

# Chapter 2

## Programming Is Fun

A programming language is a way to give commands to a computer. These commands are written as text and when we put many commands together we call that a program. Python® is a high-level, easy-to-read, interpreted language. We will see how to install Python together with an application called PsychoPy (which will be discussed in Chapter 3). We will also see how to write simple commands, perform mathematical operations, and control the flow of the program with `if`, `for`, and `while` statements.

We have seen in Chapter 1 that illusions are fun. In Chapter 2 you will see that programming is also a lot of fun. OK, maybe not in the same way, and not without a little effort, but the effort required for what we will do in this book will not take much time, and the fun will continue forever (once you have started programming).

Programming is about asking a computer to do a series of tasks. Computers are clever and there is a lot that they can do. This list of commands needs to be written as text, therefore a computer program is always a text document, called the **source code**, but we will use the more general term **program**.

In order to be understood by the computer the program needs to be in a computer language, and therefore learning to program is about learning a new language. You may be familiar with the names of several programming languages such as Fortran, Basic, Pascal, C, Java™ and Python. They are a bit like French, Italian, Russian and so on; some things are common to all languages (the distinction between a verb and a noun) and some are specific (like learning the word for **cat**).

Because programming, in addition to knowing the rules of the language, requires a certain organisation and strategy, people debate whether programming is a skill, a craft, or an art. It is probably all of these things, which is what makes it fun in the sense that it allows us to be creative.

This is a very brief introduction to one general-purpose language called Python. I will discuss how to write simple commands, and also mention the few quirky aspects that may not be intuitive, like the fact that we will have to remember to count starting from 0 instead of 1.

## Python®

It was Christmas time in 1989, in Amsterdam. **Guido van Rossum** (then 33) decided to write something that would go beyond the limitations of the languages he had worked on before, but without trying to come up with the perfect language (as a hobby project, something fun to do). We have all done that, creating something revolutionary over the winter holidays just because we were bored!

The name Python has nothing to do with snakes. Guido chose it because he was a fan of a TV show called **Monty Python's Flying Circus**. If you are too young to know who Monty Python are, do some research and watch some of their classic sketches. Then try to drag yourself away from YouTube and back to this chapter. The link with the TV show lives on because the official documentation often contains references to sketches from Monty Python (so don't be too surprised if the documentation refers to dead parrots for instance).

The project captured the interest and admiration of lots of other people and today Python is one of the most used and important languages around. True to its original philosophy, it is free and open-source, and has a community-based development model. The development is managed by the non-profit **Python Software Foundation** ([www.python.org/psf/](http://www.python.org/psf/)).

Because of its popularity you will find a large amount of useful information and tutorials online. The Python Software Foundation webpage is a good starting point. This chapter cannot be as detailed as an online tutorial; we will instead focus on the key concepts. A list of online tutorials is provided in Box 2.1. One word of caution, Python 2 is a bit different from Python 3, I will only talk about and use Python 2 (which is still the more popular version).

### Box 2.1. Online tutorials and books

There are the links to some tutorials for beginners specific to Python 2.

- This first one is directly from the Python Software Foundation:  
<https://docs.python.org/2.7/tutorial/>
- This is a tutorial created for programming games but useful for anybody even with no programming experience:  
<http://sthurlow.com/python/>
- This is a very short introduction that fits on one (long) page, created by Magnus Lie Hetland:  
<http://hetland.org/writing/instant-hacking.html>
- This tutorial is for non-programmers and is a featured book on Wikibooks:  
[http://en.wikibooks.org/wiki/Non-Programmer%27s\\_Tutorial\\_for\\_Python\\_2.6](http://en.wikibooks.org/wiki/Non-Programmer%27s_Tutorial_for_Python_2.6)
- For PsychoPy there is a reference Manual here:  
<http://www.psychopy.org/api/api.html>

These two are tutorials that introduce Python as well as PsychoPy.

- The first one is from the Technical support group of the Radboud University in Nijmegen (Netherlands):  
<https://www.socsci.ru.nl/wilberth/psychopy/index.html>
- The second is from the GestaltReVision group (University of Leuven, Belgium):  
[http://nbviewer.jupyter.org/github/gestaltrevision/python\\_for\\_visres/blob/master/index.ipynb](http://nbviewer.jupyter.org/github/gestaltrevision/python_for_visres/blob/master/index.ipynb)

Although they are not tutorials, Jon Peirce has also published a couple of journal articles about PsychoPy. These are aimed at the academic readers.

