*Programming*

# Adventure Games in Python

*Geert-Jan Kruijff*

Story Byte Studios

# Contents

# 1. Introduction

## 1.1. Creating Adventures

This book is about creating adventures. Adventures that can take place anywhere where your creativity takes you. Adventures you can play by yourself, or with others.

Yes, these adventures are computer games. And in this book that means two things. For one, this book teaches you how to write *computer programs*. We start very simple, with a text adventure game – and gradually build this out to complex massively multi-player online role-playing games. (Note the 'gradually' here – it will take some effort to get there!)

And then there is the other aspect, namely *game design*. Creating a game really starts with designing a game. What is the story? What should the player try to achieve? What can the player do? What are the obstacles that a player needs to overcome? These are several key ingredients of what makes up your adventure – which you then implement as a computer game.

## 1.2. For Whom Is This Book?

This book is for anyone with an interest in learning how to program, and to create adventure games. You do not need to have any experience in either programming or game design – we will learn things as we go.

Naturally, you may be interested in one aspect more than in the other – and that's perfectly fine. For those who are more interested in the programming side: You will learn about programming in Python, ranging from basic programming concepts like variables and control structures, to complex concepts like internet protocols, client/server architectures, and graphical user interfaces. And all of that you will then be able to apply to building a game.

For those who are more interested in creating adventure games: That is great too! All the code in this book is available online (as we will see below), so even if you do not want to dive in each and every aspect you can still run the code, and explore creating ever-more fascinating adventure games. It's all there for you.

## 1.3. Getting Things Set Up

In this section we explain how to get three things set up for your first steps in programming: the Python programming language, a graphical user interface to write Python

programs in, and *versioning system* called `git`. You will learn how to use these in Part 1, Getting Started.

### 1.3.1. Mac

To download the resources for the Python programming language,

1. Go to `https://python.org`. This is the official website for all matters Python.
2. In the menu bar on the main website, click on the "Downloads" button.
3. A new webpage will appear, stating "Download the latest version for Mac OS.X"
4. On that page you will see a (yellow) button saying "Download Python" plus a version number – at the time of writing, the latest version is 3.7.0.
5. Click on that button and follow the instructions to install Python.

We will use the Atom "integrated development environment" or *IDE* to write and test our programs. To download the resources,

1. Go to `https://atom.io`. This is the official website for Atom. Don't forget to admire that animated logo!
2. At the right hand side of the page, there is a box that says "Atom", a version number, and "mac OS", and that has a "Download" button.
3. Click on the "Download" button to download Atom. A zip file will be downloaded.
4. Once downloading has finished, open your browser's download menu, and drag the Atom icon to the "Applications" folder to install Atom.
5. Click on the Atom icon your Applications folder, to start Atom
6. A welcome screen will pop up – Welcome to Atom!
7. Click on the "Install a package" button, and then on the "Open Installer" button
8. A left pane called "+ Install Packages" appears, with a text box in which you can type package names.
9. One after the after, install the packages `atom-ide-ui` (the complete IDE), `ide-python` (language support for Python), `atom-python-run` (run your Python program straight from Atom)
10. Over the course of this book we will install more packages in Atom, but for the moment this will do! If you want, play around with *Themes* to make Atom look like you want.

Finally, we will use a versioning system called `git`. What is a versioning system, you ask? Well, think of it as an archive, where you can safely store previous versions of your programs. If you are working on something and all over sudden all goes horribly wrong (like, your cat walked over your keyboard and managed to delete half of your code), you can always go back to your archive, and take the latest version stored there. Of course, this does mean you need to regularly put your latest code into the archive otherwise ... but we will build up that "software engineering" discipline as we go along. Moreover, Atom will be a great help here as we will see.

1. `git` may already be installed on your Mac.
2. Open a Terminal window, and type in `git --version`.
3. If you get a response like "git version 2.10.1 (Apple Git-78)" then all is fine.
4. Else you will be immediately prompted whether you want to install `git` (which, of course, you do).

For the truly adventurous, you can also set up an account on GitHub. GitHub is basically "git in the cloud." You can use it to safely version everything not just locally on your laptop, but also in the cloud – useful if you want to share your code with others, or access your programs on different computers.

### 1.3.2. Windows

Too bad. Don't use Windows.

## 1.4. Conventions Used in This Book

We use the following type-setting conventions in this book:

*Italic*
for email addresses, URLs, filenames, pathnames, and emphasizing new terms when we first introduce them

`Constant width`
for the contents of files and the output from commands, and to designate modules, methods, statements, and commands

`Constant width bold`
used in code sections to show commands or text that you type in, and occasionally, to highlight potions of code

## 1.5. Using Code Examples

This book is here to help you learn more about programming, and creating adventures. The code is there to help in that. In general, you may use the code in this book in your own programs, and documentation (yes, programmers do -or should!- write documentation). You do not need to contact the author for permission unless you are using a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing the code *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Using a significant amount of example code described here in a commercial product or in your product's documentation *does* require permission.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *storybytestudios@gmail.com*.

## 1.6. Contacting the Author

You can contact the author at *storybytestudios@gmail.com*.

# Part I.

# Getting Started

# Introduction to Part 1

Let's get started! In this part we are going to cover quite some ground ... We start right away with building a simple, text-based adventure. Hello adventure world! We then continue with learning how we can let the player make decisions (i.e. actually play the game!), and how we can keep track of what all is happening. Towards the end of this first part, we build things out to a fully-fledged game engine that can run "any" kind of text adventure.

