Further Python

The alternative and more complete guide to python programming

Contents

Chapter 1. Introduction

1.1 What is this booklet, and how does it work?

This booklet is a guide to how to code where I both cover the principles of programming, which you will be able to port over to other languages you learn, and the syntax of python. Throughout I include exercises and a few worked examples. The first couple of chapters cover just the syntax and some basic principles, and towards the end I cover more of the complex and high level programming concepts.

1.2 Meet python

Python is a very high level language that supports a lot of features and is powerful in the right hands. It can be used for a very large amount of applications, such as data science, AI/machine learning, as well as some simple embedded systems such as the raspberry pi, which which runs on python. Python has a history of being a scripting language, so it's also commonly used for some system tasks and tools.
Due to python's high level nature, it does a lot of work under the hood for you, meaning it's fairly easy for people new to coding to pick up.

1.3 Setting up your environment

1.3.1 Installing python

There are a couple ways to install python:

- Using the official package:

Head over to python's website's downloads, and download and follow the instructions for installing it.

- Using a package manager:

You can also your OS's package manager for this, so `brew` on MacOS, `winget` or `choco` on windows and `apt`, `pacman`, `zypper`, `portage`, `dnf`, `xbps`, `pkg`, or any of the others I may have missed for GNU/Linux or similar. Don't worry if you don't know what this is.

Once you have python installed, there are a couple ways to run python code on your computer, I show two ways:

- Using a shell/terminal and (neo)vim

I prefer this way and know how to this way better so I detail this below

- Using a traditional text editor, like VSCode(ium) or Zed or PyCharm

This has big friendly buttons to do everything, so if you're with one of these then you should be fine.

1.3.2 Hold on, what is the python package?

When you installed the python package, you actually get the full python system, which has:

- The python virtual machine
- Python's 'compiler'
- A couple tools like `pip`

The way python runs is like this:

```
Input -> compiled to bytecode -> ran on the virtual machine
```

This is partly why python has a reputation for being slow – it doesn't compile to machine code. In this way python is interpreted, meaning it processes the input line by line.

1.3.3 Using the shell and vim

If you have a simple file, like this:

```
#!/usr/bin/env python3

def main() -> None:
    print("Hello reader!")


if __name__ == "__main__":
    main()
```

Okay that was kinda a flex for what you'll learn later on, you may as well have just written this to do the same thing:

```
print("Hello reader!")
```

But the earlier example is something known as a script, which means

on unix based systems you run `chmod +x {filename}.py` and run it with with path, for example the `./` shorthand. It tells anyone else looking at your file that it's meant to be run directly.
Anyway, now that you have the file, and content put into it, save it with the .py extension and open your terminal, navigate to the file and run `python3 {filename}.py`
If that didn't make sense I'll explain it in detail below, if it did, or you can't be bothered with this and want to just skip to using a GUI editor (with big friendly buttons) feel free to jump to the next heading.

1.3.3.1 How does a terminal (shell) work?

You may have seen the 'Terminal' application, but never known how it works. If you're on Linux I'm sure you know how it does, you can skip this part.
Really a terminal is just a UI element for a shell which uses system calls to talk to the kernel to run things. You can do almost everything you can in a GUI in a terminal, plus some other things you can't do in a GUI.
The only thing that the shell does it navigate the file system and run executable binaries.

1.3.3.1.1 The file system

On Unix-like systems, such as MacOS (actually based on BSD (Berkeley Software Distribution)) the file system uses / forward slashes and is just files and directories holding files. This is following the Unix philosophy.
To navigate it use the `cd` command (change directory) and use `ls` to list the files in the directory. Use `pwd` to see your current directory.
Here's how you might navigate it.

```
❱ ls
Desktop                                 Documents
Downloads
Pictures                                Public
Music

~
❱ cd Documents/
```

```
~/Documents
❱ ls

~/Documents
❱ mkdir code

~/Documents
❱ cd code

~/Documents/code
❱ ls

~/Documents/code
❱
```

Here all I did was list the files, go to my Documents folder and make a new folder there called code, and then went into that folder. The equivalent for doing this inside a GUI like Finder or File Explorer would be just to double click the folders to go in them, and make a folder with the right-click menu.
Also you can go into multiple subdirectories at once by chaining them together like this:
`cd ~/Documents/code`
Here I also used the ~ shorthand for your home folder. You can check the environment variable for what it is by running:
`echo $HOME`
which will give you something like `/Users/John Doe/` or on Linux `/home/John Doe`.
To go back one directory (the 'parent' directory), you use `cd ..`
Interestingly the '.' directory is the current one, and is used most commonly to run scripts, with this './' syntax. To go to the home directory run `cd` without any arguments.

Tab to autocomplete

When you type commands, modern shells like zsh, or bash have this really helpful function, where if you press the tab key (the one above caps lock that looks like ->|) it will try to guess what you are trying to type and autocomplete it. In the terminal, when you type a path, just start typing something (like `cd Docum`) and press tab to autocomplete the path.

1.3.3.1.2 Running binary executables

The other crucial part is running binary executables, which is literally anything on your system that runs. Here I'll just cover shell commands, as those are the ones you don't yet know of and need to know, but there are be many others, like your browser, and many shell commands not covered here.
On Unix based systems there's a `$PATH` which is a variable showing all the directories that the shell searches to find a command if you want to run it. If it's not there, you'll get a command not found error.
All files that are meant to be ran need to be executable, so you need to give them the execute permission, by running:
`chmod +x {filename}`, and then run it by giving the path to them. This is a security measure to anything not in your path so the system doesn't look for the executable in the entire file system, and so that you can't have arbitrary code execution.
There are a bunch of built-in shell commands, such as:

- `cd`
- `ls`
- `pwd`
- `echo`
- `grep`
- `sed`
- `awk`

And some others you install, or that come with the system, such as:

- `man`
- `curl`

- `wget`
- `make`
- `git`
- `python`
- `nano`
- `vim`

`man` is an awesome command to view the manual pages for almost all shell commands, so much so there's a saying that goes 'the only command you need to know is man'.
Another way to get documentation on how to use a command is to run it with `--help` which gives the usage options most of the time.
We need a way to write code and edit text for this booklet, as you'll need to solve exercises and try out python to actually learn it.
You could use the builtin text editor on MacOS called 'TextEdit' or IDLE which uses it, but that kinda sucks.
A solution that works okay-ish but isn't amazing is using nano, but you need to get syntax highlighting on it by downloading a bunch of syntax files for it. Here's a way to do that, just copy and paste all of this into your terminal:

```
mkdir ~/.nano > /dev/null
cd ~/.nano
curl -0 -L https://github.com/scopatz/nanorc/archive/refs/
heads/master.zip --output nano.zip
unzip nano.zip
rm -rf nano.zip
mv nanorc-master/*.nanorc .
rm -rf nanorc-master
echo 'include "~/.nano/*.nanorc"' > ~/.nanorc
cd $HOME
```

This makes the ~/.nano directory but doesn't complain if it exists already, goes into that directory, downloads the files as a zip, unzips them, deletes the zip file, copies all the .nanorc files (those are the syntax files) to the ~/.nano directory, then adds a line to the main nano config file (~/.nanorc) saying to use all the syntax files, and then goes back to the home directory.
Then to use nano you run 'nano' in the terminal.
To edit an existing file or to make a new one you run 'nano {filename}',

in our case a .py file as we are writing in python.
To exit, use C-x (hold the control key*, press x).

*For those of you thinking that you're really smart and know that on a mac the command key is the control key, unfortunately you're wrong, and in the terminal everything will be control, except for copy/paste which is still command-c command-v.

Using vim:
I think that vim is a much better text editor for general editing and for code, and that while the initial experience is a lot less nice than nano, but worth the pain. Vim has a lot more features and commands in it, as it's a modal editor with a bunch of features, so you really need a guide to get started. To do this, run `vimtutor` in the terminal to get a guide.
If this doesn't work, run vim, then press `:`, then type `Tutor`.
It takes about half an hour at first, at which point you have a good starting point for vim. Once you've done that you have a baseline, you can start using vim.
I also suggest using their baseline init.vim or init.lua (for neovim, which is the modern version of vim, but you need to install it separately).

1.3.3.2 The REPL

Python comes with a REPL (Read Evaluate Print Loop), which you can run just by typing `python` (or `python3` on MacOS) which lets you quickly try out parts of the language. You can type in it like you were in a file, and it runs all the code right there letting you play around with the language a bit easier.

1.3.4 Using a GUI editor like IDLE or Zed or VScode(ium) or PyCharm etc

There are many great editors that work, I happen to like Zed the most, but they all work, and with the python extension/plugin they all have linting and LSP and everything you may want out of the box. Just use their interfaces, save with File>Save As, Choose a filename, and then just save it as you go along, and run with the run button in them.

1.4 Running your first line of code

Let's end this chapter by introducing you to your first function, the `print` tool which allows you to `print` text to the screen.

```
print("Hi!")
```

Copy this into your text editor (whichever it may be) and run it (either with a button or `python3 hi.py`). Note the text has to be in either single (`'`) or double (`"`) quotes to work. You'll learn why later.

1.5 Summary exercises

1.5.1 Ex1

Print a triangle

Using just the print function you learned, print this to the screen:

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

Chapter 2. The Basics: Using python as a calculator

You may have heard of python used for data analysis, so let's first explore python through how it handles maths. For this it's easier to just use the interpreter shell.
Let's have a brief look at python's operators.

Operator	Name	Example	Output
+	Addition	2 + 9	11
-	Subtraction	2 - 9	−7
*	Multiplication	2 * 9	18
/	Division	2 / 9	$0.2\overline{2}$
//	Integer division (returns floored quotient)	2 // 9	0
%	Modulo (returns remainder of division)	2 % 9	2
**	Exponent	2 ** 9	512
@	Matrix multiplication	[[1, 2], [3, 4]] @ [[5, 6], [7, 8]]	[[19, 22], [43, 50]]

(You don't really have to worry about the last one, you almost never need it)

But wait, there's more!
As well as arithmetic operators, there are also logical and bitwise operators. Let's have a look at these as well.
Logical operators:

Operator	Name	Example	Output
`==`	Equality	`"" == []`	`False`
`>`	Greater than	`10 > 10`	`False`
`<`	Less than	`10 < 10`	`False`

Operator	Name	Example	Output
`>=`	Greater-than-or-equal-to	`10 >= -1`	`True`
`<=`	Less-than-or-equal-to	`10 <= 10`	`True`

Operator	Name	Example	Output
`and`	Logical AND	`True and False`	`False`
`or`	Logical OR	`True or False`	`True`
`not`	Logical NOT	`not (True and False)`	`True`

All these values return a boolean (True or False).
The last three are slightly different so I'll go over them:
The `and` compares two boolean expressions, and returns `True` if they are both True.
The `or` also compares two boolean expressions, and returns `True` if they are different, otherwise returning `False`.
The `not` is a bit different and is used to negate a boolean expression, as shown above. So if it's True it returns False and vice versa.

Lastly I want to cover bitwise operators, which are a bit more confusing and aren't used very often, but can sometimes be useful like multiple return types from functions, where you want either to return the value or to return None or an error or something.

Operator	Name
&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR
>>	Bitwise Right shift
<<	Bitwise Left shift

They work only on integers and they perform binary operations, which are beyond the scope of this guide, but if you're interested look them up!
The only ones I'll explain here are the << and >> bitwise shift

operators, as they are part of the computer science course:
What they do is take a binary number, like take 10 base 10 in binary:

```
0000 1010 ; base 10 binary, let's perform a left shift
0001 0100 ; as you see we just shifted everything to the
left, this is now 20 in base 10
```

In base ten, let's have a look how this would work so it's more familiar, so suppose you have the number 24, and we shift left, we get 240, in effect we multiplied by 10 - the base number. Now let's perform two right shifts: 240 becomes 2.4. This divided by 100. (actually it would just be 2, as this only works with integers so truncates the rest)
These shifts are really fast for shifts in computers, but can sometimes be slightly inaccurate as it works on integers and truncates when dividing (performs *integer division*), so for example `11 >> 1` becomes 5. In base 2 (binary), it ends up multiplying and dividing by 2 per shift.

You may have seen some C++ before that uses these `>>` and `<<` operators for printing to the screen. This doesn't happen in python, but for the curious readers this is how it looks:

```cpp
#include <iostream>

int main() {
  std::cout << "hi!\n"
  return 0;
}
```

2.1 Summary exercises

2.1.1 Ex1

Syntax check

- **2.0**: Get the floored quotient and remainder from $\frac{7}{3}$, and print them to the screen.

Chapter 3. Variables

3.1 What is a variable?

A variable is a crucial part of programming in any language, and is a part of main memory (memory) for storing data in a program.

In actuality, a variable is a symbolic identifier bound to a mutable reference that stores an object or value in memory. You can think of a variable as a named container for a type of data. Technically it functions as a named pointer to a memory location where data is stored, enabling indirect manipulation of the underlying value through the identifier.

Let's break down what that means:
A variable is just a name that references something - usually an object as python is an Object-Oriented Programming (OOP) language. This means that you use classes to make objects of certain data types, which can be both built in and custom. It holds the memory address which 'points' to the memory where data is actually stored, hence the name pointer.
Because of this, python doesn't actually have variables, just a name that stores a memory address. This later helps explain some things such as how functions pass parameters, and data mutation.

There are also constants, which are in almost all respects the same as variables, but they are immutable – which means that their value cannot change throughout the program.
This can be beneficial as the compiler knows the size of it at runtime, so they are memory safe to use in the stack memory (stack), unlike variables, which are usually stored in the memory heap (heap).
Therefore variables and constants must be stored in **memory** when the program is running.

3.2 Variable names

When choosing variable names there are a few rules you have to follow:

1. variable names must not have spaces - so you couldn't have `my name` as a variable, you would have to have either `my_name` or `myName` or `MyName`
2. variable names must not use any reserved keywords - you **cannot** use them as a name. See below for a list.
3. variable names must not start with a number or an operator: so you can't have `3apples` as a variable, or `+count` as a variable, however you can have numbers in your variables - so `more_than_3_apples` is ok, and you can start with an underscore - `_3apples` is ok. The last use is often used for variables which aren't used in the program.
4. If you go with a naming style keep it consistent throughout.
5. This isn't exactly a rule but follow pep8 naming conventions: use `snake_case` for variables and function names, use `PascalCase` for class names.
6. Try to use sensible names for variables

e.g for a student's name use `student_name` not `sn` or `a`. A good name is descriptive for what it is.

A full list of keywords is shown here.

```
False           class           from            or
None            continue        global          pass
True            def             if              raise
and             del             import          return
as              elif            in              try
assert          else            is              while
async           except          lambda          with
await           finally         nonlocal        yield
break           for             not
```

Also, by the end of this booklet I would have covered every single one of these, and you would know how to use them all yourself, so won't have to come back and check this to see if you're allowed to use a variable name.

