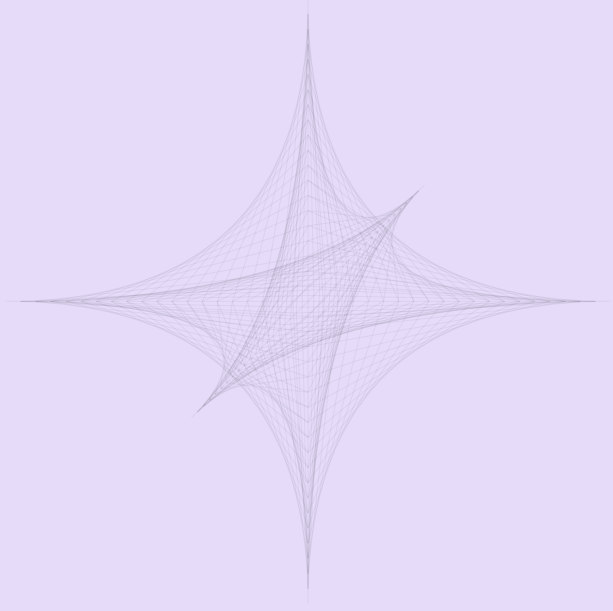
Introduction to Python & Algorithms

by Khallil Benyattou



スプレンッヂド
立派

March 3, 2024

Contents

Disclaimer

I wrote these notes for my own personal use, following the general outline of **MIT 6.00 (Fall 2008)**. I've rearranged parts, omitted others and added a significant amount of information that I learned from several other sources all over the web.

I have, regrettably, failed to reference the aforementioned sources since I didn't originally set out to share these notes with others. Sites and videos I can think of that were useful around the time of writing include (but are certainly not limited to):

- **https://stackoverflow.com/**
  - We stand on the shoulders of giants.
- **Learn Python the Hard Way** by Zed Shaw
  - Good practice for getting the basics down.
- **Python Programming Beginner Tutorials** playlist by Corey Schafer
- **So you want to be a Python expert?** by James Powell

If anybody is trying to sell you these notes:

- ~~I am flattered.~~
- These notes are freely available for public access and should not, in any circumstance, be sold or distributed for profit.

These notes are hosted in only one location, my website:
https://kbenyattou.github.io/notes/
All errors herein are my own.

Chapter 1

Overview

The goals listed in the MIT course are as follows:

- be able to use the basic tools of computational thinking to solve small scale problems,
- understand the role that computation can and cannot play in tackling technical problems,
- learn to read, write and understand programs written by others,
- understand the fundamental capabilities and limitations of computation, and
- to gain the ability to map scientific problems into a computational framework.

1.1 What is computation(al problem solving)?

The primary knowledge one should take from this course is the notion of computational problem solving - the ability to break down a problem into a sequence of simple steps that a computer can understand and execute. Such a sequence is called an algorithm:

An **algorithm** is an unambiguous sequence of instructions to solve a problem.

To compute is to execute an algorithm. How does one formulate an algorithm? This depends on the type of instructions you'd like to execute. The language of declarative and imperative knowledge give us two ways to formulate instructions.

1.1.1 Declarative and Imperative Knowledge

Declarative (or descriptive) **knowledge** refers to information (statements of fact) that describes things, their attributes and their relation to each other.

An example of declarative knowledge in mathematics is the definition of the positive square root of a non-negative real number: "$\sqrt{x}$ is the $y \geq 0$ s.t. $y^2 = x$". This definition offers a description of the square root. It doesn't give you an idea of *how to compute it*.

This can be contrasted with imperative (or procedural) knowledge:

Imperative (or procedural) **knowledge** is knowledge exercised in the performance of some task e.g. a sequence of specific instructions that tells you *how* to do something.

An example, attributed to Heron of Alexandria, for finding an approximation of the square root of $x \geqslant 0$ is written in the imperative as follows:

Algorithm 1 Heron of Alexandria's approximation of $\sqrt{x}$ for $x \geqslant 0$.

```
1: procedure HERONSQRTAPPROX($x$)
2:     $G \leftarrow g$                                  ▷ Initialise a guess $g \geqslant 0$
3:     while $G^2 \not\approx x$ do                      ▷ Loop terminates when $G$ is "close" to $x$.
4:         $G \leftarrow \frac{1}{2}(G + \frac{x}{G})$    ▷ Update $G$ for the next iteration.
5:     end while
6:     return $G$
7: end procedure
```

Line 1.3 enters a **while** loop that repeats the content indented by 4 spaces underneath it (line 1.4) *while* the condition before the **do** evaluates to `True`. Once an iteration produces a value of G for which $G^2 \approx x$, the loop terminates and the subsequent code is executed – at this point, we've found a "suitable" G.

The name given to the idea that lines of code can test whether an expression evaluates to either `True` or `False` and then change where we are in the sequence of operations is called the **flow of control** of the algorithm.

Humans are not very fast at computing things. We use computers to execute algorithms.

1.2 A Brief History of Computers

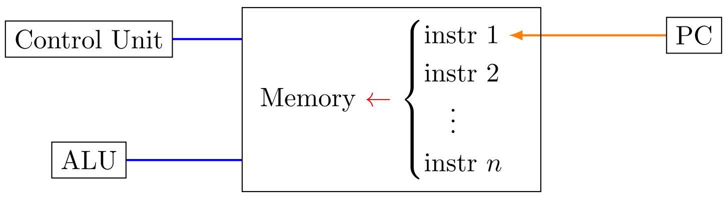
Computers weren't always the powerhouses that they are today. Naturally, there's been an evolution of what we consider a computer to be.

Initially, the idea was to design a circuit that executed a specific algorithm. Such machines were the earliest computers – fixed program computers. Examples of these include basic four function calculators and Turing's codebreaker machine. This used to be the way that all computers worked. Fixed program computers are useful but in a very limited way. They don't fully capture the idea of problem solving.

Instead of designing a circuit to execute a particular algorithm, the more abstract idea of a circuit that takes in an algorithm and "adjusts its own wiring" to perform the algorithm was the real breakthrough of computing. This was the advent of the stored program computer. Treating an algorithm/program as an input makes it indistinguishable from the data on which the program operates. To this end, programs could produce programs. This turned out to be an idea that would completely capture the idea of computation. Computers became "infinitely" flexible.

Once this became clear as the paradigm for computers, people began to think of the computer itself as a program. In particular, as a kind of program known as the interpreter. The interpreter is a program that can execute any legal set of instructions.

The basic structure of a stored program computer generally looks like the following diagram:



We feed a sequence of instructions **into** the stored computer where it's stored in memory (and treated as data). The memory is **connected** to a control unit and an arithmetic logic unit (ALU). The program counter (PC) **points** to some location in memory, typically the first instruction in the sequence.

The instructions are typically very simple e.g. take the value of two places in memory, run them through the multiplier (some piece of circuitry) in the ALU, and then stick them back to some place in memory. Having executed the first instruction, the program counter changes and points to another place in memory - the next instruction if our program is linear, or perhaps another place if a test succeeds/fails. This is how programs are executed.

Now that we have a rough outline of how a stored program works, we can focus on the algorithms that are fed into them.

1.3 Primitives and Languages

To describe an algorithm, we need a set of primitive instructions to define and manipulate data and control the flow of execution. The primitives we choose will determine the range of algorithms we can express. Thus, this choice is a very important one.

Fortunately, computer scientist Alan Turing showed that with only 6 primitives (each of which operates on a single bit of information), we can program anything that can be described by a mechanical process.

It would be painful to use only these 6 atomic primitives to code. Instead, we can abstract further to create a new set of more complex primitives.

A **programming language** is a notation for writing programs.

What distinguishes one language from another is largely due to the combining mechanisms defined to manipulate data and execution flow in a program.

We can describe programming languages on 3 notable axes:

1.4 Different Dimensions of a Programming Language

1. **High vs Low Level**
   This is a measure of how close the language is to the guts of the machine.
   At the very bottom (low level) of the spectrum, there's machine code - sequences of 0s and 1s that the machine understands and executes. A step above is assembly and is an example of a low level language. Higher level languages abstract away from the most basic primitives that drive computer interaction.

2. **General vs Targeted**
   Does the set of primitives support a broad range of applications or only a small set?
   MATLAB, for example, is a language targeted at matrix and vector operations.

3. **Interpreted vs Compiled**
   Since computers only understand binary, getting a computer to perform tasks requires some sort of translator to take source code (instructions that you type) and turn them into machine code. With compiled languages[1], a compiler takes your program and produces another program written in assembly. For an interpreted language, the interpreter typically executes commands directly from your source code.

   ◦ Compiled languages are usually faster but aren't readable by humans. Since assembly language differs between each individual computer (depending on its architecture), compiled code can only run on computers that have the same architecture as them when they were compiled.

   ◦ Interpreted languages are generally slower but are easier to debug because error codes come back in terms of the source code (which we can read very easily).

Python is a **high level**, **general** purpose and **interpreted** language.

All that follows is in the Python programming language unless otherwise specified.

1.5 Syntax and Semantics

The description of a programming language is usually split into syntax (form) and semantics (meaning).

The **syntax** of a programming language describes which strings of characters and symbols form a valid program.

Typing `3+4` into the interpreter is syntactically correct but `3 4` is not

[1]Any language can be compiled or interpreted. However, it is well understood that if we call a language itself interpreted or compiled, we really mean the particular implementation of the language that we're referring to.

By analogy with English, the syntax describes which strings of words constitute well-formed (but not necessarily meaningful) sentences e.g. “All your base are belong to us.” is syntactically well-formed but it isn’t *meaningful*:

The **static semantics** of a programming language describes what syntactically valid programs mean.