Specifically, you will learn:

- How to use *Atom*, `git`, to edit, run, and version your programs
- How to let your program show something in a Terminal (`print`), and get some input from your user (`input`)
- How to implement basic control structures in Python, for decisions (`if..then..else`) and loops (`while...` loops, and `for` enumerations)
- How to define functions, classes, and modules in Python (and what these things are! and why they are useful!)
- How to store your adventures as files, using a structured file format (*JSON*)
- How to save games, and load them, and how to load game resources

# 2. Hello Adventure World

## 2.1. Introduction

A program is basically a set of instructions what to do. There are different programming languages we can write such instructions in – Python is one of them. Other languages are for example Java, C, C++, Swift – by now (October 2018) probably close to 300 different programming languages exist! But, we will just focus on Python for the moment.

Python is actually more than "just" a language. It is also a program that can interpret your program and then translate it into instructions for your computer to execute. Let us see that in action right away.

Open a terminal, type in `idle3 &` after the $-prompt, and press return:

```
$ idle3 &
```

Up pops a little window, like the one below.

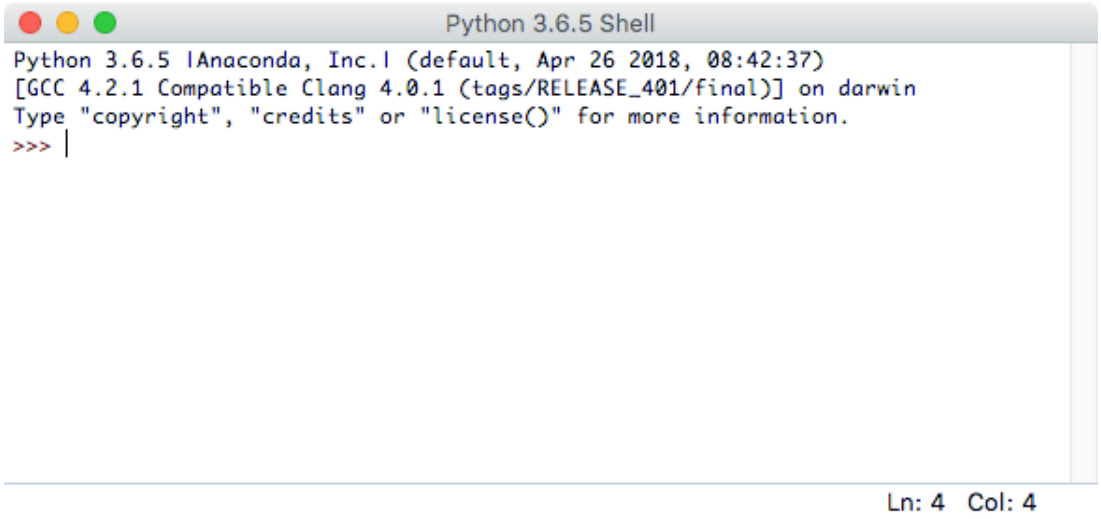


Figure 2.1.: The Python IDLE3 interactive editor

This is the `idle3` window. It is an interactive editor for Python. Everything you type in is immediately interpreted and executed – so you can see right away what it does! Let's see this at work. After the ">>>" prompt in the editor, type in the following:

```
>>> print("Hello Adventure World!")
```

And there you go! Right below your command, `idle3` shows the text "Hello Adventure World!" – you have written *and* executed (or "ran") your first line of Python code!

```
>>> print("Hello Adventure World!")
    Hello Adventure World!
```

Feel free to try out more – how do you imagine your adventure would start? Where does the player start, what does he – or she – see, smell, feel?

```
>>> print("You are in a dark, spooky house, hidden deep in the forest.
    There is only one door to the outside, and it is locked.  You hear
    strange, creaky sounds.  What do you do?")
```

Sometimes such a long piece of text looks kind of awkward. If you want to break things up a little, you can insert a `\n` or "new line" in your text. For example, if we want to have each sentence start at a new line, we can achieve that as follows:

```
>>> print("You are in a dark, spooky house, hidden deep in the forest.\n
    There is only one door to the outside, and it is locked.\n
    You hear strange, creaky sounds.\n
    What do you do?\n")
```

## 2.2. Your First Real Program

The `idle` interactive editor is useful to try a couple of things out, and see what they do. However, you can only do things line by line, and have to retype everything if you want to try again! In this section move to using *Atom*, a so-called "interactive development environment" or IDE. We will use editor to type in programs, run them, and store versions in `git`.

Start up *Atom*, and click on the "Welcome Guide" tab. The window will look something like what you see below.

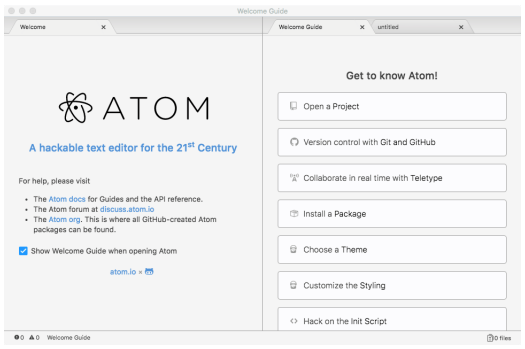


Figure 2.2.: Atom with the Welcome Guide selected

We start by creating a new project. Click on the "Open a Project" tab, and then the "Open a Project" button. A file browser pops up. Go to the Desktop, create a new folder there called "pythonadventures" and then click the "Open" button. Atom now opens a *Project* pane on the right hand side.