Note: *Equality* vs *Assignment*

In python the `=` is the assignment operator. It assigns the thing on the right of the = sign to the thing on the left. It's like telling the computer: "Make the thing on the right refer to the thing on the left". It is **not** equality, and they are **not** the same. When you say: `name = "Linus"` the variable `name` is assigned to the string `Linus`. The variable and `Linus` are **not** the same.
However to check for equality you can use the `==` operator. This again isn't the same as the = sign, but it checks for the equality. This is the equality operator, and it will give you either a `True` or a `False`.
As shown, the variable name is always on the left, and the value assigned goes on the right.
Therefore `age = 50` works and `50 = age` doesn't.
If you're wondering why this is and how it's weird, this is discussed much later in my functional section (link here) that explains an alternative paradigm. However what is shown above works in procedural/OOP/imperative languages, which is what most python is.

Now, since we assign variables, for calculations we can use variables instead of actual values. Here's an example:

```
length = 13.6

width = 5.7 # Define width and length and store them in
variables

area = length * width # Calculates area by multiplying
length by width, stores it in a variable but doesn't do
anything with it

print(area) # Outputs 77.52
```

Comments in your code

Any line that begins with a # character in python is ignored, meaning your can write 'comments' in your code, like to explain what it does/why it's there, but also you can use it to temporarily stop code from running when you're testing your code. You can also do this by surrounding it in triple """, which is known as a docstring (mostly used for documentation of functions).
Anyway, here I just use to to explain the code inside it. I go more in-depth on comments, and perhaps why you shouldn't explain how your code works in them in chapter 9 chapter 9

Using variables instead of values is a good practice in coding, as it makes your values more reusable, and often can read like a comment even if it's just used once, for example like here:

```
if time >= 300:
    ...
# Bad, this we don't know what 300 is here!

SECONDS_IN_FIVE_MINUTES = 300
if time >= SECONDS_IN_FIVE_MINUTES:
    ...
# Better, this now reads more like a comment and explains
the code nicer.
```

Assignment shorthand (augmented assignment)

Not infrequently you'll want to update a value, and normally you would have to write it like this:

```
increment = 5
value = value + increment
```

But there's a nicer way of writing this, which is by putting the operators together, like this:

```
increment = 5
value += increment
```

Which is exactly the same as the previous, but it reads a bit nicer. This works for all operators, like so:

```
a += 1
b -= 1
c *= 5
d /= 5
e **= 8

f &= 2
g |= 10
h ^= a
i ~= e
j <<= 1
k >>= 1
```

This is an example of syntactic sugar

3.3 Summary exercises

3.3.1 Ex1

3.1: Temperature conversion

First copy this code into your python file:

```
celsius_temp = float(input("Please input your celsius
temperature: "))

# Put your conversion here, and store it in a variable
called "fahr_temp"

print("The corresponding Fahrenheit temperature is: " +
fahr_temp)
```

Then use the formula: $F = \left(C \times \frac{9}{5}\right) + 32$ to convert the inputted Celsius temperature to Fahrenheit.
The way this program works is as follows:

1. A user types in a Celsius temperature
2. Python stores this in the `celsius_temp` variable
3. This is converted using the operations mentioned back in Chapter 2 (you add this to your program)
4. The corresponding Fahrenheit temperature is printed to the console

3.3.2 Ex2

3.2: Swap variables

Start with two variables:

```
a = 5
b = 10
```

Print them to the screen.
Then swap their values, so that a becomes 10 and b becomes 5.
There are three ways of doing this, if you've read a lot of python you may know the second, fancy way – and you can also use the bitwise XOR (^) to do the swap.

3.3.3 Ex3

3.3: Speed camera

Start with this code:

```
km = float(input("Distance km: "))
mins = float(input("Time minutes: "))
```

Add to it to calculate the average speed and print it to the console.

Chapter 4. Basic data types

Every variable you make has to be of a certain type, and below are the basic data types for variables.
Actually when you make a variable you initialise an instance of an object, which is made from one of the in-built classes python has. These objects have certain properties, and functions. The functions are called `methods` and are what the object can do. For example for a string, you can convert it to all upper case by having `string.upper()`. The `.upper()` is the method, which is a function on an object that allows it to do something.

4.1 String (str)

A string is just a series of characters, in quotation marks, either double or single. In lower level languages a string is an array of characters, often terminated with a null terminator ('0').

4.1.1 Character (char)

Python doesn't have the `char` data type, but many languages, including OCR reference language, do - so you need to know about it. A character - as the name might imply, is just a single character. You can think of it as a string constrained to just a single element; however this is wrong low level as it's actually the other way around, where a string is an array of characters, but you don't have to worry about this in python.

4.1.2 Common methods for strings

- `.upper` converts to upper case
- `.lower` converts to lower case
- `.len` returns the length of the string
- `.startswith(char)` returns a boolean telling you whether it starts with the specified character
- `.endswith(char)` returns a boolean telling you whether it ends with the specified character
- `.split(char)` splits your string into a list based on the given deliminator you passed in

- `.format` allows for injecting variables into lists (shown below)
- `.isdigit` returns True or False depending if only digits are in the string

4.1.3 String operations

4.1.3.1 String formatting

Suppose you have some variables you want to use in a string, for example in this situation:

```
name = "John"
age = 50
greeting = "Hello person, you are age years old"
```

Here we want to personalise this greeting to John, so we need to inject the variables `name` and `age` into the string. To achieve this we have four different options:

- String concatenation

This is like sticking the variables and the strings together. Here's how you would use it:

```
name = "John"
age = 50
greeting = "Hello " + name + ", you are " + str(age) + "
years old"
```

This also works with a `,` instead of a `+`, with the benefit that it adds a space for you. It's okay to use sometimes but it is often hard to read and tedious to use.

- `.format` method

This uses the method (which is a function on an object) `.format` to use placeholders and put the variables after the string. Here's how it's done using this method:

```
name = "John"
age = 50
greeting = "Hello {}, you are {} years old".format(name,
age)
```

Here placeholders are used - in this case the `{}` are the placeholders, which you then specify the variables to go into the placeholders. Note

the order matters, if age was specified first at the end, you would get `Hello 50, you are John years old`

- Alternative `.format` method

```
name = "John"
age = 50
greeting = "Hello {name-str}, you are {age-str} years
old".format(name-str = name, age-str = age)
```

This is extra explicit, and more readable. It also solves the order problem, as you are assigning names to the placeholders you give.

- fstrings

Short for 'format strings', this is a newer python feature, only being available on python 3.6 and up. To use it, the variable is placed inside the curly braces like so:

```
name = "John"
age = 50
greeting = f"Hello {name}, you are {age} years old"
```

It is the most readable and probably best way of using string interpolation in python. It also has the advantage that you can put calculations in the curly braces, as anything inside them is treated like code. Try to use it when you can over the other methods.

- rstrings

Short for 'raw strings', this means anything you put in the string is taken as is, so you can print escape sequences.
For example:

```
print(r"C:\Users\nathan")
```

This is an example for a use case, it's a file path in windows and uses \ for paths, which means you can accidentally include escape sequences, like \n, and to avoid having them mess up your string you use a raw string (rstring)

If you need a multi-line string you can use triple double or single quotes like this:

```
multiline_string = """This string
has
multiple lines"""
```

or

```
single_quote_multiline_string = '''
This string too,
has many lines
and it's formatted like this
'''
```

This is called a *docstring*.
It may be wiser to either use double or single quotes for your strings, for example doing this might not work:

```
string = 'This string will fail as it's not formatted
properly'
```

If we run it we get this:

```
>>> string = 'This string will fail as it's not formatted
properly'
  File "<stdin>", line 1
    string = 'This string will fail as it's not formatted
properly'
                                           ^
SyntaxError: invalid syntax
```

This is because it uses the `'` in `it's` to close the string, even though that's not the intended end of string.
The solution here is to use double quotes, instead of single quotes.

Tip: Using escape characters

In python, there is an escape character: the backslash `\` - which allows you to 'cancel' the next character.
This can be used if you have quotes in your string - for instance the string shown above:

```
string = 'This string will fail as it's not formatted
properly'
```

can be fixed like this:

```
string = 'This string won\'t fail as it\'s formatted
properly'
```

Here the `\` escapes the quote characters, fixing our strings.

There are some characters which you can't type in a string, such as new lines and tabs, so you use the escape character here.
Here's a list of the ones you can use:

- `\'` for single quotes
- `\"` for double quotes
- `\n` for new lines
- `\t` for tabs
- `\v` for vertical tab
- `\h` for horizontal tab
- `\r` for carriage return
- `\b` for backspace
- `\a` for bell (alert)

Auto concatenation for readability

Strings placed next to each other are automatically concatenated, so this:
`awesome_string = "This is" " a really awesome string"`
Which is useful if you want to spread out your string across multiple lines to make it more readable, for example:
`print("There are two papers for computer science:\n\t1. Computer Systems \n\t2. Computational thinking, algorithms and programming")` is a bit annoying to read so you probably want to split it up like so:

```
print("There are two papers for computer science:\n"
"\t1. Computer Systems \n"
"\t2. Computational thinking, algorithms and
programming")
```

But then at this point you may as well:

```
print("""
There are two papers for computer science:
    1. Computer Systems
    2. Computational thinking, algorithms and
programming
""")
```

4.1.3.2 String slicing

As you'll see later, strings are a bit like lists (and in older languages like C, the string data type doesn't exist, you instead store them as arrays of characters), which means you can treat them like lists in some aspects.
One of these aspects is string slicing, which is dividing it up to get a substring back. Here are a few examples:

```
my_string = "Lorem ipsum dolor set amet"
assert my_string[3:9] == 'em ips'
assert my_string[3:] == 'em ipsum dolor set amet'
assert my_string[:3] == 'Lor'
assert my_string[:-1] == 'Lorem ipsum dolor set ame'
```

If you hadn't guessed `assert` returns `True` or `False` after evaluating the statement. It's mostly used for testing (which I cover towards the end of this booklet), but here I just show the outcome with it.

4.2 Integer (int)

An integer is a whole number - positive or negative.
In many other languages there are signed and unsigned ints, where unsigned can't be negative.
Python's ints are unconventional in the sense that they are dynamically allocated memory - so having integer overflows can't happen.
You could use your whole memory to hold a single integer and you would be fine.

4.3 Float (float)

A floating point number, or float, is a data type that can hold decimals. It's handled as a floating point, as it can move around in the number and the memory usage is the same. Have a look:

```
a = 1.2345
b = 12.345
c = 123.45
d = 1234.5
```

All of these numbers take up the same amount of memory, but the decimal point 'floats' in the number.

4.3.1 Floating point number arithmetic

Beware, sometimes floats misbehave - because of how the computer - which works in binary - handles them. Take a look at this example:

```
>>> 0.1 + 0.2
0.30000000000000004
```

This is due to routing that occurs, a bit like how you can't store $\frac{1}{3}$ as a decimal properly in denary.

4.4 Boolean (bool)

A boolean can be `True` or `False`. This is often returned as a value for use in control flow (see Chapter Chapter 6.) They are handled as ints in python, and if you cast them as an int you get 0 and 1:

```
>>> int(False)
0
>>> int(True)
1
```

This can sometimes be used for checking if a number of values are true.

You can convert data types by 'casting' them.
This involves wrapping the variable or value in the type:

```
a = 10 # a is an integer here
a = str(a) # Here I cast it as a string
print(type(a)) # outputs "<class 'str'>"
```

You might have noticed the *class* when printing the type of a variable. That is - as mentioned before - because a variable is bound to an object of its class. The class determines its type when printing the type of a variable. That is - as mentioned before - because a variable is bound to an object of it's class. The class determines its type.

4.5 None

`None` is sort of a weird data type, but it signifies the absence of a value. You see this most frequently as either a check for presence or as a type annotation.
This is often `void` in other languages.

4.6 Summary exercises

4.6.1 Ex1

String indexing

- Print first, last, middle, reversed of `word = "python"`.

As much as I would like to give some more fun or challenging exercises here I can't really as I haven't covered the syntax required for it yet. Stick with it and you'll get to the juicy stuff!

Chapter 5. Advanced data types

5.1 Lists

A list is a mutable, ordered, indexable collection of data that can be heterogeneous. Let's explore what that means: Mutable means that it can change.
Ordered means it has an order, and it won't change, unlike with sets.
Indexable means that each value has an index, and you can view and change via indexing.
Heterogeneous means it can hold mixed data types.

5.1.1 Common methods for lists:

- `.append()` adds anything you pass as an argument in the brackets onto the end of the list
- `.pop()` removes anything you pass as an argument in the brackets from the end of the list, and also returns that value
- `.extend([])` adds a list onto the end like append, the advantage being you can add more values in one go

5.1.2 Indexing in action

Each value in the list has an index, and these are 0 based.
So in the list `[1,2,3]` the indexing assigns index 1 at index 0, 2 at index 1, and 3 at index two.
Negatives are also allowed, so index −1 is 3, index −2 is 2 and index −3 is 1. Here negative indexes start from the back of the list.

To access an index, you use this syntax: `list[index]`

5.1.3 Slicing lists

You can also 'slice' a list, meaning you return a new list containing the new, specified elements with indexing.
This works the same way it does with strings.

```
>>> list = [0,1,2,3,4] # Define an intial list
>>> list[1:3] # Slice said list
[1, 2]
```

Let's see how all this works in an example:

```
list_one = [0,1,2,3] # Initialisation of a list
list_two = [0]*10 # shorthand for making a list of 10 0s,
like this [0,0,0,0,0,0,0,0,0,0]
list_two[4] = 10 # Change the 4th element by index (so fifth
if you count from one) to 10
list_one.append(4) # list_one is now [0,1,2,3,4]
list_one.pop(4) # Removes 4 from the list
```

5.2 Arrays

Arrays are in almost all respects the same as lists, except for just some small differences as shown below

- Homogeneous data types
  Arrays must hold a single data type, unlike lists that can hold mixed data types
  - Requirement for explicit type specification
    When you make an array in python, you have to explicitly specify the type
- Stored in contiguous memory
  This means that they are much faster to read and write to
- Import required
  You have to import them with `import array`, unlike lists which are in built

5.3 Tuples

Tuples are much like lists, with they exception that they are immutable.
In all other respects they are the same, so they are indexable and can hold mixed data types (heterogeneous data types).

You declare a tuple with `()` as opposed to lists which use `[]`.

Shorthand for tuple creation

When making a tuple with the:

```
my_tuple = (0,)*10
```

shorthand, you always have to put the comma after the digit in the brackets, (as shown above) because otherwise it will be interpreted as normal brackets (which mean order of operation).

Since tuples are immutable, you can't append, extend or add them, but you can slice them, which returns a new tuple back to you.

The main advantages of tuples over lists is that you know that they won't change in your program, and they can be used if you need a hashable collection.

5.4 Dictionaries

A dictionary is a set of *key-value* pairs, is mutable and can hold heterogeneous data types.
So for example here is a mapping of people and ages:

```
people_to_ages: {"Ethan": 56, "Adam": 44}
```

To look up the age of Ethan here, you can do so like this:

```
ethan_age = people_to_ages["Ethan"]
```

If you try to look up the key of someone that doesn't exist you'll get a `KeyError`.
If you want to avoid this you can use the `.get` method like this:

```
billy_age = people_to_ages.get("Billy")
```

Which will return `None`. Alternatively for this you can specify a default return value like this:

```
billy_age = people_to_ages.get("Billy", -1)
```

This returns −1.