`3/'abc'` is syntactically correct because it’s in the form value-operator-value but Python would tell us that there’s no real meaning to this. Dividing an integer by some text is meaningless. The expression is static semantically incorrect.

Value-operator-value can be compared to noun-verb-noun in English. e.g. “I are big” is syntactically well-formed but isn’t meaningful.

The **semantics** of a programming language describe what syntactically valid programs mean.

In natural language, sentences can be ambiguous. However, in programming languages, every well-formed program has exactly one meaning. If a program is well formed and means something, then that is what it means (despite it potentially not meaning what you intended).

Depending on how well written a program is, various outcomes are possible when it’s finally executed. A program can run to completion and give you the desired output, crash, infinitely loop or even run to completion and give you an answer that you didn’t expect. To combat any undesirable outputs, one needs to develop a way (or style) of writing code that makes it easy to spot where semantic bugs come from.

Chapter 2

Core Elements of a Program

To write a program, we need a way to represent data.

Objects are Python's abstraction for data.

All data in a Python program[1] is represented by objects or relations between objects. Each object has a value, identity and type.

- The **type** of an object determines the size and layout of the object's memory, the range of values that can be stored in the memory and the set of operations that the object supports.
- The **identity** of an object can be thought of as its address in memory.
- The **value** of objects can change or not depending on the type. This partitions objects into two groups: mutable and immutable.

2.1 Data Types

The most primitive data types include numbers, strings and booleans. Numbers are there to do numerical things, strings are there to represent textual information and booleans are there to facilitate conditional logic. All other data structures, no matter how complex, can be built using those three.

A special type of object in Python is `NoneType`. This type is reserved for an object `None` that has no value. We often use this type of object as a placeholder in code. All other data types will have a value.

Examples of primitive data types include:

- **Strings** (`str`) are Python's version of `char` in other languages. We write them enclosed by apostrophes or speech marks and they represent textual/symbolic information e.g. `'pirate'` is how we write the word pirate in Python.
- **Booleans** (`bool`) are logical propositions. There are only two; `True` or `False`.
- **Integers** (`int`) are whole numbers which are typed as you normally would e.g. `25`.
- **Floating point numbers** (`float`) are approximations[2] of real numbers. We write them with a decimal point to differentiate them from integers e.g. `5.0` is the float

[1] Python code itself is an object by virtue of data and programs being indistinguishable.
[2] **Warning**: Python represents floating point numbers using the IEEE 754 floating point standard

for the number 5.

- **Complex numbers** (`complex`) are numbers in the complex plane like $3 + 2i$. In Python we write `3+2j` for such a number where `j` denotes the imaginary unit $i := \sqrt{-1}$.

- **Literals** are notation that Python recognises as syntax for writing an object directly. They are how we type them e.g. `'pirate'` and `25` are string and integer literals respectively.

The literal of an object determines its value.

We can access the type of an object by typing `type()` with an object's literal between the parentheses e.g. `type(3)` will return `<class 'int'>` which tells us that it is an integer.

There are many data types in Python. We can broadly organised into them two groups - **scalar** and **composite**. An object of scalar type can store only a single value such as an integer, boolean or float. An object of composite data type can store multiple pieces of related data and is treated as a single datum. Later we shall see examples of composite data types. For now we simply name them: lists, sets, tuples, frozen sets and dictionaries.

2.1.1 Relations Between Object Types

Given two objects, it would be useful to have some method of combining them to form a new object. We'd also need some vocabulary to describe how we type such "sentences" (statements) into the Python interpreter. To this end, we define the following terms:

- An **identifier** is a name.
- An **operator** is an umbrella term for a construct that can manipulate the value of operands.
  - A familiar subgroup of operators is arithmetic operators like `+ - *` and `/`. There are also logical (`not`, `or`, `and` etc.), comparison (`>=`, `<`, `==` etc.), assignment (`=`), membership (`in`) and identity operators (`is`).
- An **expression** is a representation of a value. They only contain identifiers, literals and operators (including arithmetic, boolean, function call `( )`, subscription/indexing `[ ]` and similar operators). Expressions reduce to a particular value e.g. the expression `3*4` is a representation of the integer `12`.
- **Statements** are everything that can make up a line (or several lines) of code. Every expression is a statement.

- a variant of scientific notation. **Do not test floating point numbers for equality** (`x==y`) because you'll have a high probability of getting `False` instead of `True`. This is because of errors in truncation/rounding that arise when numbers are converted back and forth between base 10 and binary (base 2) so that computers can do calculations.

2.1.2 Type Checking

We combine objects (operands) using operators to form expressions. When evaluating expressions, the Python interpreter does what is called type checking - a process designed to catch static semantic errors between types of objects and operators.

Languages fall on a spectrum from being **weakly** to **strongly typed**. This is a measure of how much type checking occurs before a program is run. Python is closer to being strongly typed than weakly typed but not as strongly as some may like. For example, entering `'a' < 3` evaluates to `False`.

How does Python even compare a string to a number? The answer lies in ASCII values, a character encoding standard. The `<` operator in this case compares the lexicographical ordering of symbols (including numbers) and outputs accordingly.

Some operators in Python are overloaded. An operator is **overloaded** if its meaning depends on the types of operands it acts on. For example, the "addition" (`+`) and "division" (`/`) operators:

- Adding integers corresponds to real addition e.g. `2+3` gives `5`
- Adding strings corresponds to concatenation e.g. `'a'+'b'` returns `'ab'`
- Dividing an integer by another non-zero integer corresponds to floor division i.e. `5/2` computes $\lfloor 5/2 \rfloor$ which is `2`
- Dividing a float by a non-zero integer corresponds to real division i.e. `5.0/2` returns `2.5`

Make sure you test out what an operator does before you use it. The results can sometimes be unexpected. For example, while running some faulty code where the object you're manipulating changes type, an overloaded operator would accommodate for this change of type, not throw you back an error and end up returning an output you did not expect. Thus, it's useful to exercise **type discipline** and be wary of switching types on the fly.

2.2 Variables and Assignment

We'll now take a short detour to explain the important concept of variables in Python.

- A **variable** is a name given to a storage area (place in memory) that our programs can manipulate.
- An **assignment statement** binds a name to an object. The syntax for an assignment is

  ```
  variable_name = initial_value
  ```

  Each variable has a type which is inherited from its value.

Variables make code more readable and if your code makes multiple references to a name instead of of an object, you don't need to change every object literal. You simply need to change the object that the variable points to.

Unlike other languages, we need not declare the type of a variable in Python. In fact, the type of a variable changes depending on the current value of the object it points to. Thus, we say that Python is a **dynamically typed** language.

```
# We initialise a variable named x which points to an object of type int
    and value 5
x = 5
type(x)
# The interpreter will return <class 'int'> here
# Now we make the variable name x point to a different object of type str
    and value 'abc'
x = 'abc'
type(x)
# The interpreter will return <class 'str'>
```

As an aside, when we write a sharp symbol `#` on a line of code, anything that follows it will not interact with any of the other code. We call these comments. They are there to explain the code to the reader.

2.3 More Data Types

Earlier, we gave some brief descriptions of the values primitive data types can take. We also alluded to some of the operations they support. All data types discussed thus far are built into the Python language. We call such data types **built-in**.

Now we move onto some more sophisticated/composite (but still built-in) data types. For brevity, there are only brief descriptions of the following data types. Any Python tutorial will have more information.

A **collection** or **container** is a grouping of any number (including 0) of items.

There are four data types used for grouping items in Python so that they can be operated upon in some controlled fashion:

- A **list** (`list`) is an ordered (indexed) collection of items that is changeable and and allows for duplicate entries. We create a list with square brackets `["hello", "world", "hello"]`. We can access items in a list `L` with square brackets. For example, the first item in `L` is referred to as `L[0]`, the second is `L[1]` and so on.
- A **tuple** (`tuple`) is an ordered and unchangeable collection of items. Tuples allow for duplicate members. For example, `(2,2,4,5)` is a perfectly valid tuple. A tuple with a single element is written as `(2,)` for example.
- A **set** (`set`) is a collection of items that is unordered and unindexed. Like mathematical sets, duplicates are not allowed. For example, Python interprets the set

{2,3,4,2} as {2,3,4}.

- A **dictionary** (dict) is an ordered (for Python 3.7+) collection of key: value pairs e.g.

```
pirate_dict = {
    "name": "Teach",
    "position": "Emperor",
    "age": 40
}
```

  Dictionary items can be accessed similarly to lists but we use keys instead of integers e.g. example_dict["name"] will refer to the string "Teach". We can change, add and remove items from a dictionary after it's been created. Dictionaries do not allow for two items to have the same key. An important restriction on the keys you choose is that they must be immutable data types.

2.4 Mutable and Immutable

- A **mutable** object's content (value) can be changed without changing its identity.
- An **immutable** object's value cannot be changed after it's created. Attempting to modify an immutable object's value will result in the variable name being bound to a different object with the desired value.

Recall that the identity of an object can be thought of as its address in memory. The identity of an object can be revealed by typing id() and placing an object's literal (or a variable name) between the parentheses e.g. id('abc') returns 4474752688 on my system.

We can test the mutability of an object by using id(). For example, on my system:

```
s = 'abc'
id(s)                       # returns 4383489712
s += 'xyz'
print(s)                    # prints out abcxyz
id(s)                       # returns 4385098032

x = [1,2]
id(x)                       # returns 4385029312
x.append(3)
print(x)                    # prints out [1, 2, 3]
id(x)                       # returns 4385029312
```

We can see that the location in memory of the string variable s changes when we try to change its value. This is because the string 'abc' has a fixed place in memory and cannot be changed. Thus, the variable name `s` must point to a different object when we perform line 3. This means that strings are immutable. However, the identity of the `list` object `x` does not change when we change its value. This is because list objects are a mutable data type.