- Peirce JW (2007) PsychoPy – Psychophysics software in Python. *J Neurosci Methods* 162:8–13
- Peirce JW (2008) Generating stimuli for neuroscience using PsychoPy. *Front Neuroinform* 2:10

Finally, although there is so much information available online, if you prefer to hold in your hands a book made of paper, here are two options (among many):

- Shaw ZA (2013) *Learn Python the hard way: A very simple introduction to the terrifyingly beautiful world of computers and code*. Addison Wesley Press, Reading MA  
This is the 3<sup>rd</sup> edition of a book that gives step-by-step instructions.
- Briggs JR (2012) *Python for kids: A playful introduction to programming*. No Starch Press, San Francisco  
This instead is written for kids, but nevertheless it is a proper introduction to Python and can be used by anybody new to programming.

## Setting Up

Setting up is not the most fun part, but it will not take long and it needs to be done only once. The first thing to do is to install Python on your computer. Here is the good news: it is free, easy and possible on any major operating systems. Because of wanting to create illusions we will install it together with some useful libraries and packages. In our approach this is going to be a single download. The instructions to download and install Python together with an application called PsychoPy are in Box 2.2. I will explain more about this application and these libraries later, so for now just follow the instructions as a recipe.

### Box 2.2. Downloading Python and PsychoPy

Using a web browser go to [www.psychopy.org](http://www.psychopy.org).

On the right side you will see a link called **download**. This will take you to a different location where the files are available. Here you will see a list with many different versions. We will use version 1.84.02 and therefore you can ignore everything else and select the file for the Operating system that you are using. For example:

- StandalonePsychoPy-1.84.02-OSX\_64bit.dmg (if you use a Apple Macintosh® computer)
- StandalonePsychoPy-1.84.02-win32.exe (if you use Microsoft Windows®)

It is quite possible that by the time you are reading this the latest version has a higher version number. Older versions remain available by following the link to <https://github.com/psychopy/psychopy/releases>.

The word standalone refers to the fact that in a single file you are actually obtaining everything you need, including Python itself.

Mac® users should place the PsychoPy application inside the Applications folder. After installation MS Windows® users will find a link to PsychoPy in

- Start > Programs > PsychoPy2

There is also a Configuration Wizard with more information about the setting up.

The standalone file is large and it may take a bit of time for the download to complete.

As there are different versions of Linux, if you are a Linux user you need to read and follow the more specific instructions available here: <http://www.psychopy.org/installation.html>

The programs that we are going to write were created on a Mac®. However, it is my belief that they are suitable for any environment without modification.

The fact that there are many versions of PsychoPy (we will use 1.84.02) is a clue to the fact that this software is under active development, and improvements are released regularly. However, do not worry about which version you are using. It is not a good idea to feel that you have to race and get the one just released (although it is tempting).

## Introduction to Programming

Since we have now downloaded and installed both Python and PsychoPy in a single step, we will use the PsychoPy application to learn a bit more about Python. Remember, Python is a language and for now we will focus on the structure of the language, so what we are doing is very general. We will issue a command and execute it by typing some text and then pressing the return key. This is called using Python in interactive mode.

The first step is to start the PsychoPy application that you have installed. Then inside the application go to the **View** menu and select **Go to Coder** view (on Windows® this may say **Open Coder view**). This is an important step because if you are seeing the Builder view instead of the Coder view nothing in this chapter will make any sense.

You should see a window similar to the one in Fig. 2.1 (this in particular is how it looks on an Apple Macintosh®). Note the two main horizontal panels. In the lower one there are two tabs called **Output** and **Shell**. Click on Shell and you will see a text that will inform you about which version of Python you are running. On my computer it looks like this:

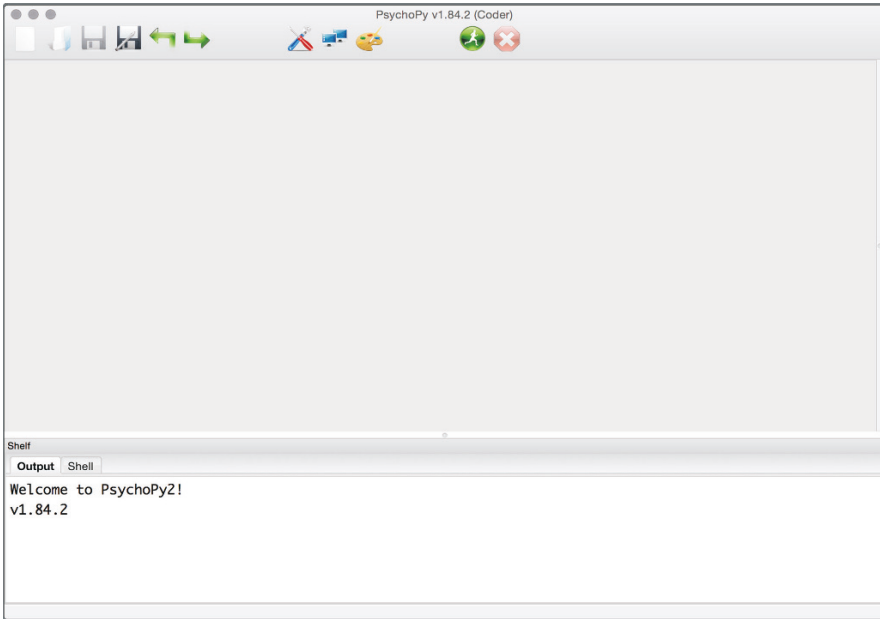


Figure 2.1. The PsychoPy application (Coder view) showing two panels. The top panel is the *Editor* and is empty for now. The bottom panel has two tabs: *Output* and *Shell*

PyShell in PsychoPy – type some commands!

```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 26 2016, 12:10:39)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We can use this window to try out some Python commands and for now you can ignore the rest of the interface. Check that you see the prompt inside the Shell window. It looks like this: >>>

This means that the Python interpreter is listening to you. In other words Python expects an input. When Python will produce an output there will not be a prompt.

It is traditional to start by asking a computer to type the words “hello there”. So please type the following followed by the return key.

```
>>> print("hello there")
hello there
```

The text should appear on a line below your command (without the prompt) as shown above. Congratulations, you asked the computer to do something and it did. Basically the shell environment in which you are working compiles and executes any command right away.

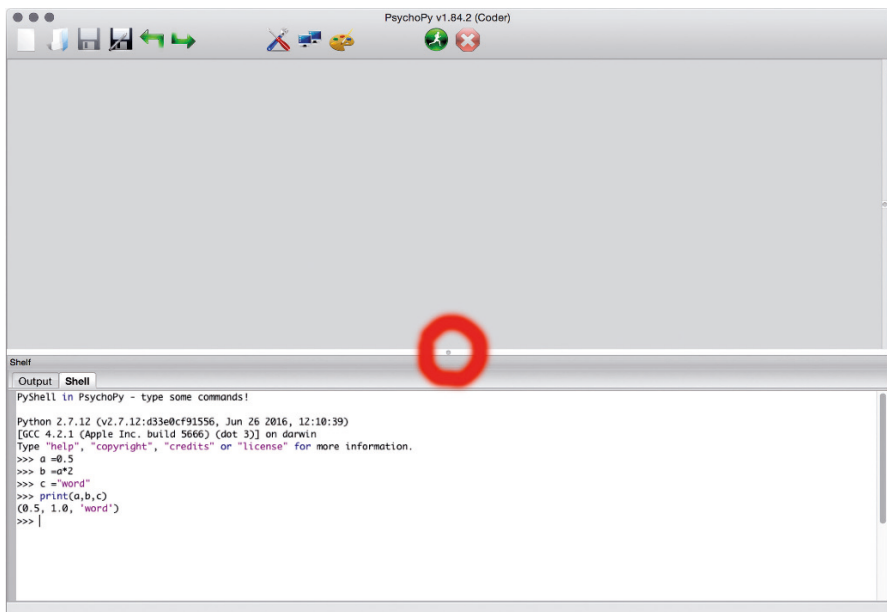
You have also learned that quotes identify a piece of text, and the text can include multiple words. Special words like `print` will appear automatically in blue, and any text like “hello there” will appear in red.

Now for some numbers and some variables. A variable is something that can take values, so the name of the variable is the name of the container.

