Programming in Python

A hands-on introduction

Bart Willems

November 2018

Contents

Chapter 1

Introduction

Python is a programming language that has steadily increased in popularity since its introduction in the late 1990s. It's easy to see why the language is so popular; easy to learn, fast turnaround time for projects, suitable for many different projects, and so on.

One of the main areas of application is "the web," although it is certainly not limited to it. But whenever claims are made that Python is a "toy language," it's safe to point out that many big web companies rely heavily on Python for their daily business: Netflix, Instagram, Facebook, Youtube…But outside the internet there are many other applications for Python. NASA and IBM use it extensively, for instance.

The applications for Python go beyond web development. Python is rapidly turning into the #1 language for data science, with powerfull packages as NumPy and Pandas at its disposal. And traditionally the language has always been used, similar like PERL, as a "glue" language to tie other processes together, or for everyday system administration tasks.

So, while universities like Stanford and the MIT use Python for their introduction classes in Computer Science, don't think that the use of Python stops there. It's a powerfull generic language that can perform many, many tasks.

1.1 Python

The Python language is a *scripting language*, as opposed to a compiled language. Compiling means that code gets translated from the human–readable code into the machine code that the computer works with. A scripting language, on the other hand, gets interpreted at "run time," which is obviously less efficient than compiling.

However, there are many advantages to scripting. For instance, it doesn't take a lot of work to change existing code. If there is an error, or a feature needs to be added or changed, it can be done quickly and with surgical precision. Fixing a bug in a compiled project means that the entire codebase needs to be recompiled; something that can take a lot of time.

Another feature of scripted languages is *platform independence*. With few exceptions, code made on a Mac or a Linux machine will work fine on a Windows computer and vice versa. Compiled programs, on the other hand, are limited to the operating system and processor that they were compiled for.

While certain tasks are certainly more suited for compiled languages, performance is in general not an issue with Python these days. There are various reasons for that:

- Processor-intensive tasks, for instance handling large amounts of numeric data, are usually performed by specialized libraries (that often *do* utilize platform-specific compiled software)
- Execution of Python code is a lot faster than most (non-Python) programmers think it is.
- Python code is high level, meaning that the programmer doesn't have to concentrate on a lot of "plumbing" that is required in languages like C or C++. It's easier to implement algorithms that are better suited for the job at hand, and in many cases using the right algorithm has a bigger impact than executing code really fast.

1.2 Python Anywhere

This course will utilize the **Python Anywhere** website (`pythonanywhere.com`). While it certainly is more comfortable to install Python on your computer and run it from your desktop, Python Anywhere has certain advantages:

- You can run it pretty much anywhere (hence the name!)—no need to install it on a computer first, or run into the brick wall of *not* able to install it on a computer (at the office, for instance)
- If setting up Python on your computer is not your thing, you don't have to worry about that

- Your Python environment will be set up exactly the way I expect it to be set up, without worrying about "this will look different on a Mac" or similar exceptions

For larger projects, or to run your code more comfortable, you can set up Python on your own computer. You will have to do that yourself though, and *you* are the one who will have to translate the differences between this course and what happens on your desktop!

1.3 Installing Python

If you do want to install Python on your computer, the recommendation is to use a scientific package like **Anaconda** or **WinPython**. While it certainly is possible to install a vanilla Python from the Python website (`python.org`), the resulting Python installation is extremely sparse with only the most basic functionality.

And while Python's *batteries included* philosophy certainly means that the basic installation is more than capable for a lot of tasks, it's also lacking a lot of the tools that cause Python to be very popular. Installing some of these tools, like **NumPy** can be very challenging, and that is where the pre-packaged installations come in.

1.3.1 Anaconda

Anaconda is the 800–pound gorilla in the Python distribution package ecology. It's by far the most popular, well rounded package with versions available for Windows, Mac and Linux. You can download your Anaconda package from the Anaconda website:
`https://www.anaconda.com/download/` but be prepared to wait a little while—it's huge!

Anaconda comes with its own package manager, making it easy to install or remove libraries. It also comes with an editor. Anaconda is a great way to get started with Python!

1.3.2 WinPython

WinPython is the Windows-specific alternative for Anaconda. You can find it at `https://winpython.github.io/`. There are only a few benefits of using Win-

Python over Anaconda, but one of them is a gigantic one: WinPython can be installed as a portable program. Even when you're not allowed to install software on your work computer, you can still run WinPython from a memory stick or even straight from a folder on your desktop (or in *My Documents*. The fact that you can actually use it gives, in my opinion, WinPython a gigantic advantage over Anaconda.

Chapter 2

Python Anywhere

Python Anywhere allows you to run Python from a webserver. While this provides an environment with less control than an at-home installation, there are certain advantages to doing so as well:

- Your Python environment is set up and maintained by dedicated professionals
- Many of the commonly used external packages are already installed
- You can run code even when your computer (work, school, library) doesn't allow it
- You can run code while you're not around (webserver, scheduled scripting)
- You can interact with web-hosted data, even if the firewall on your network doesn't let you do that

This chapter will touch upon using **Python Anywhere** , and how to set up the environment used in this course.

2.1 Setting up Python Anywhere

Let's start by using your favorite browser[1] to navigate to `https://www.pythonanywhere.com/`.

[1] Hopefully not Internet Explorer

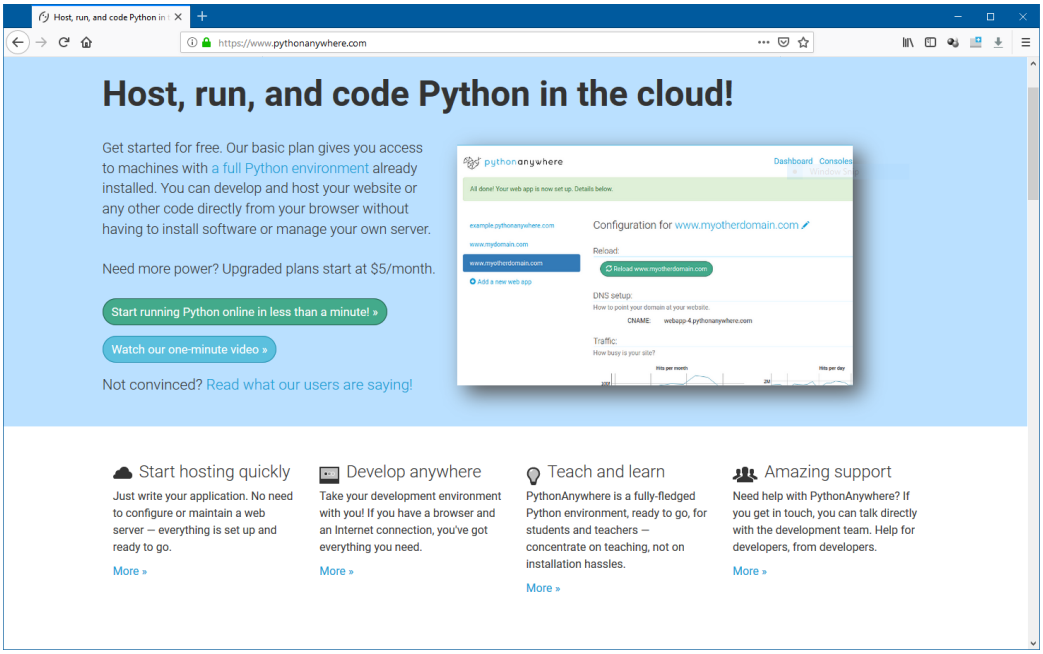

Figure 2.1: Python Anywhere Welcome Screen

At the very top of the web page is a menu bar; click on **pricing & setup** to navigate to the sign-up screen. There are various membership options available; the most prominently visible one is free!

Signing in requires a few details. The site tries to protect your account and utilizes tools as the **Google Authenticator** to provide two-factor authentication.

After signing up (and confirming your identity through email) you will be led to the **Dashboard**.

Should you pay for it?
Until you find programming "your thing," sticking to the free plan is definetely the best choice.
If you find yourself using **Python Anywhere** for an extended amount of time, consider getting a paid "Hacker" account; at only $5 per month it's well worth it.

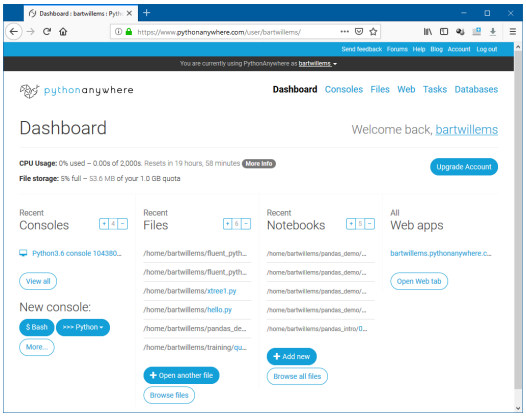

Figure 2.2: Python Anywhere Dashboard

2.2 Navigation

The dashboard is the central hub of your **Python Anywhere** site. On the top right-hand side are the main navigation locations of **Python Anywhere**

Dashboard	Consoles	Files
Web	Tasks	Databases

Of particular interest for us are the dashboard, consoles, and files.

2.2.1 Dashboard

The dashboard provides a quick single-glance overview of what is going on at your **Python Anywhere** site. You can see how many consoles are running, if the web-server is running, what the most recently edited files are, etc.

2.2.2 Consoles

A console contains a session that (usually) runs a Python interpreter. Either a console runs an interpreter by virtue of the script that it runs, or it runs all by itself, in *interactive mode*

A free account is limited to running only two consoles simultaniously; the **Consoles** section is where you can close them (click on the "x" next to an active console to terminate it).

To start an *interactive session*, simply open a new console:

- navigate to the **Consoles** section
- under **Start a new console** click on the console type of choice. Generally the highest Python number is recommended
- after a short initialization period, your new console is up and running!
- click on the **Python Anywhere** logo to return to the home screen
- navigate back to **consoles**
- close the console window

2.2.3 Files

In this partion of **Python Anywhere** , you manage your files. Since Python is pretty much file-driven (scripts are files, program input is usually in a file, output is generally a file), this section is pretty important! On the top left side, right underneath

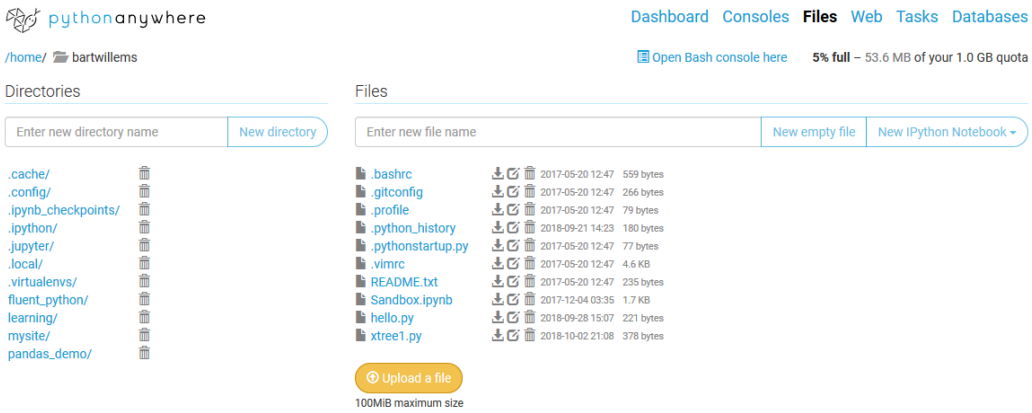

Figure 2.3: Python Anywhere Files

the **Python Anywhere** logo, is a *breadcrumbs trail* that shows the active directory ("folder") that's being shown. Use the breadcrumbs to retrace yourself back into higher directories in the file system.

