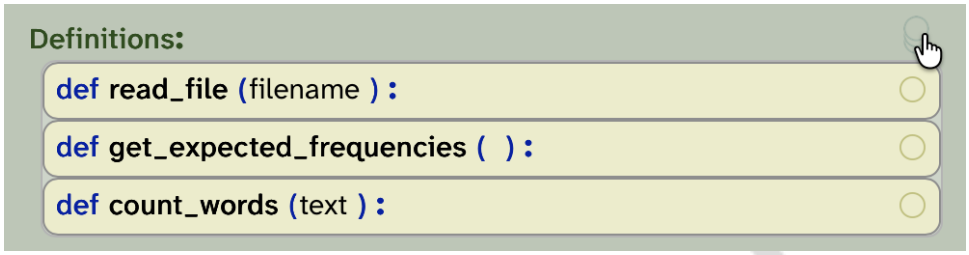



*Figure B.4: Folding all functions at the same time*

A global folding control is available at the top of the definitions area (Figure B.4). Using this control, all functions can be folded into the same state.

In addition to folding of functions, folding is also available for classes (Figure B.5).

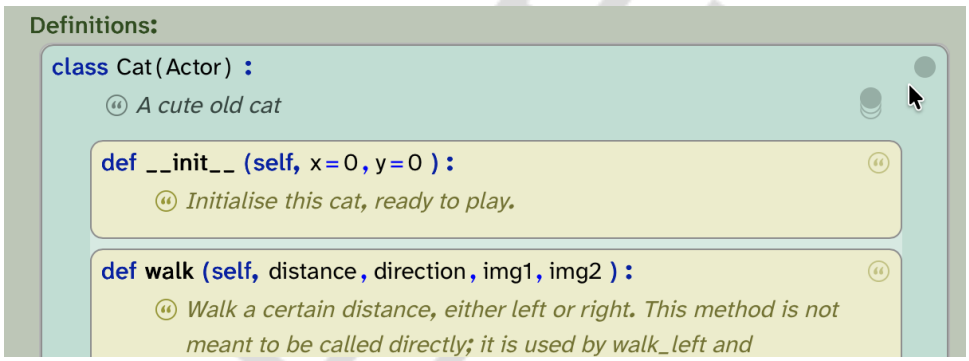



*Figure B.5: Folding classes*

For classes, two folding controls are available: one for folding all methods within the class, and one for folding the class itself. By making good use of these options, classes can be either entirely collapsed, or they can be displayed with a listing of their method headers (with or without documentation). The best way to understand these options is to try them out.

Making good use of Strype's folding functions can help with navigating a project and concentrating on the segments of code currently under construction.

#### Freezing functions and classes

In addition to folding, functions and classes can be frozen. Freezing is available via the right-click context menu. When a function or class frame is frozen, it is displayed with a small snowflake icon in its top right corner (Figure B.6).

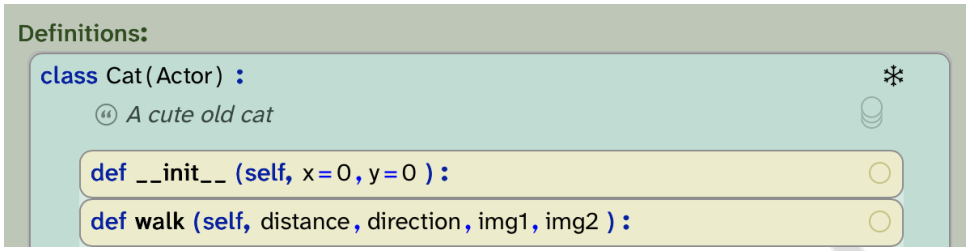



*Figure B.6: A frozen class*

A frame is typically frozen in a project to indicate to the programmer that this part of the program is finished in its implementation, and is not intended to be further modified.

A typical case is a project provided to students with some helper functions implemented, which the students are expected to use, but not modify. Freezing these functions sends the signal that students are not expected to modify or understand the implementations, but can use them as library functions.

When frames are frozen, the folding control toggles only between the two header options (with or without comment). The implementation will not be displayed.

Note that freezing sends just a hint to the student how they are expected to use this code. It does not prevent the function from being modified. Any user can, at any time, unfreeze a function to study or modify its implementation. Thus, freezing provides an aid to viewing and using a project, not a security feature.