```
>>> a = 0.5
>>> b = a * 2
>>> c = "word"
```

Variable `a` will have a value of 0.5, variable `b` 1.0 and `c` will be a string (text). No need to worry about the fact that these values are of different types, you can rely on Python to be able to cope with any kind of data. How? Basically by making an educated guess. This is known as **duck typing**. This name comes from the saying that if something walks like a duck, sounds like a duck, and swims like a duck, then it is probably a duck. I am not kidding, duck typing is a genuine technical term. For example by noticing that you used quotes around “word” Python knows that it is a string and the text editor has colour coded it as red.

Note that when you typed these lines nothing happened. The command `print` made something happen, but to fill a variable with a value will not generate an out. If you want to see the content of a variable you need to type its name followed by return, or ask for it to be printed.



**Figure 2.2.** This is the PsychoPy application (Coder view) showing a few lines of commands in the Shell window. Here every Python command that is typed is immediately executed. Note that the size of the panels can be adjusted by dragging the handle (*small dot*) in the middle (highlighted in red)

**Table 2.1. Python data types.** This is a list of the main built-in Python data types, divided in *numerals*, *sequence* types, and *mapping* types. For each I give two examples

Data type	Name	Example 1	Example 2
<b>Numerals</b>			
int	integer	4	5 912
float	floating point	7.23	3.14159
<b>Sequence</b>			
str	string	'hello'	"another example of a string"
list	list	[2,4,'a',9]	['Sheba','Simba','Smudge']
tuple	tuple	(2,4,'a',9)	('Sheba','Simba','Smudge')
<b>Mapping</b>			
dict	dictionary	{'one': 1, 'two': 2}	{'Sheba': 'black', 'Simba': 'ginger', 'Smudge': 'tabby'}

In the case of `print` you can ask for several variables to be printed by writing the names separated by commas.

```
>>> print(a,b,c)
(0.5, 1.0, 'word')
```

In a new line, without the prompt, you can see the values 0.5, 1.0, and ‘word’, as shown also in Fig. 2.2. This time the text ‘word’ is within single quotes (`'word'`) rather than double quotes (`"word"`). Either way is OK for Python.

A list of the main built-in data types in Python, with examples, is given in Table 2.1. We have started to see values for integers (**int**), floating-point numbers (**float**), and strings (**str**). We will discuss lists next, they are very important.

Even though they are in Table 2.1 I will not say much about dictionaries because we will not have a need for this type of data. Briefly, they map a unique key (before the :) with a value (after the :). So you can see how they would be quite useful if you wanted to store for instance details of your cats (for each one a colour, a birthday, and so on). In this book we will focus on programming visual illusions, and therefore the most important types are the numerals, the strings for the text, and the lists for things like positions of vertices.

## Lists and Strings

We often need to store a list of values. To do so we format the values with square brackets and separated by commas.

```
>>> aList = [12,8,4]
>>> aListOfLists = [[12,8,4],[2,0]]
```

Lists are extremely important and we will make good use of them. The first example is a list with three elements, the second is a list of two elements (each with its own elements inside) and therefore it is a list of lists. You may have already guessed, but you could also have a list of lists of lists, and so on. Also, lists can contain elements other than numbers.

```
>>> myCats = ['Sheba', 'Simba', 'Smudge']
```

They can even contain elements of different types.

```
>>> myList = [1, 'Sheba', 2, 99]
```

Next we need to know how to read values from a list. They are accessed with an index for the position of the element, starting from zero. Therefore try the following.

```
>>> print(aList[0])
12
>>> print(aList[2])
4
>>> print(myCats[0])
Sheba
```

Note that `aList[3]` is out of range. That means that we are looking at a position that does not exist and we will get an error. This is because our list has only three positions, 0, 1 and 2. Python is not the only language that counts from 0 instead of 1, it is something that computers like to do so we will have to get used to it.

You can also see a convention about naming variables. To make them easy to read we capitalise any new word within a complex name, for example we wrote `aListOfLists`. This is optional, not something necessary in Python, but we will follow this custom in writing variable names in this book.

Indices are very clever, for example you can give a range. `aList[0:2]` is a list with two elements (the first two). Another clever thing is that you can count from the back using negative numbers, so `aList[-1]` is the same as the last element which in our example is `aList[2]`, and therefore it has a value of 4.

Given that we can specify a specific location we can use this to change a value inside the list. We specify the element using the index, and then use `=` to assign it a new value.