Immutable types include `bool`, `int`, `float`, `str` and `tuple`.

Mutable types include `list` as seen above, `dict` and `set`.

We can convert an object of one type to another using built-in functions like `float()`, `int()`, `str()` etc. as long as the conversion is sensible to Python. For example, trying to convert a list `[1,2]` into an integer raises an error:

```
TypeError: int() argument must be a string, a bytes-like object or a number, not
'list'
```

Now that we've familiarised ourselves with how data is represented in Python and the types of data that exist, we move onto the next fundamental aspect of writing an algorithm - how to control the flow of execution.

2.5 Control Flow

Python offers several control structures that allow for controlling the flow of execution. These are usually written in code in the form of keywords that Python recognises as a means to change the order of execution or a way to raise an exception/error and halt the program entirely.

A **keyword** is an identifier that is reserved for the language. They cannot be used as ordinary identifiers. Examples include `if`, `return`, `while`, `for` etc.

- Python supports logical comparisons like equality `a == b`, inequality `a != b`, inequalities like `a <= b`, `a > b` etc. We can combine these with `if` statements to emulate conditional logic. The keyword `elif` offers more conditions to test should the prior (above) tests fail and `else` will execute the code it encompasses if all tests fail.

```
x = 20
y = 40
if x < y:
    print("x is less than y")
elif x > y:
    print("x is more than y")
else:
    print("x is equal to y")
```

- The `while` loop is the most basic loop. It repeatedly executes a block of code while the given condition written immediately after the keyword `while` is `True` e.g.

```python
a = 1
while a <= 10:
    print("Hello")
    a += 1
```

Notice here that the loop terminates when the value of `a` exceeds 10 because that is precisely when the condition `a <= 10` becomes `False`. This while loop will print the string `"Hello"` 10 times.

- Another type of loop is the `for` loop. This type of loop iterates over a collection of items and executes the indented block of code for each item until the entire collection has been traversed. As an example, if one would like to iterate over a range of integers, we can use the `range()` function.

```python
for i in range(0,10):
    print(i)
```

This loop will print the numbers 0 through 9 (inclusive).

2.5.1 Multiple Conditions

In the case that I'd like to check that an object satisfies several conditions, using logical operators like `and` or `or` becomes unwieldly above 3 or 4 conditions. This is where the `all()` and `any()` functions come in handy.

- `all(collection)` returns `True` if all items in your collection are `True`
  e.g. `all([False,True,True])` returns `False`.
- `any(collection)` returns `True` if any item in your collection is `True`
  e.g. `any([False,False,True])` returns `True`.

To keep my code succinct, I usually pair these with list comprehensions. List comprehensions are a Pythonic way of creating lists. I'll explain what these are in the miscellaneous section at the very end of these notes.

2.6 Exceptions

These are everywhere in Python. If you've ever written a program, you've encountered one before. For example, trying to find the 12th item `test[11]` of a list with only 2 items `test = [1,2]` yields an `indexError`. Anything that ends in `Error` is a type of **exception** in Python. In particular, `indexError` is an example of an unhandled exception; one which causes a program to crash.

As one would expect, there are mechanisms for handing exceptions in Python. Sometimes programs are written with the intention to raise an exception so we can catch it and do something useful with it. One way we can do this is via a **try-except block** of code:

```
try:
    test_block
except:
    block_if_error
```

If `test_block` doesn't encounter any errors, the `except` block is skipped and the program continues to execute the rest of the program below the block.

We frequently use exceptions when a program takes a user input.

2.6.1 Defensive Programming

The keyword `assert` followed by an expression e.g. `assert 2 + 3 == 5` is an expression that evaluates to `True` or `False`. If it evaluates to `True`, nothing happens and the program continues as normal. In the event that it reduces to `False`, this serves as a way to terminate the program.

Adding assertions on the actual parameters to which formal parameters are bound acts as:

- a form of defensive programming to prevent the program from continuing when a user inputs something nonsensical, or
- as a way for a programmer to check if a value at a certain point in a code is as they expect for when it sometimes isn't clear if the code is transforming variables in an expected way.

Chapter 3

Machine Interpretation of a Program

3.1 Modules

Whenever one quits the Python interpreter, all the definitions that one has made are lost. To make up for this, programmers write programs in text editors and save them as files for the interpreter to run. Such a file is called a script. In Python, such files have the extension `.py` and we also call them modules.

A **module** is a file that contains Python code.

Modules can be imported into other modules using import statements like `import math`. The `math` module is a collection of Python definitions and statements that model mathematical operations and objects. Modules are useful for grouping related information into a single file.

We access information in modules via **dot notation**. For instance, `math.log()` is how we access the command `log()` which is a model of how one computes the natural logarithm of a positive real number.

Dot notation offers **disambiguation**. Consider two different modules named `moda` and `modb`. Suppose that each module contains a variable named `elem`. We can avoid a name conflict by directly accessing each from its respective module i.e. `moda.elem` and `modb.elem`

There is a built-in function called `dir()` that takes a module name as its argument and lists all the information inside said module. This is very useful.

3.2 Functions

As programs increase in scale, the likelihood for errors also increases and a need for a clear structure arises. One way of addressing this is with the implementation of functions.

A **function** is a self-contained block of code, usually in the form of related statements, that performs a particular task.

Before we show the syntax for defining a function, we first list a few reasons why they are useful:

- Decomposition/Modularisation - Functions allow us to divide our programs up into pieces, giving them structure.
- Reusability - Once a function is defined, it can be used as many times as we'd like (and even in other programs).
- Abstraction - Functions suppress details and allow us to call a function as though it were a black box - we need only know how to use it (e.g. its name, what it does, valid parameters one can pass and an expected output) and not what the inner workings do.

The syntax for defining and calling a function are as follows:

```python
def function_name(formal_parameters):
    """check it out!! I am an optional docstring to explain how to use the
    function"""
    some_commands
    return something

function_name(actual_parameters)
```

In the first line of the function definition, one can choose to include an optional string literal called a **docstring**. These serve to summarise the behaviour of the function, document its arguments, what it returns and any exceptions it raises. These are conventionally written within triple quotes and the docstring appears as a popup in an IDE when one is calling the function.

We now look to a basic example to explain the other terms in a function. This example takes in a number as input, adds 1 to it and returns the new number. After the function definition, a variable `x` is bound to an object of type `int` with value `3` and the function is called on this variable and the result is printed out.

```python
def f(x):
    ans = x + 1
    return ans

x = 3
print(f(x))
```

- The `def` keyword lets Python know that we are defining a function and Python does its magic in the background (which we shall uncover soon).
- The `x` written in the first line is called a **formal parameter** - it is simply a name bound to an object with no value.
- The `return` keyword tells the interpreter to immediately terminate the function and passes execution control back to the caller. It also provides a mechanism by which the function can pass data back to the caller.
- When `f` is called, the formal parameter `x` is bound to an **actual parameter** or **argument**. In this case, the actual parameter is the variable `x` which refers to the

`int` object with value `3`.

> **Important aside:** There are two types of arguments. Consider the following function.
>
> ```python
> def difference(first, second):
>     return first - second
> ```
>
> **Positional arguments** are passed into a function based on the exact order in which the formal parameters are defined after `def`.
> - Calling `difference(2,1)` returns 1 and `difference(1,2)` returns −1.
>
> **Keyword arguments** are passed into a function and are identifiable by specific formal parameter names. Unlike positional arguments, the order in which keyword arguments are passed into a function is irrelevant as long as you assign values to the appropriate parameter names.
> - Calling `difference(second=2, first=1)` returns −1.

Notice that `x` is the name given to two different things. How does Python not get confused?

3.3 Namespaces

In any given program, one will create multiple identifiers (names for objects). Naturally, there has to be a system in place to keep track of all these names so they don't interfere with each other. Python does this with the ideas of namespaces and scope.

A **namespace** is a mapping from names to objects.

Whenever a variable is defined, Python needs a way to remember both its name (identifier) and the object it's pointing to. Python does this by storing them in a dictionary (a set of key:value pairs) where the keys are the currently defined names and the values are the objects themselves. This dictionary is an example of a namespace and it serves as a lookup table. Whenever a name is referenced, the interpreter searches the namespace for the corresponding object. If found, it returns the object. Otherwise, you receive a `NameError`.

3.3.1 Lifetimes, Creation and Types

Namespaces have different lifespans and are created at different times.

There are 4 types of namespaces in Python:

1. The **built-in** namespace:

   This comprises of all the names of Python's built-in objects. The list of these objects can be accessed with `dir(__builtins__)`

2. **Global** namespace(s):

   This contains all names defined at the level of the main program. This namespace is created when the Python interpreter runs a script (which is known as the main module). This namespace is not unique and when another module is imported, the interpreter creates a global namespace for it too.

3. **Local** namespaces:

   A new namespace is created whenever a function executes. The benefit of this is that variables may be defined and used within a function even if they share a name with a variable defined in another function or the main program. Such namespaces are discarded when the interpreter exits the function.

4. **Enclosing** namespaces:

   When two function definitions are nested e.g.

   ```
   def f():
       def g():
   ```

   we call the namespace generated with `f` an enclosing namespace.

3.4 Scope

Despite modules having their own respective global namespaces, not all names can be accessed from anywhere in the module. This is where the idea of scope comes in.

A **scope** is a textual region of a Python program where a namespace is directly accessible (without a prefix).