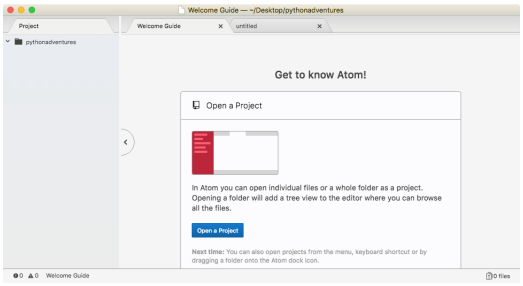


Figure 2.3.: Atom with the Project pane and browser

Next, we want to add a `git` "archive" or *repository* to our project. There are two ways in which we can do that. One, we can scroll down in the Welcome Guide tab, till you see the "Version control with Git and GitHub" tab. There are two buttons there, "Open the Git panel" and "Open the GitHub panel." If you want you can click on the "Open the Git panel" – or wait and see how we can get to that panel more quickly. If you hover with your mouse on the middle of the right side of the window, a half moon button pops up with a "<" inside. Click on that button to directly get to the Git panel!

Now, because we already set up our project, the Git panel has a button that suggests we create a repository directly in our project directory. As that is exactly what we want, press the button, and then click "+Init" in the window that pops up to initialize the repository ("repo") in the directory as shown.

Finally, select the "untitled" tab, so that we can get coding! Atom now looks like below, (and you may have noticed already that the Project browser has been updated with a ".git" folder – our repository).

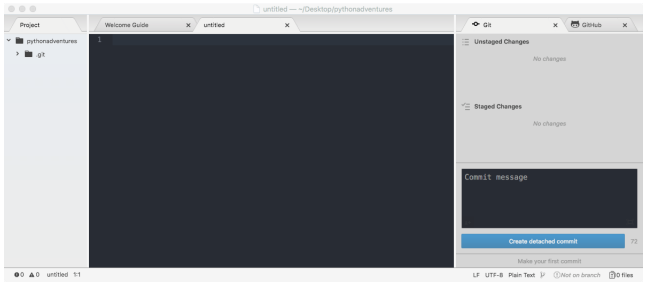


Figure 2.4.: Atom with the Project pane, the Git pane, and the code editor

Type in the following lines of code in the editor. The line numbers below reflect those you see in the editor.

```
print("You are in a dark, spooky house, all alone out in the forest.")
print("There is only one door to the outside, and it is locked.")
print("You hear strange, creaky sounds.")
```

Once you have entered the code, and save it in the project directory as *p1ch1-pythonadventure.py*. In the Project panel the file now appears, and `git` has also spotted that there is something now! Notice that our file is listed under "Unstaged changes." Before `git` puts something into the repository, you need to "stage" it. Click on "Stage all" to move your file to the "stage." Then, in the *Commit message* you should type a simple and short description of what is new or different in the file(s) you have staged. Right now this is simple: "everything is new!" Next, push the *Commit* button. Because our repository is empty, the button says "Commit detached." Push the button, to make your first commit to the repository. Well done!

EXPERT KNOWLEDGE 1 (THE `GIT` MASTER, AND BRANCHES) The Commit button now says "Commit to master." The *master* is the main archive. People often visualize the archive as a tree, where the main archive is the "trunk," from which you can also create "branches." A branch initially diverges from the trunk, allowing you to try things out without these experiments ending up in the main archive. This can be handy: If we decide the experiments are no good, we cut the branch and no harm is done to the main archive. If instead we decide that all is great, then we can merge the branch with the trunk to have the main archive reflect all the changes we made. ◀

## 2.3. Letting the Player Do Something

So the player finds himself (of herself) in a dark, spooky house. Now what ? How can we let the player do something? Most games nowadays let you move around a character in the world using a mouse, a gamepad, or a keyboard. Here, we let the user type in what he wants to do, just like players in a role-playing game like *Dungeons & Dragons* would tell the Game Master what their character is doing next!

Type in the code below, right after the lines you just typed in.

```
command = input("> ")
print("Boo!")
```

Now let's run the code. We need a Terminal for that, and fortunately we can do that right here in *Atom*. In the menu bar, click on *View*. This gives you a drop-down menu, where all the way at the bottom there is a *Terminal* ▶ option. Click on *Terminal*, and then select *New Terminal Window*. A new tab called `terminal` appears next to you code tab.

Go to the `terminal` tab, and start your program:

```
% python p1ch1-pythonadventure.py
```

Voila! As you may have expected, your program prints ("tells") the beginning of the story – just like a Game Master ...

```
% python p1ch1-pythonadventure.py
You are in a dark, spooky house, all alone out in the forest.
There is only one door to the outside, and it is locked.
You hear strange, creaky sounds.
What do you do?
>
```

And there, after the introduction, is a prompt > waiting for you to say what you want to do. What would *you* do? Go ahead, type it in, press *Enter*.

```
> look around
Boo!
%
```

The excitement never ends! In the next chapter you learn how you can do something with what the player typed in – you may have already noticed the bit "`command =` " in line 5 in your code ...