```
>>> aList = [12, 8, 4]
>>> aList[0] = 1
>>> print(aList)
[1, 8, 4]
```

What happens if we use round instead of square brackets? Actually more or less the same, you can also write lists that way, but in Python these are not called lists, they are called tuples. Once they are created tuples cannot be changed, and therefore are less



flexible than lists. For instance what we did in assigning a new value to `aList[0]` would give an error for `aTuple[0]`.

```
>>> aTuple = (12,8,4)
```

If we were to check whether `aList` is equal to `aTuple` we would find that this is `False` (they have the same numbers inside, but they are not the same thing). We will mainly use lists rather than tuples.

```
>>> aList == aTuple
False
```

This comparison is `False`, but note that we need two equal signs to compare two variables. One equal sign simply replaces what is in the first (right side) with the second (left side). It's an easy mistake to make. Try to remember that `=` is always actively doing something, not making a simple comparison. The main Python operators, like `==`, are listed in Box 2.3.

In a complex program we will need comments as well. Comments in Python start with the hash character, `#`, and extend to the end of the line (so there is no need to have an `#` at the end).

### Box 2.3. Python operators

This is a list of the main built-in Python operators to perform comparisons. The result from any such comparison can only be `True` or `False`.

- `==`      equal to
- `!=`      not equal to
- `>`        larger than
- `>=`     larger than or equal to
- `<`        smaller than
- `<=`     smaller than or equal to

This is a list of the main built-in Python math operators (to perform arithmetic operations). The result from any such operation is a number. Some are not used very often, so for example although the modulo operation (`%`) can be quite useful do not worry if it seems unfamiliar to you at the moment.

- `a + b`    addition
- `a - b`    subtraction
- `a * b`    multiplication
- `a / b`    division (can give an integer or a float)
- `a // b`   division and rounding of the result to the lower whole number
- `a % b`    modulo. It gives the remainder of `a` divided by `b`.  
Examples: `11 % 2` is 1, and `11.5 % 2` is 1.5 (what is left from the division)
- `a**2`    `a` to the power of 2, the same as `a` multiplied by itself: `a * a`
- `a**b`    `a` to the power of `b`
- `-a`       negation (changes the sign of `a`)

## Python As a Pocket Calculator

Now let's practice some simple maths. Here is a bit of arithmetic (followed by a comment, which of course plays no part in the maths, it is shown in green here and comments will be green also in all our programs). What is the result of this operation?

```
>>> 4 + 3 * 2 #a bit of arithmetic, four plus three times two
10
```

The result is 10. If you thought it should be 14 this is a mistake (easy mistake to make, it was a bit of a trick question). Just remember the rule of precedence of operations: **power, division, multiplication, addition, subtraction** (PoDMAS). So the multiplication happens first even if it is written after the addition. We can avoid any ambiguity by using brackets when necessary, as in  $(4 + 3) * 2$ .

```
>>> (4 + 3) * 2 #a bit of arithmetic, four plus three, and the result of that times two
14
```

Next, what is the result of this operation?

```
>>> 10 / 4 #ten divided by four
2
```

That's easy, it should be 2.5. Well, your maths is correct but unfortunately this was another trick question. Really tricky this time. Because we expressed the division in terms of integers (whole numbers without decimals) we get an integer and the result is 2. Annoying perhaps, but easily solved by remembering that if we want operations with decimals we should include a decimal point.

```
>>> 10. / 4.
2.5
```

The result now is 2.5, and you don't even need to type the zero because 10. (ten followed by a full stop) is the same as 10.0 for Python.

OK, enough trick questions, now for text, you can use single or double quotes around text. The latter has the advantage that it copes with having apostrophes (single quotes) inside. So you can write "Father's day" using double quotes around it.

```
>>> print("Father's day")
Father's day
```

You can do clever operations with strings, using `+` and `*`.

```
>>> s = "hello" + " there"
>>> father = "pa" * 2
```

You may experiment and use `print` to see the results. Quite logically the results are “hello there” (placed inside a variable called `s`) and “papa” (placed inside a variable called `father`).

We can get the length of a string with `len()`. Therefore, `len("Father's day")` is 12, the total number of characters including spaces. This command applies to lists as well, for example `len([1,2,8])` is 3.

Just like `len(aList)` provides the length of the list, there are other similar built-in functions, such as `max(aList)`, `min(aList)` and `sum(aList)`. They find the maximum value, the minimum value and the sum of all values.