Now suppose we want to add Billy to this dictionary. Here's how we add a new value:

```
people_to_ages["Billy"] = 7
```

Dictionaries can hold mixed key-value pair types, so we could add an integer to a boolean here like this:

```
people_to_ages[69] = True
```

To delete a key-value pair you can use python's built in `del` operation like this:

```
del people_to_ages["Adam"]
```

Note: Keys must be hashable

When making a dictionary you can set anything as the value for anything, but you have to have the `key` be hashable.
In python, any immutable type is hashable, so: ints, floats, strings, booleans and tuples are all hashable as they are immutable.
You cannot use lists, sets or other dictionaries as keys.

5.5 Sets

Sets are mutable, unique, unordered, hashable collections of data that can hold mixed data types.
This means that you cannot have multiple of an element, and the order is not preserved. To make a set you use curly braces like this:

```
set = {1,2,3,4}
```

Note if you want to make an empty set you have to use the `set()` function as empty `{}` curly braces are interpreted as an empty dictionary. Here's how to do that:

```
empty_set = set()
```

You can also use the `set()` function to convert a list to a set like this:

```
set_from_list = set([0,1,2,3])
```

5.5.1 Common methods for sets

- `.add` adds an element to a set
- `.remove` removes an element from a set

Just like dictionaries you can only add hashable data types to sets. You can't index a set as the elements are unordered, but instead you can use the `in` keyword for it like this:

```
my_set = {1,2,3}
print(0 in my_set)  # Prints False
```

This is mostly what sets are used for, as checking if something is in a set is much faster than in a list.

Using the `in` keyword

The `in` keyword works for all container data types (lists, arrays, tuples, sets, dictionaries etc), and also custom types with the `__contains__` dunder method (though this is part of the OOP of python, see chapter 13 for an explanation.)
This works exactly the same way as the `.find` method or a normal search, but it's much faster as it's a python in built.

Note: Type annotations

Python is a **Dynamically typed** language, which means that when you create or use variables, you don't have to explicitly state the type, like you might have to in some other languages, such as C, C++, Go or Rust.
For example, if you wanted an integer `a` with a value of 10, you wouldn't have to write `a: int = 10` where you explicitly state the type, you would just be ok with `a = 10`, and the translator figures it out. But this `: int` type annotation can be really helpful in making your code more readable, obvious, and easier to debug, as often your editor will give you a hint that something's wrong more often if you use type annotations.
To use type annotations, you can add `: {type}`, with any type, including custom types in OOP, and functions, and `None` for no return.
To show a return type from a function (link to functions chapter here) you can use this: `-> {type}`
so for example:

```
def simple_function() -> None:
    print("This function doesn't return anything")
```

Here you can also use the bitwise operators you learned way back in chapter 2 , to have multiple return types, or to have annotations for if you might return something, or return None. Here's how you might use them:

```
def check_length(list: list) -> int | None:
    return list.len if list is not None else return
```

This function returns the length of a list if the list exists, otherwise return None.

Tip: Data types used in advanced data types

As well as storing basic data types in the above advanced data types, you can use them to store almost anything you want, (unless explicitly mentioned otherwise) so for example you have have a list of lists (which is called a matrix or a 2d list/array), or a dictionary of dictionaries, or a list of dictionaries, and so on. This can at times be very useful.

Mutability of data types

Some data types like lists and dictionaries are mutable, and this makes sense, if you change the object (like appending a value to a list) it actually changes the value. However this isn't fully true with all data types. Here's a list of mutable and immutable objects:

Immutable

- Strings
- Integers
- Floats
- Booleans
- Tuples

Mutable

- Lists
- Sets
- Dictionaries

What this means for you is that if you wanted to have a function that uses a list but doesn't change it, you need to create a slice of it, because, as we will see later, python holds the memory address of variables, not values, which means if you were to try to make a copy of a list, like so:

```
list_copy = list
```

they would point to the same address in memory. Really you would need to do either this:

```
list_copy = list[:]
```

or this:

```
list_copy = list(list)
```

5.6 Summary exercises

Loop a list

suppose you have a list like this:

```
epic_list = [1, 2, 3, 4, 5, 6]
```

And you want to take the first element and put it at the end, to get this list at the end:

```
epic_list = [2, 3, 4, 5, 6, 1]
```

See if you can write only **one** line of code to make this work. This should teach you about the order code gets executed, and how functions/methods order, and when calls open and close. You'll probably understand it better when we get to the functions chapter, but this is a nice introduction.

Chapter 6. Control flow and programming constructs

Control flow encompasses how the program runs - in which order and how the code is executed. There are three you need to know of:

- Sequence
- Selection
- Iteration

Let's talk about each one in detail:

6.1 Sequence

This is when one piece of code gets executed in a linear fashion, one after another.

6.2 Selection

This is when there is a 'choice' as such, where either one option is taken or another - like a fork in the road of which way your code could execute.

Note: Indentation matters!

Everywhere you see a `:` at the end of a line it will always be followed by an indented block in python. Python uses indentation over curly braces to define scope, so keeping neat indentation is crucial. Always use **4 spaces** for indentation (you can change your editor's tab key to be 4 spaces).

Note: Truthy and Falsy values

All of python's standard data types have truthy and falsy values:

```
int   -> 0
float -> 0.0
str   -> ''
list  -> []
tuple -> ()
dict  -> {}
set   -> set()
```

In all of these cases, they evaluate to `False` in a boolean context, which means that you can make if statements with just

```
if variable:
    # Some code
```

Any other value for these will evaluate to `True`

6.2.1 Using `if`, `elif` and `else`

Here's how it works:

You use the `if` keyword followed by a condition that returns a boolean or has a truthy value, and if it evaluates to True, then the code in the indented block runs.

Let's see an example:

```
if age > 18:
    print("You can (legally) drink ethanol")
```

You can also check for more than one condition using the `and` and `or` keywords.
Let's see an example for this too:

```
if age > 18 and country_of_residence != "USA":
    print("You can (legally) drink ethanol")
```

Warning: using more than one condition

Remember that when you have more than one condition like above, you have to specify the full condition.
So for example if you have this code:

```
if number < 100 and number > 0:
    # Some code
```

You have to still write out `number` the second time, python doesn't remember it.
This code would **not** work:

```
if number < 100 and > 0:
    # Some code
```

However this would, as it's one expression:

```
if 0 < number < 100:
    # Some code
```

This last part is a python specific feature, very few other languages do this.

Now suppose you want to have something else run if the condition isn't met, in this case we use the `else` keyword like this:

```
if age > 18 and country_of_residence != "USA":
    print("You can (legally) drink ethanol")
else:
    print("You probably shouldn't drink ethanol")
```

You'll notice that the `else:` block doesn't take a condition, because it acts as a 'catch all' type of case, it's the default case.
If you wanted to have something run if the initial condition was false,

but not in the catch-all case you would use `elif` (short for else if), which allows you to pass another condition, like this:

```
if number < 100 and number > 0:
    print("Number less than 100 but greater than 0")
elif number > 100 and number < 200:
    print("First if evaluated to false, ran this if as
number is greater than 100 and less than 200")
else:
    print("Number could be anything more than 200 or less
than 0, this is the default case")
```

`elif` runs only if all of the statements above evaluated to false.

> **Tip: Using else for error handling**
>
> One way to handle errors to cover everything in `elif` blocks and leave `else` behind to catch anything you may have missed, rather than using it to catch wanted data. However don't over rely on this, as I later discuss when I talk about inversion to avoid nesting.

Lastly for this if block, using `not` negates the boolean, so for example:

```
if age > 18 and not country_of_residence == "USA":
    print("You can (legally) drink ethanol")
else:
    print("You probably shouldn't drink ethanol")
```

This would do the same thing that the code previously did, but using `not`.
This is a worse way of doing it in this case, and is less Pythonic, however in some cases it might be very useful.

Just a quick tip, if you have an identity check (`if a is b`) you should negate it with `if a is not b` not `if not a is b`.
Example:

```
if getkey(user) is not None:
    return
```

Walrus operator

Suppose you wanted to write this:

```
line = input("enter string: ")
if line == "quit":
    break
```

But this feels a bit unnecessarily verbose, so there's actually a shorthand to assign and immediately use. This is using the `:=` operator, and it's used like this:

```
if (line := input("enter string: ") != "quit"):
    exit()
```

(It's called this because it kind of looks like a walrus' face)

6.2.2 Using `match/case`

If you have a lot of conditions to check (upwards of 3) it may be impractical to use a really long chain of if/elif/elif/elif, where a much nicer way to do that is to the `match/case` syntax, which is basically a switch block from other languages (including OCR). Here's how it looks:

```
def match_grade(score: int) -> str:
    match score:
        case 9:
            return 'A'
        case 8:
            return 'A'
        case 7:
            return 'A'
        case 6:
            return 'B'
        case 5:
            return 'B'
        case 4:
            return 'C'
        case _:
            return 'F'
```

You pass in the thing you're matching after the `match` keyword, and then you specify the cases after case.
The _ is the default case which is ran if nothing else matches.

You can also use the bitwise operators here, cleaning the above code up a bit:

```
def match_grade(score: int) -> str:
    match score:
        case 9 | 8 | 7:
            return 'A'
        case 6 | 5:
            return 'B'
        case 4:
            return 'C'
        case _:
            return 'F'
```

(page over)

And you can also use the `if` keyword in a match statement, like so:

```
def match_grade(score: int) -> int:
    match score:
        case s if s >= 90:
            return 9
        case s if s >= 80:
            return 8
        case s if s >= 70:
            return 7
        case s if s >= 60:
            return 6
        case s if s >= 50:
            return 5
        case _:
            return 0
```

This looks a little strange, but translated this '`s if s >= ?`' really means "match anything and bind it to s, but only if s is greater than or equal to ?".
You can't directly write `case >= ?` because `match case` matches patterns, not arbitrary values.

6.3 Iteration

This means looping, and repeating certain parts of your code, often with some mutation in each iteration.

Python has two kinds of loops: `for` and `while`[1]:

Let's explore each of these:

6.3.1 For loops

For loops are made to iterate over an iterable.
An iterable is anything that can give out its members one at a time.
To iterate over an iterable we use this syntax:

```
example_list = [1,2,3]
for v in example_list:
    print(v)
```

[1]Well actually there's another kind that is universal in almost all languages which is called *recursion*, which is when a function calls itself. I go more in-depth on recursion later on

Here it prints the value.

To instead iterate over a range of numbers – in effect creating a count controlled loop, you can use this:

```
for i in range(10):
    print(i)
```

This makes an iterable that ranges from 0 to 9.
`range()` also takes optional arguments, so here it takes a start index as well:

```
for i in range(3, 10):
    print(i)
```

This prints the values from 3 to 10 (not including 10).
Then to include a step size you can use this:

```
for i in range(3, 10, 2):
    print(i)
```

This counts in twos, so the output from this program would be:

```
3
5
7
9
```

A common use case for the range function would be to iterate over a list by index, like this:

```
example_list = [2,4,6]
for i in range(len(example_list)):
    print(example_list[i])
```

But python has a much better way of doing this, with the `enumerate` function:

```
example_list = [2,4,6]
for i, v in enumerate(example_list):
    print(f"Index: {i}, Value: {v}")
```

Suppose you now wanted the number that the value is in the list, you might want to try do this:

```
example_list = [2,4,6]
for i, v in enumerate(example_list):
    print(f"Number: {i + 1}, Value: {v}")
```

Which gives this:

```
Number: 1, Value: 2
Number: 2, Value: 4
Number: 3, Value: 6
```

Luckily this isn't necessary, and you can actually just pass in the start index into the `enumerate` function like this:

```
example_list = [2,4,6]
for i, v in enumerate(example_list, 1):
    print(f"Number: {i}, Value: {v}")
```

And if you wanted to be extra legible and expressive you can even use a 'key-word argument' (more on them later), like this:

```
enumerate(example_list, start=1):
```

Another helpful function for loops is `zip()`:

```
list_one = [1,2,3]
list_two = [4,5,6]
for v1, v2 in zip(list_one, list_two):
   print(f"v1: {v1}, v2: {v2}")
```

Here we iterate over two lists at the same time, printing their values on each iteration.
If your iterables aren't of the same length, the loop terminates when the shortest of the iterables has finished.

6.3.1.1 Comprehensions

This is a python specific feature (and also seen in functional languages) that allows you to express iteration over an iterable, making a new iterable in a much more concise manner.
Here's how to use them, and a side-to-side comparison of using the standard syntax and a comprehension:
Standard:

```
squares = []
for i in range(10):
    squares[i] = i*i
```

but using a comprehension:

```
squares = [i * i for i in range(10)]
```

6.3.2 While loops

This loop continues looping until the condition specified is True. For example this loop continues forever:

```
while True:
    print("Still true")
```

But this one will loop the user types `n`:

```
running = True
while running:
    _input = input("Do you want to continue? [Y/n]
").lower()
    if _input == 'n':
        running = False
    else:
        print("Answer not 'n', continuing")
```

You can also evaluate statements that return a boolean in the `while` loop:

```
lives = 5
health = 100
while lives > 0:
    print("Continue playing")
    if not health:
        lives -= 1
```

6.3.2.1 Keywords for `while` loops

There are two keywords you need to know for while loops:
`break` and `continue`
`break` breaks you from the loop, no matter the condition
`continue` skips the rest of the loop below and continues onto the next iteration

6.4 Summary exercises

6.4.1 Ex1

Odd or even?

Get an input from a user and print whether that number is even or odd. (Remember to use `int(input("enter int"))` to make it an integer!)

6.4.2 Ex2

Multiplication table

Ask a user for a number and print the multiplication table for that number from 1 to 12

6.4.3 Ex3

Countdown

Create a countdown, where instead of the normal:

```
for i in range(1, 11):
    print(i)
```

you make it count in reverse. You might have to think a bit more for this one, and remember that there are multiple ways to loop.

6.4.4 Ex4

Prime checker

Given a positive number print whether or not the number is prime.
Think about how you (as a person) normally check if a number is prime. Apply similar logic.

6.4.5 Ex5

Word counter

Given an input, perhaps even multiple lines (although this is harder as input returns on \n), count the number of words in the file. Remember that whitespace can be \t, \n, \r, and ‘ ‘.

6.4.6 Ex6

Character replace

Read in a string and then replace all whitespace with underscores (_) , and all vowels with asterisks (*)

6.4.7 Ex7

Isogram checker

An isogram is a word or phrase in which no letters repeat, for example ‘uncopyrightable’ and ‘subdermatoglyphic’ are isograms whereas ‘eleven’ isn’t (the ‘e’ repeats). Write a program that takes an input and prints whether or not it is an isogram.