The location of a name's assignment in one's source code determines the namespace it will live in, and hence the scope of the name's visibility to one's code.

At any time in a program that has at least one function defined, there are at least 3 nested scopes:

- The scope of the current function you're in
- The scope of the module
- The scope of the Python built-ins

When an unqualified (prefix-less) name is used inside a function, Python will search up to four namespaces (inside-out) in the following order:

1. **Local** - the local namespace (of the current function),
2. **Enclosing** - the namespaces of any enclosing `def` and `lambda` functions,
3. **Global** - the global namespace, and finally

4. **Built-in** - the built-in namespace.

This is called the **LEGB** rule of scoping.

The discussion of namespaces and scope is a nice place to stop before things get too complicated. We can now begin to focus on problem solving and learning about algorithms. We'll return to the discussion of implementations and creation of data types in chapters 6 and 8-9.

Chapter 4

Problem Solving

4.1 Enumerative Exhaustion

Most programs can be solved with brute force. One of the most basic types of brute force algorithms is a process called enumerative exhaustion. This is done with a basic `for` loop that iterates through a collection (list, tuple, dictionary, set). As one can imagine, brute force algorithms are not always the most efficient. To this end, we consider a different type of searching algorithm:

4.2 Bisection Search

The idea of a bisection search is to halve the search space at each iteration of a loop.

e.g. Finding an approximation for the square root of a number x.

```python
x = float(input("Input a number to approximate its square root: "))
y = float(input("What's your margin of error?: "))

guess = x/2
lower = 0
upper = x

while (abs(x - guess**2) >= y) and (guess <= x):
    if guess**2 > x:
        upper = guess
        guess = (lower+upper)/2
    else:
        lower = guess
        guess = (lower+upper)/2

print(f"The square root of {x} that we found with a tolerance of {y} is
    {guess}.")
```

The built-in function named `input()` prompts the user for an input. We then convert `x` and `y` to floats so that we can perform float operations on them.

We begin by making an initial guess g_0 which is the midpoint $x/2$ of the search interval $[0, x]$. If our guess is too high (or too low), we redefine the bounds of the search interval to remove half of the search space that definitely will not contain our desired output. If our guess is within the user's chosen margin of error, we terminate the loop. Otherwise, we iterate over the loop again using the remaining half of the search interval.

Once the loop completes, we print out what is called a formatted string literal (or **f-string** for short). This is a succinct way of formatting a string so we can incorporate variable values directly into the string. We put variable names inside curly parentheses so they print out mid-string.

When using any search method, we're relying on the fact that the answer lies in the region we are searching.

The algorithm above does not work for numbers $x \in (0,1)$ because $\sqrt{x} > x$ in this interval and our algorithm relies on searching for a root in $[0,x]$. Take `x` as $1/4$ for example. We'd be searching in $[0,1/4]$ but $\sqrt{1/4} = 1/2$ which isn't in the search interval.

4.3 Recursion

Recursion is the way of defining a problem/process (or designing the solution to a problem) in terms of a simpler version of itself.

Recursion fits in well into the idea of problem reduction.

Consider the exponential function $f\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ with base $b > 0$ defined by $f(b,n) = b^n$. We can redefine f recursively by writing:

$$f(b,n) = b^n = b \times \underbrace{b \times \cdots \times b}_{(n-1)\text{ times}} = b \times b^{n-1} = b \times f(b,n-1)$$

How can we guarantee that this recursive definition doesn't nest infinitely? There's nothing in our above definition to stop the second argument from descending $n-1 \mapsto n-2 \mapsto ... \to -\infty$. To rectify this, we define a base case for our recursion - a point at which the function is easy to calculate and the recursion terminates once it's reached.

Since n is a natural number and we're only interested in multiplying b by itself n times, the base case for f can be taken when $n = 0$ (so $f(b,0) = 1$).

The combination of this base case and the recursive step guarantee that the recursion terminates and so f is defined for all natural numbers n.

$$f(b,n) = b^n = \begin{cases} 1, & n = 0 \quad \text{base case} \\ b \times f(b,n-1), & n > 0 \quad \text{recursive/inductive case} \end{cases}$$

In terms of a loop, we write:

```python
def f(b, n):
    if n == 0:
        return 1
    else:
        return b*f(b, n-1)
```

We now consider a popular example of recursion.

4.3.1 The Tower of Hanoi

There is a temple in Hanoi. In this temple are 3 tall jewel encrusted spikes and there are 64 golden disks in total - all of increasing size. The stack starts out on one of the spikes, starting with the largest on the bottom and uniformly decreasing with size until the smallest disk on the top. The goal is to move all the disks to another spike but the caveats are that:

- you can only move on disk at a time, and
- you cannot move a disk that is larger on top of a disk that is smaller.

Label the spikes:

- `f` for 'from' - the spike we begin at
- `t` for 'target' - the destination spike
- `s` for 'spare' - the spike used for intermediate steps

This problem can be approached recursively. Suppose that there are n disks to move.

The base case is moving a single disk from `f` $\to$ `t`.

The recursive step comes from noticing that the problem can be broken up into a sub-problem of the same type:

- We first move the stack of size $n-1$ from `f` $\to$ `s` using the target spike `t` as an intermediary.
- Then we move the bottom disk from `f` $\to$ `t`.
- Finally, we move the stack of size $n-1$ back from `s` $\to$ `t` using `f` as an intermediary.

```python
def hanoi(n, f, t, s):
    if n == 1:
        print(f"Move from {f} to {t}")
    else:
        hanoi(n-1, f, s, t)
        hanoi(1, f, t, s)
        hanoi(n-1, s, t, f)
```

As you might imagine, increasing the number of disks, n, increases the number of movements required and so it increases the time taken to solve. A natural question to ask is how many steps it takes to solve the problem and how does it vary as the number of disks varies. We begin to formulate the language to describe this in the next section.

Chapter 5

Time Complexity and Order of Growth

You've coded a solution to your problem. Fantastic! You know it works because ~~you're a genius~~ it was an easy problem. You hit enter!

. .. .

It's been 2 hours and your laptop can now cook an egg. What went wrong? Your algorithm actually needed 98763457 years to complete.

A main focus of programming is getting your algorithms to complete in good time. The program needs to be more efficient. Perhaps you made the same calculation many times when you could've done it just the one time.

Ideally, a program will complete as quickly as possible and be as conservative with memory consumption as it can. Sometimes this is achieved with a few lines of clever coding. More often it's about choosing the correct algorithm to solve your problem.

An obvious starting point is to study how long an algorithm takes to run.

> *"My machine is faster than yours so it clearly takes 2 minutes instead of the 5 it took you."*
>
> *"We were both looking for the number 3 and my list began with 3 so this algorithm is very fast."*

Clearly there's something amiss here.

How long a program takes to run on a single machine with a certain input is dependent on:

- the speed of the machine,
- the language one has coded the solution in,
- the implementation of the language, and
- the input itself.

Therefore, we should try and find a stable measure (independent of all the factors in the list above) of how long a program takes to run. The correct question to ask then seems to be less about the time taken but perhaps in terms of the number of basic steps needed in relation to the input size.

Thus, we need to define two things:

- The **input size** is a number that represents the size of an object passed into an algorithm. The input depends on the type of problem (e.g. number of elements in a list, length of a string) so we need to be clear about specifying what we mean by input size.
- A **step** is an operation that completes in constant time.
  e.g. arithmetic operations, assignment, comparisons, memory access etc.

5.1 The Random Access Model

Despite not being completely true, for our purposes we'll be using a model of a computer known as a **random access machine (RAM)**. In such a machine, instructions are executed sequentially and we assume[1] that the time taken to access any object from memory at random is constant.

5.2 Running Time

The running time of an algorithm can be approached in three different ways. Take a linear search (sequentially checking elements in a collection) for example:

Best Case: The minimum running time over all possible inputs.

- Say the first element in the list is 3 and we're searching for 3. We find it right away and stop.

Worst Case: The maximum over all inputs of a given size.

- The worst case for linear search is that the element we're searching for is not in the space.

Expected Case: The average over all possible inputs.

- The expected case seems like the one we should most care about but in practice it's too hard to compute. To compute it, we need to know a distribution on the inputs - are all the inputs equally likely or are they going to depend on other things? The input distribution depends on the user so we throw that out.

As a result, analysis almost always focuses on the worst case scenario which provides an upper bound for how long our programs take to run (so there are no surprises).

We aren't only interested in the running time for inputs of fixed size. More often than not, programs are designed to scale for very large data sets. We use the word **complexity** to describe the relationship between the growth of the input size and any quantities that change as a result.

[1]In old computers, this model wasn't accurate because memory was often a tape and reading something at the end of the tape took much longer than at the start of the tape. Modern computers, on the other hand, have a memory hierarchy (levels of memory - cache and actual memory) and depending on whether or not data is in the cache, the retrieval time changes. The idea of processes happening in parallel is another viewpoint that adds extra detail but for practicality, such details are omitted.

We think about complexity in two dimensions - space and time.

- The **time complexity** of an algorithm is the relationship between the growth of the input size and growth of operations executed.
- The **space complexity** of an algorithm is the relationship between the growth of the input size and growth of space needed.

5.3 Calculating Complexity

Complexity calculations are done inside-out. Namely, if you have nested functions, one begins from the innermost loop and works their way outwards.

5.3.1 Loops

Consider the exponentiation function implemented with a `while` loop defined below: code

```python
def exp_while(a,b):
    ans = 1
    while (b>0):
        ans *= a
        b -= 1
    return ans
```

How many steps does this function take to run?