## B.2 Project and file storage

### Open and save

You can save Strype programs to your local filesystem, or to a cloud storage system. Currently, two cloud stores are supported: Google Drive and Microsoft OneDrive.

Strype does not manage or access your user data on those platforms. To use a cloud drive, you must have an account with one of those providers and access it in the usual manner. Strype never sees your account details or password and stores no user data.

### File access: Local device vs cloud

Saving systems to your local computer may be easier, since it does not require an

account with a cloud provider. However, if you would like to work on the same project across different computers (for example, in a school lab), then saving to the cloud can be more convenient.

Another reason to save your project to the cloud is data file access: For security reasons, web browsers are not allowed to access your local filesystem (without an explicit file open dialogue). Because of this, your Strype programs cannot read data files from your local file store, and programs that attempt to access local files will fail.

To work with your own data files, store your project in a cloud file system, and then place the data file into the same folder as your program. The program can then access the file without a full path (using just the file name).

## B.3 Export / import

#### Opening Python programs

You can open many standard Python programs directly from within Strype. To do so, just use the standard 'Open' function and select a Python source file (**.py** suffix).

Not all Python programs can be opened. Firstly, Strype will attempt to re-order code in the file to move all imports to the top, and all definitions above any free-standing code. This is necessary to conform to Strype's restrictions. Secondly, some (rarely used, advanced) Python constructs are not supported, and programs using them cannot be opened. The great majority of Python programs, however, will work with Strype.

#### Saving Python programs

Strype programs can be saved in standard Python format using the "Convert to Python file" function from the Strype menu. The resulting program will run in a standard Python runtime, as long as it does not use any of the Strype-specific libraries (imports starting with **strype.**).

#### Pasting python code

You can paste standard, text-based Python code directly into the Strype editor. For this to work, you should copy complete Python statements (one or more). Pasting will fail if the paste buffer does not hold complete statements.

### Download as image

A selection of Strype frames can be downloaded to your local file system as a PNG image. This can be useful for creating documents (e.g. teaching material) including Strype code snippets.

To do so, select the desired frames, then use "Download as image" from the frame's context menu. The image will be downloaded to your default browser download location.

Similarly, you can download an image of the graphics output or text console via a right-click menu in the output area.

### Project sharing

A copy of an existing program can be made available to others. This can be useful, for example, to make a starter project available to students, or to give a copy of your program to a friend. (The book projects in this book are made available in this manner.)

To share your project, use the "Share" function from Strype's menu.

Two different sharing options are available:

**Cloud link**: To use a cloud link, your project must be saved in a cloud storage system (Google Drive or OneDrive), and it must be in a saved state. The **cloud link** dialogue will give you a link which you can then send to others (via email, a web page, or any other means). Any user who clicks on the link will be taken to Strype, with a copy of your project loaded. They can then save their copy to their own storage; this will not affect your copy.

Cloud link provides a *live copy*: If you make and save changes to your program after sending the link, other users will see those changes when they later click the link.

**Snapshot link**: A snapshot link encodes the entire Strype program in a URL. Clicking this option places this link into your paste buffer, ready to be pasted elsewhere. The advantage over the cloud link is that no cloud storage is needed.

The link is static: it encodes the program at that moment; later changes will not affect the link. **Note**: Most browsers have a maximum URL length. This option will not work if the program is very long. Therefore, programs including images or

sounds will often not work with this option (the image or sound file makes the program too big). For those programs, use a cloud link.

## B.4 Other

### Local language setting

Strype's interface language can be changed using an option in the Strype menu. Currently, a limited set of languages are available.

If you are interested to contribute a new translation to a language you speak, please contact us at team@strype.org. We welcome volunteer contributions of interface translations.

### The Strype Teachers' Lounge

The Strype Teachers' Lounge provides a community site for teachers who are interested in Strype. If you are a teacher at a school, you can request access. The forum provides discussion and resources that can help your teaching (project ideas, worksheets, etc.). A link to the Teachers' Lounge is available on the Strype website at **strype.org**.

# Appendix C: Dealing with errors

Errors are an inherent part of programming. Everyone runs into errors in programming: from the most expert programmers, to the person writing their very first program. They are not a sign of deficiency; they are an expected part of the process. Indeed, programming can sometimes feel like the journey from one error message to the next!

Strype is specifically designed to eliminate some errors that you can get from Python, such as mismatched brackets, or incorrect indentation. Nevertheless, you will encounter errors in Strype. Each error arises from a unique situation so we cannot predict exactly how to fix each and every error. This appendix gives general strategies on how to deal with errors.