6.4.7.1 Ex8

Collapse an matrix

Suppose you have a spreadsheet which you read in as a 2d array (list of lists) that has records of employee bonuses every month. It may look something like this:

```
   bonuses = [
       ["James", 2000],
       ["James", 2200],
       ["James", 1800],
       ["Jill", 4005],
       ["Jill", 4005],
       ["Jane", 6900],
       ["James", 1148],
]
```

And you want to collapse the table so that each person only shows up once with a cumulative bonus, so it looks like this:

```
   bonuses = [
       ["Jill", 8010],
       ["Jane", 6900],
       ["James", 7148],
]
```

Be aware that lists in python are mutable, and because of how they work you need to make sure that you either use the same table all the way through, or you make a completely new list that doesn't point to the same memory address.
This is quite a hard exercise to do manually (not using fancy python tricks with default dictionaries or the pandas library, which we haven't covered so don't use them), so try to be creative about it and break it down into multiple parts. There are many ways of doing this, try to find one that works.
Good luck.

Chapter 7. In-built functions

Let's quickly go over how to use a function:

```
  value = cool_function_here(6, 7) # <-- a function *call*
# ^      |__________________|______|
# |      | function name    |^  ^
# |       ----------------| ||  |
# ---captured return value| |'6' and '7' are arguments
#                         | | supplied in parentheses
```

This is a function 'call'. A function call has a name[2], followed by parentheses, which can have arguments in them. The arguments are the real values you give the function when you call it.
Here `value` is the thing returned by `cool_function_here` which is taken into a variable. Not all functions return values. I cover how to write your own functions in the next chapter, here I just show python's in-built functions that do useful things.

Python has a number of handy, in built functions – which I will quickly go over. You have probably used some of these before.

- `print`

Function: Prints something to the console
Usage: Pass the thing you want to print in parentheses as an argument:
`print("Hello")`

- `input`

Function: Take input from standard input and returns it, and can be used to put into a variable, but also as a parameter for another function.
Usage: `var = input()`. Here `var` was the variable but you can put anything. It can be passed as a parameter like this:

```
>>> print(input("Please input something: "))
Please input something: 3456
3456
```

[2]Well actually not all functions have names, anonymous functions called lambda functions don't, but that's covered in chatper 14

Here it was passed to `print`.
Note that the input function takes input as a string unless you cast it otherwise.

Also you can put a string to be printed to standard output as an argument to the input function in parentheses.

- `len`

Function: Returns the length of the thing that was passed into it
Usage: `len(item)`

- `sum`

Function: Returns the sum of an iterable
Usage: `sum(iterable)`

- `abs`

Function: Return the absolute value given (always positive, think of this as distance from 0)

- `round`

Function: Rounds value passed to it to the degree of accuracy specified
Usage: `round(1.23456, 2)  # Returns 1.23`

- `range`, `enumerate` and `zip`

Covered in my control flow section.

- `max`/`min`

Function: Return the largest or smallest value of an iterable or a comma separated list given
Usage:

```
max([1,4,2,5])  # Returns 5
min(0, 4, 8, 10)  # Returns 0
```

- `sorted`

Function: Take any iterable and return a sorted list
Usage: `sorted({5, 2, 7, 8})  # Returns [2, 5, 7, 8]` Optionally it takes `reverse = True` to reverse the returned list.

- `all`/`any`

Function: Check if all or any (respectively) values are true/truthy. Returns a boolean.
Usage: `any(False, '')  # -> False`

- `eval`

Function: Take a string and evaluate as a python expression.
Usage: `eval("print(\"Hello reader!\")")  # Prints "Hello reader!"`

- `map`

Function: Takes in a function and applies it to every item in an iterable.
Usage: `map(function, iterable)`

- `filter`

Function: Filter an iterable based on the given function, then return a generator. Look this up for more details.

- `ord` and `chr`

Function: Convert a character into a Unicode code, and a Unicode code number into a character.
Usage: `ord('A')` is 65, and `chr(65)` is 'A'. This is helpful for doing some fancy string operations where you need the letters to be treated as numbers to be able to use number operations on them.

- `str()`, `int()`, `bool()`, `float()`, `set()`, `list()`, `dict()`, `tuple()`

Function: Convert the passed value to that type. This is known as casting.
Usage: `int(my_var)`
An example use case for this is to combine with `input` as the default case is always a string.
`integer = int(input("Please enter an integer: "))`
Using this:
`my_var = int(my_var)` Which can be used to convert a type of a variable, is known as *shadowing*. This is the practice of making a variable with the name of an existing variable, masking the older variable with the new one.

- `help`

Function: Provide documentation for the thing passed to it
Usage: `help(thing)`
This is super useful for providing help.
Here's something you should know however:

```
>>> help(str)
```

Try running this! You will get something that starts with:

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If
encoding or
 |  errors is specified, then the object must expose a data
buffer
 |  that will be decoded using the given encoding and error
handler.
 |  Otherwise, returns the result of object.__str__() (if
defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as
described by format_spec.
```

This makes more sense after I cover Object Oriented programming in python in a later chapter, but you'll notice the class, and how a bunch

of dunder methods are defined here, some which you may have used, such as `__format__` used as `.format`.

7.1 Summary exercises

7.1.1 Ex1

Word counter (again)

Revisit the word counter program we had in the last chapter. Try to make it more efficient by using the `ord()`, `chr()` and `sum()` functions.

7.1.2 Ex2

Using `chr()` and `ord()` more, Caesar cipher encryption

You may have heard of this method of very old and ineffective encryption known as the Caesar cipher, where each letter is shifted a certain amount in the alphabet forward/backward, and anything too far wraps. For example `zig` shifted forward by 2 becomes `bki`, or
`Python is interpreted!`
shifted by 13 becomes:
`Clguba vf vagrecergrq!`
Notice how the `!` is unaffected.

Chapter 8. Functions and modular code

As seen in the previous chapter, python has in built functions, but we can also create our own.
A function is a self-contained block of code that do a specific task. It returns a value.

Functions in python can take any number of input parameters, and return any number of values.

8.1 Defining functions

You define a function with the `def` keyword, like this:

```
def my_function():
    print("I'm in a function!")
```

By default if there is no `return` statement at the end of a function it implicitly returns the `None` type, if you want to be extra explicit you can use a bare `return` with no arguments. You use `return` to exit the function (as the name implies), meaning you often use it where otherwise you might want a 'break'. This is helpful for keeping your code tidy as you don't need to have more if checks. I talk more about 'early returns' in chapter 9.

As we've seen before, python expects indented code after the `:`, and if you want to keep your function, you can use the `pass` keyword to have no other code there, like this:

```
def todo_function():
    pass
```

But this isn't a great idea as if you call `todo_function` here, you won't get anything, so the error passes silently. A better alternative is:

```
def todo_function():
    raise NotImplementedError("here i explain why i didn't
implement this yet, or what needs to be done")
```

This means the program raises this error and crashes if you try to call this function.

8.2 Scope

All languages have this as an concept, and it tells you where a variable or object is valid for, and can be used. Previously we've defined all our objects and variables and code in something known as `global` scope, meaning it is valid everywhere in the script. When you define a function you create a `local` scope, meaning something that you create in the function doesn't exist outside the function as you destroy the scope and any objects when the function returns.
Local scope takes precedence over global, or nonlocal scope. Now let's demonstrate all this:

```
x = 21
def func():
    print(x)
```

This will print 21.

```
x = 21
def func():
    print(x)
    x = 0
    print(x)
```

This is invalid and will fail as python thinks you are trying to change a global variable. Here:

```
x = 21
def func():
    global x
    print(x)
    x = 0
    print(x)

print(x)
```

This will print 21, 0, 0 as you modified the global variable by using the `global` keyword.
Now let's look at local variables:

```
x = 21
def func():
    x = 0
    print(x)
print(x)
```

This will print 0, 21 as you made a local variable with the same name as a global variable, but didn't change the global variable.
But what if you have nested functions? This is known as `external scope` and you need to use the `nonlocal` keyword to access variables that are external to the scope but not global. Otherwise this works the same way that global does:

```
def func():
    x = 21
    def inner_func():
        nonlocal x
        print(x)
```

This doesn't work the same way with mutable objects, which behave a bit different as python's variables all hold memory addresses not values. You'll see how this works shortly.

8.3 Arguments and parameters

To avoid this annoying scope idea, we use parameters in our functions, and then when we call them we supply arguments. A parameter is like a placeholder variable that you use in the function, which isn't actually real until you call it with an argument. The argument is the real value that you give to the function, that replaces the parameter with a value. When you define a function, you write in these placeholder parameters like so, and place them in parentheses like this.

```
def greeting(name):
    print(f"Hello, {name}")
```

Then later in your code you need to call this function like so:
`greeting(John)`
Here `John` was passed in as the name, and the function is called with parameter John taking the place of the argument `name`.

Note: *Argument* vs *Parameter*

A parameter is like a placeholder you put in your function when you define it.
An argument is the actual value you give the function when you call it.

8.3.1 Default values

If you want a value to be optional to a function, you can specify a default value, like so:

```
def greeting(name = 'John'):
    print(f"Hello, {name}")
```

This way if we just call `greeting()` we'll get "Hello, John" printed to the screen. Obviously this is kinda weird and doesn't make sense, but it does have its use cases, like for when you are designing a function that has extra functionality, so you can pass in `None` as a default value if it's not given.

8.3.2 Unpacking values and `*args` and `**kwargs`

Suppose you wanted to pass a list as values to a function:

```
def print_items(a, b, c, d):
    print(a)
    print(b)
    print(c)
    print(d)

list = [1, 2, 3, 4]
print_items(list[0], list[1], list[2], list[3])
```

We can also do this with star unpacking like this:
`print_items(*list)`
We can denote any number of normal arguments to a function like this in the declaration:
`def print_items(*args)`
Where `args` becomes a tuple of arguments.

We can place any 'normal' arguments before the `*args` like so:
`def print_items(p1, *args)`

8.3.2.1 Keyword arguments

You can pass in arguments by name rather than position like so:

`print_items(1, 2, 3, p2 = 4)`
So long as the function has p2 in the declaration like so:
`def print_items(p1, *args, p2 = 4)`

> **Argument order**
>
> Keyword arguments must come after any positional arguments, so this wouldn't work:
> `def print_items(p2 = 4, p1, *args)`

You can unpack a dictionary into keyword arguments with the double star notation like this:

```
def print_a_b(a, b):
    print(f"a: {a}")
    print(f"b: {b}")

d = { "b" : 1, "a" : 0}
print_a_b(**d)
```

Which means the function call gets unpacked into:
`print_a_b(a = 0, b = 1)`

You can also do the reverse of this by specifying that you need keyword arguments in the function declaration like this:
`def print_items(**kwargs)`
Which gives you a dictionary of items mapping the keyword to the argument, so you can access the values by the key.

8.4 Returning values

When you want to give back anything you did inside the function, you use the `return` keyword like this and it gives you a value when you call the function, like this:

```
def add(a, b):
    sum = a + b
    return sum
```

We could have also written the return statement like this:
`return a + b`
To return multiple values, separate them with a comma like this:

```
def divide(dividend, divisor):
    result = base // divisor
    remainder = base % divisor
    return result, remainder
```

Under the hood python actually returns a tuple of these values, like this: `return (return, remainder)`
Then when you call the function, you can either store the returned values in variables like this:

```
result, remainder = divide(65, 7)
```

Here, we unpacked the tuple.
Note that the variable names don't have to be the same as the return names.

You can also pass the returned values to a function like this:

```
print("The result of the division was: ", divide(65, 7)
```

This returns a tuple with the two values. If you wanted them separately, you would have to pass them into variables to unpack the tuple, as shown previously.

Another example of passing the returned value to a function:

```
print(f"6 + 7 is: {add(6, 7)}, and that's the total!")
```

Note: Python uses *Pass by Object Reference*

What this means is that if the object you give the function is mutable, any change you make to said object in the function is reflected outside the function as well. This is because they actually point to the same object in memory.
Knowing this, you frequently don't return lists from functions, rather an input and an output list are given to the function, and the function mutates the output list.

8.5 Yield statements

Instead of a return statement, you can also have `yield` statements in a function, which turns your function into a generator, which returns a generator object. A generator is like an iterable, which is something able to give out its elements one at a time, lazily evaluated. It's kind of like a list, and this way you can have some features of lazy evaluation such as infinite lists, as they are only evaluated when they are needed.

Yield saves the state of the function after every yield, giving out its values over time, not evaluating everything at once like with `return`.

You iterate over it either with a `for` loop like with a list, or with the `next()` function.

Under the hood yield implements the iterator protocol, but the way this works is beyond the scope of this booklet.

Let's look at an example to better understand how this works:

```
def count_to_n(n: int):
    for i in range(n):
        yield i

# Using the generator
gen = count_to_n(5)  # Creates a generator object
print(next(gen))  # Output: 0
print(next(gen))  # Output: 1
```

```
print(next(gen))  # Output: 2

# Or iterate with a for loop
for num in count_to_n(5):
    print(num)  # Output: 0, 1, 2, 3, 4
```

While this is quite artificial you get the point of how the syntax works:

- You first get an object (a generator object) when you call the function, and then you perform operations with this generator.

8.5.1 Using send to make the generator receive values as well

Using `send()` you can also get values from the generator caller like so:

```
def counter():
    count = 0
    while True:
        received = yield count
        if received is not None:
            count = received  # Update count with sent value
        else:
            count += 1

gen = counter()
print(next(gen))  # Output: 0 (starts the generator)
print(gen.send(10))  # Output: 10 (sends 10, yields 10)
print(next(gen))  # Output: 11 (increments from 10)
```

8.5.2 Using a comprehension like syntax for concise generators

You can also get a simple generator without having to define a function like so:

```
gen = (x**2 for x in range(5))
print(list(gen))  # Output: [0, 1, 4, 9, 16]
```

The syntax is the same as for comprehensions just using parentheses as opposed to square brackets.

8.6 Function wrappers and decorators

Wait, what's a function wrapper?

A function wrapper is function that adds or or modifies the functionality (behaviour) or another function without modifying the code of the original function.

You might use a wrapper if your function had a prerequisite that needs

to be fulfilled for the function to work, or if there's some block of code you find repeated in multiple functions.
Here's a primitive example to illustrate how they work, first here's the non-wrapper example:

```
def sign_in():
    username = input("Enter username: ")
    password = input("Enter password: ")
    print(f"[SIGN IN] Attempting to sign in user:
{username}")
    if username == "admin" and password == "secret":
        print("Sign in successful!")
    else:
        print("Invalid credentials.")


def sign_up():
    username = input("Enter username: ")
    # password = input("Enter password: ")
    print(f"[SIGN UP] Creating account for user:
{username}")
    print("Account created successfully!")
```

And now here's the same logic using a function wrapper:

```
def require_credentials(func):
    def wrapper(*args, **kwargs):
        username = input("Enter username: ")
        password = input("Enter password: ")
        # Call the original function with username and
password
        return func(username, password, *args, **kwargs)

    return wrapper


def sign_in(username, password):
    print(f"[SIGN IN] Attempting to sign in user:
{username}")
    if username == "admin" and password == "secret":
        print("Sign in successful!")
    else:
        print("Invalid credentials.")
```

```
def sign_up(username, password):
    # Example sign up logic
    print(f"[SIGN UP] Creating account for user:
{username}")
    print("Account created successfully!")


if __name__ == "__main__":
    while True:
        choice = input("Do you want to (1) sign in or (2)
sign up? (q to quit): ")
        if choice == "1":
            wrapped_func = require_credentials(sign_in)
            wrapped_func()
        elif choice == "2":
            wrapped_func = require_credentials(sign_up)
            wrapped_func()
        elif choice.lower() == "q":
            break
        else:
            print("Invalid choice, try again.")
```

And looking at this it's kinda tedious because every time you want to use the functions sign_in or sign_up you have to call the wrapper, and use the returned wrapped function. Luckily we don't actually have to code like this because we have decorators: which are syntactic sugar for function wrappers which make it way nicer to use them. Here's the same example using decorators:

```
def require_credentials(func):
    def wrapper(*args, **kwargs):
        username = input("Enter username: ")
        password = input("Enter password: ")
        # Call the original function with username and
password
        return func(username, password, *args, **kwargs)
    return wrapper

@require_credentials
def sign_in(username, password):
    # Example sign in logic
    print(f"[SIGN IN] Attempting to sign in user:
```

```
{username}")
    # Here you'd check against stored credentials, etc.
    if username == "admin" and password == "secret":
        print("Sign in successful!")
    else:
        print("Invalid credentials.")