```
>>> sum([1,2,10])
13
```

What if a list has text inside instead of numbers? `max` and `min` will treat strings in alphabetical order, so `max` will give the element that comes last alphabetically, but `sum` will not work with strings. Here are a couple of simple examples using “Father’s day”.

```
>>> len("Father's day")
12
>>> max("Father's day")
'y'
```

One final point about numbers and strings. And here you will see how Python can be really clever and flexible with data types. Suppose you have a number in a variable called, say, `number`.

```
>>> number = 5
```

What you want is to use it as a string, mixed in with some other text perhaps. You can turn the number into a string by saying the following:

```
>>> numberAsText = str(number)
```

Using the built-in function `str()` we are asking Python to treat what is inside the brackets as a string. And you can also go the other way:

```
>>> numberAsNumber = int(numberAsText)
```

This gets back a number again, something on which we can do some maths. This works for other types of variables as well. So here is another way to obtain 2.5 (the correct result) when dividing 10 by 4. We can use `float()` to make sure that an integer is treated as a floating point number.

```
>>> float(10) / float(4)
2.5
```

As we have seen, inside the Shell window Python tries to execute each command typed after the prompt, and it will provide an error message if there is a problem. We can

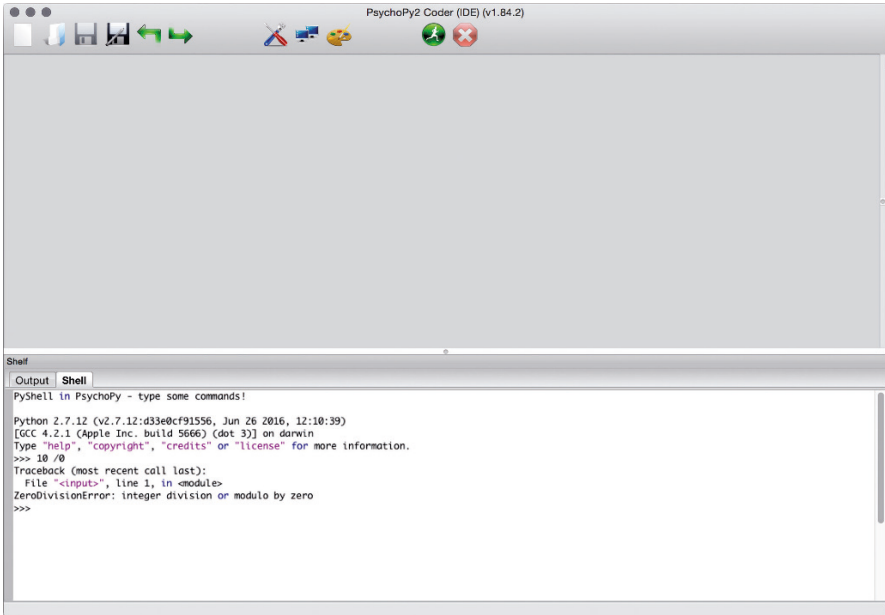


Figure 2.3. The Shell window of PsychoPy showing the error message generated when trying to divide 10 by 0

generate an error for example if we try to divide a number by zero (something that is not possible as there is no defined result). You can see the error message below and in Fig. 2.3.

```
>>> 10 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This is another example of an error generated when trying to find the sum of a string. In this case the message says that the operation is unsupported for this data type.

```
>>> sum("Father's day")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

## Controlling the Flow

We have seen how to issue simple commands, and put values in variables. But the real power of a program is when we write a series of commands and control how they are executed. This is done with special instructions that control the **flow of the program**. There are three control flow statements in Python: **if**, **while**, **for**.

Here is a first example that uses a `for` loop.

```
>>> aList = [1,2,3,4,5] #creates a list with numbers
>>> for item in aList: print(item) #loops over all the elements of the list and prints each one
1
2
3
4
5
```

This loop prints all the numbers in the list. To do that it uses a new variable `item` not used before. This variable is created within the `for` statement. Also note the : before the `print` command, this punctuation is necessary for any control flow situation, and you can read : to mean *do the following* (do say it aloud as you read the program, it does help).

Special words like `for`, `while`, but also `not`, `in`, `print` and others are written in blue in this book and are automatically coded as blue by default in PsychoPy.