Directories

On the left side is a list of all directories inside the active directory; you can click on one of them to open that particular directory. The **Python Anywhere** file browser is exceptionally limited and offers only one option: delete a directory (by clicking on the garbage can icon). *It's an exceptionally bad idea to delete any of the directories whose name starts with a period!*

Above the directories list is an empty box where you can type the name of a new directory, if you want to create one; type in the word `course` and click on **new directory** to create a new directory with that name. You can click on the breadcrumbs to go back to your user directory (the one with your name)

In similar fashion, create two directories called `scripts` and `data` inside your `course` directory. We will use those later in the course.

Files

In similar fashion you will find a list of all files on the right side. Text files are created in the same ways as directories (and deleted as well). In addition you can upload files (big button at the bottom), download files (the little down-pointing arrow icon), and edit files (the pencil icon).

Navigate to the `data` directory inside `course` and create a text file called `mary.txt`. Edit its contents as follows:

```
Mary had a little lamb,
whose fleece was white as snow.
And everywhere that Mary went,
the lamb was sure to go.
```

Bash

The editing options within **Python Anywhere** are very limited. Seemingly the only way to rename files is to copy them and delete the old version, which is neither practical nor efficient.

Luckily, there is the option to open a **bash** console. If you're familiar with DOS or the Windows Command Prompt it's relatively easy to use. Google "bash commands" if you want to know more, but here's a short list of the most useful commands:

command	description	example
cd	change directory	`cd course`
cp	copy file	`cp image.png course/image.png`
mv	move and/or rename	`mv image.png course/image.png`
ls	list folder contents	`ls course`
mkdir	make directory	`mkdir course`
rmdir	remove directory	`rmdir course`

2.2.4 Interactive Mode

One of the ways to use Python is in *interactive mode*. It's a great way to try out small things, or to use Python to quickly get some results on your desktop. One of the ways Python is used in interactive mode is as a very advanced calculator.

Start a console by navigating to the Consoles section and click on the latest Python version (depending on your account, likely 3.6 or 3.7). When the command prompt (`>>>`) appears, type a small calculation like `5+7` and press **Enter**.

```
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 4.8.4] on linux
Type ”help”, ”copyright”, ”credits” or ”license” for more information.
>>> 5+7
12
>>>
```

Close the console (navigate to Consoles) once done.

2.2.5 Creating and running a script

A script is nothing more than a text file containing Python code. Navigate to the Files section, and create a new file called `hello.py` (make sure you add the `.py` extension).

Edit the file with the following line of code:

```
print('Hello, world!')
```

Make sure to click the **Save** button, and then click **Run**. The script will run in a mini-console window underneath the edited script.

Congratulations! You just created and ran a Python script!

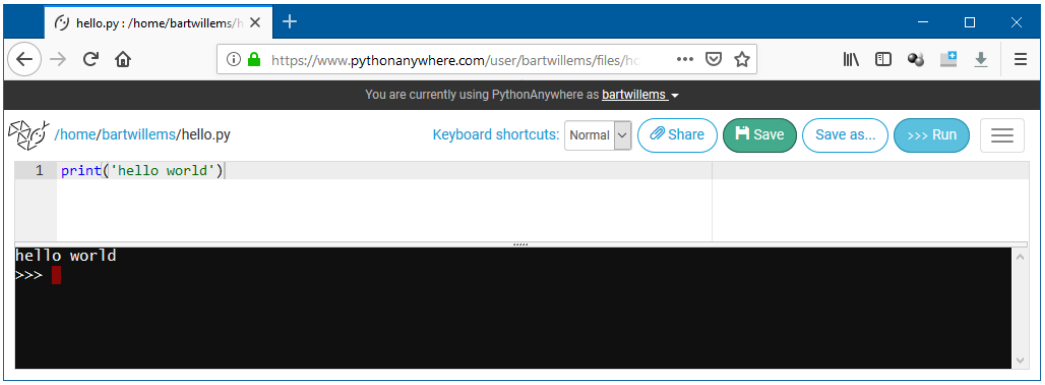

Figure 2.4: Python Anywhere Script

Chapter 3

Creating Scripts – The Christmas Tree

It is time to start coding with Python! In this chapter we'll focus primarily on the process of writing a script and running it. The program code will be very primitive and really not do a lot, and is aimed at getting into the habit of writing code and verifying that it works as envisioned.

3.1 The Christmas Tree

All code in this chapter will be written around a program that prints a christmas tree on screen, like this:

```
    *
   ***
  *****
 *******
    *
```

It's not the most exciting tree! But it can be expanded a bit, and have a message underneath it—enough to write some code around. At the end of the chapter we'll have a real program that can create trees for us in all kinds of variations.

There are various approaches to developing software projects. The classic method is what's called a *waterfall model* where the development "trickles down" from high management decisions to the implementation engineers. Requirements are analyzed, specifications are written up and handed down to the software developers, who on a smaller scale develop a plan on how the software is going to meet those

specifications and eventually they'll write code for it. It may sound very bureaucratic but it's not a bad idea to think before you start writing code.

The opposite approach is what we're going to apply in this chapter; *fast prototyping*. Based on initial requirements a working prototype is developed which is unlikely to be the final product, but it can be used to evaluate the output. Based on that, both requirements and implementation can be finetuned.

It is always good to think ahead before you start coding, even with fast prototyping. Keep that in mind when working through the assignments; the approach might seem very ad-hoc, but that is mainly because there's no foundation to start with. As we progress in the course we'll see a more thoughtful approach towards the first prototypes.

3.2 Creating a script

Before code is written, a canvas is needed to paint the code on.

- Make sure there is a directory `course` in your user directory
- Create a text file named `xmas_tree.py` inside the `course` directory
- Enter text *exactly* as shown below:

```
print('   *   ')
print('  ***  ')
print(' ***** ')
print('*******')
print('   *   ')
print('Merry Christmas!')
```

- Click on the **save** button
- Click on the **run** button

3.3 Looking at the script

While it seems obvious what the script is doing, it's good to step back and take a closer look at what *exactly* is going on when **run** is clicked.

At first, Python is instructed to open the file `xmas_tree.py` and load it into memory. Nothing is actually *done* with the code at this point; the Python interpreter

is purely making sure it can load all the code in the file (and is not encountering a broken file).

Then—and this will become important later on—it will run or "execute" the code one line at a time, working from top to bottom. In this case, each line is pretty similar; they all invoke the `print` function.

A function is a mini program that performs a certain task. Python comes with many functions, and later on we will define our own functions! Not surprisingly, the `print` function will take whatever you give it and output it to the console window.

Functions are always called with parenthesis, similar to using functions in Microsoft Excel. If a value is passed on to the function, it's placed inside the parenthesis. Such values are called the *arguement* of the function. If there are multiple arguments they are separated by commas; if there are no arguments the parenthesis still need to be put in place.

To distinguish text from program code, all literal text inside Python code is placed inside quote marks. These can be either single or double quotes (although, as a set, they need to match; you cannot use a single quote on the left side and a double quote on the right side). One does not work better than the other; in general, single quotes are used as they provide less visual clutter, but if your text contains a single quote (causing confusion to Python), feel no restraint to use double quotes.

3.4 Exercises

- Modify `xmas_tree.py` so that the tree only has three "branches" instead of four.
- Modify the script again, this time make a tree with five branches
- Return the script to the original form, showing four branches

3.5 Improving the program

It's great to have some output. At the same point, the Christmas Tree program has some undesirable aspects. There is no "intelligence" in the code. We tell it exactly what to do, and how to do it. Changing the size of the tree requires a lot of work, and so does changing the symbol used, if desired.

Clearly, some improvements are needed! A daunting task when all you know is the `print` statement; it's time to learn some more code. But first, let's outline a step-by-step plan for a *real* Christmas Tree program:

- Using *variables* will allow for different symbols in the tree
- Utilizing Python will make it easier to format the tree without counting spaces
- Using variables will also allow for "building branches" in an automated fashion

3.6 Variables

A variable is a placeholder for a value. As the word suggest, the value can change[1], which means we can do things with it. In our program, the asterisk ('*') is used as the symbol for the tree, but that can be stored inside a `symbol` variable.

We can then build a set of `branch` variables (numbered 1, 2, etc.) based on the symbol of choice and print it out. You simply create a variable in Python by assigning a value to it. Python has about[2] four "primitive" (basic) variable types:

```
my_str = 'text values'
my_int = 15
my_float = 17.3
my_bool = True
```

3.6.1 Variable names

There are a few rules around variables that need to be taken into account; both strict rules and conventions:

- Variable names cannot contain spaces. You can use underscores instead
- You cannot use *reserved words*. The type names (str, int, float, bool) are for instance reserved; that's why the prefix "my" is used in the above list of basic types.

[1] technically, some variables are *immutable* but that is not for practical consideration right now.
[2] depending on your definition

- Names are case sensitive; the variable `name` is *not* the same as the variable `NAME` or even as the variable `Name` — these are three different variables!
- The convention in Python is to only use lowercase for variable names, and use underscores to separate words (unlike for instance Java, where CamelCasing-Words is used). There are some exceptions though
- Another convention is to distinguish variables from *constants* by using uppercase. Python does not have specific "grammar" for constants, by using a name in uppercase you signal to whoever is reading your code[3] "hey! don't change this value!"

In practice all of this is a lot easier than this list suggests; don't bother learning these "rules" as you'll figure them out very quickly as you go along.

3.6.2 String variables

Text variables are named "strings." Why not simply "text?" Because historically, that designation is used for a wider family of variable types. Ancient languages would use a different variable type `char` (for "character") to denote individual characters, and longer text would be represented by "a string of characters," for which the language would maybe have a built-in type (for instance C doesn't have a string type, but C++ does).

The convention has stuck, and a text variable type that can contain multiple characters is referred to as "string" in practical every modern language, including Python. For brevity, the word uses inside Python is `str` (pronounced "sturr") but in general it's just called "string."

When you're typing code you will need a way to differentiate your strings from code; they're both text, after all. For that reasons strings are always represented by putting them inside quotes. Python allows both single and double quotes (but not both); in general single quotes are used as they produce less visual clutter, and double quotes are used in case you're enclosing single quotes:

```
good_string = 'Hello, world'
another_good_one = ”There's no place like home”
bad_string = 'the quotes must match”
```

[3] including yourself!