Errors can occur while you are editing, or they may occur when you run your program. If you are in the middle of editing you may want to ignore some errors until you believe you have finished. For example, if you create an if-frame and don't fill in the condition, an error will be displayed telling you that the condition is empty. But that is expected, and you may want to ignore the error until you are ready to fill in the condition.

Errors that occur during editing will prevent you from running the program and you will need to fix them first. Other errors may only occur when that particular piece of code is executed — or the error might even only happen based on what happened earlier in that particular execution of the code.

## C.1 What to do when you get an error

First: stay calm. Errors are one of the ways in which the programming system communicates back to you, to tell you it has encountered a problem. Your role is to take that error and work out how to resolve it.

The next thing to do, which may sound obvious, is to read the error message. This

can be challenging. Error messages can be confusing, or full of jargon. Sometimes the error does not make sense, or will only make sense when you understand the problem. Some people new to programming get scared of trying to read the message at all, but make sure you do read it. It will be much harder to fix if you do not know what is wrong.

**STRYPE TIP**

The error message should pop-up in a bubble when it occurs. If you want to see it again later, you can move your mouse over the little exclamation mark (!) triangle on the right of the frame where the error occurred. Failing that, you can try running the program to try to make the error re-appear.

Check the code for spelling mistakes. It is easy to write "prnit" where you meant "print". This includes capitalisation. If you called your variable "myString" and then try to use it with "mystring", Python counts this as two different things, and you will get an error.

A further step which can help is to compare the code with an error to similar code without an error. For example, you might compare your code to code in this book (which should all be error free!), or to programs that you have previously written that worked. Pay close attention to punctuation like speech marks or brackets; often code errors arise from missing vital punctuation that is easy to overlook.

Finally, some errors, especially in larger programs, can actually be caused by mistakes earlier in the program. If the message does not make sense where it is displayed, look earlier in the program for possible related mistakes. Often the relation is the variables that are used on the line with the error.

## C.2 Deciphering errors: example 1

Let us look at an example of deciphering an error. You might want to print the text *Hello* on the screen, so you write this:

```
print(Hello)
```

At first glance, this looks reasonable. There are no spelling mistakes here. But if we click *Run* we get this error:

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

That is quite confusing. But let's not just disregard it. Let us see what it is trying to tell us. It says the name Hello is not defined. But that is stupid: Hello is not meant to be a name. Maybe that gives you a hint, but maybe it is not enough. So let us compare to some similar, working code. Here is some code from earlier in the book:

```
print( "This is a string" )
print( 'This is also a string' )
```

Let us compare our code very carefully to that code. Do you see the difference?

It is the quote characters around the string. We are missing the quotes! And with that, the error message makes more sense: because we did not have the quotes, Python thought Hello is the name of a variable and did not find such a variable. We can add the quotes to fix the error:

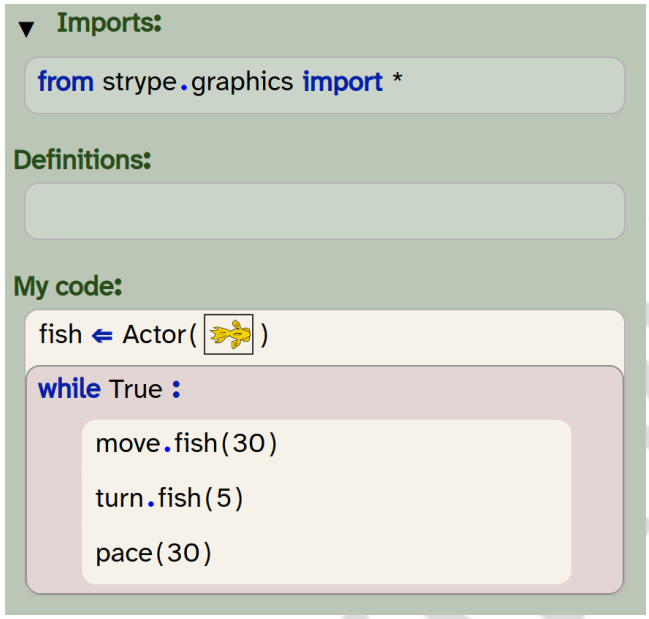
```
print( "Hello" )
```

**STRYPE TIP**

If you have some text in Strype where you forgot the quotes, the easiest way to add them is to first select the text in question (either with the mouse, or shift and left/right keys) then type a quote character. This will surround the selected text in quotes.