The key part is inside the while loop. Each time the loop executes, 3 steps are performed: a comparison `(b > 0)` and two arithmetic operations `ans *= a` and `b -= 1`. We go through the loop `b` times so this is $3b$ steps in the loop. Outside of the loop, we have an initialisation and a return keyword so these are 2 more steps, giving a total of $3b + 2$ steps.

5.3.2 Recursive Algorithms

In the case of a recursive algorithm, we can calculate the number of steps it takes to exponentiate by solving a recurrence relation.

```python
def exp_rec(a,b):
    if b == 1:
        return a
    else:
        return a*exp_rec(a,b-1)
```

Let $T(b)$ denote the number of steps it takes to solve the problem of size b. In $T(b)$ we have 1 comparison test, a subtraction and a multiplication plus the number of steps it takes to solve a problem of size $b - 1$.

$$\begin{aligned}
T(b) &= 3 + T(b-1) \\
&= 3 + 3 + T(b-2) \\
&= \dots \\
&= 3k + T(b-k) \\
&= \dots \\
&= 3(b-1) + T(1) \qquad \text{the recursion terminates when } b - k = 1 \\
&= 3b - 1
\end{aligned}$$

As the input size grows, we see that additive constants don't contribute greatly to the growth of the running time. The same goes for constant coefficients. Ideally, we'd like a (mathematical) language that helps us characterise growth in such a way. To this end, we introduce asymptotic notation.

5.4 Asymptotic Notation

Asymptotic notation is used to give a quick measure of a function's $f(x)$ behaviour compared to a simpler function $g(x)$ as x grows large. The language of asymptotic notation finds a natural place in describing how the running time (or space requirements) of an algorithm grows as the input size grows.

For our purposes[2], we'll let $T\colon \mathbb{N} \to \mathbb{R}^+$ denote the running time of an algorithm and $f\colon \mathbb{N} \to \mathbb{R}^+$ be the comparison function.

We list below the precise mathematical definitions followed by their informal descriptions.

The notation $f \in \blacksquare(g)$ is to be read as "f is in $\blacksquare$ of g".

[2]The general definitions involve letting U be an unbounded subset of $\mathbb{R}^+$, $f\colon U \to \mathbb{C}$ and $g\colon U \to \mathbb{R}$ be such that $g > 0$ for all sufficiently large $x \in U$. Then, for example, f is in big-O of g if $\exists x_0, C > 0$ s.t. $x > x_0 \implies |f(x)| \leqslant Cg(x)$.

- $T(n) \in \mathcal{O}(f(n))$ if
$$\exists n_0 \in \mathbb{N} \text{ and } \exists C > 0 \text{ such that } n > n_0 \implies |T(n)| \leqslant Cf(n).$$
- $T(n) \in \Omega(f(n))$ if
$$\exists n_0 \in \mathbb{N} \text{ and } \exists c > 0 \text{ such that } n > n_0 \implies c|f(n)| \leqslant T(n).$$
- $T(n) \in \Theta(f(n))$ if
$$\exists n_0 \in \mathbb{N} \text{ and } \exists c_1, c_2 > 0 \text{ such that } n > n_0 \implies c_1 f(n) \leqslant |T(n)| \leqslant c_2 f(n).$$
- $T(n) \in o(f(n))$ if
$$\forall C > 0 \ \exists n_0 \in \mathbb{N} \text{ such that } n > n_0 \implies |T(n)| \leqslant Cf(n).$$
- $T(n) \in \omega(f(n))$ if
$$\forall c > 0 \ \exists n_0 \in \mathbb{N} \text{ such that } n > n_0 \implies cf(n) \leqslant |T(n)|.$$

Table 5.1: An informal description of asymptotic notation.

$f \in \mathcal{O}(g)$	f grows asymptotically no faster than g
$f \in \Omega(g)$	f is bounded below by g asymptotically
$f \in \Theta(g)$	f grows asymptotically as fast as g
$f \in o(g)$	f is dominated by g asymptotically
$f \in \omega(g)$	f dominates g asymptotically
$f \sim g$	$f/g \to 1$

In practice, since we're concerned with behaviour for large x, multiplicative and additive constants are usually subsumed e.g. $10n + 5 \in \mathcal{O}(n)$.

Definition 5.4.1 (Asymptotic Equality) *We say that f and g are asymptotically equal, $f \sim g$, if*
$$\lim_{x \to \infty} \frac{|f(x)|}{g(x)} = 1.$$

To say that $f \sim g$ means both functions grow at the same rate in a stricter sense than $f \in \Theta(g)$. This notation is usually reserved for when the growth of a function is well-understood up to some small error terms e.g. Stirling's Approximation $n! \sim \sqrt{2\pi n}(n/e)^n$.

5.5 Efficiency

Say we have an algorithm that completes in constant time, regardless of input size n. Such an algorithm's running time[3] $T(n)$ is said to be in $\mathcal{O}(1)$. Some algorithms depend linearly on the size of the input e.g. for a linear search through a list, $T(n) \in \mathcal{O}(n)$.

Table 5.2: A list of common running times $T(n)$.

$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log(n))$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n\log(n))$	Log Linear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^a)$ for $a \in \mathbb{Z}$	Polynomial
$\mathcal{O}(a^n)$ for $a > 0$	Exponential
$\mathcal{O}(n!)$	Factorial

If your input size is large, a general rule of thumb is to avoid doing anything that costs more than log linear time.

[3]In practice we use $\mathcal{O}$ notation e.g. $T \in \mathcal{O}(f)$ and suggest that the worst case growth of T is f. Any other function that grows faster than f can in principle be used. The way $\mathcal{O}$ notation is used in practice mimics the meaning of Θ slightly more.

Chapter 6

Search Methods

A very important type of algorithm is one which searches a set of data. These are called searching algorithms.

6.1 Linear Search

Suppose we'd like to search a collection of integers that is sorted in ascending order. The following algorithm searches through a list `L` for an element `e` and outputs `True` if the element is found and false otherwise.

```
def linear_search(L, e):
    answer = None
    list_length = len(L)
    while i < list_length and answer ==  None:
        if e == L[i]:
            answer = True
        elif e < L[i]:
            answer == False
        i += 1
    return answer
```

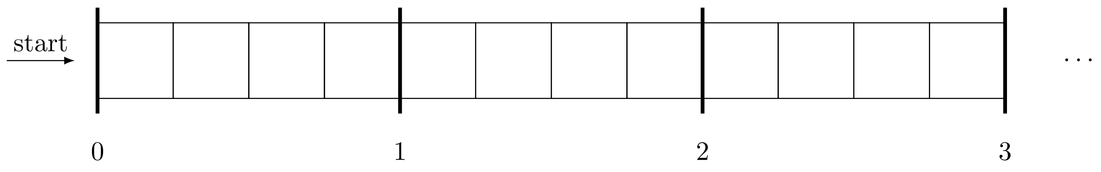
Assuming that list access is a primitive operation (one that completes in a single step), the complexity of the linear search is linear i.e. $T(n) \in \mathcal{O}(n)$ where n is the length of the list `L`.

However, not all programming languages implement lists in the same way and so list access may not be constant.

6.2 Aside: How Python Implements Lists

6.2.1 Elements of Fixed Size

Say we have a list of integers. Let's say, to allow a fairly large range of integers, each integer takes up four memory cells in a row. Suppose that the first element is located at `start`.

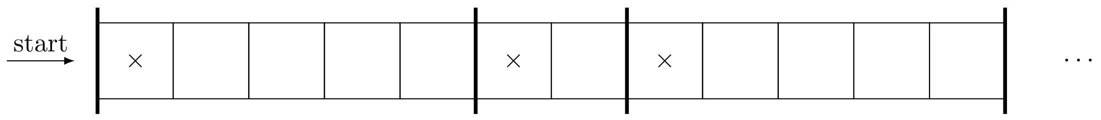


To access the n^{th} element, we can see that it begins at the memory location that is at `start + (4*n)`. Assuming the random access model, it takes a constant time to access a point in memory given its location.

This works because our list is composed of a single data type of equal size. This is not true for a general list. Often, we'd like to include a variety of information types in a list. What if our list contains data structures of differing size?

6.2.2 Elements of Variable Size (Linked Lists)

One of the standard ways to implement a general list is to use what is called a **linked list**.

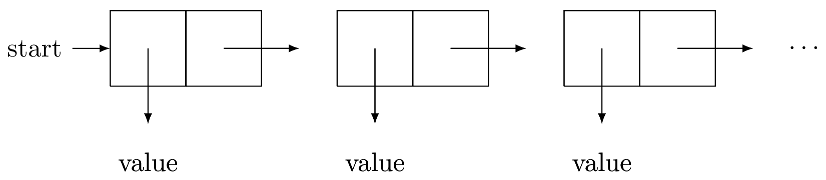


In the first memory cell of each list item, we store a pointer to the beginning of the next item in the list. We mark this with an $\times$. The remaining memory cells are reserved for storing the objects themselves. For the final linked list entry, one puts a value `None` as the final pointer to signify the end of the list.

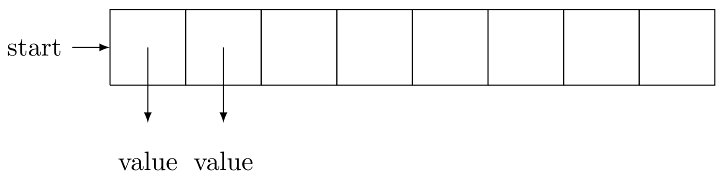
One of the trade-offs with a linked list is that list access is not constant. Indeed, to find the n^{th} element of such a list, one would have to traverse all pointers leading up to the element itself. Thus, the complexity of list access is linear. This is not what we want.