String Operators

Variables are not variables if you cannot change them; and that is where *operators* come in place. Technically an operator is a symbol that represents an operation, but that definition is as meaningful as "GNU stand's for GNU is Not Unix." A simple example will clarify things, and you can type these lines one by one in a Python Console:

```
>>> name = 'Arthur'
>>> name = name + 'King of England'
>>> print(name)
ArthurKing of England
```

The equal sign is the *assignment operator* and tells Python "put the value on the right in the variable on the left." More specifically, the word "expression" is used instead of "value," as we don't have to assign a single value (that would be very limiting) but we can assign the outcome of a "formula" as well—and the technical name for such a formula is "expression."

That, of course, is exactly what happens in the second line, where the expression (resulting in a value…) `name + 'King of England'` is assigned to `name`. The **plus operator**, when used with strings, will concatenate values. The end result is a bit disappointing, because we forgot to include a space! An improved version shows that you can also build more complex expressions by "stringing together" multiple operators:

```
>>> name = 'Arthur'
>>> name = name + ', ' + 'King of England'
>>> print(name)
Arthur, King of England
```

This is a good example of *operator overloading*. Obviously, the plus sign is intended for adding numbers together, but when used on strings, Python interprets it as concatenation. Operator overloading is obviously a feature supported by Python, although it's encouraged to *only* implement it in those cases where it's really obvious; as is the case with string concatenation.

Another overloaded operator for strings is the **multiplier operator** or asterisk. For strings the operation is basically "repeat *n* times" as the following example (Console!) shows:

```
>>> symbol = '*'
```

```
>>> print(symbol * 3)
***
>>> print(symbol * 5)
*****
```

String Methods

Aside from operators, strings also have *methods*. A method is a built-in function that operates on the object it is attached to. An example, in this case, means more than a thousand words:

```
>>> name = 'Arthur was the king of England.'
>>> name.upper()
'ARTHUR WAS THE KING OF ENGLAND.'
>>> name.lower()
'arthur was the king of england.'
>>> name.title()
'Arthur Was The King Of England.'
>>> name.capitalize()
'Arthur was the king of england.'
```

The method is attached to the variable using a dot (period). This example also shows that Python, in interactive mode, will automatically output an expression if it returns a value. It's different from `print` in a subtle way, as it will *show* the value, where `print` will properly format the value for output (remove quotes, apply whitespace properly, etc).

A method is a function, and functions only work properly when you add parenthesis. Hence, it's `name.upper()` and not `name.upper` (without "parens"). Sometimes functions or methods take arguments, like `print` does. Sometimes they don't, like the methods listed above; you will still need to add parenthesis!

There are many more string methods but the above ones are very useful. There's one more method that is very useful for our Christmas Tree; `center`. This method will take one argument: the number of spaces that you want a string value centered in.

```
>>> symbol = '*'
'   *   '
```

Now you don't have to count out spaces to line up the branches of the tree! You don't even have to do the math to figure out where to place them; `center` will do it for

you! In the next section we'll see how to implement this into our program, but we'll take a look at the other variable types first.

Finally, here's a couple of string methods:

method	**description**	**arguments**
upper	convert to uppercase	
lower	convert to lowercase	
capitalize	capitalize the entire string (first letter upper case, the rest lower case)	
title	capitalize each word inside the string	
center	center across *n* spaces	n (total width)
ljust	left align across *n* spaces, pad with spaces on the right	n (total width)
rjust	right align across *n* spaces, pad with spaces on the left	n (total width)

3.6.3 Integer variables

Integers are whole numbers. Some languages distinguish between small integers and large integers; Python doesn't[4]. You can create integers of any size. The Python designation for integers is `int`, just like `str` for string.

Integer Operators

The basic four arithmetic operators don't need an introduction, but there are a couple more operators that can be used with integers. Here's a demo, followed by an explanation:

```
>>> a = 29
>>> b = 11
>>> a + b
40
>>> a - b
18
```

[4] internally it does, but when writing code that's not something to be concerned about

```
>>> a * b
319
>>> a / b
2.6363636363636362
>>> a ** b
12200509765705829
>>> a // b
2
>>> a % b
7
```

The `**` operator is for exponents; `a ** b` really means a^b. In this case, $29^{11} = 12200509765705829$—as said before, integers have no limits in Python!

The `//` operator is for *integer division*, which is a regular division but with the fraction truncated.

That fraction can be extracted with the `%` *modulo operator* which returns the remainder from a calculation. In this case, the integer division is 2, and the remainder is 7; so we can write $27 = 2 \times 11 + 7$

3.6.4 Float variables

Floats (in Python called `float`) are "floating point numbers"—numeric values with a fraction in it. Computers store these in a way that is similar to "scientific notation" taught at high schools; as a number between 0 and 1.0, multiplied with a power of ten. Here's how some popular numbers are stored:
$\pi = 3.141592653589793 = 0.3141592653589793 \times 10^1$
$g = 9.81 = 0.981 \times 10^1$
$Pop_{earth} = 7,530,000,000 = 0.753 \times 10^{10}$
This allows a computer to store these numbers in a reasonably compact form. The number between 0 and 1 is called the *mantisse*, and for standard floats has a precision of approximately 14 decimals[5]. The power of ten that is used is called the *exponent* and it can range between -307 and +308.

For very large numbers, *exponential notation* can be used. For instance, instead of writing the population of Earth as 7,530,000,000 (7.53 billion), it can be written in Python as `7.53E9` meaning 7.53×10^9. There's an inherent risk associated with

[5]Computers work in Binary numbers, and there's no one-to-one translation between binary fractions and decimal fractions. Hence, "approximately"

floats when combining large numbers with very small ones; when two numbers are combined, the smallest exponent is scaled to match the largest one as this example shows (you can only add mantisses when they have matchin exponents:
$0.314 \times 10^1 + 0.753 \times 10^{10} = 0.000000000314 \times 10^{10} + 0.753 \times 10^{10}$
$= 0.753000000314 \times 10^{10}$
When the difference between the exponents gets too large, the mantisse of the smallest number will simply be zero, as there are not enough digits to represent the value. This is easy to show on the Python console:

```
>>> pop = 7.53E9
>>> cells = 30E12
>>> large = pop * cells
>>> large
2.259e+23
>>> large + 100000 - large
0.0
```

In this example, we take the population of Earth (7.53 billion) and multiply it with the number of cells in the human body (30 trillion or 30×10^{12}). The product is a number that represent the total number of human cells on the planet; it's assigned to a variable named `large` and it's truly large (2.258×10^{23}—a two followed by 23 zeroes).

And while 100,000 is usually a large number, it's dwarfed by `large`. As a result, adding `100,000` to `large` makes it drop off the radar, and when we subtract `large` from the result, we end up with 0.

This is not a Python bug; it's inherit to the way computers work with (large) fractional numbers. There are ways around it; banks for instance work their large numbers with "scaled integers." In a simplistic way, you'd represent $15.78 as 1,578 although in reality the accuracy is quite a bit more than just dollar cents.

3.6.5 Boolean variables

The final primitive variable we take a look at are *Boolean variables*—by now it will not coma as a surprise that these are named `bool` in Python! A Boolean[6] value is either `True` or `False`. which seems quite limited. The power is in those limits

[6]named after English mathematician George Boole (1815–1864), who developed the groundbreaking mathematics on which modern computers are based

though; "if it's not true, it must be false, and vice versa!" This allows computers to evaluate and combine "conditions" in nearly unlimited ways.

As we'll see, whenever Python needs to make a decision, it will be based on a "Boolean expression," something that either evaluates to `True` or `False`. They usually involve **Boolean operators**:

```
>>> a = 7
>>> b = 5
>>> a == b
False
>>> a != b
True
>>> a > b
True
>>> a < b
False
```

The following Boolean operators are used in Python:

operator	meaning
==	equal
!=	no equal
>	greater than
<	less than
>=	greater than, or equal
<=	less than, or equal

3.7 Take two on the Christmas Tree

It's time to revisit our Christmas Tree program. Open `tree1.py` and take a look at its code:

```
1 print('   *   ')
2 print('  ***  ')
3 print(' ***** ')
4 print('*******')
5 print('   *   ')
6 print('Merry Christmas!')
```

Using the `center` method of strings, we can build the branches and the stump without the need to count out spaces! Save your file as `tree2.py`[7] and rewrite the code to take advantage of string operators and methods. Save the file and test it:

```
symbol = '*'
print((symbol * 1).center(7))
print((symbol * 3).center(7))
print((symbol * 5).center(7))
print((symbol * 7).center(7))
print(symbol.center(7))
print('Merry Christmas')
```

A variable `symbol` was used to store the symbol used for the tree. If we decide to use something else, like a # or a then that is easily done now.

To get the right size of each branch, `symbol` is repeated *n* times, depending on the size, using the `*` operator. For consistency's sake, the first branch (size 1) is created in a similar fashion, just to show that each branch is created the same way, its only difference being the actual size of the branch.

The `center` method is used on the outcome of `symbol * n`. Because writing `symbol * 5.center(7)` will confuse Python—it will think you want to use the `center` method on an integer, and those don't have such a method—parenthesis are used to force a proper order of operations (first repeat `symbol`, *then* apply `center`)

This version of the program represents a small step forward. No longer is there a need to count out spaces. We can simply copy & paste lines of code if a bigger tree is required, or remove lines if a smaller tree is sought after. Of course there are some nagging issues…

- Every time the size of tree is changed, the width changes, and `.center(7)` has to be changed into `.center(n)`, depending on what *n* is
- Changing the size of the tree requires adding/removing lines of code

The first issue almost contains the answer in itself. Obviously, we can replace the width argument to the `.center` method with a variable! It is a good habit to do that anyway whenever a number is reused multiple times throughout the code—so-called "magic numbers." Of course you can use find & replace to replace all `7`s with `9`s, but what if your code contains *two* sets of `7`, each with a different meaning (one indicates the maximum width of the tree, the other refers to a font size somewhere?).

As a result, the first improvement is to use a variable `max_size` to store the maximum size of the tree. Update and save `tree2.py`:

[7] it's a good habit to *start* with renaming a file before you change it—if you change it and then carelessly press **Ctrl S** to save it, you overwrite your original!

```
symbol = '*'
max_size = 7
print((symbol * 1).center(max_size))
print((symbol * 3).center(max_size))
print((symbol * 5).center(max_size))
print((symbol * 7).center(max_size))
print(symbol.center(max_size))
print('Merry Christmas')
```

The second improvement involves making our code much, much smarter, and turning into a real program.

3.8 Looping code

What makes a program a program? Why is writing HTML generally *not* considered programming? When a browser processes a web page, it will process each piece of HTML code just once. Up to this point, that was the same for our Christmas tree program, but that is going to change now. The ability to arbritarily execute code—or skip it altogether—is what differentiates programming languages from "markup languages" like HTML.

The first type of statement we'll use in the Christmas Tree code is the `while` statement. It instructs Python to repeat executing code *while* a certain condition is met. When looking at the `tree2.py` code, lines 3–6 are identical, except for the actual size of the branch that is used. We can write a better program by:

- create a variable `size` and give it an initial value of 1
- *while* `size` is less than, or equal to `max_size`, do the following:
- print out the branch, and increase `size` by 2

Save your program as `tree3.py` and rewrite it as follows:

```
symbol = '*'
max_size = 7
size = 1
while size <= max_size:
    print((symbol * size).center(max_size))
    size = size + 2
print(symbol.center(max_size))
print('Merry Christmas')
```

We're using a variable `size` to keep track of the branch size we're printing. It's initialized with a value of 1 in line 3; as that is the starting size of the tree at the top.