# 3. Making Decisions

## 3.1. Introduction

At the end of the previous chapter we saw how we could get the user to type something in, and then advance the story. In this chapter, we explore this further. We start with the user simply pressing *Enter* to advance the story, just like turning a page. Then, we look at *variables*, for example *String* variables that can hold text. We use a variable to pick up what the player just typed in – remember `command`? Once we have that, we inspect what the player wants to do (e.g. look around, or go south) and drive the story forward accordingly. For that, you learn more about the `if..then..else` control structure in Python, and basic String comparison `==`. By the end of the chapter, we will have a game where the player can roam around various rooms in the house, look around, and find all manner of horrific things ...

## 3.2. The Story So Far

So far, the story is that you are in "a dark, spooky house, all alone out in the forest." Scary thing is, you are inside – and you cannot get out! Whereas, hearing strange noises, getting out is clearly what you want ...

Game Design 1 (Victory condition) Most games have a *victory condition*: What you need to achieve to win the game. Collect all the little shiny boxes, like in *Fez*, or defeat all the monsters, like in *Dark Souls*, or free the princess in *Super Mario*. (Not all games have apparent victory conditions, for example look at an exploration game like *Dear Esther*.) Key is of course that the player has some idea about what the victory condition might be. In our story, the first few lines already make it clear: Escape! Which then turns into the question, how .... ◀

Having some idea about what you need to achieve is good, but even better is knowing *how* you can achieve it! This is where game mechanics and events come into play (literally).

Game Design 2 (Game mechanics and events) A *game mechanic* defines how the game works. It is a rule we implement in code, determining what a player can do. If the player does "this," then "that" is going to happen. For example, many games use the *A* button on the (Xbox) controller to make your character jump. That is a mechanic. Or, in text adventures, you type in a command – that is another mechanic. When you are playing a game, you use these mechanics to make things, *events*, happen. Things may

go one way or the other, depending on what you decide to do. Take again *Dark Souls* – as the monster attacks you, do you press *L1* to raise your shield and block the attack, or do you use your left stick and *O* to roll to the side? Different mechanics, allowing you to take different actions, resulting in different ways in which the game might play out ... ◀

Let's use that to work out our story a little bit more. The victory condition is to get out of the house but the only door outside is locked. Let's assume that that means the player needs to find a key. As the game wouldn't be particularly exciting if the key would be laying there right in front of the player's nose, we need some game mechanics for the user to go around the house, and look for objects. And that should, of course, lead to some "interesting" events ...

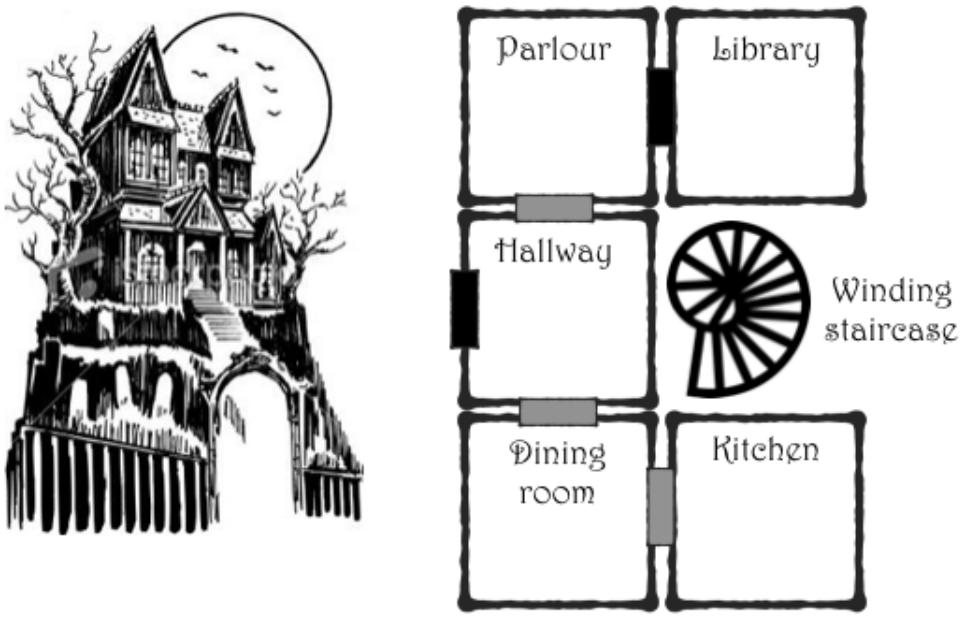


Figure 3.1.: The Haunted House In The Forest

Figure 3.1 shows the map of the house. The black and grey triangles indicate doors. Grey means the door is open, black means that door is closed. The player needs to find the right key to open a closed door. The player can find the key to the *Library* in the *Parlour*. The key to the door outside is in a handbag in the *Kitchen*.

Now, this would not be a scary story if there would not be any monsters around ... So let's place a Ghoul in the kitchen, to guard the Key To Outside. This is going to be our *Boss Fight*!

We should not allow the user to simply waltz into the kitchen, completely ignore the Ghoul, and just grab the key. No – if the player does that, "You Died!" Instead the player needs to find a weapon to defeat the Ghoul. The Library is a good place for that. We can put a sword above the mantelpiece. For good measure we also put a Ghost in

the Library. The Ghost floats in front of the sword.

Those are our key objects: the key to get into the Library (found in the Parlour), the sword to kill the Ghoul (found in the Library), and the key to get out of the house . The player can find that final key in a handbag in the Kitchen. To spice things up, we can let the Ghoul carry the handbag around its neck.

We now have our locations, our objects, a monster – what we need now still is define what the player can do.

We have objects, so a player should be able to **take** an object – "take sword" or "pick up key."