## C.3 Deciphering errors: example 2

Here is a different example in a short program, like the ones you will write early in the book. Imagine you wrote:

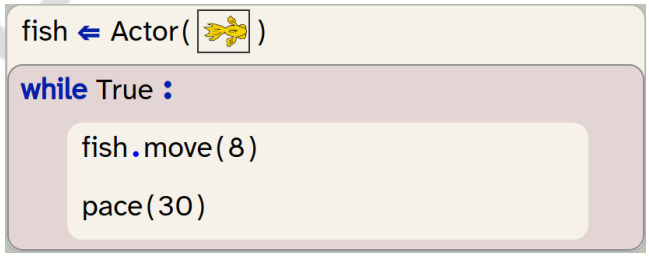



This looks fine, and valid sensible code. But when you run it you get an error:

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

So what has happened here? It says NameError so it's something to do with a name. It says move is not defined. That's not too bad in terms of jargon; it does not seem to know about move. But the import is present, you've created the fish, why is "move" not known?

A good strategy here is to compare carefully against code from the book. Here's a piece of code we showed earlier in the book that you might have been copying:

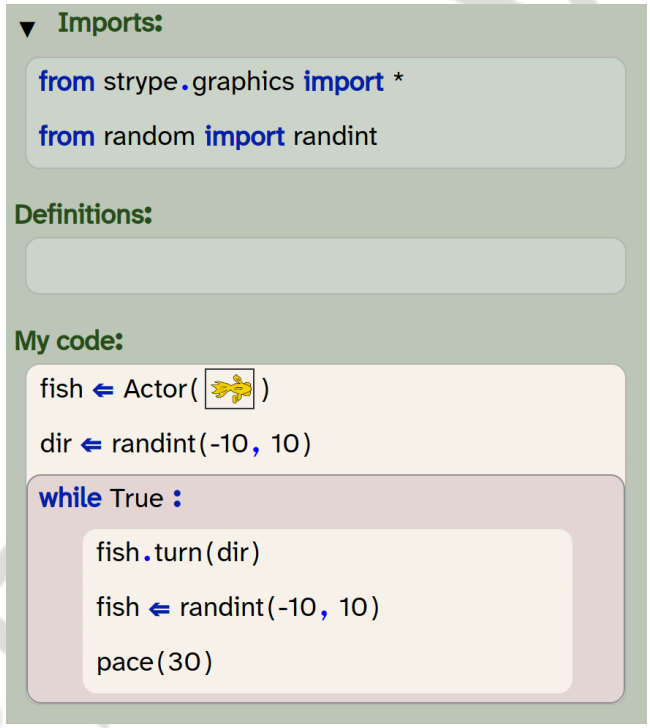



Perhaps at first glance that looks the same to you. But go through the code very carefully and be precise. You will find that the book says fish.move and your code says move.fish. This is the source of the error; the code is trying to call the fish

method on a move object, and Strype is saying it doesn't know what move is. So the fix is about the order of the words. These kind of precise details matter very much in programming, even if on first read it can seem plausible and seem identical to what you were copying.

## C.4 Deciphering errors: example 3

Let us look at another example, in a slightly longer program, which produces a confusing error. This program is intended to make a fish character turn randomly:




If you run this, you will get an error on the fish.turn(dir) line which says:

```
AttributeError: 'int' object has no attribute 'turn'
```

This is already a bit confusing in its wording as it is using different terminology to that which we have used here. It talks about attributes, which may not make sense. But it mentions having no attribute 'turn'. The line has the method turn so it somehow has not found that.

At this point it is not a bad idea to look up if turn was the correct name for the method (maybe it was rotate and you misremembered). But you will find in the code samples here and the documentation that turn is correct. So what does the other part of the message mean? It mentions 'int' object. Well, we know dir is an int, so int seems to make sense but what else could it be referring to on this line? The only other item is fish. Could it be a problem with fish?

Usually errors are caused by code that executed before, so we scan upwards in the code. The fish variable is clearly assigned an Actor, so that looks fine. But when dealing with loops, remember that code that **executed** before could be a later part of the loop that executed in the previous loop iteration. So we should also look below. You may notice fish is mentioned on the line below... where it is assigned a random number!

What has happened here is this: we accidentally assigned to fish when we meant to assign to dir. We got a quite confusing error because of the incorrect assignment on the line **after** the error. Programming often involves these intricate issues.