@require_credentials
def sign_up(username, password):
    print(f"[SIGN UP] Creating account for user:
{username}")
    print("Account created successfully!")

# Example usage
if __name__ == "__main__":
    while True:
        choice = input("Do you want to (1) sign in or (2)
sign up? (q to quit): ")
        if choice == "1":
            sign_in()
        elif choice == "2":
            sign_up()
        elif choice.lower() == "q":
            break
        else:
            print("Invalid choice, try again.")
```

This `@func` syntax does the same as the above, just in a much neater way. You'll see decorators elsewhere in python, for example static methods or dataclasses, but now you can use your own!

8.7 Recursion (part one)

Recursion is when you have a function that calls itself - which in effect creates a loop. It comes mostly from the functional programming paradigm, and isn't often seen in what we've seen so far, the procedural, imperative and maybe OOP style of code.
Here's an example (although this is still the procedural paradigm):
Suppose you have a problem where you need a user to enter a number between 1 and 20, and get them to keep entering a number until it is in the valid range.
You could do it like this:

```
def enter_number() -> int:
    print("Please enter a number between 1 and 20")
    while True:
        number = int(input())
        if 1 <= number <= 20:
            print("Number in range, continue")
            return number
        else:
            print("Please enter a valid number between 1 and
20")
```

Firstly note that the `break` is implied here, since you have the return statement.
While this is a perfectly valid way of achieving this, we can leverage functions calling themselves - which is called recursion. Here's the same code but using recursion:

```
def enter_number() -> int:
    number = int(input("Please enter a number between 1 and
20: "))
    if 1 <= number <= 20:
        print("Number in range, continue")
        return number
    else:
        print("Number invalid")
        return enter_number()
```

Recursion is less safe to use over traditional loops, because every new function you call, you don't terminate the previous function, you just put a new one on the call stack. This could potentially cause a call stack (stack) overflow.
Recursion is very much linked to the functional paradigm, and it doesn't make much sense to go further in it here, so I go over it in much more detail in my Functional Programming in Python chapter. (link here)

8.8 Summary exercises

8.8.1 Ex1

Returning values (syntax check)

Write a really simple function: `doubleit(x: int) -> int` that literally just returns the doubled number. This is so you can play around with understanding what names have the be the same, and how return values work.

8.8.2 Ex2

Returning values: grade checker

Now for the juicy functions that you have the basic syntax down. Create a program that will do the following:

1. Collect a student's name
2. Collect the student's computer science teacher's name
3. Collect marks for the last three homework assignments
4. Calculate the average out of them (but not print it)
5. Print a corresponding message depending on what the mark was:
   If the mark was ≥ 8, print this:

   ```
   f"Well done, {student_name}, {teacher_name} is very
   pleased with your effort"
   ```

   If $6 <$ mark ≤ 8 print this:

   ```
   f"A good effort, {student_name}."
   " However {teacher_name} thinks you should check your
   work more carefully."
   ```

   Lastly if the mark was ≤ 5, print this:

   ```
   f"{student_name} this is poor. "
   "{teacher_name} has asked you to try harder."
   ```

Tip: Importing functions

For basic things (like means) there's often an existing function for it in python, either in-built or you can import it.
For this case, there's a `mean` function, but you have to import it with
`from statistics import mean`
at the top of your document, which you can use for the above exercise.

8.8.3 Ex4

Collatz sequence

Write the Collatz conjecture sequence as a function in python. If you don't know what that is (firstly watch the Veritasium video on it), here's how it works:

- You take a number (any number)
- If it's even you divide it by two
- If it's odd you multiply it by 3 and add 1

So far what all tested numbers, though not proven, will end up at 1, which is in a loop:

$$1 \times 3 = 4$$
$$\frac{4}{2} = 2$$
$$\frac{2}{2} = 1$$

Make a function that given any number will follow the chain and then return the number of steps it took to get to 1.

8.8.4 Ex5

Digit sum

Given any integer, sum its digits. So 69420 will be $6 + 9 + 4 + 2 + 0$ which is 21.
This is trivial, but there's a fancy way of doing it so it's a one-liner. If you can do that try for a root sum, where you keep summing until there's only one digit, so $6 + 9 + 4 + 2 + 0 = 21$ and $2 + 1 = 3$

8.8.5 Ex6

Collatz sequence (with yield)

Take the previous collatz function that you wrote and turn it into a generator that yields until 1.

8.8.6 Ex7

Fibonacci generator

Write a generator function with yield that produces and infinite Fibonacci sequence.

Chapter 9. Better code structure

Up until here, I've just given you a bunch of tools and how to use them, but not suggested when to use them.
Here's just some basic advice on how you should be writing your code:

- Use functions for almost everything

This is good practice - use functions for everything that happens, each function should ideally be no more than ≈ 60 lines long for readability. If you are feeling the need to go over that you should break it up into smaller functions. pep8, the official python style guide, states that you shouldn't have cyclomatic complexity greater than 15 in your functions.

- Always use type hinting

This makes your code more readable, clear, and your editor can help you debug it easier. Always use them.

- Try to follow pep8

This is the styling and writing guide for python. Read through it and try to stick to it.

- Setting up your file structure:

Here's generally how I set my file structure up, and it makes more sense why later:

```
# Your imports/dependencies

# Your functions

def main() -> None:
    # Call your functions here

if __name__ == "__main__":
    main()
```

Basically have a main function that you call in the if block at the end.

This is a `dunder` method that checks you are running the file directly, not importing it as a package/module. Again explained fully in the other files chapter.

In a large number of cases you won't actually want a `main` function you call, but often something else you want to call to start execution, for example the init from a class.

9.1 Writing comments

You may think that as your programming skills grow and the length of your programs increase, you should place more and more comments in your code to explain what it does, so that anyone reading it will more easily understand it.
While this does have an element of truth, I'd like to present the more controversial view that you probably shouldn't be using comments in your code in most cases.

9.1.1 Replacing comments with writing code

Here's the argument, which I touched on in chapter 3: If your code isn't obvious, you should endeavour to make the code more easily readable and comprehensive rather than trying to use natural language to explain it.
Here I'll explain how you might do this with some examples:
This is the example I'll use for a couple points:

```
# function returns the circumference of a circle
def function(a):
    # a is the diameter taken into the function
    # 3.141 is pi, which we multiply the diameter to get the
circumference
    return a * 3.141
```

0. Using descriptive names
   This might be really really obvious but using good, descriptive, clear and mostly short names for functions variables and classes is really important to keep your code easy to read and use. Here we had to explain what the function did and what the parameter was. Instead if we had written this:

```
def find_circumference(diameter):
    # 3.141 is pi, which we multiply the diameter to get the
circumference
    return diameter * 3.141
```

Now we don't need the two extra comments.

1. Using variables to avoid magic numbers
   A magic number is a number that isn't clear what is to the reader. In the above function, there is this 3.141 which is kinda just floating around, while this is rather simplistic and we can all recognise it as π it still may be unclear, and makes the code harder to follow.

To make this better, we can define it as a constant:

```
def find_circumference(diameter):
    PI = 3.141
    # 3.141 is pi, which we multiply the diameter to get the
circumference
    return diameter * PI
```

Now we can remove the comment as it reads the same as the constant PI. Or, even better if we import it from math:

```
import math
def find_circumference(diameter):
    return diameter * math.pi
```

This was the point I made in chapter 3 when I introduced variables.

2. Using types instead of comments
   Suppose we had this function here:

```
    # Function looks for a user and returns it if the user
    # is in the users dictionary of username: user,
otherwise returns None
    def find_user(self, username):
        return self.users.get(username)
```

But we can actually replace the comment with a type annotation:

```
    def find_user(self, username: str) -> User | None:
        return self.users.get(username)
```

And it says the exact same thing as the comment, just that now you don't have the problems with the comments.

As an additional benefit your code editor probably also now can tell you more about this function.

9.1.2 Problems with comments

Comments get bugs, just like code. Often this happens when you update the code but not the comment, and you get comments like this:

```
# Verify the users password is at least 8 characters long
if len(password) < 10:
    return False
```

The thing is, we have tests for code, and the syntax and sometimes semantics are enforced by the translator, whereas comments don't have this.
This means that often when you read a program's code to figure out how it works, it often makes most sense to read just the code, not the comments, as comments lie.

Equally there's often no need for comments that describe what the code does about 99% of the time. This is because code is self-descriptive, so you have code the explains what it does, and you know what it does just by reading it, if you know the language.

9.1.3 When to actually write comments

You should write comments if the code does something non-obvious because of performance reasons, (or any reason good enough to grant this), you should explain the code if you judge that the reader may fail to do so themselves.
The other reason to write comments is to explain *why* a piece of code is there, not *how* it works. A reader will be able to understand how it works just by reading it, but might not know why it's there. This is when you may consider leaving comments in your code.
Lastly if you used a function from somewhere, like if you imported it, you may consider writing a comment linking to it.

9.2 Code documentation

This is different from code comments, as it describes the high-level architecture of the code, rather than the inner working of how the code works. Explains how to use the code. Luckily the documentation

for code can be written with the code, and then using tools like doxygen or pydoc you can automatically generate documentation.

9.2.1 Python specifics

Python has its own specific way of documenting code, so I'll go over the way to do it here:

9.2.1.1 Functions

Functions in python use docstrings (triple quotes) right under the definition like this:

Unfinished

This part here is unfinished, at some point I'll come back and add this content.

9.3 Never nesting

Kind of a misnomer here, but basically the idea is that you don't nest your code more than 3 times — i.e your indents aren't more than 3. This is generally a neater way to read and write code, and you have two tools at your disposal to de-nest your code:

- Extraction

This is when you take part of your function/method/class and make it its own thing, and in your main code you call the function/method.

- Inversion

This is where you make the 'unhappy' part of your code first into the ifs, and then leave the happy to the else block. This means you can actually omit else and let it run as the default case. Here's a nonexample example:
This is the first type of code:

```
def func():
    if condition_a:
        print("Check passed")
        if condition_b:
            print("Check two passed")
        else:
            print("Condition b failed")
```

```
else:
    print("Condition a failed")
```

And here's an inverted logic, removing the unnecessary else statements:

```
def func():
    if not condition_a:
        print("Condition a failed")
        break  # Alternativly exit()
    print("Check passed")
    if not condition_b:
        print("Condition b failed")
        break
    print("Check two passed")
```

Often this is done in a loop, where if the check fails you want to skip the happy part of the code and run the checks again. Here use `continue` to achieve this.

```
def func():
    while True:
        if not condition_a:
            print("Condition a failed")
            continue
        print("Check passed")
        if not condition_b:
            print("Condition b failed")
            continue
        print("Check two passed")
        break  # Since both checks passed we break out of
the loop
```

And if you want a count controlled loop to 'redo' this iteration, use a while loop again, with a manual counter, like this:

```
def func():
    i = 0
    while i < 5:
        if not condition_a:
            print("Condition a failed")
            continue
        print("Check passed")
        if not condition_b:
            print("Condition b failed")
            continue
        print("Both checks passed")
        i += 1
```

9.4 Summary exercises

Unfinished

I still have yet to add the exercises here, at some point I will update this.

Chapter 10. Error handling

In most high-level languages[3], there are several kinds of errors:

- Syntax errors:
  Syntax is the rules of a language, so a syntax error simply means you didn't write the code correctly – in the sense that you violated rules that python has to follow.
  For example you missed a `:` after a scope definition like an `if` statement, or you missed a parenthesis or made a spelling error. Here you broke the grammar rules In this case Cpython (the interpreter) will point to you where the error was caught in compiling.

  > **Note: How is python translated?**
  >
  > So far I've told you that python is interpreted, which is true - you don't compile python directly to machine code.
  > However, python is actually compiled to *bytecode* which is then executed by the python virtual machine.

- Logical/Semantic:
  This is where the code *could* be correct as grammatically it is, but it either causes a runtime crash, or is caught by python or via exceptions (explained below). Here's an example of code that runs but gives an unexpected output:

```
def average_pair(a: int , b: int) -> float:
    return a + b / 2
```

Here when the lexical tokens (tokens) get parsed by the language compiler, the precedence of `/` is greater than that of `+` due to the order of operations, so it is read in like this:
`return a + (b / 2)`
Which is not the intention, as it yields incorrect results. What we wanted is this: `return (a + b) / 2`, so we have to manually put this

[3]Here I mean imperative Object Oriented high-level languages, like C++ and Java, but excluding things like Haskell (functional) or Zig (Zig has its own error system)

in.
None of these violated the rules or grammar of the language, but were nonetheless incorrect. I cover logical errors and testing more later in in chapter 15: Writing tests.
Earlier I mentioned that a spelling error is a syntax error, but often it is actually a semantic error. Here's an example:

```
# Print a greeting to the reader
pront("Greetings, reader!")
```

Clearly there is a spelling error, but this is actually a logical/semantic error as this *could* be right, but the language parser doesn't find a function called `pront` so fails and gives us this crash:

```
~/code/python via  v3.13.9 took 8s
❯ python test-errors.py
Traceback (most recent call last):
  File "/home/larry/code/python/test-errors.py", line 1, in
<module>
    pront("Greetings, reader!")
    ^^^^^
NameError: name 'pront' is not defined. Did you mean:
'print'?
```

This is different from a syntax error which may be missing a quote pair like this:

```
~/code/python via  v3.13.9 took 9s
❯ python test-errors.py
  File "/home/larry/code/python/test-errors.py", line 1
    print("Greetings!)
          ^
SyntaxError: unterminated string literal (detected at line
1)
```

The reason that the first case could work is because we have a valid function call, just to a function `pront` that doesn't exist. We can define this function like so:

```
def pront(*args):
    print(*args)
```

and now our code runs just fine. I'll prove this further with exceptions later on.

- Exceptions:
  This is a high-level error feature of python, that allows you to catch semantic and logical errors from the language.
  As the python documentation puts it:

> "Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs."
>
> — Python Docs

10.1 Catching exceptions

By default if an exception is *raised*, the program will crash. To avoid this we run them in a `try/except` clause. To catch an error, you need to know what might fail. There are many different types of exceptions, for example: ZeroDivisionError, ValueError, FileNotFoundError, NameError, TypeError and so on. Here's how to catch them with the `try/except` clause:

```
def divide(dividend, divisor):
    try:
        result = dividend // divisor
        remainder = dividend % divisor
        return result, remainder
    except ZeroDivisionError:
        print("Cannot divide by zero")
```

Warning: Do NOT use bare try/except clauses

Suppose you didn't know that the division by zero error was called `ZeroDivisionError`, and you tried to write this instead:

```
def divide(dividend, divisor):
    try:
        result = dividend // divisor
        remainder = dividend % divisor
        return result, remainder
    except:
        print("Cannot divide by zero")
```

While technically this does work, it has a huge problem: It catches everything indiscriminately, which includes bugs and something like a user pressing control-c to terminate the program.
If you still wanted a catch-all solution, you could use this:

```
def divide(dividend, divisor):
    try:
        result = dividend // divisor
        remainder = dividend % divisor
        return result, remainder
    except Exception:
        print("Error occured")
```

And if you wanted to also catch what caused it use this:

```
def divide(dividend, divisor):
    try:
        result = dividend // divisor
        remainder = dividend % divisor
        return result, remainder
    except Exception as err:
        print(f"Error occured: {err}")
```

Then you might want some cleanup in the try/except clause, where you have code to run no matter what error occurred - in actual examples of real code, this is often things like closing off connections to databases and closing files (but for this just use a context manager

(`with`)).
To do this use the `finally` keyword at the end of your clause like this:

```
try:
      # Some code that might fail
except ValueError:
    ...
except NameError:
    ...
except Exception:
      # As many exceptions as you want to try to catch, you
      # can have as many of these blocks as you want
      # 'Exception' is a 'catch all' solution you don't
usually want
finally:
      # A single finally block here
```

10.2 Raising exceptions

However sometimes you also want to deliberately cause an error to crash the program, and then either catch this or let it terminate the code.
For this you use `raise` keyword, like this:

```
def check_age(age:int):
    if age < 0:
        raise ValueError("Age can't be negative!")
```

If you now call this function with a negative value you get this:

```
Traceback (most recent call last):
  File "/home/user/raise.py", line 5, in <module>
    check_age(-1)
    ~~~~~~~~~^^^^
  File "/home/user/raise.py", line 3, in check_age
    raise ValueError("Age can't be negative!")
ValueError: Age can't be negative!
```

This is usually the way to write functions that *can* fail, and then call them in a try/except block, like this:

```
def main():
    try:
```

```
        check_age(-1)
    except ValueError as err:
        print(err)
```

Which just outputs:
`Age can't be negative!`

10.3 Catching typos

As I previously mentioned you can catch spelling mistakes if the python grammar is not the problem. The specific name of for this is 'NameError'. Knowing this let's write a try/except block to catch this:

```
try:
    user_value = imput("Greetings: ")
except NameError as e:
    print(e)
```

Which will output:
`name 'imput' is not defined`
However I would suggest to not just run your whole code inside a massive try/except block to catch typos, it makes more sense to let the runtime crash happen so it tells you exactly what to fix and where. This ends up being more helpful in fixing programs and just generally writing code.

10.4 Custom errors

At some point you may want to write your own errors to be raised and caught. For this you need to define a custom error class, which is part of the object oriented part of python, but that's covered in chapter 13 (link here). For now, just try to follow along:

```
class MyCustomError(Exception):
    def __init__(self, message: str) -> None:
        super().__init__(message)
        self.message = message
```

Here we made a custom class that makes an error object inheriting from the `Exception` class.
This now introduces the `MyCustomError` as a valid error that can be raised. You can supply a message to it as well, which will allow you to catch and do something with it.

You can also create instances of this class by assigning it to variables like this:
`simple_message = MyCustomError("hi, you've raised an exception")` And then raising the error like this:

```
try:
    raise simple_message
except simple_message as e:
    print(e)
```

10.5 Summary exercises

10.5.1 Ex1

Handle erroneous input

Previously we've handled input with `x = int(input('int please'))`, which has a problem: if a user inputs something other than an integer, the program crashes. This time, wrap the input in a try/except block to catch the input, and inside a loop to keep prompting the user for input until they enter a valid input.

Unfinished

I still have yet to add the exercises here, at some point I will update this.

Chapter 11. Using external files

In this chapter we will cover:

- Setting up a python project structure
- Using extra modules and libraries, both from standard and non standard libraries

11.1 Splitting your project across several files

In real python projects, you might have a directory structure that looks like this:

```
python_project
├── README.md
├── __init__.py
├── __main__.py
├── control
│   └── LICENCE
├── pyproject.toml
├── requirements.txt
└── my_library
    ├── package_one
    │   ├── module1.py
    │   └── module2.py
    └── package_two
        ├── module1.py
        └── module2.py
```

Let's talk about what each of these files are:

- `__init__.py`

This just tells python that this is a package, enabling importing of its modules

- `__main__.py`

This is the main entry point of execution of your program, and has a main function, which has the:

```
if __name__ == "__main__":
    main()
```

code block.

To import a package from your library (in the `__main__.py` file), you use this at the top of it:
```
from my_library import package one
```
Then to run a function from the module from the package you would run:
`package_one.module1.my_func()` If you want the whole library you just run:
```
import my_library
```
or if you just want a single function from a module you can use this:
```
from my_library.package_one.module1 import my_func
```
Where you can just call the function as normal (as if it were defined in that file).

To go back to the dunder `__name__` variable, it's set by the interpreter depending on how the file was run. If you run a file directly (no matter the name of the file), you will get `__main__` back, however if you run it by importing it from another file (as a module or a package), you will get where it was imported from (the file path).

- `requirements.txt`

Often you'll go beyond the standard libraries, so you will need to install some requirements for the project to work. You place them here so that people can automate installing them.

- `package_one`

A package is a directory which is a collection of modules which all relate to the same thing.

- `package_one/module1.py`

A module is a single file.

11.2 Installing dependencies and using non-standard libraries

When you want to go beyond the standard libraries (like datetime, random etc), you will first need to install them.
The way you would do that is with a package manager - `pip`, which stands for `preferred installer program` and it allows you to install python libraries and packages. It came when you installed python so

you don't need to additionally install it to use it.

11.2.1 Python virtual environments

When you have a project you are working on it's best not to install your dependency packages globally as they may break you system packages, and/or conflict with other projects. The best way to avoid all of this is via virtual environments:
A virtual environment is like a sandboxed directory with its own python installation.
You should install all your python packages there.
Here's how to use virtual environments:
First make sure your source code is all in a single directory with nothing else in it.
Then run this in the project directory:

```
python -m venv env
```

This makes a directory called `env` that houses your environment. To activate it run this:

```
source env/bin/activate
```

Now you are ready to install packages with pip. Installing them with the package manager is really easy, just use `pip install {package name}` to install and `pip uninstall {package name}` to uninstall. If you want he help page just run `pip` with no arguments.
Here's an example:

```
pip install matplotlib
```

installs the `matplotlib` package (for graphing in python).

Then you import it just like a standard library:

```
import matplotlib.pyplot
```

If you don't want to constantly write out `matplotlib.pyplot.graph` you can alias it in the import like this:

```
import matplotlib.pyplot as plt
```

11.3 Summary exercises

Unfinished

I still have yet to add the exercises here, at some point I will update this.

Chapter 12. Reading/writing to/from external files

As of now, you've only stored data in memory[4] in variables, which means that when the program terminates the data is lost. If we want data to be stored persistently, we need to write to external files and save them to the disk, for example, you may want to store logins and usernames or user data in an external file.

Here's how python handles this:

Python as an inbuilt function for this, called `open`. The way this works is it makes a file object that you can use, and operate on. More on objects later though. This allows you to open a file, with a mode.
Here are the valid modes you can use, and what they do:

Mode	Function
'r'	Open for reading (default)
'w'	Open file for writing, truncating the file first (overwrites contents)
'x'	Make a file, throws an exception if the file already exists
'a'	Open for writing, append to file if it exists
'b'	Binary mode
't'	Text mode (default)
'+'	Open for updating (read and write)

```
def return_file(filename: str, mode: str) -> TextIO:
    my_file = open(filename, mode)
    return my_file
```

This gives you the file object.
Then you have the following methods for the file object:

- `.read()` reads in the file into a variable
- `.write('Text to write')` write text to a file

[4]Most often this is actually CPU cache, and when machine code executes on the CPU, parameters from functions are most likely passed in directly into the CPU registers (Memory Data Register in Von Neumann Architecture)

- `.close()` closes the file if you opened it with the `open()` function
- `.readline()` reads a single line from the file
- `.readlines()` reads in the whole file as a list of lines

If you want the file as a list you can also use these two options:

```
for line in f:
    print(line, end='')
```

This works because the file object is an iterable, meaning that you can loop over it with `for` as with any other iterable.

Equally you could just do this to get a list:
`list(f)`

12.1 Context managers, using the `with` keyword

A context manager does the opening and closing for you, which is more convenient. Here's how you use `with`:

```
with open('file') as f:
    data = f.read()
```

The read mode is default, so we didn't need to specify it. And now, you don't have to call `file.close()` as `with` handles it for you.

`with` isn't only used for files, it's used when you need to connect and then close off connections, for example:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

12.2 Using json as a format for storing data

The `.json` format is pretty helpful for reading and writing data to files in a way that a computer program like python can easily read in as a data structure.

Unfinished

This part here is unfinished, at some point I'll come back and add this content. For now, look this up!

12.3 Summary exercises

12.3.1 Ex1

Reading a file line by line

Write a generator function (with yield) function that reads in a file line by line, and with each call returns the line.

Unfinished

I still have yet to add the exercises here, at some point I will update this.

Chapter 13. Object Oriented Programming in Python

In python, everything is an object - you can test this by wrapping anything in the `type` function.

```
x: int = 12
print(type(x))
```

This gives us: `<class 'int'>`
What this means, is that `x` (the variable name) is an object (you can tell as it says 'class'), of type `int`. `int` is a built-in data type into python, one of the many we explored in chapter 4, and all those mentioned there are really classes, which are blueprints for objects.
Not just variables and data are objects, everything in python is an object. Here's a function:

```
def simple_function() -> None:
    pass

print(type(simple_function))
```

We'll get `<class> 'function'>`.
This doesn't mean that the `simple_function` is a data type per se, but rather that it's an object, which is an instance of a class. A class is like a blueprint for an object.

You may have noticed that I've previously mentioned methods for objects - such as the `.add` option for integers.
What this really is, is a method in the class `int`. Methods are basically functions that are in classes.
Alright, let's start making our own classes, with this example we are making a simple maths object:

- First start with the `class` keyword, followed by a name (PascalCase for the name), and the `:`

```
class ArithmeticClass:
```

- Next we have everything in the class indented, and we will start by defining an `__init__` constructor method. This is a function that runs every time you initialise the class.

```
class AritheticClass:
    def __init__(self, value):
        self.value = value
```

- Then we can add our own methods (which are really functions) to the class:

```
class AritheticClass:
    def __init__(self, value):
        self.value = value

    def add_value(self, x):
        return self.value + x
```

- Now we have a class made (which is like a blueprint for an object), but we still need to initialise the object. Here's how we do that:

```
a = ArithmeticClass(69)
```

Any parameters in the `__init__` dunder method are ones you have to give when you make a class.
Now we can't just print this class out - this is what we get when we do:

```
<__main__.ArithmeticClass object at 0x7fc09fec7eb0
```

We just got the memory address of it but not the value of it. This is because we didn't return anything in the `__init__` method - and we can't as it must return the `None` type.
If you wanted to have an output from print you need to use the dunder method `__str__` which prints a string to the console when printing the object normally. Here's how to use it:

```
class ArithmeticClass:
    def __init__(self, value):
        self.value = value

    def add_value(self, x):
        return self.value + x

    def __str__(self):
```

```
        return str(self.value)


a = ArithmeticClass(69)
print(a)
```

The `__str__` method only returns a string, so you need to wrap it in the `str()` if it's not a string.

The only point of `__init__` is to initialise the class the variables, not to do anything else. Hence the other dunder methods - such as `__str__` for string representation.

Any dunder method (double underscore), sometimes called a magic method is a method that does something special to python, for example there's the `__len__` method for defining a result when ran in the `len` function.

13.1 Static methods

Now suppose we want to add some more general functions to our `ArithmeticClass`, that do general operations but don't change the state of the class (any values we defined that belong to the class, like the `self.value`. We could try do this at first:

```
class ArithmeticClass:
    def __init__(self, value):
        self.value = value

    def add_value(self, x):
        return self.value + x

    def add_two_values(a, b):
        return a + b
```

But this throws an error as we need to have the self parameter on an instance method, which is the type that you've seen so far. What error? Here:

```
TypeError: ArithmeticClass.add_two_values() takes 2
positional #glspl("arg") but 3 were given
```

Python is telling us that we gave it two many arguments, you'll get the same error with this example:

```
def function(a, b):
    return a + b


result = function(0, 1, 2)
```

You may be confused as we only gave it `a, b` in the function call, but this error occurs because we have an instance method, which takes `self` (this can be called anything but by conversion it's called `self` in python, so python assumes you called the `self` instance parameter `a`). One way to fix this is to add self in the function definition, like this:

```
    def add_two_values(self, a, b):
        return a + b
```

But this doesn't make any sense as we aren't using self, so why should we pass it in?
This is where `static` methods come in, they are a type of method that doesn't do anything to the instance. This is usually used for functions that you generally organise into a class for neater code, but there isn't another reason that they are in a class. You may have seen them in other languages shown like this, for example in java:

```
class HelloReader {
    public static void main(String[] args) {
      System.out.println("Hello reader!")
    }
}
```

This method is static because it doesn't do anything to the class state. But also python doesn't do this because it isn't a boilerplate (meme) language like java.
Anyway, to use static methods in python you add the `@staticmethod` decorator to the method, like so:

```
class ArithmeticClass:
    # --snip--
    @staticmethod
    def add_two_values(a, b):
        return a + b
```

13.2 Inheritance

The idea behind inheritance is that it allows you to more easily write similar classes by abstracting them to a 'parent' class and the 'child' classes inherit their properties, state, and behaviours. Then in the child class you can add separate functionality without having to copy and paste the whole parent class.

For example (and for this example I'm going to abstract the ideas, so it may not actually be a viable option if you were going to try build this), if you were building a city simulation game, and in the city there are many buildings.

Without inheritance, you might make a class for a school, one for a home, one for a skyscraper and so on; but all these classes have lots of things in common, like similar materials, having doors and windows, and structural elements.

Here's some representative pseudocode:

```
class SkyScraper {
    int counter_number = 0

    private:
      int space;
      int cost;

    fn SkyScraper(int space, int cost)
    {
      this.cost = cost;
      this.space = space;
      counter++;
      int doors = 6;
      int windows = 256;
      int bedrooms = 0;
    }

    fn pay_tax(this, UserAccount account)
    {
      account -= VALUE
      record payment
    }
}

class House {
```

```
    int counter_number = 0

    private:
      int space;
      int cost;

    public:
    fn SkyScraper(this, int space, int cost)
    {
      this.cost = cost;
      this.space = space;
      counter++;
      doors = 2;
      windows = 16;
    }

    public:
    fn pay_tax(this, account)
    {
      account -= VALUE;
      record payment;
    }
}
```

But using inheritance, you can take this out into its own parent class of a general building, and then have all the other specific buildings inherit from the parent.