We are not going to place these objects in plain sight, as that would be too easy. The player should **search** – "search room" or "look in handbag."

Naturally, the player needs to be able to get around the house. We can do that "old school"-style using **go** with a compass direction – "go north", "go south." From the Hallway you can go North to the Parlour, or South to the Dining Room, etcetera.

Finally, as we have monsters, the player needs to be able to **attack** a monster – or **retreat** if the monster is too scary!

Okay, there we go. All we need now is the story ... and the code.

## 3.3. Getting The Story Going

Let us have a look again at the code we have so far.

```
1 print("You are in a dark, spooky house, all alone out in the forest.")
2 print("There is only one door to the outside, and it is locked.")
3 print("You hear strange, creaky sounds.")
4 command = input("> ")
5 print("Boo!")
```

As our story is going to continue well beyond the cheap jump scare in line 5, we can delete the "Boo!" Instead, we should take stock of the situation. What do we know? Well, we know that the player does *not* have the main key, nor the key to the library, nor the sword – nor the handbag for that matter. All of that is key to achieving the victory condition:

1. Search the Parlour to find the key to the Library
2. In the library, attack the Ghost, then take the sword
3. In the kitchen, use the sword to attack the Ghoul
4. After the Ghoul has been defeated, take the handbag
5. Search the handbag, find the main key

Once the player has the main key, he should go to the Hallway, and escape the haunted house. Game over, "You Win!"

This means we need to track what the player already has. We can use variables for that. A variable is essentially a "container" you can put a value in (assign a value to). For example, look at line 4 again:

```
command = input("> ")
```

We introduce here a variable `command`, to which we assign a value we get from the function `input` – i.e. command stores what the user has just typed in. In Python, there are various types of variables. The `command` variable is a variable of type *String*. A String stores text. If the user types in *go south* and presses enter, then the *value* of `command` becomes 'go south'.

EXERCISE 1 *Try this out in* `idle`*. Type line 4 in* `idle`*, and then type in a command. After you have pressed enter, type in* `command` *(the variable name), and press enter again.* `idle` *shows the value assigned to that variable.* ◀

There are also various other types of variables. Two types you will frequently encounter are *Integer* variables, and *Boolean* variables. An Integer variable stores (whole) numbers.

EXERCISE 2 *Try this out in* `idle`*. Type in* `score = 0`*. You have zero points! To check that, type in* `score` *(and press enter), and you will see – 0. If you want a higher score than that, simply add that to the variable, like so:* `score = score + 100`*. This means, "take the current score, add 100, and store the resulting number again in score." Now add another 10 points, and check what value is assigned to* `score`*.* ◀

EXPERT KNOWLEDGE 2 (INTEGER ADDITION SHORT CUT) If all you want to do is add a number to the current value of an Integer variable, and assign the result again to that variable, (like we did above), then Python has a shortcut for that. Instead of `score = score + 100` you can also use the short-cut `score += 100`. ◀

A Boolean variable is a variable that is either `True` or `False`. We can use such variables to track whether the player has certain items. Does the player have the sword? The main key? At the beginning – none of that, so we set all those variables to `False`.

We often call such Boolean variables "flags" A flag variable is a variable you define to have one value until some condition is true, in which case you change the variable's value. For example, `hasSword = False` until the player has taken the sword in the Library – then it becomes `True`. It is a variable you can use to control the flow of a function or statement, allowing you to check for certain conditions. So, only once it is the case that `hasSword = True`, the player can attack the Ghoul and stand a chance to survive that encounter.

Type in the code below, before the first `print` line. Something new: Lines 1 and 6 start with `#`. This indicates a comment; Python happily ignores those lines. Note that you do not need to retype all the print statements; the full code listing is included for illustration.

```
# Initialize flags
hasMainKey = False
hasHandbag = False
hasSword = False
hasLibraryKey = False
# The story begins -- the Hallway
print("You are in a dark, spooky house, all alone out in the forest.")
print("There is only one door to the outside, and it is locked.")
print("You hear strange, creaky sounds.")
command = input("> ")
```

What is the first thing that the player might do in such a situation, other than scream "lemme out!?!"? Well, either of two things. Either the player indeed says "help," or something like "look around." If the player says "help", we can help him (or her) out by saying some soothing words, and briefly explain what actions can be performed. Type in the code below. After the code we explain what this new `if..` construction means.

```
11 if (command == "help"):
12     print("Don't be scared. You can do a couple of things:")
13     print("- 'go' in a compass direction, e.g. 'go south' ")
14     print("- 'search' to 'search room', or look into an object")
15     print("- 'attack' to attack a monster")
16     print("- 'take' an object, e.g. 'take key' ")
```

The `if` statement in line 11 is a check. If the player typed in *help*, then do whatever comes after the `:` – in this case, several print statements. For Python it is very important that the block of code that should be executed, is properly *indented*. As you can see, the `print` statements do not begin at the same position as the `if` position, but a little further. Atom does this automatically.

Another thing you may notice is that we use a double "==" and not a single "=" to make a check. This is important, and makes all the difference. A single "=" *assigns* a value to a variable – a statement `score = 10` assigns the value 10 to the variable `score`. A double "==" compares the value of the variable, to the value provided. So, `score == 10` is a check whether the score is 10 (or not) – or, as in line 11, we check whether the player's command was "help."