On line 4, the `while` statement does one thing: it checks the condition (a Boolean expression) that is used to repeat the code included in the *while* loop. That code is easy to recognize; it's indented. Whenever you have a case of code "belonging" to a line of code above it, it has to be indented. All of those lines need to have the same indentation, although you're free to choose what you see fit. In general, an indentation level of 4 spaces is recommended (but you can choose 1, 2 or even 25—you're in control). Finally, lines of code that "enforce" indentation (as `while` does) are always closed with a colon.

Inside the `while` loop two things are done: the current branch is created (symbol × size) and printed, and `size` is increased with 2. This is repeated as long as the new value of `size` does not exceed `max_size`

Once Python exits the `while` loop it continues with "the rest of the code," printing the tree stump and the holiday message. Merry Christmas!

Exercises

- Change the script so it makes a tree that is 5 units wide
- Change the script so it makes a tree that is 11 units wide
- How much easier is changing the tree size compared to the `tree1.py` script?

3.9 Branching with if

There is always room for improvement. At one point you decide that it's time to make the Christmas Tree program even better—why not center the tree on the holiday message underneath it. A quick count reveals that the length of `Merry Christmas` is 15 characters, and a new version, `tree4.py` is created:

```
1 symbol = '*'
2 max_size = 7
3 width = 15
4 size = 1
5 while size <= max_size:
6     print((symbol * size).center(width))
7     size = size + 2
8 print(symbol.center(width))
9 print('Merry Christmas')
```

If it was *that* easy there wouldn't be a need for a fourth version of our script! Where's the catch? The problem arises when a tree size is picked that exceeds the width of the message underneath it. Try `max_size = 25` and see what happens:

```
        *
       ***
      *****
     *******
    *********
   ***********
  *************
***************
*****************
*******************
*********************
***********************
*************************
        *
Merry Christmas
```

Well, *that* doesn't work out as planned! At one point the branches get larger than the size that's been given to `center`, to which Python reacts with "well, I'm supposed to pad it left and right with spaces to fill up the entire width—I can't do that, so I won't add spaces." The result is that the bottom branches get left aligned, and an ugly tree is the result.

An easy solution would be to set the width to a very high number, and sometimes that is the kind of solution that you can use. The downside of that solution, in this case, is that when you're sending the output to a console (and we are), all that extra space might wrap around to the next line, and then the results look even funkier.

A mucb better solution, and not completely by accident this coincides with the theme of this section, is to choose what we pick as the basis for the width argument. Do we pick the size of the message? Or do we go with the maximum size of the tree?

Just like the `while` statement, an `if` statement checks a condition. The difference is, that unlike `while`, Python runs the code inside it *only once*. Here is how it works:

```
1 symbol = '*'
2 message = 'Merry Christmas'
3 max_size = 25
4 width = 15
```

```
5  if max_size > width:
6      width = max_size
7  size = 1
8  while size <= max_size:
9      print((symbol * size).center(width))
10     size = size + 2
11 print(symbol.center(width))
12 print(message.center(width))
```

Not only did we add an `if` statement on line 5, we also apply the `center` method to the holiday message. And while we're at it, the message is now stored as a variable, making it that much easier to change it to "Feliz Navidad!" or something similar.

In this particular case, the program sets the width to a certain value, and then overrides it if the maximum size is bigger. The `if` statement allows you to make that choice inside it, using the `else` clause. And to make the program truly indepedent of the chosen message, the length of the message is not hard-coded inside the program, but calculated using the built-in `len` function:

```
1  symbol = '*'
2  message = 'Merry Christmas'
3  max_size = 25
4  if max_size > len(message):
5      width = max_size
6  else:
7      width = len(message)
8  size = 1
9  while size <= max_size:
10     print((symbol * size).center(width))
11     size = size + 2
12 print(symbol.center(width))
13 print(message.center(width))
```

Exercises

- Change the maximum size back to 7 and confirm that the output looks as desired
- Change the message to `Feliz Navidad!` and check the output
- Change the message to `Merry Christmas, and a Happy Newyear!` and check the output

Chapter 4

Structuring your code

With just two statements—`if` and `while`—we've created a flexible script that can quickly generate a wide variety of Christmas trees and messages. It's not the most complicated script in the world, but it's easy to see how, when you're building a script, it will grow and grow…Until it's not that small anymore.

It's nearly unavoidable to have a script that contains dozens, or even hundreds of lines of code, when you're trying to achieve something complex. The approach to keep such a mountain of code managable is to chop it up into smaller bits—functions.

4.1 Case: temperature conversion

For a start, create a simple script called `temperature1.py` that will print a table of temperatures from ° F to ° C. The conversion for Fahrenheit to Celsius is:
$T_{Celsius} = \frac{T_F - 32}{1.8}$

```
1 print('Create a temperature conversion table')
2 print('From Fahrenheit to Celsius')
3 f = -20
4 while f <= 220:
5     c = (f - 32) / 1.8
6     print(f, ':', c)
7     f = f + 20
```

The script is very straightforward; set an initial Fahrenheit temperature `f` to -20, calculate the Celsius temperature, print the two side, by side, and increase the Fahrenheit temperature with 20° while it reaches 220° F.

The output leaves a lot to be desired, and before jumping into functions we'll fix that first—it provides a good illustration on how functions can clarify complex code.

```
Create a temperature conversion table
From Fahrenheit to Celsius
-20 : -28.88888888888889
0 : -17.77777777777778
20 : -6.666666666666666
40 : 4.444444444444445
60 : 15.555555555555555
80 : 26.666666666666664
100 : 37.77777777777778
120 : 48.888888888888886
140 : 60.0
160 : 71.11111111111111
180 : 82.22222222222221
200 : 93.33333333333333
220 : 104.44444444444444
```

4.1.1 Rounding numbers

Python has a built-in function aptly named `round(<value>, <decimals>)` that returns the rounded version of a numeric value. Here's a little demonstration using a Python console[1]:

```
>>> pi = 3.141592653589793
>>> round(pi, 2)
3.14
>>> round(pi)
3
>>> two = 2
>>> round(2, 2)
2
>>> mega = 123456
>>> round(mega, -3)
123000
```

[1] feel encouraged to follow along whenever "console" is mentioned

The use of `round` is pretty straightforward. Provide value, decimals, and you're in business. A few things to note:

- If `decimals` is not provided, `round` will use a **default value**. A lot of Python functions show this kind of behavior. For `round`, the default value for `decimals` is, not surprising, `0`.
- The `decimals` value can be negative; this is useful if you need to round to the nearest ten, hundred, thousand-fold, etc.
- The function will round to the *nearest* decimal, which means it will round upward from .5; 3.5 will be rounded to 4.0, while 3.49 will be rounded to 3.0
- The input type is the same at the output type. Even though `round(n, 0)` will return an integer value, its type will still be `float` if *n* was a float to begin with.

We can now update our program `temperature1.py` to show all values with zero decimals:

```
1 print('Create a temperature conversion table')
2 print('From Fahrenheit to Celsius')
3 f = -20
4 while f <= 220:
5     c = (f - 32) / 1.8
6     print(f, ':', round(c))
7     f = f + 20
```

That already looks a lot better, but not perfect! For that, we need to align all numbers.

4.1.2 Converting Types

Aligning text is easy; the `rjust` method takes care of that. It works very similar to `center` except that it pads all the spaces on the left side, resulting in "right justified" text (which is where the method name comes from).

The problem though, is that our numbers are *not* strings, but numeric values; and numeric types do not have a `rjust` method. What to do?

Each Python type has a matching conversion function[2]; as long as the input type is valid, it will return a value of the specified type:

[2] technically it's a *constructor* but that's for later to worry about

- `str` will return a string type variable. Whatever a numeric value gets printed as, is the string value that `str` will return
- `float` will return a float type variable. If the value is not a pure numeric value, Python will produce an error message. `'125'` is pure numeric, `'12E67'` is not, even though it's valid scientific exponent notation, and `'12 lbs'` is *definitely not* numeric.
- `int` will return an integer type variable. The same restrictions as for `float` apply. In addition, if there's a fraction, `int` will simply drop it. 6.1, 6.9 and 6.99 all get converted to 7; -6.1 and -6.9 get converted to -6 (and *not* rounded down to -7, as some languages do). `int` can convert floats to int, and `str` as well, but the string value has to represent an integer value.
- `bool` will return a boolean type variable. Pretty much *anything* can be turned into a `bool`; zero and empty values get converted to `False` and everything else becomee `True`

Armed with that information we can convert the tempartures to `str` values and then justify them with `rjust`:

```
print('Create a temperature conversion table')
print('From Fahrenheit to Celsius')
f = -20
while f <= 220:
    c = (f - 32) / 1.8
    print(str(f).rjust(3), ':', str(round(c)).rjust(3))
    f = f + 20
```

At this point it might not be perfectly clear anymore what the code is doing. One of the ways is to fix that with comments.

4.1.3 Adding comments to code

The ability to add "strictly for humans" code to your program is something virtually every language supports. They're usually little fragments of text that clarify what's going on, so when you read your code later it will be easier (or rather, *possible*) to understand what it's doing.

In Python anything following a pound sign (#) on a line of code will be considered a comment and ignored by the Python interpreter. You can put comments on the same line although it's generally considered a better practive to put them on separate lines, especially when the comments get a little bit longer.

```
print('Create a temperature conversion table')
print('From Fahrenheit to Celsius')
f = -20 # start temperature
while f <= 220:
    # convert celcius to fahrenheit
    c = (f - 32) / 1.8
    # turn temps into string, and right justify them
    print(str(f).rjust(3), ':', str(round(c)).rjust(3))
    # next temperature to show is 20 degrees higher
    f = f + 20
```

It is definitely a good idea to sprinkle code with comments. Make sure comments are *meaningful*; the comment on line #9 explains *why* variable `f` gets increased with 20; a comment that states `# Add 20 to f` does not and is meaningless.

There is a school of thought that states that comments are "the deodorant of coding; they are used to cover up nasty smelling code." The thought is that properly written code doesn't need comments; it should make its intention clear by itself.

That is not something that is easy to achieve, and it's better to put comments on code than to assume your code is "readable" when it's really not.

4.2 Writing Functions

With a "solid" case under our belt we can study how functions improve your code. In its simplest form, a function just returns a value, as this console example shows:

```
>>> def five():
...     return 5
...
>>> five()
5
```

Press **Enter** after the first line, and instead of the regular >>> prompt, Python will now show a ... prompt to indicate that you're still inputting code. As you need to indent your code, don't forget to press **Tab**!

On the next empty line after entering `Return 5` just press **Enter** to indicate that you're done entering text.

Now that you've defined a function it van be called (don't forget the parenthesis) and it will show its return value. Of course, a function that returns the value of 5 is not very useful. But a function that returns the Celcius temperature is:

```
>>> def celcius(t):
...     return (t - 32) / 1.8
...
>>> celcius(100)
37.77777777777778
```

In this case an argument `t` was added between the parenthesis. When the function gets called, `t` is treated as a variable with the value that was passed along to the function (in this case 100°).