```
class Building {
  int counter = 0

  fn Building(this, int space, int cost, int windows, int
doors, char[] name)
  {
      this.cost = cost;
      this.space = space;
      this.windows = cost;
      this.doors = cost;
      this.name = name
      counter++;
  }

  public:
  fn pay_tax(this, UserAccount account)
```

```
  {
    account -= VALUE;
    record payment;
  }
}

class SkyScraper(Building) {
  fn SkyScraper(this, int space, int cost, int windows, int
doors, char[] name, floors) {
    parent.Building(this, int space, int cost, int windows,
int doors, char[] name)
    this.floors = floors
  }
}

class House(Building) {
  fn House(this, int space, int cost, int windows, int
doors, char[] name, bedrooms) {
    parent.Building(this, int space, int cost, int windows,
int doors, char[] name)
    this.bedrooms = bedrooms
  }
}
```

Here we reused a bunch of things between our classes, like the cost and the space, just changing what was unique to the class.

13.3 Dataclasses

In a lot of other languages, there's a way to group data together neatly in something called a 'struct' but here in python that doesn't exist, but we do have classes, and a special kind of class called a dataclass, can be very powerful and probably the best way to pass data around in python. Here's an example:

```
from dataclasses import dataclass

@dataclass
class User:
    id: int
    email: str
    hash: str

user = User(420, "john.doe@provider.com",
"914c13f1b81e1486202203634c9173aeb6b30d301172c2f5b34a9ec671b008f7")
```

What this basically does is add the constructor and set the values for you, so without it you might have done the same like so:

```
class User:
    def __init__(
        self,
        id: int,
        email: str,
        hash: str):

    self.id = id
    self.email = email
    self.hash = hash
```

Then you can index the values like normal:

```
accessed_hash = user.hash
```

You can also set default fields like so with the equals operator:

```
@dataclass
class User:
    id: int
    email: str = ""
    hash: str = ""
```

Now if you initialise without the email or hash fields, we get an empty string. Note this only works for immutable values.
You can also modify some values and how the dataclass behaves when you pass in parameters when you use the wrapper. Here are some useful ways to change how it behaves:

1. `repr` by default is true and modifies the representation of the class.

By default if you print a dataclass it gives you all the values of the dataclass, for example if we printed our earlier dataclass user we would get:

```
User(420, "john.doe@provider.com",
"914c13f1b81e1486202203634c9173aeb6b30d301172c2f5b34a9ec671b008f7")
```

Which might not be all that useful, so you can implement your own repr dunder method like so:

```
@dataclass(repr=False)
class User:
    _id: int
    email: str = ""
    _hash: str = ""

    def __repr__(self): -> str:
        rep = f"User email: {self.email}"
        return rep
```

2. `frozen` by default is false and means that after you create your dataclass you can't change it. This is useful if you want to later hash your dataclass.

```
@dataclass(frozen=True)
class User:
    _id: int
    email: str = ""
    _hash: str = ""
```

13.4 Summary exercises

Unfinished

I still have yet to add the exercises here, at some point I will update this.

Chapter 14. Functional python

This chapter might make your head hurt

Here I cover the more functional side of python, but in doing so have to cover some functional programming concepts and how it ties into maths, and a little bit of lambda calculus's functions to explain how it works. Try to follow along but if you find it doesn't make any sense then don't worry too much as this is only small part of python.

Python being a multi-paradigm language, it has the option to also write in a more functional way.
What does this mean?
The idea behind functional programming and languages like F#, Haskell, Elixir etc is that:

- The only abstraction allowed is a function (no classes, objects, variables, only functions)
  - Thus functions are treated a little bit differently, so functions are 'first class citizens' meaning they are treated like anything else, they can be passed around as anything else, and be returned from other functions, and taken in as parameters.
  - Functions are pure, meaning that they have no side effects, don't mutate any variables, and always take one input and return one output.
- There is no state.
  - State is anything that is stored in the program at runtime and changes. This means that all variables are immutable - meaning when a value is set it cannot change. In fact, variables are just names, rather than what we traditionally think of as a variable.

Thinking in a functional way, functions are much more like the functions you get in maths, which are known as pure functions. This

means that they have no side effects, always take a single input and always return a single output.
Take this function for example:

```
def square(x: int) -> int:
    return x * x
```

Which can be written as this:

$$f : \mathbb{Z} \to \mathbb{N} \text{ such that } x \mapsto x \cdot x$$

(the $\mapsto$ means maps to, like mapping from one set of standard Cartesian coordinates to another)
or simply like this:

$$f : f(x) = x \cdot x$$

Quickly I'll go over lambda functions as it makes it easier to translate between the maths functions and python functions.

14.1 Anonymous (lambda) functions

All this is is a function without a name, but you can assign it to a variable, which really is just a name. It allows to express functions in the more 'mathsy' way. This is often used to pass to other functions like map, slice, sorted, filter etc. Here's how you use it.

```
square = lambda x: x * x
```

This is the more functional way of writing a function.
You can also take and return any number of parameters:

```
sum3 = lambda: x, y, z: x + y + z
```

As you can see it can take any number of arguments. Lambda functions can also return any number of values, by bundling them together as a tuple or list, like so:

```
diffAndSum = lambda x, y: (x + y, x - y)
```

14.2 Higher order functions

This is where instead of having large recursive function that run your program, you break it into smaller functions, and to really utilise the power of these functions we have functions that return functions.

We've seen this already when we covered function wrappers and decorators, but let's see a really simple example of one first.

```
def return_add_n(x: int) -> func:
    def add(y: int) -> int:
        return x + y
    return add

add_42 = return_add_n(42)  # add_42 is now a function that
given an integer will add 42 to it

print(add_42(27))  # prints 42 + 27, so 69
```

14.3 Currying

This is named after mathematician Haskell Curry, and is a way to 'bind' parameters to strictly pure functions that always take one parameter and always return a single value.
Let's first look at the impure way of doing it:

```
sum = lambda a, b: a + b
```

In lambda calculus you instead express it as this:

$$f : f(x) = \lambda x.(\lambda y.x + y)$$

or in more familiar maths functions like this:

$$f : f(x) = (f(y) = x + y)$$

```
add = lambda a: (lambda b: a + b)
```

This is kind-of impractical in most cases as tuples are a thing and we don't need to follow lambda calculus rules.

14.4 Function composition over procedural commands

Suppose you wanted a function that returns the sum of two squares like this:

```
def sum_two_squares(a, b):
    return (a ** 2) + (b ** 2)
```

And if we separate the operations we get this:

```
def sum(a, b):
    return a + b

def square(x):
    return x * x

def sum_two_squares(a, b):
    x = square(a)
    y = square(b)
    return sum (x, y)
```

While these functions are pure, they still are a concatenation of commands, a procedural way of writing code. Instead of this let's first rewrite the last function as an expression:

```
def sum_two_squares(a, b):
    return sum( square(a), square(b) )
```

But we're still using a command, `return`, so let's rewrite this with anonymous functions instead:

```
sum = lambda a, b: a + b

square = lambda x: x * x

sum_two_squares = lambda a, b: sum(square(a), square(b))
```

There's still a problem here though, as in original lambda calculus functions have a fixed arity of 1, meaning they can only take one parameter. So let's fix this with currying:

```
sum = lambda a: (lambda b: a + b)

square = lambda x: x * x

# make name shorter to fit on the page
sum_t_s = lambda a: (lambda b: sum(square(a))(square(b)))

# How to use this function
six_squared_plus_seven_squared = sum_t_s(6)(7)  # = 85
```

This is a bit harder to read, so python actually prefers you to use this syntax for functions:

```
def sum(a):
    return lambda b: a + b

def square(x):
    return x * x

def sum_two_squares(a):
    return lambda b: sum(square(a))(square(b))

sum_two_squares(4)(5)
```

Even though it's not as 'functional' as the previous.

14.5 Recursion (part 2)

Since no variables can be mutated, we instead use recursion, which is when functions call themselves, to get a kind of iteration.

As I mentioned earlier, recursion when not used properly is risky as it can go on forever filling up the call stack until it breaks your program. To avoid this, we must always have a termination in the recursion, with a terminal case:
We must have three types of cases in recursion:

- Recursive Case: The part of the function where it calls itself with modified arguments, moving the problem toward the base case.
- Base Case: The simplest instance of the problem, where the function does not call itself and returns a direct answer. The base case prevents infinite recursion and stack overflow errors.
- Progress Toward Base Case: Each recursive call must modify the input to ensure that the base case is eventually reached.

Here's an ideal use case for recursion: you want to have a factorial function (perhaps because you didn't know python had one from the math library), and you decided to write it yourself. First let's write the normal, standard, way of writing code, no recursion.

```
def factorial(x: int) -> int:
    if x < 0:
        raise ValueError("can't factorial a negative")
    elif x == 0:
        return 1
```

```
    result = 1
    while x > 1:
        result *= x
        x -= 1
    return result
```

By this point, this should all be self-explanatory to you, if it's 0, you get 1, if it's negative we throw an error, and we use a loop that multiplies each element for by a number, which is set to the previous element. We return when we hit 1.

This same code can be written like this:

```
def factorial(x: int) -> int:
    if x < 0:
        raise ValueError("can't factorial a negative")
    elif x == 0 or x == 1:
        return 1
    else:
        return x * factorial(x - 1)
```

or in a more concise way like this using python's ternary operator:

```
def factorial(n):
    return 1 if n == 0 or n == 1 else n * factorial(n - 1)
```

This took me a long time to get my head around, but the easiest way to actually get it is to get a piece of paper and write down what happens at each step yourself. Nevertheless, I'll try to explain what happens. Crucially you return a function call, which means that before it can return, it evaluates the function, which returns a result allowing the 'parent' function to return. This is what happens at first:
Suppose we get a number like 5 thrown into the function.
It hits the else block, which returns 5 (which is x) multiplied by what happens when you pass 5-1 (4) into function.
Recursion works this way when you have a task that you can take as the same task minus one little part you evaluate, like here, we took out and said that the factorial of 5 is really the factorial of 4 multiplied by 5.
The function calls itself until it hits the base case which has a 'normal' return, returning 1, and pushes this back up the chain to close the function calls.

The next function call up, which is when we called `factorial(2)`. Here this evaluates to 2 * factorial(1) which returned 1, which then evaluates to 2. The next call is factorial(3) and so on. This continues back up returning from each function until we finish.

If you're familiar with C style loops, which I'll show below, we can take each part and turn it into the recursive function, making a normal loop a recursive loop. Let's start with a normal loop:

```c
#include <stdio.h>

void print_arr(int* arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}

int main() {
    int arr[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
    int n = sizeof(arr)/sizeof(arr[0]);
    print_arr(arr, n);
    return 0;
}
```

I apologise for not sticking with pure python, but C has the most universal `for` loops, so I felt it would be more appropriate. Here all we do is define the array, the array's length as n, and pass it to the function print_arr.
It iterates over each element and prints it.
The for loop has three parts: the initialisation, (int i = 0), the condition (i < n), and the post (i++).
Equivalently in python it's this:

```python
i = 0
while i<10:
    i += 1
```

To turn this into a recursive loop, which is used in functional paradigms, where all variables are immutable, we need to take care of this mutation, as it's not allowed to change variables.
In functional code, all you have is a function, so let's first take out this main body of our C into its own function, like so:

```
void print_index(int* arr, int index)
{
        const int item = arr[index];
        printf("%d", item);
}
```

All this function does is print an item given an index, which is just what is being done for every time we iterate in the for loop.
To print the whole array we need to call this function for every item in our array.
First to get looping we add the function call at the end.

```
void print_index(int* arr, int index)
{
        const int item = arr[index];
        printf("%d", item);
        print_index(arr, index);
}
```

But this is no good, as we are calling this function indefinitely and it just prints the same item over and over again.
To increment for every function call and no mutation, we add a new constant which is the next index, and call print_index with the next index each time.

```
void print_index(int* arr, int index)
{
        const int item = arr[index];
        printf("%d", item);
        const int nIdx = index + 1;
        print_index(arr, nIdx);
}
```

But this goes until the end of the list until the program crashes as it tried to read an index past the length of the list.
To stop this we have our base case, which corresponds to our condition. We only run it so long as the index is in range, so pass the list's length, and have an if check.

```
void print_index(int* arr, int index, int n)
{
        if (index == n)
                return;
```

```
        const int item = arr[index];
        printf("%d\n", item);
        print_index(arr, index + 1, n);
}
```

And there, we've swapped normal iteration for a recursive function call, rewriting our loop in a more functional way. This works the same way in python, so try to apply what I've shown you here.

14.5.1 Interview question with recursion

There's a famous-ish interview question that goes like this:

> "You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?"

This includes all permutations, so for example, if the staircase had 5 steps there would be 8 ways to climb the staircase:

```
1. 1, 1, 1, 1, 1
2. 1, 1, 1, 2
3. 1, 1, 2, 1
4. 1, 2, 1, 1
5. 2, 1, 1, 1
6. 1, 2, 2
7. 2, 1, 2,
8. 2, 2, 1
```

Here I'll walk through the way to solve it recursively step by step:

- We need a base case:
  - If it has 0 or 1 steps there's only one way to climb it, so let's define this as a starting point.