Expert Knowledge 3 (Single or double quotes?) Finally, once we are talking about things you may be noticing: When you used `idle` to check the value of a String variable, the value was returned with single quotes, e.g. `'help'`. But now, line 11 is using double quotes – `"help"`! Why, what is the difference? Simple answer: There is no difference. You can use double quotes or single quotes, and to Python it is all the same. In general, we use double quotes here, and in lines 12 through 16 you also see we are using single quotes inside double quotes ... not a problem. ◀

The following code provides the player with more details about the Hallway, should

he be looking around. Looking at the map in Figure 3.1 (on p.20) again, we see that the player has several possibilities.

```
17 if (command == "look around"):
18     print("You find yourself in the hallway.")
19     print("To the east is a winding staircase, on the verge of collapse.")
20     print("To the west is the door to the outside.")
21     print("You can go north, and south.")
```

Before we let the player go to the Parlour, let's handle the case where somebody is really scared and desperately wants to get out of this house as fast as possible. What if the player says `go west`? Well, if the player has the main key – fine, go ahead, you win. But if not ... So what we need to do, after checking whether the command is `go west`, is what the status of our `hasMainKey` flag is. No key? No exit.

```
22 if (command == "go west"):
23     if (hasMainKey == True) :
24         print("You have escaped the house! You can breathe easy now.")
25     else:
26         print("The door is locked. You need a key...")
```

Line 23 checks whether the player has the key. If that is the case (`hasMainKey == True`), then the player wins. If that is *not* the case, then we get to the *else* statement in line 25. Note that this other case means `hasMainKey` is `False`, as a Boolean can only have one of these two values. The player is informed that he cannot go ("oh nooooo....") but he does get a little hint: go look for the key ...

EXERCISE 3 *Run the program in the terminal. Type in a command. What happens, after you have typed in the command?* ◀

(In case you read this already without doing the exercise first – no cheats or shortcuts!) Indeed. After you have typed in a command (help, look around, go west), the program ends. That's not much of an adventure of course, so we should change that. Add another statement `command = input("> ")` at the end of the long `print` blocks for "help" and "look around", and at the end of the "else" block (after line 26). Make sure to use the right indentation!

EXERCISE 4 *Run the program in the terminal. Type "help." After that, try "look around." Does that work? Now try "go west." What happens if you* now *type in "look around"? Can you explain why?*[1] ◀

It's time to get the player moving around. First, we go north, to the Parlour. There the player can find the key to the Library, and once we are in the Library, there is a Ghost to deal with before the player can then take the sword ...

[1] The answer: We go step-by-step through the program, and in this program you cannot go back. So once you are passed "look around" the only next step you can take is "go west" as the program no longer checks for "help." Not great, indeed, but it's a start. Later on we will see how to do better.

```
27 if (command == "go north"):
28     print("You go through the door, into the Parlour.")
29     print("Heavy curtains cover most of the windows. ")
30     print("It smells dusty here.")
31     print("There is a door to the west, which appears locked.")
32     command = input("> ")
33     if (command == "look around" or command == "search room"):
34         print("In the faint light, you see several chairs, and a coffee
               table.")
35         print("A key is laying on the coffee table.")
36         command = input("> ")
37     if (command == "take key"):
38         hasLibraryKey = True
39         print("You found the key to the library!")
40         command = input("> ")
```

Once you have typed in the code above, let's have a closer look.

First of all, notice the `or` in line 33. This means: if the player typed in *either* "look around" *or* "search room", execute the block of code after that. So we offer the player a little bit of flexibility here in how he can phrase what he wants to do.

EXERCISE 5 *If you like, go back and change the code in line 17, to match line 33.* ◀

Another thing is that we are starting to change our flags. Assuming the player has seen there is a key, and then decided to "take key" we can now set the flag `hasLibraryKey` to `True`.

Alright, so now the player has the key to the library. If the player now says, "go west" – no problem. Or, if the player has not looked around so far, and thus does not know there is a key to pick up, then "go west" should not take him into the Library. Just like we saw in line 23, we can check our flags, and handle that too.

Or so it seems ... but first things first. After the block for taking the key, add the following. Make sure that the indentation is at the same level as the `if`-statements for looking around, and taking the key.

```
41 if (command == "go west"):
42         if (hasLibraryKey == False):
43             print("The door to the library is locked.")
44             print("You need the key to the library to open the door.")
45         else:
46             print("You unlock the door, and enter the Library")
```

EXERCISE 6 *Do you already see the problem that we are running into? See also Exercise 4.* ◀

Everything is fine as long as the player takes the steps exactly in the order as we programmed them. Look around, take stuff, go somewhere. But, oooh, what if the

player does things in another order? The program is going to be messy – "spaghetti code" as that's sometimes called. If we handle everything in a very strict, linear fashion ("on rails") then we need to take care of every possible order in which the player might be doing something. Suffice to say that that's not a good way to do it.

So why did we do it that way then in the first place? Well, to show you what it is like. Understand the problem, and then in the next chapter, we discuss how to properly solve it!