4.2.1 Using functions in scripts

How does this work inside a script? Pretty much the same way, with one catch. Only when Python *executes* the function code it will load the function into memory. Until that happens, the function *does not exist*. Try it out in our script (new version, `temperature2.py` by adding the function at the bottom:

```
1  print('Create a temperature conversion table')
2  print('From Fahrenheit to Celsius')
3  f = -20 # start temperature
4  while f <= 220:
5      # convert celcius to fahrenheit
6      c = celcius(f)
7      # turn temps into string, and right justify them
8      print(str(f).rjust(3), ':', str(round(c)).rjust(3))
9      f = f + 20
10
11 def celcius(t):
12     return (t - 32) / 1.8
```

The result when you run the code, is a giant **kaboom**!

```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit
(AMD64)] on win32
Type ”help”, ”copyright”, ”credits” or ”license()” for more information.
>>>
 RESTART: C:\Course\temperature2.py
Create a temperature conversion table
From Fahrenheit to Celsius
Traceback (most recent call last):
  File ”C:\Course\temperature2.py”, line 6, in <module> c = celcius(f)
NameError: name 'celcius' is not defined
```

The problem is, as stated before, is how Python executes code. In order to execute the `celcius` function, the function needs to be loaded into memory, and the code that does *that*, is the `def celcius` statement and all code that belongs to it.

That code is listed *underneath* where `celcius` is called, and hasn't been executed yet. As a result, the program crashes because it's trying to call a function it doesn't know about (yet).

The proble is easy to solve: change the order in which your code appears! After changing `temperature2.py` to the following it will run just fine:

```
1  def celcius(t):
2      return (t - 32) / 1.8
3
4  print('Create a temperature conversion table')
5  print('From Fahrenheit to Celsius')
6  f = -20 # start temperature
7  while f <= 220:
8      c = celcius(f)
9      # turn temps into string, and right justify them
10     print(str(f).rjust(3), ':', str(round(c)).rjust(3))
11     f = f + 20
```

The empty line between the function definition and the rest of the code isn't required by Python. It's still needed though, as it makes the code a lot easier to read.

A more subtle change in the program happened on line #8: the comment about the Celcius conversion was dropped. By calling the function, it's perfectly clear what is happening. This is an example of *self-documenting code*. By structuring the code in a particular way, and by choosing meaningful variable names, less documentation is required.

The big problem that remains in the code is the jumble on line 10. There's a double set of nested function calls required to convert the temperatures to strings and to right-align them. The solution is simple: put it in a function!

```
1  def celcius(t):
2      return (t - 32) / 1.8
3
4  def formatnum(n):
5      # format numbers, turn them into strings
6      # and right-align them
7      r = round(n)
8      s = str(r)
9      return s.rjust(3)
10
11 print('Create a temperature conversion table')
12 print('From Fahrenheit to Celsius')
13 f = -20 # start temperature
14 while f <= 220:
15     c = celcius(f)
16     print(formatnum(f), ':', formatnum(c))
```

```
17      f = f + 20
```

4.2.2 Using a main function

The attempts to clean up the code have been very successful. The functions `celcius` and `formatnum` to their job and keep the main portion of the code clean.

In accordance to Murphy's law it does introduce another problem now; the actual program starts in the middle of the code. For a 16-line script like this one it's not a big problem, but for larger projects is obviously is.

The solution is to adapt a program structure that is adopted by nearly all Python programmers:

- Put the main part of the code in a separate function
- Put this function at the very beginning of the code. Remember: executing a function definition will merely load it into memory, it will *not execute the code* until the function is called.
- At the very bottom of the code, once all other function definitions have been executed (and the functions have been loaded into memory), write a line of code that calls the main function.
- You're absolutely free to call this main function `SpongeBob` or anything else you desire. However, the standard naming convention is to call it the more obvious `main`

Save your script as `temperature3.py` and update it with a `main` function. Note how the main code is now indented inside the `main` definition, as it's part of it:

```
1  def main():
2      print('Create a temperature conversion table')
3      print('From Fahrenheit to Celsius')
4      f = -20 # start temperature
5      while f <= 220:
6          c = celcius(f)
7          # turn temps into string, and right justify them
8          print(formatnum(f), ':', formatnum(c))
9          f = f + 20
10
11 def celcius(t):
12     return (t - 32) / 1.8
13
14 def formatnum(n):
15     # format numbers, turn them into strings
```

```
    # and right-align them
    r = round(n)
    s = str(r)
    return s.rjust(3)

main()
```

Docstrings

A special class of comments are called *docstrings* and they are used to add a little bit of information to a function. It works really simple: the first line of "code" inside a function definition can function as the docstring. Here's how to `celcius` function inside `temperature3.py` looks with a docstring:

```
def celcius(t):
    '''Return temperature converted from Fahrenheit to Celcius'''
    return (t - 32) / 1.8
```

Of course you could do the same with a regular comment. We'll see later that using a docstring has additional benefits, making it much more suited for the job than a plain comment.

In some cases your docstring will contain multiple lines, especially when you're documenting the function arguments as well. Regular quotes don't allow line breaks; for that, Python has *triple quotes* (either single or double).

The convention is to *always* use triple quotes for the docstring, and put them on their own line when writing a multi-line docstring, like this:

```
def celcius(t):
    '''
    Convert Fahrenheit to Celcius

    Output:
    temperature in degrees Celcius

    Arguments:
    t - temperature in FahrenHeit
    '''
    return (t - 32) / 1.8
```

To be fair, a simple function like `celcius` probably doesn't need such an elaborate docstring; a single line saying "fahrenheit to celcius conversion" would likely be informative enough.

However, it is a good habit to *always* include a docstring for every function you create, even if it's just a single line (still use triple-quoting for that single line though).

Exercise

Add a docstring to the `formatnum` function that reads:

```
Return a right-aligned string that represents
the rounded value of the input
```

4.2.3 Default Arguments

Some functions only need arguments sometimes. The `formatnum` function is a good example. It could benefit from some customization, where the user of the function can specify both the number of digits to be used for rounding, as well as the number of spaces used for right alignment, in case something else than 3 spaces is needed. Save the script as `temperature4.py` and update the formatnum function:

```
def formatnum(n, ndigits, size):
    '''
    Convert a number to a string and format it

    Arguments:
    n - the number to be converted
    ndigits - number of decimals for rounding
    size - number of spaces to right-align in
    '''
    r = round(n, ndigits)
    s = str(r)
    return s.rjust(size)
```

Running the program will now produce an error; the call to the function (with a single argument) does not match the function definition that uses three arguments. That is what this error message is all about:

```
TypeError: formatnum() missing 2 required positional
arguments: 'ndigits' and 'size'
```

The obvious solution is to fix `main`:

```
def main():
    print('Create a temperature conversion table')
    print('From Fahrenheit to Celsius')
    f = -20 # start temperature
    while f <= 220:
        c = celcius(f)
        # turn temps into string, and right justify them
        print(formatnum(f, 0, 3), ':', formatnum(c, 0, 3))
        f = f + 20
```

But that will make *most* of the calls to the function overly complicated, when in general we only want zero decimals and three spaced-formatting.

We saw earlier that the `round` function uses an `ndigits` argument, but that it can be left out if the rounding digits are zero. We can do something similar for `formatnum` by providing a default value in the function definition:

```
def formatnum(n, ndigits=0, size=3):
    '''
    Convert a number to a string and format it

    Arguments:
    n - the number to be converted
    precision=0 - number of decimals for rounding
    size=3 - number of spaces to right-align in
    '''
    r = round(n, ndigits)
    s = str(r)
    return s.rjust(size)
```

It's a good idea to update the docstring with the default values as well; if someone is reading the docstring (we'll get to that later) they'll know that those arguments have default values.

Now we can revert `main` to its original state (calling `formatnum` with only one argument) and perhaps extend it a little bit; why not display the Celcius temperature with one decimal? Of course, that'll require adjusting the `size` to 5—two extra spaces are needed to accomodate the decimal sign (period) and the extra digit:

```
def main():
    print('Create a temperature conversion table')
    print('From Fahrenheit to Celsius')
    f = -20 # start temperature
    while f <= 220:
        c = celcius(f)
        # turn temps into string, and right justify them
        print(formatnum(f), ':', formatnum(c, 1, 5))
        f = f + 20
```

4.2.4 Named vs Positional Arguments

The nomenclature in Python for arguments with default values is *named arguments*, and arguments without default values are *positional arguments*.

Positional arguments, as the name suggests, can be inserted in a function call by their order of appearance. For the `formatnum` function that is `n`, `ndigits` and `size`.

In a lot of cases positional arguments make sense. For the `round` function it's rather intuitive what `round(16.43, 1)` is going to do.

The argument position becomes a bit shaky when default values are involved. Suppose you're calling `formatnum(3.667, 5)`—with the intention of the second argument being the size; "I want zero decimal rounding, so I don't have to specify it." Why, yes, but how would Python know that the 5 means `size` and not `ndigits`?

To solve that problem, you can specify an argument by name and for arguments with default values that is usually the preferred way; hence the moniker "named arguments." In this particular example, the function call would be `formatnum(3.667, size=5`. Normally you put spaces around equal signs (by convention), but when assigning values to named arguments it's convention to not use spaces.

One of the Python mantra's is *Explicit is better than implicit*, so it's recommended to call named arguments by name (especially when their meaning is not intuitive—compare the `round` function with `formatnum`).

You can *enforce* the use of argument names in a function call by inserting an asterisk (`*`) in the function definition, as is done with the `formatnum` function here:

```
def formatnum(n, *, ndigits=0, size=3):
    '''
    Convert a number to a string and format it

    Arguments:
    n - the number to be converted
    precision=0 - number of decimals for rounding
    size=3 - number of spaces to right-align in
    '''
    r = round(n, ndigits)
    s = str(r)
    return s.rjust(size)
```

Once again, our program produces an error message:

```
TypeError: formatnum() takes 1 positional argument
but 3 were given
```

To fix this, the line in `main` that alls the function needs to be updated, so that it's explicitly naming the named arguments:

```
    print(formatnum(f), ':', formatnum(c, ndigits=1, size=5))
```

4.3 Refactoring

Refactoring is the process of (significantly) altering the code, to improve its structure, without breaking it. Entire library shelves are filled with books about refactoring and by no means is this meant as even an *attempt* to teach refactoring.

In this section you will see an example of refactoring code, allowing eventually for new functionality, in a structured, step-by-step process.

The Request

The temperature conversion table is a big hit; you get requests on a daily basis to email people such a table. However, the table doesn't always provide the answer people are looking for, as this email shows:
Hello!
Your temperature conversion table is great. However, I export meat and need to specify the temperature for the refrigerated containers that I am using. Not only are the increments of 20° far too coarse, for my purpose it would me more useful if the table ran from -60° to +40°, and then in 5° steps if possible.
You've gone through the nightmare of changing program code with the Christmas Tree program, and obviously that is *not* how to fulfill this request. In a perfect world your program would contain a call to `create_temp_table` and provide arguments with the start temperature, end temperature and interval. But how to get there?