6.2.3 Python's List Implementation

Python stores collections in a different way with the key idea being **indirection** - the ability to reference something using a name instead of the value itself.



Each chunk of memory contains a pointer to the value of the object, and a pointer to the next chunk of memory. We can reorder this diagram by taking all the first cells and sticking them together so our list becomes a list of pointers.



This is a nice setup because it mimics the original list in **6.2.1** where each element has a fixed size in memory that we could search in constant time.

6.3 Binary Search

An example of an algorithm running in logarithmic time is the bisection search from **4.2**. When the output of a bisection search is a precise result in a list, we call it a **binary search**.

Take, for example, the problem of finding a particular word in the dictionary. In pseudocode, such an algorithm goes as follows:

Algorithm 2 Pseudocode algorithm for finding a word in a book via binary search.

```
1: procedure BINARYSEARCH(dictionary, word)
2:     pick up the dictionary
3:     open the current dictionary to the middle page
4:     search the page for word
5:     if word is in the page then
6:         stop and return current page
7:     else if word is earlier in the dictionary then
8:         open the left half of the dictionary
9:         go back to line 2
10:    else if word is later in the dictionary then
11:        open the right half of the dictionary
12:        go back to line 2
13:    else
14:        quit
15:    end if
16: end procedure
```

Cutting the search window in half for each step in an algorithm means that doubling the input size only increases the depth of recursion by 1. This is characteristic of algorithms with logarithmic running times[1] $T(n) \in \mathcal{O}(\log n)$. For the binary search to work, however, the collection needs to be **sorted**.

[1]We omit the base of a logarithm because in asymptotic analysis, everything is modulo a constant multiplier and the change of base rule for logarithms means that different bases differ by a constant multiple e.g. for constants a, b:

$$\log_a(n) = \frac{1}{\log_b(a)} \log_b(n).$$

Chapter 7

Sorting Algorithms

We've already seen an algorithm that depends on a list being ordered, the binary search. Other algorithms, like the linear search, don't require a sorted input.

Suppose that we could find an algorithm to order a list and that it has a time complexity of order $S(n)$. Would it be faster to order a list and use the binary search to find an element, or to abandon sorting it and search it in a linear fashion? We'd ideally like our sorting algorithm to perform in sub-linear time if the sort and binary search is to have a better time complexity.

Can we sort a list in sub-linear time? Seeing as we need to access each list element at least once in order to sort the list, a sub-linear time is not possible. The same goes for linear time since sorting usually involves iterating over a list and visiting each element at least a constant number of times for comparison purposes.

Another important question to ask is how many times we wish to search a collection and whether sorting it once beforehand makes much of a difference to the overall time complexity.

7.1 Amortisation Analysis

It's usually the case with searching algorithms that we wish to search a collection several times. If we can sort our list once and search it many times, the cost of the sort can be split over the searches and so is not very significant in the overall complexity. This amortises the cost of the sort.

For k searches, the time complexities are as follows:

sort + k binary searches	$\mathcal{O}(S(n) + k\log(n))$
k linear searches	$\mathcal{O}(kn)$

We'll proceed to find the time complexities of a few sorting algorithms and come back to this table to replace $S(n)$ with a suitable expression and compare these two search methods.

7.2 Selection Sort

Given an unsorted array `A`, a selection sort goes through the array and at each index `i` in `range(0, len(A)-1)` of the array, sets `A[i]` as a pivot and checks the remaining elements of the array on the right for the smallest number less than the pivot. If such an `A[j]` for `j in range(i+1, len(A))` is found, one swaps `A[j]` and the pivot `A[i]`. One then moves onto the next index and repeats the same comparison process on the rest of the unsorted list.

```python
def selection_sort(A):
    L = len(A)
    for i in range(L-1):
        current_min = i
        for j in range(i+1,L):
            if A[j] < A[current_min]:
                current_min = j
        if current_min != i:
            A[current_min], A[i] = A[i], A[current_min]
    return A
```

The order of growth for the selection sort is in $\mathcal{O}(n^2)$ because the worst case scenario is performing a linear scan through the list for each of the n elements.

7.3 Merge Sort

Given an unsorted array `x`, the idea of a merge sort is to break[1] `x` down into n subarrays of length 1 (which are sorted by definition) and recombine them in the correct order. The first order of affairs is to define a function called `merge` that takes two sorted lists `a` and `b` and combines them into a single sorted list. The method used to define `merge` is called the two finger pointing method. Then we can recursively define `merge_sort` which is called to `merge` the left and right "halves" of the array `x`.

```python
def merge(a,b):
    c = []
    a_index,b_index = 0,0
    while a_index < len(a) and b_index < len(b):
        if a[a_index] < b[b_index]:
            c.append(a[a_index])
            a_index += 1
        else:
            c.append(b[b_index])
            b_index += 1
    if a_index == len(a):
        c.extend(b[b_index:])
    else:
        c.extend(a[a_index:])
    return c

def merge_sort(x):
    if len(x) <= 1:
        return x
    left,right = merge_sort(x[:len(x)//2]),merge_sort(x[len(x)//2:])
    return merge(left,right)
```

[1] The space complexity of a merge sort is larger than a selection sort because we need to keep track of the extra structure.

7.3.1 Complexity Calculation

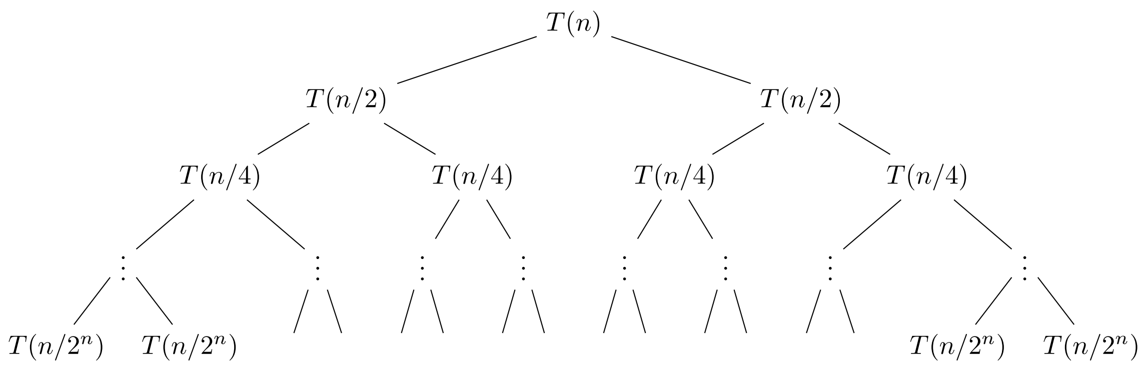
Let $T(n)$ denote the running time of the `mergesort` algorithm. The running times of the constituent parts of the algorithm are as follows:

- Dividing an array of length n into 2 subarrays of length $\approx n/2$ entails
  - calculating the midpoint which takes constant time,
  - creating two empty arrays of size $\lfloor n/2 \rfloor$ (or `n//2`) and $n - \lfloor n/2 \rfloor$ (or `n//2`), which is also constant, and
  - populating those arrays which has running time $c_1 n$ for some $c_1 > 0$.

  Thus, the running time for the divide step is $c_0 + c_1 n$ for some constants $c_0, c_1 > 0$.
- The `merge` function picks up one element from either left or right at a time and adds it to a new array. Thus, the complexity of merging two sorted arrays of size $\approx n/2$ into an array of size n is $c_2 n + c_3$ for some $c_2, c_3 > 0$.
- We have two recursive calls in `mergesort` which have a running time of $T(n/2)$.

$$\therefore \quad T(n) = \begin{cases} C & \text{if } n = 1 \\ 2T(n/2) + cn + c' & \text{otherwise, where } c' = c_1 + c_3, c = c_2 + c_4 \end{cases}$$

The tree diagram of time complexities is as follows:



Since each step in the recursion divides the array length by 2, the depth of the recursion (height of the recursion tree) is how many times we can divide n by 2 (plus the root node) i.e. $\log_2(n) + 1$. Starting from the root $n = 0$, the n^{th} row in the tree has 2^n items.

$$\begin{aligned}
T(n) &= 2 \cdot T(n/2) + c' + c''n \\
&= 2^2 \cdot T\left(n/2^2\right) + 2cn \\
&= 2^3 \cdot T\left(n/2^3\right) + 3cn \\
&= \dots \\
&= 2^{\log_2(n)} \cdot T(n/2^{\log_2(n)}) + \log_2(n) \cdot cn \\
&= nT(1) + cn\log_2(n) \\
&\in \Theta(n\log(n))
\end{aligned}$$

7.4 Amortisation Analysis 2: Electric Boogaloo

Log linear time complexity is the best we can do for a sorting algorithm.

This means that for a single sort, the linear search wins out with a linear time complexity $\mathcal{O}(n)$ compared to $n\log(n)+\log(n) = (n+1)\log(n) \in \mathcal{O}(n\log(n))$ for sorting and binary searching.

For k searches, the linear search has time complexity in $\mathcal{O}(k \cdot n)$, whereas the sort and binary search wins out with a running time of $n\log(n) + k\log(n) = (n+k)\log(n)$ $\in \mathcal{O}(n\log(n))$.

Chapter 8

Hashing

If a language didn't have dictionaries, one could build similar functionality using lists:

```
def keysearch(L, k):
    for elem in L:
        if elem[0] == k:
            return elem
        return None
```

So why bother with dictionaries? The answer lies in how long it would take to find an element in a list using such a function. On average, it would take `len(L)/2`. This isn't an efficient lookup time. Dictionary retrieval on the other hand is done in constant time (irrespective of list size).

Hashing is the method with which dictionaries are implemented in Python. They are very efficient in search time but the trade-off is that they require space.