EXPERT KNOWLEDGE 4 (REFACTORING) It often happens that you start programming, and as you go along, you find out that your solution is not quite what it should be. Even though there are local improvements you can still make, you probably realize that ultimately this particular implementation is a dead-end. And that is fine! You have learnt an important lesson – how *not* to do. Now what you can do is take the pieces of code that you are happy with, and put them together in a different, better way. We call that *refactoring your code*. Remove the spaghetti and the bad ideas, take what is good, then build a new and better solution. ◀

Still. Although we know now that this is not the way to go, we can still go and extend our code a little bit. There is still a Library to explore, and more importantly, there is a Ghost to scare away.

```
41      if (command == "go west"):
42          if (hasLibraryKey == False):
43              print("The door to the library is locked.")
44              print("You need the key to the library to open the door.")
45          else:
46              print("You unlock the door, and enter the Library.")
47              print("You immediately notice the free floating full torso
                    vaporous apparition.")
48              print("In other words, there is a ghost.")
49              print("It is floating in front of a sword, hanging above the
                    mantelpiece.")
50              command = input("> ")
51              ghostPresent = True
52              if (command == "look around" or command == "search room"):
53                  print("What are you doing?! There is a ghost flying around!")
54              if (command == "attack" or command == "attack ghost"):
55                  print("You slowly walk towards the ghost.")
56                  print("The ghost appears to be reading a book.")
57                  print("As you approach, she looks towards you, about to say
                        'Shh!'")
58                  print("You respond with a questionable 'Boo?!' ")
59                  print("The ghost disappears, whispering 'the kitchen,
                        danger...'.")
60                  ghostPresent = False
```

Just for the purpose of this individual scene, we introduce a *local* variable called `ghostPresent`. It simply tracks whether the ghost is still here, or not. We initialize it in line 51 to `True` is the ghost is in the Library when we enter. Only once we have chased it away, with a rather feeble "Boo?!" at that, we set it to `False` (line 60). If the player now tries to take sword, all is fine. If the ghost however is still there, see what happens.

```
59              print("The ghost disappears, whispering 'the kitchen,
                    danger...'.")
60              ghostPresent = False
61              command = input("> ")
62              if (command == "take sword" and ghostPresent == False):
63                  print("You reach up to the sword, and take it.")
64                  print("Be careful not to cut your fingers, as it is
                        sharp.")
65                  hasSword = True
66          if (command == "take sword" and ghostPresent == True):
67              print("As you try to reach thru the ghost, to reach the
                    sword,")
68              print("She turns into a frighteningly old lady, flies through
                    you,")
69              print("and covers you in sticky ectoplasmic goo.")
70              print("The ghost is still floating around, muttering angrily.")
```

The code listing above repeats a bit of the previous code, to make sure it is clear how to indent the code. The first "take sword" block is *within* the overall "attack" block, whereas the second "take sword" block is indented at the *same* level as "attack" block. In other words, if you immediately go and grab the sword without attack the ghost first, you will get slimed (rather than get the sword).

One new thing in the code above is the use of the `and` operator. Whereas `or` indicates that one of the conditions for the `if` must be true, the `and` indicates that they must *all* be true.

EXERCISE 7 *Look at line 62. Is the second condition* `ghostPresent == False` *needed? How about the statement* `ghostPresent == True` *in line 66?*[2]

That's it. Make sure to stage and commit the code to `git` (cudos if you have been doing that after each step!), and run the adventure. If you stick to the right order, it is definitely fun to play! But, yes, indeed, there is room for improvement. In the next chapter, we look at bringing in the required flexibility, and we will complete the story.

[2]The answer: Look at what the player must have done to arrive at line 62. He must have attacked the ghost first. That means that there is no need to check, as we know that the ghost is gone. Something similar holds for line 66. We know that the player has not attacked the ghost, so the ghost must still be there. Again, we do not need to check for the condition, as there is only one possible value for `ghostPresent` at that point. However, if the code would support the player to perform these actions in any order, then these conditions are of course necessary!

# 4. Till The Bitter End

## 4.1. Introduction

In the last chapter we saw how our code gradually became ever more complex. The code reflected a very strict order in which the player had to perform the actions ("the right way – and the only way"). Any divergence from that resulted in us having to introduce more and more `if`-statements to cover those alternatives.

Things became so complicated that we didn't even manage to get all the way to the end! So that is what we are going to do this chapter. We are going to refactor our code, (basically write a new program with bits from the old one), using two new constructions in Python: *loops* and *functions*.

A function is, simply put, a block of code that is outside of your "main" program, and that you can call to do something. Especially when you have to do that something many times, implementing it as a function is very useful. You only have to implement it once, rather than repeat the code at every point where you need it. And, if you need to make a change, you only need to make it once (namely in the function) – rather than at every point where you are using that code!

We have used functions in our code already as well: `print()` is one, `input()` another. The `print` function does something for us and that's it, whereas `input` does something too and then actually returns a result.

What is more, looking at our code, there is plenty of repetition there. A good example is handling "help." In principle the player can ask for help anytime he wants. However, it would be very cumbersome to include all the help text in our code, every time we need to handle "help." So that is a good candidate for a function!

That brings us to *loops*.

## 4.2. Loops

A loop is a, literally, something that goes round. And round. And round.

Sometimes you have a block of code that needs to be repeated like that. And rather than copying-and-pasting it a zillion times, so that you can execute that code that many times, Python offers different ways to formulate loops in your code.

The kind of loop construction we use in this chapter mostly is the `while` loop. In a `while` loop we repeat a block of code, *while* a certain condition holds:

```
1 while (...some condition...):
2       #do something
```

For example, for our adventure game, we want to continue going *while* the player is still alive, and has not yet escaped the house:

```
while (isAlive == True and hasEscaped == False):
    #do something
```

What is important to understand is that somewhere in the `#do something` code block, these flags should change. At some point, the player should be able to escape, setting `hasEscaped` to `True`. At another point, the player may die an unfortunate death, setting `isAlive` to `False`. Why is that important? Because if the flags used in the condition would never ever change, then the loop would never end. You would end up with an infinite loop.