The Process

In refactoring you take little steps that allow you to ensure that your code continues to work correctly. For this process we could:

- Move the code from `main` to a new function `create_temp_table`
- Initialize variables at the beginning of `create_temp_table` that define start, finish and interval
- Convert those variables into named arguments
- Adjust the call to `create_temp_table` with different arguments

Step 1

- Save your file as `temperature5.py`
- Rename the function `main` into `create_temp_table`
- Insert a new function `main` above it, and make it call the function `create_temp_table`
- Test your code

```
def main():
    create_temp_table()

def create_temp_table():
    print('Create a temperature conversion table')
    print('From Fahrenheit to Celsius')
    f = -20 # start temperature
    while f <= 220:
        c = celcius(f)
        # turn temps into string, and right justify them
        print(formatnum(f), ':', formatnum(c, ndigits=1, size=5))
        f = f + 20
```

Step 2

- At the top of the function, insert lines to set the process variables:
  ```
  t_start = 20
  t_finish = 220
  t_interval = 20
  ```
- Replace the “magic numbers” inside the function with the process variables
- Test your code

```
def create_temp_table():
    print('Create a temperature conversion table')
    print('From Fahrenheit to Celsius')
    t_start = -20
    t_finish = 220
    t_interval = 20
    f = t_start
    while f <= t_finish:
        c = celcius(f)
        # turn temps into string, and right justify them
        print(formatnum(f), ':', formatnum(c, ndigits=1, size=5))
        f = f + t_interval
```

Step 3

- Move the process variables from the code to inside the argument list
- Test your code

```
def create_temp_table(t_start=-20, t_finish=220, t_interval=20):
    print('Create a temperature conversion table')
    print('From Fahrenheit to Celsius')
    f = t_start
    while f <= t_finish:
        c = celcius(f)
        # turn temps into string, and right justify them
        print(formatnum(f), ':', formatnum(c, ndigits=1, size=5))
        f = f + t_interval
```

Step 4

With the conversion complete, we can now call `create_temp_table` with different arguments to get a different table:

```
def main():
    create_temp_table(-60, 40, 5)
```

And that concludes the refactoring! With very tiny steps we ensured that the code kept working the way we intended it to work.

However, the new structure does introduce hazards that were not there in the past. What if a steel mill requests a table that runs from 500° to 2000° in intervals of 100°? Surely the one decimal for the Celsius temperature wouldn't be needed and we'd need more space to show the Fahrenheit temperature. Adding additional `size` and `ndigits` arguments could take care of that.

But even worse, what if the wrong parameters for the `create_temp_table` function are provided? A negative interval results in a `while` condition that will never be met; the program will run indefinitely.

These are not issues we're going to address right now; but they are something to ponder over, and illustrate how improving a program, even when it's not introducing bugs at the surface, can still cause problems down the line.

That's not intended as an example to not improve your programs! Quite the contrary—just be aware that your work is usually not finished when you think it is!

Chapter 5

Loops and Lists

The ability to repeat code is crucial in any programming language. We already encountered the `while` function ("Looping code," page 24) which provides one way of repeating code.

The `while` loop is primarily intended for "undeterministic" loops, which is a fancy way of saying "up front we have no clue how many times this has to run." We'll start this chapter with studying a classic case of such an undeterministic loop.

5.1 Calculating a square root

The square root of a number is number that is chosen in such a way that, when multiplied by itself, it returns the original number. The notation of the square root of *n* is $\sqrt{n}$.

In some cases, the square root is trivial to determine. For instance, $\sqrt{36} = 6$ and $\sqrt{81} = 9$, but others are much harder. What is, for instance, the square root of 40? In fact these numbers are *rational* numbers, meaning that they cannot be represented by a fraction. We can, however, calculate a number that gets reasonably close to such a square root.

Calculating square roots is done often, and the Python language has its own built-in function for it. However, rolling out your own square root estimator is a good exercise. The method used here is not how Python calculates square roots, but it's a good practical example of how to solve a problem that, at first, might seem very hard to solve.

Problem Analysis

Looking at the square root of 40, we can be certain of two things:

- 40 is more than 36 or 6^2, so the square root of 40 has to be more than 6
- 40 is less than 49 or 7^2, so the square root of 40 has to be less than 7

What do you do if someone says "it's in between these two values?" Why, use the average of course! The average of 6 and 7 is 6½. If we square that, we get 42¼ which is clearly too large; obviously the square root has to be between 6 and 6½, so let's try 6¼, and so on…

This is a method that requires some accounting, as you'd have to keep track of the previous two attempts, and decide if your answer is too high or too low. That's way too complicated!

Programming is often like that; you start with a solution, and then realize that this might be too complex. *There has to be another way!*. Let's take another look at the problem: we're looking for a value *s* (square root), so that $s \times s = n$. Aha! Rewriting that equation tells us that $\frac{n}{s} = s$…But what if *s* is *not* the square root of *n*? If our chosen estimate is too large, the quotient will be too small to be the square root, and if our estimate is too small, the quotient will be too large. *So why not average those two?*

Let's try this method; and for the first estimate *s* we simply pick half of the value we want the square root of (40):
$s = \frac{40}{2} = 20$ Now we calculate the quotient *q*; if it's reasonably close to textits we're done. Otherwise we use it as our new estimate, s_2 and repeat the process:

$$\frac{40}{2} = 20 \qquad \frac{40}{20} = 2$$
$$\frac{20+2}{2} = 11 \qquad \frac{40}{11} = 3.636363$$
$$\frac{11+3.6363}{2} = 7.318181 \qquad \frac{40}{7.318181} = 5.465839$$
$$\frac{7.318181+5.465930}{2} = 6.392056 \qquad \frac{40}{6.392050} = 6.257768$$
$$\frac{6.392050+6.257768}{2} = 6.324909 \qquad \frac{40}{6.324909} = 6.324202$$
$$\frac{6.324909+6.324202}{2} = 6.324555 \qquad \frac{40}{6.324555} = 6.324555$$

At first, the estimates swing violently between 20 and 2, but once the value starts converging it stabilizes rapidly at 6.324555—and plugging in 6.32555×6.32555 on a calculator[1] does indeed yield 39.999999—close enough for practical purposes!

Implementation

The same process can be done much quicker with Python. Here's the `square_root.py` script to use the same algorithm:

```
'''square root estimation'''

def main():
    n = 40
    s = square_root(n)
    print('The square root of', n, 'is', s)

def square_root(n):
    '''Estimate the square root of n'''
    s = n / 2
    q = n / s
    while abs(q - s) > 0.00001:
        s = (s + q) / 2
        q = n / s
    return s

main()
```

The program uses a "new" function, `abs`, when calculating the difference between the estimate of the square root `s` and the quotient `q`. That's because the difference can be negative if `q` is less than `s`. The `while` loop will exit if the difference between the two is less than 0.00001, but, for instance, -5 is definitely less than that, but not a condition we want to bail out on!

The `abs` function ("Absolute") returns the positive value of any number (positive or negative) and bypasses that problem; now the difference can be checked correctly.

Deterministic?

The whole point of this demonstration is to show a use case of the `while` statement. How many times the `while` loop runs is not known—at least not for the program—until it ends. For instance, when calculating $\sqrt{4}$, Python doesn't even enter the `while` loop as the first estimate *is* the final answer. For $\sqrt{40}$, it runs 5 loops, and for $\sqrt{4000}$ it runs 9 loops.

[1] remember, you can use the Python console as a calculator!

Those numbers are not pulled out of thin air; the program can be modified a little bit to keep track of how many times the code inside the `while` loop gets executed:

```
def square_root(n):
    '''Estimate the square root of n'''
    s = n / 2
    q = n / s
    c = 0 # counter; how many WHILE loops?
    while abs(q - s) > 0.00001:
        s = (s + q) / 2
        q = n / s
        c = c + 1
    print(c, 'while loops')
    return s
```

Exercises

- Modify the code as shown above to keep track of how many times the `while` loop is executed
- Test the program by calculating the square roots of 5, 10, 25, 100, 684, and 1000
- What calculation requires the most loops?
- Do you think that a bigger input number by definition means more loops?
- What if you try other initial values than `n/2`? Does it really matter?

5.2 Another way of looping

The `while` statement is one way of repeating code; it is easy to understand, which is why it was shown first. But there's another way of repeating code in Python, and it's in fact used much more often. That is the `for` statement. Unlike the `while` statement the duration is (in most cases at least) predetermined[2]. The general pattern of the `for` loop is:

```
for thing in sequence_of_things:
    lines_of_code
```

[2] in reality, "determined vs. undetermined" isn't really that important but it can help you to figure out if you're in doubt about using `while` or `for`

But what is a "sequence of things?" And what is that `thing` for starters? Well, you could quite literally, give Python a list[3] of values, as this console[4] example shows:

```
>>> for name in 'Anton', 'Bertha', 'Carl', 'Diana':
...     print(name)
...
Anton
Bertha
Carl
Diana
>>>
```

As we can see, "thing" is just a place holder for the variable that gets assigned to each of values, one by one, as Python works its way through the "sequence of things." You're free to pick any variable you want, but it's preferred to pick something that doesn't confuse you.

Here's another example of a `for` loop, this time going through a sequence of numbers:

```
>>> total = 0
>>> for num in 100, 20, 675, 25, 25, 30:
...     total = total + num
...     print(total)
...
100
120
795
820
845
875
>>>
```

On purpose we're doing something different then merely printing `num` here, to show that we can actually *do* something with the values we're looping through!

The official term for going through a sequence of things is *iterating*, and the official moniker for "a sequence of things" is *iterable*. The easiest definition of what

[3] technically what the example shows is in Python terminology not a *list*, but let's not nitpick on details here…

[4] Console means "follow along!"

an iterable in Python is? "You can run a `for` loop over it." In general though, an interable is "a bunch of things."

The `for` loop is at the very core of what defines Python as a programming language, and pretty much *anything* is iterable. You can iterate through a string—it's a *string of characters* after all. You can iterate through a text file (on a line-by-line basis). You can iterate through a database table (on a record-by-record basis). You can iterate through a PowerPoint presentation (on a slide-by-slide basis). Keep in mind that in Python "iterate" is shorthand for "go through it using a `for` loop."

But how does Python know what the sequence of smaller things is that an iterable is made up of? How does it know that a PowerPoint file is composed of Slides, but a text file is made up from lines of text?

There's no magic involved; it's all up to the programmers who wrote the software to interact with a database or a PowerPoint file in the first place. They made the decision to add some functionality to their library: "if the user iterates through the Powerpoint object I provide, I will provide them with a sequence of Slide objects, in the same order as they appear in PowerPoint." That may look obvious, *but someone made that decision and wrote code for it.*

There's no magic involved, or surprises what you get when iterating through a data object; it's predetermined by the type (characteristics) of the object you're iterating through.