## C.5 Welcome to the wonderful world of errors

When you program, you will encounter many more errors than these. We ourselves often encounter new errors we have not seen before, and we make use of similar techniques to find them. So remember:

- Stay calm.
- Read the error carefully for parts you **do** understand (and don't worry if you do not understand it all).
- Check the line with the error for obvious mistakes, especially spelling mistakes or punctuation problems.
- Compare the code to other code which you know works; what are the differences and how might they explain the error?
- Look at earlier code for possible causes, especially lines which involve the same variables as the line with the error.
- Remember that earlier code can include things that happen in a function call, or in a loop where the current code resides.
- You can also ask a friend, ask the teacher, or try searching the Internet for help.

# Appendix D: Python operators

This appendix lists the Python operators you are likely to need, organised by category.

## D.1 Arithmetic operators

| Precedence | Operator | Name | Description |
|---|---|---|---|
| Highest | -x | Unary minus | Negation |
| High | * | Multiplication | Multiply values |
| | / | True division | Floating-point division |
| | // | Floor division | Integer division (floor) |
| | % | Modulo | Remainder |
| Normal | + | Addition | Add values |
| | - | Subtraction | Subtract values |

## D.2 Boolean operators

| Precedence | Operator | Name | Description |
|---|---|---|---|
| Highest | not | Logical NOT | Inverts truth value |
| High | and | Logical AND | True if both operands are true |
| Normal | or | Logical OR | True if either operand is true |

## D.3 Comparison operators

| Precedence | Operator | Name | Description |
|---|---|---|---|
| Normal | <, <=, >, >= | Ordering | Relational comparisons |
| | ==, != | Equality | Value comparison |
| | is, is not | Identity | Object identity test |
| | in, not in | Membership | Membership test |

## D.4 More operators

In this appendix we have omitted some advanced operators. There is a full list of operators in the Python documentation.

# Index

**A**

abstraction, 91
Actor, 17, 25, 52
  movement, 25, 27
  tagging, 74
assignment, 23-24, 49-51
auto-completion, 109

**B**

body, 30
bool, 75
bounding box, 77
brackets, 17

**C**

class, 54
code quality
  abstraction, 91
  comments, 93
  naming, 88, 90-91
  program structure, 84
  readability, 88-89
  refactoring, 93
  reuse, 93
colour, 38, 41
comment, 37
condition, 30
context menu, 108
coordinate system, 20

**D**

default value, 44
documentation
  function comment, 92
  library, 53

**E**

else-case, 59-60
errors, 119
execute, 12

**F**

float, 46
folding, 39, 112
for-loop, 64-66
frame, 105
  deleting, 108
  disabling, 110
  inserting, 106
  moving, 109
frame cursor, 106
frame palette, 106
freezing, 45, 113
function
  call, 16, 35
  commenting, 93
  defining/creating, 86
  definition, 38
  key_pressed, 30-31
  pace, 27
  print, 25
  randint, 68, 100-101
  range, 65-67
  show_text, 100
  stop, 100

**I**

identifier, 90-91
if-statement, 29-30, 58, 76
import, 15, 69
init, 58
int, 46

**K**

keyboard control, 29

**L**

library
  documentation, 53
  import, 69
  random, 69
loop, 65
  for, 64-66
  while, 27-28

**M**

method, 27, 54-55
  is_at_edge, 79
  is_touching, 73-74
  move, 26
  play, 97, 99
  remove_touching, 76, 79
  turn, 28, 100

**O**

object, 23, 51
operators, 125

**P**

pace, 28
parameter, 16, 36
  actual, 42
  formal, 42
  optional, 43
program structure, 84
project
  *fatcat*, 56
  *fireworks*, 36, 47
  *knock-knock*, 52
  *yellow-fish*, 14, 29, 31, 63, 70, 79, 94, 97

**R**

random
  behaviour, 68, 70, 100
  library, 69
  numbers, 68
readability, 88-89
refactoring, 93
return type, 75
return value, 74-75
run, 12

**S**

self, 56
slot, 106
  frame slot, 106
  text slot, 106
sound
  library, 97
  playing, 99
  recording, 98
  working with, 97
Sound (class), 97
str, 46
string, 41

**T**

type, 40-41, 46
  bool, 75
  float, 46
  int, 46
  str, 46

**V**

variable, 24, 49

**W**

while-loop, 27-28
world, 20
    coordinates, 20-21