  ```
  def count_ways(stairs: int) -> int:
      if stairs == 0 or stairs == 1:
          return 1
  ```

  This is the only number we'll (manually) return from this function, the recursive calls do the rest.
- And we need a recursive case:
  - Thinking about it, there are two ways you can reach the top step: either having gone from two steps up or from one step up. So let's

return those two cases, and all the ways to get to those two steps as well, like this:

```
def count_ways(stairs: int) -> int:
    if stairs == 0 or stairs == 1:
        return 1
    return count_ways(stairs - 1) + count_ways(stairs - 2)
```

This function now calls itself modifying the input until the value is 1, and then closes all opened function calls feeding the number up the chain until it returns a single integer.

14.6 Summary exercises

14.6.1 Ex1

Rewrite in python

Given what you've learned, take the C code I used as the example and rewrite it in python. It's easier in python as you don't have to worry about the length of the array in the fancy way as you just use `len()` or the method.

14.6.2 Ex2

Binary search

Using recursion, so no mutation of variables, write a binary search function, where given a list and an item will return a boolean depending on whether it was found. Binary search cuts in half repeatedly until the item is found, and only works for sorted data.
This is an ideal problem to solve with recursion, as it can easily be broken down, and there are clear cases. Try to apply what you've learned.

Unfinished

This part here is unfinished, at some point I'll come back and add this content.

Chapter 15. Testing

So far we've just been writing code and hoping that it works, then running it and testing it 'interactively' that way. This isn't ideal, and you can't see just from looking at code whether it does what you want it to. Instead, I'll introduce you to how to write code to test your other code. This is also crucial for finding logical and semantic errors that otherwise go under the radar, as I showed in my Errors chapter.

15.1 A very simple example

To start off with, the general way of writing code to test code is to mimic the directory and file structure, and have test functions that test your functions. Here's an example:

```
# file maths.py
def add(a, b):
    return a + b
```

```
# file test_maths.py
from maths import add

def test_add():
    a = 10
    b = 15
    result = add(a, b)
    assert result == 25
```

And that's it for the basic case, `assert` will raise an error if something went wrong, and you can pass an error message to assert like so:
`assert result == 25, f"{a} + {b} returned {result}, should have been 25"`

15.2 Key concepts in testing

- Test Case: A single unit of testing, defined as a class inheriting from `unittest.TestCase`. Each test case contains methods that test specific functionality.
- Test Suite: A collection of test cases or test methods to be run together.

- Assertions: Methods like `assertEqual`, `assertTrue`, or `assertRaises` to check if the code behaves as expected.
- Test Runner: Executes tests and reports results.
- Setup and Teardown: Methods to prepare the environment before tests (`setUp`) and clean up after (`tearDown`).

15.3 Unit tests using the unittest module

Just using assertions can work but also gets a little tedious, so let's use the standard library's unittest module to help us out here.
To start off, let's rewrite our previous test using it so you can get to see the syntax:

```
import unittest
from maths import add

class TestMaths(unittest.TestCase):
    def setUp(self):
        # Runs before each test method
        self.num1 = 10
        self.num2 = 5

    def test_add(self):
        result = add(self.num1, self.num2)
        self.assertEqual(result, 15, "Addition failed")

    def tearDown(self):
        # Runs after each test method (optional cleanup)
        pass

if __name__ == '__main__':
    unittest.main()
```

Now suppose we add a division to our `maths.py` file like so:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Here we have two cases to test for, a normal case and the raise case where we want to raise an error. Here's how you test for this:

```
class TestMaths(unittest.TestCase):
     # -- snip --
```

```
    def test_divide(self):
        result = divide(self.num1, self.num2)
        self.assertEqual(result, 2.0, "Division failed")

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(self.num1, 0)
```

Here we add a new method for every case so we test everything and debugging is easier.

15.4 Mocking functions for tests

Throughout this, we've very often used the 'input' function within our functions, but you may see a problem here while testing, as it's not straightforward to define code that puts something into the input function. Instead we write use mock functions that supply inputs to the input function (or similar). For this you need `unittest.mock`. Here's an example for where we would need to use it, and how to use it. Suppose we have a function:

```
def get_info():
    name = input("enter name: ")
    credit_card = int(input("enter credit card number: "))
    return name, credit_card
```

since `get_info()` doesn't take any arguments we need to mock input by patching it like this:

```
import unittest
from unittest.mock import patch

class TestUserInfo(unittest.TestCase):
    @patch('builtins.input')
    def test_get_user_info(self, mock_input):
        # Set up the mock to return specific values for
consecutive calls
        mock_input.side_effect = ["Alice", "25"]

        # Call the function
        result = get_user_info()

        # Assert the result
```

```
        self.assertEqual(result, {"name": "Alice", "age":
25})

        # Optionally, verify that input() was called with
the correct prompts
        mock_input.assert_any_call("Enter your name: ")
        mock_input.assert_any_call("Enter your age: ")

if __name__ == '__main__':
    unittest.main()
```

15.5 Test driven development

This is a way of developing your code where instead of writing a bunch of code, then writing more code to test it you write the tests before the code, and you write code to pass your tests. The tests become your code's requirements.

15.6 Summary exercises

Unfinished

This part here is unfinished, at some point I'll come back and add this content.

Chapter 16. Asynchronous programming and parallelism

Asynchronous programming means you have a program that doesn't start executing all at the same place, instead it starts at different places that all finish their tasks at different times. We do this with co-routines. There are different models for parallelism:

- Threading
  This is for low resource tasks that run at the same time and may need to share data
- Processes
  This is for much more intensive or long computational tasks that run on different cores of a CPU
- Async
  When you don't want the program to hang on a certain task, and to continue doing other things, where the program starts executing in multiple different places.

16.1 Async

First let's import it:
`import asyncio` And get a simple function that runs asynchronously, with the `async` keyword.

```
async def main() -> None:
    print("YOOO i'm running async!")
```

But you can't just run this function normally, as it returns a *coroutine object*, which then needs to be 'awaited'.
To actually run this we use this like of code: `asyncio.run(main())`, so our very first async program looks like this:

```python
import asyncio

async def main() -> None:
    print("YOOO i'm running async!")

if __name__ == "__main__":
    asyncio.run(main())
```

<record scratch> Hold on what's actually going on?
Let's explain the theory before we go any further:

16.2 The Theory

Asynchronous programming is a way to handle tasks that might take time to complete (like fetching data from a website, reading a file, or querying a database) without blocking the execution of other tasks. It allows a program to start a task and move on to other work while waiting for the task to finish, improving efficiency, especially for I/O-bound operations (tasks that spend a lot of time waiting for input/output, like network requests).

16.2.1 The event loop

This is like the nucleus/core of asynchronous programming, that manages tasks, schedules them to run, checks to make sure that they don't block each other etc. It's like a central hub that manages everything else.

16.2.2 Coroutines

This is like a task, which in python is a special type of function that can pause its execution and give control to the event loop - like `yield` does. In python we define them with `async def`.

16.2.3 Await

This is a keyword (the last one! We've covered them all now!) that is used within an async function to pause execution until another coroutine/task finishes. It then tells the event loop that it can run other tasks in the mean time.

16.2.4 Tasks and Futures

- A *Task* is a wrapper around a coroutine that schedules it to run in the event loop.

- A *Future* is a lower-level object representing a result that will be available later. Tasks are built on top of Futures.

16.3 The (Theoretical) Practice

Now that the theory is out of the way, let's see how to properly apply all of this inside of python.

- We use `async def` to define a coroutine function
- We use `await` to – you guessed it – wait for a coroutine function to finish without blocking the event loop. You can only use await inside a coroutine function.
- `asyncio.run()` runs the main coroutine
- `asyncio.create_task()` schedules tasks

Remember that `async def` coroutine functions return a coroutine object, and need to be run inside the event loop (like with `asyncio.run()`)

16.4 The Practice Practice

Here I'll show some examples and talk you through them, and I'll be using `asyncio.sleep()` as an async version of `time.sleep()`

```
import asyncio

# Define an async coroutine
async def say_hello():
    print("Starting to say hello...")
    await asyncio.sleep(1)  # Simulate a 1-second delay
(non-blocking)
    print("Hello, world!")

# Run the coroutine
if __name__ == "__main__":
    asyncio.run(say_hello())
```

16.5 Multiple coroutines

The previous example was very basic, let's look at multiple coroutines now:

```
import asyncio
```

```python
async def task1():
    print("Task 1 started")
    await asyncio.sleep(2)  # Simulate work
    print("Task 1 finished")

async def task2():
    print("Task 2 started")
    await asyncio.sleep(1)  # Simulate faster work
    print("Task 2 finished")

async def main():
    # Run tasks concurrently
    await asyncio.gather(task1(), task2())

# Run the event loop
if __name__ == "__main__":
    asyncio.run(main())
```

Explanation:

- `asyncio.gather()` runs multiple coroutines concurrently and waits for all to complete.
- `task2` finishes before `task1` because it has a shorter sleep time, but they run concurrently, not sequentially.
- Total runtime is 2 seconds (the longest task), not 3 seconds (sum of sleep times).

Output:

```
Task 1 started
Task 2 started
Task 2 finished
Task 1 finished
```

16.6 Using tasks

```python
import asyncio

async def fetch_data(id):
    print(f"Fetching data for ID {id}")
    await asyncio.sleep(id)  # Simulate varying work time
    return f"Data for ID {id}"

async def main():
    # Create tasks
```

```
    task1 = asyncio.create_task(fetch_data(1))
    task2 = asyncio.create_task(fetch_data(2))

    # Wait for tasks to complete and get results
    result1 = await task1
    result2 = await task2
    print(f"Results: {result1}, {result2}")

asyncio.run(main())
```

Explanation:

- `asyncio.create_task()` schedules a coroutine to run in the event loop immediately.
- Tasks run concurrently, and we use `await` to get their results.
- The total runtime is 2 seconds, as tasks run in parallel.

Output:

```
Fetching data for ID 1
Fetching data for ID 2
Results: Data for ID 1, Data for ID 2
```

16.7 Error handling:

We may also want to catch exceptions that may be thrown from async functions, can to avoid an async/await try/catch hell we can neatly do it like this:

```
import asyncio

async def risky_task(id):
    print(f"Starting task {id}")
    await asyncio.sleep(1)
    if id == 2:
        raise ValueError(f"Task {id} failed!")
    return f"Success for task {id}"

async def main():
    tasks = [risky_task(i) for i in range(1, 4)]
    try:
        results = await asyncio.gather(*tasks,
return_exceptions=True)
        for result in results:
            print(f"Result: {result}")
    except Exception as e:
```

```
        print(f"Caught error: {e}")

asyncio.run(main())
```

Explanation:

- `return_exceptions=True` in `asyncio.gather()` ensures that exceptions are returned as results instead of crashing the program.
- Task 2 raises an error, but tasks 1 and 3 complete successfully.
- This approach is useful for robust async programs.

Output:

```
Starting task 1
Starting task 2
Starting task 3
Result: Success for task 1
Result: Task 2 failed!
Result: Success for task 3
```

Unfinished

This part here is unfinished, at some point I'll come back and add this content.

Chapter 17. References

17.1 Tables

Operator	Name	Example	Output
+	Addition	2 + 9	11
-	Subtraction	2 - 9	−7
*	Multiplication	2 * 9	18
/	Division	2 / 9	$0.2\overline{2}$
//	Integer division (returns floored quotient)	2 // 9	0
%	Modulo (returns remainder of division)	2 % 9	2
**	Exponent	2 ** 9	512
@	Matrix multiplication	[[1, 2], [3, 4]] @ [[5, 6], [7, 8]]	[[19, 22], [43, 50]]

Operator	Name	Example	Output
=	Assignment	`architecture = "x86_64"`	
:=	Walrus	`print( x := 3 )`	3

Operator	Name	Example	Output
==	Equality	`"" == []`	`False`
>	Greater than	`10 > 10`	`False`
<	Less than	`10 < 10`	`False`
>=	Greater-than-or-equal-to	`10 >= -1`	`True`
<=	Less-than-or-equal-to	`10 <= 10`	`True`

Operator	Name	Example	Output
`and`	Logical AND	`True and False`	`False`
`or`	Logical OR	`True or False`	`True`
`not`	Logical NOT	`not (True and False)`	`True`

Operator	Name	Example	Output
&	AND		
|	OR		
~	NOT		
^	XOR		
>>	Right shift	11 >> 1	5
<<	Left shift	11 << 2	44

17.2 Links

- Official Python Docs
  - This is pretty helpful, especially the tutorial for getting the hang of python, but it's meant for programmers (who know how to code in other languages) rather than complete beginners.

17.2.1 Youtube videos I found useful

- Learn Python Fast by Alex Mux
  - A very helpful video for python's syntax, explained in a 'no fluff' fast way
- Why you shouldn't nest your code
  - From code aesthetic, where I learned the 'never nesting' philosophy
- Don't write comments
  - Again from code aesthetic, where I got my views on comments which I explain in chapter 9
- Dear functional bros
  - Also from code aesthetic, from which I learned a lot of the functional concepts from chapter 14. The examples are in JavaScript, but it's pretty easy to understand while knowing no JavaScript, as the concepts are what matter, not the syntax.
- OOP for beginners
  - This is where I learned most of the OOP chapter from.

Chapter 18. Glossary

argument: In a function call, the values put in place to substitue the parameter placeholders are called arguments, e.g sum_result = add(6 + 7), here 6 and 7 are the arguments given to the function add. This is referred to as the 'actual argument'. 4, 4, 10, 33, 33, 49, 55, 55, 55, 56, 60, 62, 62, 62, 62, 62, 63, 63, 63, 63, 64, 64, 64, 64, 64, 64, 64, 64, 64, 92, 100, 108, 111, 120

boolean – **boolean value**: A value that can only hold one of two values, True or False. This was named after George Boole, coming from boolean algebra. 3, 16, 16, 16, 16, 25, 25, 31, 31, 36, 43, 43, 45, 51, 57, 116

stack – **call stack**: In computer science, a call stack is a LIFO (Last-In-First-Out) stack data structure that tracks active subroutine invocations during program execution. It manages control flow by storing stack frames (activation records) containing:

Return addresses (location to resume after subroutine completion)

Local variables (temporary data scoped to the subroutine)

Parameters (arguments passed to the subroutine)

Bookkeeping data (e.g., saved registers, frame pointers)

When a recursive function causes a 'stack overflow', it is this stack that ran out memory for 'frames' being pushed to it 71, 111

class: A class is a construct in object-oriented programming that defines an object with its characteristics and shared aspects from the class. 18, 25, 32, 76, 88, 97, 97, 97, 101, 101, 101, 103, 104, 107

heap – **memory heap**: A memory heap is a region of process memory reserved for dynamic allocation of data structures whose size and lifetime are determined at runtime. It operates as a pool of unstructured memory managed through explicit programming

pointer: A pointer is a primitive data type storing the memory address as its value - which enables indirect data access. They are used for manual memory management. 18, 18

stack – **stack memory**: This is a high-speed part of memory on CPU that works as a LIFO data structure, and where the memory usage must be determined at compile time, therefore only immutable data can be stored here. 18

syntactic sugar: Part of language syntax that makes it easier to use/read for humans. It makes the code 'sweeter'. 22, 69

token – **lexical token**: A lexical token is a discreet part of the language, a string with an assigned meaning. It has a name and a value. For example you can have keywords, such as 'def', 'class', 'return', but also punctuation like parenthesis or ':', or operators, whitespace, identifiers and literals 83

variable: In programming, a variable is a symbolic identifier bound to a mutable reference that stores an object or value in memory. Technically, it functions as a named pointer to a memory location where data is stored, enabling indirect manipulation of the underlying value through the identifier. 2, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 20, 20, 20, 20, 20, 21, 24, 24, 25, 25, 25, 26, 26, 26, 26, 26, 26, 27, 32, 32, 32, 32, 32, 38, 40, 55, 55, 61, 61, 62, 62, 65, 65, 76, 77, 77, 89, 94, 97, 99, 107, 107, 107, 107, 111, 113, 113, 116