5.2.1 A practical example

In the `square_root.py` example of the previous section there was an exercise to check the outcome of the square roots of 5, 10, 25, 100, 684, and 1000. There was little choice but to update the code and run the program every time, but not anymore! Load `square_root.py` and modify the `main` function as follows:

```
def main():
    for n in 5, 10, 25, 100, 684, 1000:
        s = square_root(n)
        print('The square root of', n, 'is', s)
        # compare square of s with actual number
        d = abs(n - s * s)
        print('The difference with', n, 'is', d)
        # empty line
        print()
```

Keep in mind that the differences, because they are so small, are printed in scientific notation. 9.53027e-6 (9.53027×10^{-6}) really stands for 0.00000953 ("six zeroes in front of the nine"). At one point we'll get around formatting that properly, but not

right now!

5.2.2 Creating sequences of numbers

In the previous section a sequence of numbers was provided by listing them as a range of numbers in code, separated by commas. There are many cases where that is convenient, but the downside is that such a list is hardcoded inside the program.

If a regular sequence of numbers is needed, Python can help out with the `range` function. This function is used incredibly often, so it's a good thing to get familiar with it. Follow along in the Console for the next few examples!

Using `range` with one argument

In its simplest form, `range` will produce a range of *n* numbers, starting at 0. For instance:

```
>>> for n in range(5):
...     print(n)
...
0
1
2
3
4
>>>
```

Because the sequence starts at 0 (there are good reasons for that), calling `range(5)` will end at 4. This can be confusing at first! Writing a lot of Python code will cure you of that confusion; believe it or not, it's actually beneficial that it's set up like this.

Fixing the output

Another annoyance when we're printing out these sequences is that `print` always advances to a new line. It takes up a lot of space without any benefit. Luckily, the print function takes a named argument `end` that specifies what kind of character the function should add after the last print value.

By default this is the newline character (written as
`n` but we can override that, and make it a space:

```
>>> for n in range(5):
...     print(n, end=' ')
...
0 1 2 3 4 >>>
```

That also puts the prompt on the same line as the output, as Python doesn't advance to a new line even after the last element of our sequence! That's a small price to pay though!

Using `range` with two arguments

When adding a *second* argument to `range` the behavior of the function changes slightly. The first argument now indicates the *start* value, and the second argument indicates the *final* value–but it will *not* include that value!

That seems insane. and in some ways it is. But in practice it works really convenient. Suppose you want a list of *n* items, starting at 1:

```
>>> n = 7
>>> for i in range(1, 1 + n):
...     print(i, end=' ')
...
1 2 3 4 5 6 7 >>>
```

No complex math is involved to get the correct sequence; if the final number was included, you'd have to remember to "subtract one to account for the start value." There are other benefits; we'll get to those when looking at `range` with three arguments.

Using `range` with three arguments

The third argument in `range` is the *step* or interval value; it tells the function what the increment (or decrement) in the sequence is. To display for instance the numbers from 10–100, we can use:

```
>>> for i in range(10, 101, 10):
...     print(i, end=' ')
...
10 20 30 40 50 60 70 80 90 100 >>>
```

Once again, it looks stupid that the final value is 101; if 100 was picked it wouldn't be included in the sequence. It makes more sense when we think of the `range` arguments as something to be calculated. Let's generate the same sequence in a slightly different way:

```
>>> start = 10
>>> interval = 10
>>> count = 10
>>> for i in range(start, start + interval * count, interval):
...     print(i, end=' ')
...
10 20 30 40 50 60 70 80 90 100 >>>
```

The choice to *not* include the final value makes a lot more sense with `range` parameters that are somehow computed. Those are (compared to using fixed numbers) the harder cases, and for those cases that particular choice of the Python designers makes it a lot easier for us!

Creating a numeric sequence for the temperature table

Let's revisit `temperature5.py` and its `create_temp_table` function:

```
1 def create_temp_table(t_start=-20, t_finish=220, t_interval=20):
2     print('Create a temperature conversion table')
3     print('From Fahrenheit to Celsius')
4     f = t_start
5     while f <= t_finish:
6         c = celcius(f)
7         # turn temps into string, and right justify them
8         print(formatnum(f), ':', formatnum(c, ndigits=1, size=5))
9         f = f + t_interval
```

Before we do anything…Safety first, and save this version as `temperature6.py`

Instead of using a `while` loop to create all entry temperatures in the table, we can use built-in Python functionality to do so. In general it's a good idea to avoid writing code for functionality the platform, in this case Python, already provides.

Of course there's that pesky "the `range` function will not include the final value" detail, but that can be solved by adding 1 to the final value…Or can we?

What if, for some reason, we want a *decreasing* scale? In the current implementation with `while` that would never work, as the program would jump out of the `while` loop at the very first check (the "current" temperature would be higher than the "final" temperature, after all)

The `range` function can work fine with a decreasing order (as long as a negative interval is provided). Of course we can use an `if` statement:

```
if interval > 0:
    t_final = t_finish + 1
else:
    t_final = t_finish - 1
for f in range(t_start, t_final, interval):
    ...
```

But that is ugly, and besides, now we're writing *more* code, when the whole point of using `range` was to write *less* code and let Python do the work. There has to be a better way!

And there is! We can simply add the interval value to the final temperature. Assuming the interval matches the "direction" of the temperature scale[5] it will automatically "extend" the final temperature in the desired direction:

```
def create_temp_table(t_start=-20, t_finish=220, t_interval=20):
    print('Create a temperature conversion table')
    print('From Fahrenheit to Celsius')
    for f in range(t_start, t_finish + t_interval, t_interval):
        c = celcius(f)
        # turn temps into string, and right justify them
        print(formatnum(f), ':', formatnum(c, ndigits=1, size=5))
```

Not only is this implementation (marginally) shorter, it's also *better*. The function can now be used to provide a decreasing temperature scale, for instance:

```
def main():
    create_temp_table(60, -40, -5)
```

5.3 Lists

We've now seen two ways to provide "a sequence of things" to a `for` statement; simply typing a bunch of values in code, separated by commas, or using the `range` function. But there's a third way, using a special object: the list.

We've encountered *primitives*: the basic variable types (`str`, `int`, `float` and `bool`). They contain a single value of some specific type, and that's it.

A list is a *container* type. It doesn't represent a value by itself, but it contains other objects—either primitives or other containers. But before we go down that path, let's take a look at a simple list in the Console:

[5] a dangerous assumption! But we'll deal with *not* making that assumption in a later stage

```
>>> astronauts = ['John Glenn', 'Neil Armstrong', 'Sally Ride']
>>> for astro in astronauts:
...     print(astro)
...
John Glenn
Neil Armstrong
Sally Ride
>>>
```

The `for` statement works exactly the same; an iterable (sequence) is an iterable, after all. One can argue that it's really not that different from what was done in the beginning of the chapter, when we provided a sequence of numbers to the for statement. Now we're just putting some square brackets around it, right?

The difference is that we're storing the sequence of names inside a list object `astronauts`. That object can be modified, passed around and taken apart if needed; *that's* the difference, and a pretty significant one!

Continuing in the same console we can add a few more astronauts, using the list's `append` method:

```
>>> astronauts.append('Alexei Leonov')
>>> astronauts.append('Valentina Tereshkova')
>>> for astro in astronauts:
...     print(astro)
...
John Glenn
Neil Armstrong
Sally Ride
Alexei Leonov
Valentina Tereshkova
>>>
```

5.3.1 Lists of lists

A list is a *container object*; it can contain other objects, regardless of what they are…even other lists! Imagine you are storing the test scores of students. One way to store them could be like this:

```
scores = ['angela', 86, 'bob', 32, 'colette', 62, 'donald', 71]
```

The downside of that method is that when you're looping over the contents of the list, you're extracting a name one time, and a score another time. And while it's certainly possible to make that work, it's also complicated; and complexity is a sure way to introduce errors in your code. But there's a better way; score each couple of student name and test result as its own individual list:

```
>>> scores = [
        ['angela', 86],
        ['bob', 32],
        ['colette', 62],
        ['donald', 71]]
>>>
>>> print(scores)
[['angela', 86], ['bob', 32], ['colette', 62], ['donald', 71]]
>>>
```

First of all, this example shows that when entering a list, you can do so using multiple lines. It is totally optional to put the line breaks where the commas are, but it's a good idea to do so, for readability purposes. However, indentation *is* mandatory!

When printing the list the formatting used to input it is not kept in the same format—the formatting is purely intended for us, when we write the code, to see what it is doing.

Of course when iterating over `scores`, the returned values are the mini-lists *inside* the `scores` list:

```
>>> for score in scores:
print(score)

['angela', 86]
['bob', 32]
['colette', 62]
['donald', 71]
>>>
```

That's nice, but how do you access the values inside a list? There are actual techniques for that, but given that each sub-list has a fixed set of very limited size, we can use a technique called *unpacking* to get a hold of them. Here's how unpacking works, with a simple example based on our `scores` list, that contains the scores of four students:

```
>>> print(scores)
[['angela', 86], ['bob', 32], ['colette', 62], ['donald', 71]]
```

```
>>> a, b, c, d = scores
>>> print(c)
['colette', 62]
>>>
```

In other words, if you put *n* variables on the left side of an assignment, and a list with *n* variables on the right side, each variable gets assigned to one of the elements (in order of appearance) inside the iterable. This works regardless of the iterable; you could do `a, b, c = range(3)` for instance. Packing works the same way; it takes a group of values and turns them into a very special kind of iterable, called a *tuple*:

```
>>> a, b, c, d = scores
>>> new_scores = a, b, c
>>> print(new_scores)
(['angela', 86], ['bob', 32], ['colette', 62])
>>>
```

Tuples are unmutable (unchangeable) lists—we'll see more about them later on—that use parenthesis, instead of square brackets, to visually distinguish them from lists. Packing and unpacking happens on-the-fly (by definition) and is a very powerful technique in Python. Here's how we can apply it to the student scores:

```
>>> for student, score in scores:
print(student.ljust(10), str(score).rjust(3))

angela      86
bob         32
colette     62
donald      71
>>>
```

Since sub-list is always a list of a student and their score, they can be unpacked and assigned to variables "on the fly." At that point, a little string formatting is applied; display the names in a 10-space slot, so the second column is aligned, and convert the numbers to strings, right-aligned in a 3-space slot.

Lists are incredibly rich objects. It has been mentioned before; they form the very core of programming in Python. Virtually all data in a Python program is stored inside some form of list, and most problem-solving techniques evolve around transforming data from one list into another list. There is no way everything about lists can be shown in a single chapter.

For that reason, there will be several chapters focusing purely on the various aspects of lists; but for now, priority is knowing that a list is a container object that can be iterated over with a `for` statement.

Chapter 6

First Recap

While the very basics of programming have been covered it will still be very hard to write even simple programs. There is extended functionality in the statements covered so far—`if`, `while`, `for` and function definitions, and there a couple of useful built-in functions that haven't been touched upon.

This chapter is a short recap of the subjects discussed previously to round up your knowledge about the basics of programming in Python.

6.1 Python Documentation

The Python documentation website and it's not a bad idea to bookmark the location of it:

`https://docs.python.org/3`

On this site there a couple of particularly useful sections:

- **Tutorial**—mainly intended for experienced programmers, the tutorial provides a high-density overview of most of the functionality Python has to over.
- **Language Reference**—insights in all statements available to Python. It's a bit dry, with very formal notation that takes a while to get used to; but it *is* the definitive guide on how things work in Python
- **Library Reference**—saving the best for last! The Python documentation site tells you to "keep this under your pillow" for a reason. This section is the

go-to for about any question that comes up during programming. If you're curious—it contains a full list of all Python functions with extensive documentation on each. But there's much, much more in the *Library Reference*!