We'll begin by assuming that we're hashing a set of integers (i.e. that we want to build a set of integers and want to detect whether a particular integer is in the set). Let $i \in \mathbb{Z}$. Define the following map

$$\begin{aligned} \text{hash}\colon \mathbb{Z} &\to \mathbb{Z}/k\mathbb{Z} = \{0, \dots, k-1\} \\ : i &\longmapsto \text{hash}(i). \end{aligned}$$

We'll use the integer $\text{hash}(i)$ to index into a list of lists e.g. `[[a,b,c,d], [e,f,g,h], [i,j,k,l]]`. Each of these constituent lists is called a **bucket**.

We've already seen that we can find the i^{th} element of a list in constant time. When we ask whether an integer z is in this set, we'll hash it (find $\text{hash}(z)$) and go immediately to the correct bucket and search the list for the integer.

- The hash function is a many-to-one function.
- When two different elements hash to the same bucket, we have what is called a collision.
- A good hash function has the property that it will widely disperse the values that you hash.

If the number of buckets is large relative to the number of elements we insert into this table, then lookup is roughly constant because the lists are shorter. This is the trade-off between space and time - the larger the space dedicated to a table, the less time it will take to search it.

Chapter 9

Classes

We've already seen that every object in Python has a type. With only the basic built-in data types provided by Python, we'd be limited in how we model and manage more complicated types of data. To this end, Python offers a means to extend our language and create a new, user-defined data type. The blueprint[1] for such a type is known as a class.

- A **class** is a user-defined blueprint for a new *abstract* data type for objects. More concretely, a class defines a set of data that characterises any object created from the class. Such data can be split into two categories:
  - **Attributes** are variables associated with a class.
  - **Methods** are functions associated with a class. We use these to interact with the object (and class) itself.

  We access attributes and methods via dot notation (as we did with modules).
- We use classes to create objects. The process of creating an object (or instance) is called **instantiation**.

Accordingly, classes allow us to logically group data and functions in a way that's easy to re-use and build upon. An important benefit of using classes is that if an object is passed from one part of a program to another, the new part of the program automatically has access to the functions associated with that type of object. This property is a fundamental idea for **object-oriented programming**.

We call them abstract data types because we define an interface that explains what the methods do at the level of the user and not how they do it. This is how built-in types work. For instance, we knew how to create dictionaries and use them but it wasn't until the last section that we understood how the `dict` type works with hashing. This is because the people who wrote Python provided an interface which explained how to use `dict`.

Some classes are so useful that somebody along the line decided that they should be part of the Python language itself, have efficient implementations, and that nobody should have to reimplement them. Examples of these include `list`, `dict` and `set`. These are called built-in classes.

[1]By analogy with car manufacturing, the class would be the design blueprints for a type of car, the creation step would be the building process, and the object is the end result - an instance of a car that abides by the blueprints.

9.1 Creating and Instantiating Classes

A class is created with a class statement which uses the keyword `class`. This is syntactically similar to `def` for functions. The indentation works the same too. Below, we create an empty class:

```python
class Coordinate:
    """optional docstring: this class is actually empty"""
    pass
```

To instantiate a class, we pass it as a variable.[2] The syntax is reminiscent of a function call.

```python
v = Coordinate()
w = Coordinate()
```

9.1.1 The `__init__` Method

We can manually assign attributes to objects with statements like `v.first_coord = 2`. You can imagine that with many objects, manually assigning attributes would repeat a lot of code. Instead, we can define a method within our class definition to initialise our objects with attributes. We use a special method for this called `__init__`.

```python
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When an object of type `Coordinate` is created, the `__init__` method is the first thing that is run and initialises[3] our object. Methods are defined just like functions outside of classes but with one key difference - they have an extra parameter, the placeholder `self`. You can use any name you'd like for this placeholder but we conventionally use `self` to refer to a particular instance of the class. We access `self` using dot notation, as you can see when we initialise the x and y coordinates in our object.

The variables `x` and `y` are specific to each object/instance. These are what we call **instance variables**.

Since we now have an `__init__` method in place, we can create new objects of type `Coordinate` with their desired coordinate attributes as follows:

```python
v = Coordinate(3,4)
w = Coordinate(0,0)
```

[2]We can also pass instances into functions and do everything else that we with basic data types. More on this later.

[3]Such a method is called a **constructor**.

The `__init__` method clearly takes 3 parameters. Where is `self` here? Well, Python implicitly understands that `self` is the object `v`. Thus, we need not declare it directly. This is a level of abstraction that makes classes more user-friendly.

9.1.2 Instance Methods

We can define other methods similarly. For example, we define the Euclidean distance between two `Coordinate` objects below:

```
    def euc_dist(self, other):
        x_diff_sq = (self.x - other.x)**2
        y_diff_sq = (self.y - other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

- `other` is a parameter that refers to another instance of the class (another object).
- `euc_dist` takes two objects of type `Coordinate` and returns the Euclidean distance between them defined[4] for vectors with n-coordinates by:

$$\mathrm{d}(v, w) = \sqrt{\left(\sum_{i=1}^{n} (v_i - w_i)^2\right)}$$

We can call `euc_dist` in two equivalent ways. Python understands both of the following calls:

```
print(v.euc_dist(w))
print(Coordinate.euc_dist(v,w))
```

9.1.3 Class Variables

What we've defined so far are instance variables - specific to each instance of the class. Classes also allow us to define variables that are shared by all objects of the same type. These are known as **class variables**. They are defined inside a class but outside of any methods in the class.

e.g. As an example, if one wanted to keep track of how many points are in our space, we could use a class variable that increments by +1 every time a new object is created. We do this by incrementing it within the `__init__` method because that's automatically called when a new object is initialised.

```
class Coordinate:
    num_of_points = 0
    def __init__(self, x, y):
        ...
```

[4]Note that $a_1 + \cdots + a_n =: \sum_{i=1}^{n} a_i$

```
        Coordinate.num_of_points += 1
```

We use dot notation to retrieve the class variable from the class itself. Not the particular instance. This is an important distinction.

9.2 Object Representation

Now we know how to initialise a class and define a method that interacts with objects created from the class. Let's say that you're debugging a geometry program and you want to print out a coordinate `v`. The console returns this:

```
<__main__.Coordinate object at 0x105f73580>
```

When `print` is used on a class object, Python calls a special method called `__str__` on the object. This particular default representation of our object `v` is not very informative and we'd like to define some ways to meaningfully represent our object.

In general, there are two types of representation of an object:

- `__str__` is a more user friendly representation of the object.
- `__repr__` is generally defined to return an unambiguous representation of the object. Namely, we define it to return a string literal of the exact string you would need to type into the console to create an instance of the object.

`__str__` is called automatically when we call `print()` on an object.

`__repr__` is called automatically when we call `repr()` on an object.

By re-defining `__str__` and `__repr__`, we override their previous implementations:

```
class Coordinate:
    def __init__(self, x, y):
        ...
    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"
    def __repr__(self):
        return f"Coordinate({self.x},{self.y})"

v = Coordinate(3,4)
print(v)
repr(v)
```

The output is as follows:

```
>>>  <3,4>
>>>  'Coordinate(3,4)'
```

9.3 Special (Dunder) Methods

So far we've been seeing a lot of methods that are surrounded by double underscores. We call them **dunder**[5] methods for short. In fact, we've seen a few more that have been hiding in plain sight, working in the background of high level syntax.

- When adding two objects together, the top level syntax `+` corresponds to the special method `__add__`. For strings, addition `+` corresponds to string concatenation.
- Another method we've seen is `len`. Typing `len('test')` returns `4` and what's going on behind the scenes is that a corresponding special method named `__len__` is being called on the `str` object `'test'`.

Now that we've seen a few examples of special methods, a pattern has begun to emerge. Every time we want to implement some custom behaviour on a Python object, we do it by implementing a `__function__` which ties to some top level syntax/function and we implement it in terms of that thing itself.

Take our `Coordinate` example.

```
class Coordinate:
    def __init__(self, x, y):
        ...
    def __add__(self, other):
        return Coordinate(self.x + other.x, self.y + other.y)
```

When we call `+` on two `Coordinate` objects, our code delegates back to `__add__` and `__add__` is implemented by calling `+` on the components of the coordinates.

When implementing a corresponding special method, it's good practice to look to the official documentation to see any caveats and conventions we should abide by. For example,

There are many more examples of dunder methods in the standard library.

[5]I prefer to call them **data model methods** because the first or second result in a search engine by searching 'data model python' is a link to the official documentation that lists such methods and tells you what they do.

Chapter 10

The Pillars of Object-Orientation

10.1 Abstraction and Encapsulation

Encapsulation is the idea of bundling together similar attributes and methods that operate on those attributes. A benefit of encapsulation is that it hides the complexity of code and allows one to prevent access to implementation details. **Abstraction** is a way to show only implementation details that are relevant to the user. In this sense, encapsulation enables us to implement one's desired level of abstraction.

Encapsulation is also a means to an end known as **data hiding**.

10.1.1 Data Hiding

Directly accessing variables of a class is generally discouraged and for good reason:

> Imagine that you got a message saying "IDLE has changed, please download a new version" and it had a new implementation of `list` that caused all your programs to stop working. You'd be pretty mad. This won't happen. Why? This is because your programs do not directly depend on how the developers of IDLE chose to implement the built-in type `list`. You've only programmed to the specification of these types, not their implementation. This means that the implementer can implement a built-in type however they please as long the code meets the specification that the user sees.

The minute a user goes in and directly accesses (and potentially alters) variables of the class, they've used things that do not appear in the specification laid out[1] for them and if the implementer changes something in the implementation, the program might break. To prevent this from happening, we can restrict, from outside, direct access of certain attributes/methods inside an object.