```
>>> if len(aList) > 0: print(aList[0])
1
```

This will print the first element in the list. We checked that the list was not empty with an `if` and the `len()` command. This is useful because we can avoid the problem of trying to print from an empty list (which would give an error).

An `if` can also be combined with an `else`. After the word `else` and : there will be commands executed only when the `if` condition is False. For more complex situations where we need to deal with multiple possibilities we can use an `elif`, which is a contraction of the words *else if*. It does exactly what it says, it checks something (but only if the first `if` condition is False).

In a program you may want to use a `for` loop over a large range of numbers. Instead of preparing a long list we can generate the list using `range()`.

```
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

This command is very useful and we just have to remember a few facts about how it works. `range(20)` is a list of numbers from 0 to 19. Those are 20 elements in total because we start from 0, so it does make sense. To get the numbers from 1 to 20 we need to specify the starting number as well as the stopping number: `range(1,21)`. Again this will include 20 but not 21.

Therefore, as we have seen, when `range()` is written with just one parameter it produces all the numbers from 0 to the number just before the one given (the stopping number is never included). If `range()` is used with two parameters the first is the start and the second is the stopping number. Apart from that the result is as before, that is a sequence of numbers excluding the stopping number.

Finally we can even use `range()` with three parameters, in which case the third is the step.

```
>>> range(1,21,2)
[1,3,5,7,9,11,13,15,17,19]
```

This is a list of numbers stepping up two at a time. In this case this means all the odd numbers from 1 to 19 (again no 21, as before the last is always excluded).

You see here an example of how flexible Python is, `range()` can take one, two, or three parameters. The three parameters are **start**, **stop** and **step**. Stop is not optional, it has to be there (we need to know when to stop). If one of the others is missing there is a default value, so if the start is missing the start will be 0, if the step is missing the step will be 1.

Let us make a loop that prints odd numbers from 1 to 19.

```
>>> for item in range(1,21,2):print(item)
1
3
5
7
9
11
13
15
17
19
```

Now, before reading the solution (just below, but don't look!), try and write down (in the Shell window) a way to print all the even numbers from 2 to 20.

```
>>> for item in range(2,21,2):print(item)
2
4
6
8
10
12
14
16
18
20
```

Well done. You can start from 2 go until 21, and use a step of 2. Remember that you still have to put 21 as the end of the range, because if you enter 20 you will only see the numbers up to 18.

We have seen the **for** and the **if**, the **while** control loop works in a similar fashion.

```
>>> item = 0
>>> while item < 10: item = item + 1
>>> print(item)
10
```

Table 2.2. Iteration-dependent values of our variables

Iteration	Value of item	
	as we enter	as we come out
first	0	1
second	1	2
third	2	3
fourth	3	4
fifth	4	5
sixth	5	6
seventh	6	7
eighth	7	8
ninth	8	9
tenth	9	10

This loop will increment the value of item from 1 (the first time, when we add 1 to 0) to 10 (the last time, when we add 1 to 9), but will not execute anything when item is greater than 9 (therefore we will never add 1 to 10).

Sometimes it is useful to write down a little table showing the values of our variables on each turn (called iteration) of the loop. That is the value as we get in and as we come out. Table 2.2 shows how it would look like for our simple example.

A special command is called `pass` and does absolutely nothing. This can be useful when we get into a control flow situation that should have no effect.

```
>>> if myStomachIsFull == True and IAmSittingByTheFire == True: pass
```

I have taken the opportunity here to show a `if` followed by two conditions, joined by the logical operator `and`. You can see how this is easy to read, and special words like `and` and `or` work exactly as expected. The fact that they are automatically coded in blue also helps in making clear that they are special words.

You could read this line aloud like so: if it is true that myStomachIsFull and it is true that IAmSittingByTheFire, then do the following: absolutely nothing.

Here is a different example that uses `or` instead of `and`. Note that the two statements with `==` are either True or False, and only one of them needs to be True for the `if` statement to be executed. After the `:` the command with `=` assigns a value to the variable `isTimeToEat`. `=` and `==` should not be confused.

```
>>> if isLunchTime == True or isDinnerTime == True: isTimeToEat = True
```

## Indentations

There is an important feature of Python, which is very specific to Python and that I cannot easily demonstrate from the prompt. Often we need to group a whole set of commands together. For example we may need to do that after a `if` or a `for`. Instead of using brackets Python uses indentations. This makes the text very clean and uncluttered, but it requires careful attention to the size of the indentation, as there can be multiple levels.