EXPERT KNOWLEDGE 5 (BREAK STATEMENTS) Strictly speaking, you *can* escape an infinite loop using a `break` statement within the loop.

```
while True:
    # do something
    if (the right conditions): break
```

Sometimes you may have too many different conditions to check. Sometimes, it may be that "depending on the situation..." you want to have sometimes these, sometimes those conditions drive the decision whether to stop looping or not. When you need to check for such *local* conditions, (rather than a few *global* conditions like `isAlive == True`, then an infinite loop with suitable `break`s may be a better solution. ◀

Another useful loop construction is the `for`-loop.

```
for <something> in <object>:
    #do something
```

A `for`-loop in Python cycles over *something*'s from some *object*. For example, imagine that we model the player inventory as a list of items `["sword", "library key", "handbag"]`. If the player then asks, "what do I have," you can use a `for` to go through the list, and print one after other.

```
playerInventory = ["sword", "library key", "handbag"]
for item in playerInventory:
    print(item, end=', ')
```

EXERCISE 8 *Type in the code above, and run it. Change the* `end` *bit into* `end='--'` *and run again. What if you completely remove the* `end` *part, (and the comma before it)?* ◀

## 4.3. Restructuring Our Code

Now let us get started with our new-and-improved code. Best you create a new file, and save it under any name you like.

We begin with setting up our flags. Like before we use a small set of flags to track *player inventory*, e.g. `hasMainKey`. And, to control our loop conditions, we also introduce a number of *player status* flags.

```
# Python Adventure v2.0

# Initialize player inventory flags
hasMainKey = False
hasHandbag = False
hasSword = False
hasLibraryKey = False

# Initialize player status flags
isAlive = True
hasEscaped = False
isEntering = True
room = "Hallway"
```

The flags `isAlive` and `hasEscaped` should be clear by now. We will come to `isEntering` and the `room` variable in a minute.

Let us first set up the function for "help." As we said before, a player should be able to ask for help at any point in the adventure, but we don't want to have to repeat typing the help text every time. Instead, we can make a function call.

We define a function using the `def` keyword. After that keyword comes a list of parameters, but as our function is not going to take any, we leave that aside for the moment.

```
# Function to show the help text
def showHelp():
    print("Don't be scared. You can do a couple of things:")
    print("- 'go' in a compass direction, e.g. 'go south' ")
    print("- 'search' to 'search room', or look into an object")
    print("- 'attack' to attack a monster")
    print("- 'take' an object, e.g. 'take key' ")
```

Whenever the player now types in "help", all we have to do is call this function, and the help text is shown!

```
if (command == "help") : showHelp()
```

Time to start defining our `while`s. If you look at our adventure, there are essentially two loops.

One is the main loop: continue the adventure while the player (character) is alive, and he has not escaped yet.

The second is loop is *when the player is in a room.* Think about it for a moment – why a loop? Well, one way to approach that question is by looking at the trouble we ran into. It became difficult to allow the player to perform actions in any order. And it became difficult to then deal with moving to other rooms.

Now imagine that whenever a player is in a room, you loop. In that loop, you handle whatever action the player wants to perform. You *escape* that loop when the player moves to another room – where a similar loop then begins again. Get it? Let us look at the basic structure of the code, and then all will become clear.

```
21 # Main loop - as long as the player is alive, and has not escaped, continue
22 while (isAlive == True and hasEscaped == False):
23
24     if (room == "ROOMNAME"):
25         if (isEntering):
26             # print the room description
27             isEntering = False
28         # Loop within the room, as long as player is not leaving, and alive
29         while (isAlive == True and isEntering == False):
30             command = input("> ")
31             if (command == "go DIRECTION"):
32                 # change the room, set flag that player is entering
33                 room = "ROOMNAME"
34                 isEntering = True
35             if (command == "look around" or "search room"):
36                 # print description
37
38             if (command == "attack"):
39                 # if there is something to attack, handle attack
40                 # otherwise provide a witty statement
41             if (command == "take OBJECT"):
42                 # check whether the object can be taken; if so take
43             if (command == "help") : showHelp()
```

Line 22 sets up the main `while`-loop. We saw this one before: run the adventure as long as the player is alive, and hasn't escaped yet. Next, remember that earlier on (in line 13 – how ominous!) we set up the `room` variable. We initialized it with the value `"Hallway"` as this is the room the player starts in.

Within the main `while`-loop we then check which room the player is currently in. Assume for the moment that line 24 reads `room == "Hallway"` so that we enter that block. Then the first check we make (line 25) is whether the player is *entering* that room. Why? Well, only upon entering a room are we going to display the room description. As soon as we have done that, we set the flag `isEntering` to `False`. The player is now in the room.

Next comes the next loop, within the room itself (line 29). As long as the player is not leaving, and is still alive, we let him do whatever he wants (sort of). At the beginning of the loop, we get the next player command (line 30), and then we have several `if`-blocks to handle the various commands.

If you look at the `if`-block for "go", you see how we escape from the room loop – by going to another room! Depending on the direction and where the player decides to go, we set the `room` variable to the appropriate name (e.g. "go north" from the Hallway sets `room="Parlour"`) *and* we set still true at this point), and then the new value for `room` gets us into the code block for a different room. Where the next room-specific `while`-loop then starts.

# Index