6.2 Built-in functions

Python has about 68 built-in functions. Most of them will not be covered in this section, but it is good to know of the existence of a few more than the ones we've encountered (which will be covered here as well).

6.2.1 help

The `help` function is very helpful when in console mode. It will display the docstring (see page 36) of any function that is entered as its argument. Here's an example (in the console) about using it for the `print` function:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

If you provide a docstring with a function (which is why it's such a good idea to always write a docstring), `help` will show that too—one of the reasons to write docstrings.

```
>>> def mult(a, b):
...     '''Multiply a with b'''
...     return a * b
...
>>> mult(3, 4)
12
```

```
>>> help(mult)
Help on function mult in module __main__:

mult(a, b)
    Multiply a with b
```

You can even call `help` on a variable—you will get the docstring of the associated variable type. That's useful because you'll find out what the variable type is in the first place (although there are other ways for that) as well as what to do with it.

```
>>> a = 5
>>> help(a)
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no
 |  arguments are given.  If x is a number, return x.__int__().
 |  For floating point numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a
 |  string, bytes, or bytearray instance representing an integer
-- More  --
```

If there's more output than can fit on a screen, `help` will pause the output—that's what the texttt-- More -- at the bottom means. Pressing the spacebar will output the next page to the console; pressing **Enter** will advance it a single line.

Sometimes `help` will output a lot of text that you're not interested in, and it feels like you have to press **Enter** forever! In that case you can escape out of the help output by pressing **Ctrl C**.

6.2.2 type

The `type` function returns the type of a variable (or value). Consider it the shorthand version of `help`; you'll get information about a variable, but without the entire story. Here's a console example:

```
>>> a = 5
```

```
>>> b = 5.0
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>>
```

The reason the types are listed as `<class 'int'>` and `<class 'float'>` (as opposed to merely `<int>` and `<float>`) is because everything[1] in Python is an object, and "class" is the language used for "object type."

In some languages, like C++ or Java, types are exceptionally important and enforced. That's not to say that types are irrelevant in Python, they're just not as strongly enforced as in C++ or Java. Many outsiders mistake this for "weakly typed" but the Python community prefers "strongly typed but weakly enforced," or in short, "duck typing[2]."

In some cases though it can be convenient to check the type of a variable ("if it's a `str`, do this; if it's an `int`, do that"); be aware that in those cases you want to check against the object *type* and not the name:

```
>>> type(a) == int
True
>>> type(a) == float
False
>>> type(a) == type(b)
False
>>>
```

Something else to keep in mind is that, because of Duck Typing, Python's test for equality is based on the value, not on the type.

```
>>> a = 5
>>> b = 5.0
>>> a == b
True
>>>
```

In some languages (again, C++ or Java) this would be `False` as the types are different—or even generate an error message as you're comparing two different types. If you require two variables to have the same value *and* the same type, you'll have to test for both.

[1] Rabbit hole alert: in Python, classes are objects too!

[2] If it walks like a duck, and quacks like a duck…

6.2.3 input

The `input` function "reads a line of text from the standard input" as the docstring mentions; which is usually the console. It takes one argument, the prompt. Providing a prompt is a great idea, otherwise the user of the program will not realize some input is required.

The `input` function always returns a string; if the user input is expected to be a number, there will have to be some code to convert it to an integer or a float.

```
>>> text = input('Please enter a number>')
Please enter a number>12
>>> text
'12'
>>> num = int(text)
>>> num
12
>>>
```

The input function is a great way to add some exciting interactivity to your program.

6.2.4 range

The `range` function has already been discussed; it provides a range of numbers that can be iterated over using a `for` loop. The syntax is a bit unique as the *first* argument is optional[3]:

```
range([start], stop, [step])
```

remember that *the stop value is not included in the sequence*; if you want to include it, add the step value to it.

6.2.5 round

Another function already encountered is the `round` function which will round the input value on *n* digits.

There's one important detail to notice in the documentation: "The return value is an integer if ndigits is omitted or None. Otherwise the return value has the same type as number."

[3] More correctly: if there's one argument, it's the "stop" value; if there are two arguments, they're the "start" *and* "stop" values

This means that `round(3.14)` is *not exactly* the same as textttround(3.14, 0); the first one returns an integer, the second one returns a float. Usually that doesn't matter—as mentioned in the `type` section—but there are cases where integers are required (array indices, for instance) and using `round(value, 0)` won't do in that case.

6.2.6 len

Not only does `len` return the number of characters in a string, it will also return the number of items in a list. Technically that's the *only* thing it does; the docstring mentions "len returns the number of items in a container," with a string being a container of characters.

```
>>> my_text = 'My parrot is dead'
>>> my_list = [1, 2, 'three', 'four']
>>> len(my_text)
17
>>> len(my_list)
4
>>>
```

While seemingly very limited, `len` is incredibly useful and it's rare to see a Python program that doesn't use it.

6.2.7 min, max

The functions `min`, `max` return respectively the lowest or the highest value from a sequence. That sequence can either be a range of arguments, or a container object:

```
>>> min('the lowest letter in this text')
' '
>>> min('TheLowestLetterInThisText')
'I'
>>> min(12, 3, 5, 18)
3
>>> min([8, 16, -3, 12000])
-3
>>>
```

Letters are evaluated by their ASCII value; hence spaces are ranked lower than letters, and capitals are ranked lower than minuscules (this may come as a surprise; but the ASCII code for the capital letter *A* is 65, and for the minuscule letter *a* is 97).

Something else to be aware of is that these functions will return the highest/lowest value either from the inside of the container (if that container is the single argument) or from the list of values provided, regardless of what those values are.

In other words, when you feed *two* lists into `min` you won't get the lowest value from both lists, but rather the "lowest ranked list," which is based on the first value in the list:

```
>>> min([10, 20, 30], [100, 10, 0])
[10, 20, 30]
>>>
```

It's also possible to define a *key* that is used for ranking. `key` is a named argument, and it should provide a function that takes the values as argument and returns a value that can be compared (a numeric value or a text string).

For instance, we can use `max` to return the longest name from a list by using the `len` function as the key:

```
>>> names = ['aristotle', 'plato', 'socrates']
>>> max(names) # no key; alphabetical order is used
'socrates'
>>> max(names, key=len) # longest name
'aristotle'
>>>
```

Using a key *does* allow you to return the lowest value from a list of lists; first you use the key `min` to find the list with the lowest value, and then you take the lowest value from that list:

```
>>> nums1 = [100, 32, 17]
>>> nums2 = [20, 64, 128]
>>> nums3 = [1000, 100, 10]
>>> min(nums1, nums2, nums3) # we know this doesn't work
[20, 64, 128]
>>> min(nums1, nums2, nums3, key=min) # but this does!
[1000, 100, 10]
>>> min([1000, 100, 10]) # run min on the winning list
10
>>> min(min(nums1, nums2, nums3, key=min)) # all at once
10
>>>
```

Being aware of the textttkey argument can provide many shortcuts when using the `min` and `max` functions.

6.2.8 sum

The last function to look at is `sum`. You'd think that, similar to `min` and `max` it would either take a variety of arguments or a list, but that's not the case; this function **only** works with a list (container) of numbers.

The function does take a second argument, which is the "start" value. It's rarely used, but it allows to add the sum to a starting value, for instance when you keep track of a running total.

As an example, revenues for a shoe store are listed by quarter, inside a bigger list. Note the special notation used to enter a large list; as long as the closing bracket isn't entered you can continue on the next line:

```
>>> revenue = [
...     [12000, 13673, 11894],
...     [14573, 18760, 9734],
...     [11385, 12809, 19765],
...     [14577, 9351, 18375]]
>>>
```

There are better ways to store this, but there it is, a list with the revenues per quarter, each quarter being a list with the revenues by month. We can now write a few lines of code to show the total per quarter, as well as the running total for the year:

```
>>> for qtr in revenue:
...     running = sum(qtr, running)
...     print(sum(qtr), running)
...
37567 37567
43067 80634
43959 124593
42303 166896
>>>
```

Hardly ever will you see `sum` with two arguments, but this is a practical application of using it that way.

6.3 if

The `if` statement in its simplest form is quit easy to understand. It is quite flexible though and can be used as alternative for the "select" statement that is offered by some other languages, by using the `elif` clause:

```
if <first condition>:
  <first condition code>
elif <second condition>:
  <second condition code>
elif <third condition, etc>:
  <third condition code>
...
else:
  <code when none of the conditions passed.>
```

Using the extended form is obviously not mandatory; an `if` statement can be as short as a single "if" (and it's associated code) and nothing else. On the other hand, it can also be used for more complex cases as this example shows:

```
1  def main():
2      print('Grades of students')
3      student_grades = get_student_grade_list()
4      # note 'unpacking' results from get_student_grade
5      for student, score in get_student_grade_list():
6          grade = calculate_grade(score)
7          print(student.ljust(20), grade)
8
9  def get_student_grade_list():
10     return [
11         ['angela', 86],
12         ['bob', 32],
13         ['colette', 62],
14         ['donald', 71]]
15
16 def calculate_grade(score):
17     if score >= 90:
18         return 'A'
19     elif score >= 80:
20         return 'B'
21     elif score >= 70:
22         return 'C'
23     elif score >= 60:
24         return 'D'
25     else:
26         return 'F'
```

Chapter 7

About this document

This document was created with LaTeX[1], a typesetting system that is based on TeX[2]. In the late 1970s, computer scientist Donald Knuth developed this system because he felt that there wasn't a system that would allow him to (automatically) typeset his manuscripts in a visually appealing way.

Latex has become the standard for writing materials in the academic world and is, fourty years after its original inception, still relevant. It's a lot *easier* to use a product like Microsoft Word, but at the same time it's a lot less *frustrating* to use Latex. Despite actually being *more* automated than Word, Latex tends to be less intimidating, and it overriding decisions less often than Word does.

In addition, when special formatting for code is desired, Latex tends to be a lot easier. Code can simply be copied and pasted into the Latex file, identified with a marker, and no other formatting is needed. While Styles in Word are easy to use, this is even easier!

Because Latex documents are written in plain ascii, in a way very similar (yet different!) to html, you can edit your documents anywhere, even if you don't have Latex installed on the device you're working on. And because the Latex file is effectively a script that instructs Latex on how to generate output, it is trivial to switch, for instance, between A4 size and Letter size, without having to reformat half the document.

[1] pronounced "*lay–tek*"
[2] pronounced "*tek*"

7.1 Getting started

LaTeXis an open source project and there are many implementations. A good place to start is `https://miktex.org` who provide a "turnkey" implementation of LaTeX. Download the latest version, install it, and you're off to the races!

`http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf`
This is a great place to start. The manual is pretty short but it gets you going with a few *Hello, World* type of examples.

`http://mirrors.ibiblio.org/CTAN/info/lshort/english/lshort.pdf`
This manual is still aimed at beginners but provides a bit more insights and reference for features that are often used.