You will see many examples of indentations in the programs but here is a snippet of code with two levels.

```
for x in range(-10,11):#remember these are numbers from -10 to 10 (not 11)
→ for y in range(-10,11):
→ → position = [x,y]
```

Now I will write something very similar but with the wrong indentation. This will produce an error.

```
for x in range(-10,11):#remember these are numbers from -10 to 10 (not 11)
→ for y in range(-10,11):
→ position = [x,y]
```

This is known as an indentation error. After : (read it as *do the following*) Python expects some commands, but the next line is not indented and therefore it is not part of what should happen within the second `for` loop. As there are no commands after the second : this is an error.

When running a script the error message will appear in the Output panel (the one next to the Shell panel). For an indentation error it will print out the file name (containing the script) and the line on which the problem occurred.

File "/Users/marco/Projects/PythonIllusions/myProgram.py", line 78

```
position = [x,y]
^
```

IndentationError: expected an indented block

This is very useful and most of the times you can then fix exactly the line where the problem is. The example above refers to the name of a file from my computer. We will learn more about how to save scripts in files in the next chapter.

Box 2.4 illustrates the way to look at indentations. Basically indentations create blocks, and these blocks are separate groups of commands. Therefore a change in indentation has a direct effect on the execution of the script.

We can fix the problem by moving the third line to be just after the :

```
for x in range(-10,11):#remember these are numbers from -10 to 10 (not 11)
→ for y in range(-10,11):position = [x,y]
```

However, in most cases it is not practical to write what follows after : on a single line. We have done it in this chapter because we were just practicing simple one line commands. In the scripts that we will write as full programs we will need indentations. When we will have a loop, the many lines of commands that need to be executed within the loop should increase by one level of indentation.

In the example above we have seen a message about the indentation error generated by Python. However, for complex scripts an indentation error may also simply change the flow of the commands in a way that does not generate an error. The consequence is that what should be inside a block ends up outside that block. These



errors are harder to find and solve. The moral is that we always have to pay close attention to indentation.

By controlling the flow of the commands programs can be clever. We will also see how to group commands so that these groups will carry our specific tasks. When they are given a specific definition (a name and maybe some parameters) we will call these structures, logically, **functions**, but they will have to wait a bit.

#### Box 2.4. Indentations

Indentations in Python create blocks of commands. For example block 2 may only be executed after a condition is checked with an **if** in block 1, and block 3 may be part of a loop and be controlled by a **for** in block 2. In the book the tabs that create indentations are always shown as arrows. You will not see these arrows when you type in the Editor.

Moreover, you can use tabs or you can use spaces (Python will replace tabs with spaces behind the scenes). In general is a good idea not to mix tabs and spaces.

```
Line 1 in block 1
Line 2 in block 1
Line 3 in block 1
→ Line 1 in block 2
→ Line 2 in block 2
→ → Line 1 in block 3
→ → Line 2 in block 3
→ → Line 3 in block 3
→ Line 3 in block 2
→ Line 4 in block 2
Line 4 in block 1
```

#### Box 2.5. The Zen of Python in 20 aphorisms

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one – and preferably only one – obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than \*right\* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

This chapter has probably set a record as the briefest and most incomplete introduction to Python. Not to worry, what we have learned is enough to start putting together some interesting programs, or at least it will be after we learn about the PsychoPy commands in the next chapter. What is necessary is, however, not to simply read about the way that Python works but to experiment by typing lots of slightly different things, lists, strings and operations from the prompt. Do spend at least ten minutes doing that from within the Shell panel before moving on to Chapter 3.

Finally, before we start using Python you may also be interested in the famous 20 aphorisms written by long-time Python developer Tim Peters (Box 2.5). They give a flavour of the programming philosophy, and some may apply to life in general (and yes, there are only 19 in the list, it's a Zen thing).

<http://www.springer.com/978-3-319-64065-5>

Programming Visual Illusions for Everyone

Bertamini, M.

2018, XI, 221 p. 49 illus., 22 illus. in color., Hardcover

ISBN: 978-3-319-64065-5