To "hide" an attribute, name it with a double underscore prefix. This will indicate that the attribute should not be directly accessed from outside of the class definition. Python isn't great at data hiding at all and doesn't have mechanisms to hide information like other languages.

Instead of accessing variables directly, one can define methods to retrieve and adjust class attributes. These are known in Python as **getters** and **setters**. In the context of our `Coordinate` class, we can define a getter to return the first coordinate and a setter to renew the value of the first coordinate:

[1] A docstring that explains how to use the class is one such example.

```
def get_first_coord(self):
    return self.x

def set_first_coord(self, newx):
    self.x = newx
```

Notice that the getter doesn't pass any parameters. This is a benefit of object-orientation. Procedural code, on the other hand, more often has functions passing multiple parameters.

10.2 Inheritance

In the event that we'd like to create two data types that share some base-level functionality but differ in extra information, we'd likely be repeating a lot of code in our classes. In this case, the idea of inheritance in object-orientation comes in handy and we can create another class from which our two similar classes can inherit. More formally:

Given a parent (or base) class with some functionality (methods and attributes), we can create a child class (subclass) that inherits all of the base class' information. This is the idea of **inheritance** in object oriented programming.

This idea helps to cut down on repeated code, is convenient for projects as they grow in size, and sets up a hierarchy of classes. The syntax to create a subclass from a parent class is:

```
class ChildClass(ParentClass):
    pass
```

10.2.1 Extending Functionality

If we'd like to extend our subclass to contain more information (attributes) than its parent class, instead of redefining the `__init__` method with all the parent's attributes, we can use the `super()` method to let the parent class handle the inherited attributes:

```
class Person:
    def __init__(self, name):
        self.name = name

class Student(Person):
    def __init__(self, name, year_group):
        super().__init__(firstname)
        self.year_group = year_group
```

10.2.2 Overriding

Overriding a method inherited from a parent class is as simple as re-defining the method with the same name under the derived class.

10.2.3 Useful Built-in Functions

Python has two built-in functions that give us information about inheritance. They return booleans as their output:

- `isinstance(x,y)` checks to see if an object `x` is an instance of a class `y`
- `issubclass(x,y)` checks to see if the first input `x` is a subclass of a second class `y`

10.3 Polymorphism

Polymorphism means multiple forms. We've already seen polymorphism in operators in the form of overloaded operators that take on a different meaning depending on the objects they relate. For classes, in particular, it works similarly. Take the following example of two methods called `perimeter` that are wrapped in two different classes.

```
class Rectangle:
    def __init__(self, width, length):
        self.w = width
        self.l = length

    def perimeter(self):
        return 2*(self.l + self.w)

    def __str__(self):
        return "a rectangle of side lengths " + str(self.l) + " and " +
    str(self.w)

class Circle:
    pi = 3.14159
    def __init__(self, radius):
        self.r = radius

    def perimeter(self):
        return 2*(Circle.pi)*(self.r)

    def __str__(self):
        return "a circle of radius " + str(self.r)
```

Running the following code will take advantage (by virtue of cutting down on code) of the same name being used for two different methods that are associated with different classes:

```
rec = Rectangle(1,2)
circ = Circle(4)
```

```
for shape in [rec, circ]:
    print(f"The perimeter of {shape} is {shape.perimeter()}")
```

The output will be:

```
The perimeter of a rectangle of side lengths 2 and 1 is 6
The perimeter of a circle of radius 4 is 25.13272
```

Chapter 11

Miscellaneous Useful Things

11.1 Iterables and Iterators

We've already seen several times that all of Python's built-in collections like lists, dictionaries and tuples can be iterated over with a `for` loop. Now that we have an understanding of classes, we can begin to explain how `for` loops work.

An object is **iterable** if it can be looped over. More formally, an object is iterable if it supports a special method called `__iter__`.

We've seen examples of iterables like lists, tuples, strings, dictionaries etc. To check if an object is iterable, we can list out all of its methods and attributes by calling `dir()` on the object itself. For example, calling `dir(list)` in the interpreter yields the following:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
    '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
    '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
    '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
    '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append',
    'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
    'reverse', 'sort']
```

Note that `__iter__` can be found as the first entry on the fourth line. Thus, a list object is iterable.

Akin to how we've seen top-level functions like `len` calling `__len__` in the background, the Python interpreter recognises the keyword `for` and calls the special method `__iter__` on the object. What `__iter__` does is return an object called an iterator:

An object is an **iterator** if it has some way to remember its current state and supports a method (called `__next__`) that can retrieve its next value.

We can call `__next__` using the top level syntax `next(object_name)`. This will return the next element in the object. Once there are no more elements to traverse over, `__next__` will raise a `StopIteration` exception.

We know that lists are iterable. Are lists also iterators?

Notice that `dir(list)` above does not list a `__next__` method. Therefore, a list is iterable but not an iterator.

11.1.1 Custom Iterators

Studying iterables and iterators for their own sake is fun and all but we can actually add the iterator protocol to our own custom classes. Not only can we add this behaviour, but also customise the behaviour.

```
class Reverse:
    """Blueprint of an iterator object for looping over a sequence in
    reverse."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

As a brief summary of the above code, the `__iter__` method has to return an iterator (an object that supports the `__next__` method). We can simply define a `__next__` method in `Reverse` and have the `__iter__` method in `Reverse` return `self` (so that initialising `Reverse` will create an object that we can call `__iter__` on to return an object that supports `__next__`).

That was a mouthful.

11.2 Cartesian Products

```
from collections import product
a = [1,2]
b = [3,4]
prod = product(a,b)
print(list(prod))
>>>  [(1,3), (1,4), (2,3), (2,4)]
```

11.3 `map`, `filter` and `reduce`

These are three important functions that share the general syntax `function_name(function, sequence)`.

- The `map` function takes a sequence and applies a function to each element of the sequence.

```
a = [1,2,3,4]
b = map(lambda x: x**2, a)
print(list(a))
>>>  [1, 4, 9, 16]
```

- The `filter` function takes a sequence and filters out elements of the sequence depending on a rule given by a function.

```
a = [1, 2, 3, 4, 5, 6]
b = filter(lambda x: x%2 == 0, a)
print(list(a))
>>>  [2, 4, 6]
```

- The `reduce` function (from `functools`) takes a function that has two arguments and repeatedly applies it to elements of the sequence and returns a single final value.

```
from functools import reduce
a = [1, 2, 3, 4, 5, 6]
product_a = reduce(lambda x,y: x*y, a)
print(product_a)
>>>  720
```

11.4 List Comprehensions

List comprehensions condense a lot of syntax into a fairly easy-to-read expression in order to create a list of objects. The disadvantage is that it doesn't translate well into other languages. A typical list comprehension looks like:

```
[expression for variable in iterable if conditions]
```

Using functions, we can emulate a list comprehension with:

```
list(map(expression, filter(condition, iterable)))
```

We need to use `list()` on the object created by `map()` because it's an iterator (an object representing a stream of data, returning the data one element at a time), not an iterable.

As an example, consider the following list comprehension.

```
example_list = [n**2 for n in range(1,5) if n\%2 == 0]
```

What this particular list comprehension does is:

- Creates the list `[1,2,3,4]`.
- The `filter` function creates an iterator whose elements satisfy the condition that the element is divisible by 2, leaving 2 and 4.
- It then applies the function $n \mapsto n^2$ to each element of the iterator and creates a list out of them.

11.5 Arbitrary Arguments

So far we've only seen functions with a fixed number of arguments. Python has a way of permitting functions that can take on an arbitrary number of values.

- If you mask a parameter with `*` then you can pass any number of positional arguments to your function.
- If you mask a parameter with `**` then you can pass any number of keyword arguments to your function.

  ```
  def foo(a, b, *args, **kwargs):
      print(a,b)
      for arg in args:
          print(arg)
      for key in kwargs:
          print(key, kwargs[key])
  ```

  In tutorials and other literature, `*args` is used for arbitrary positional arguments and `**kwargs` for arbitrary keyword arguments. You can preface any formal parameter name, e.g. `*weights` if you'd like to pass an arbitrary number of positional arguments for weight to a function.

  In the example above, `*args` is a tuple and `**kwargs` is a dictionary.

Now suppose that we call `foo(0, 2, 3, 4, six = 6, eight = 8)`. The arbitrary positional arguments are `3` and `4` and the arbitrary keyword arguments are `six = 6` and `seven = 7`. What is returned is as follows:

```
0 2
3
4
six 6
eight 8
```

11.6 Tuples

A tuple can be unpacked into variables e.g. `details = (40, "Teach", "Emperor")` can be unpacked into `age, name, position = details`. Using a `*` before a variable name creates a list out of the elements between its enclosing values (`i` and `j` in the example below):

```
a = (0,1,2,3,4)
i, *j, k = a
print(i)    # This will print 0
print(j)    # This will print [1,2,3]
print(k)    # This will print 4
```

Lists are generally larger (size-wise) than tuples and tuples are faster (more efficient) to iterate over. However, tuples are immutable.

Chapter 12

Extra Things That Pop Up

- Generators (some kind of analogue to iterators?)
- Composition (something to do with classes being composed of other classes like a human class consisting of body parts classes that have their own attributes and functions?)
- Parallel processing and multi-threading (genuinely no idea beyond the name)

Fin

This is the summary at the end of the MIT course. Looks like a decent summary.

- You should now have a sense of how to represent knowledge with data structures.
- Be familiar with good computational metaphors e.g. using iteration, loops and using recursion as a good way to break down problems into simpler problems of the same type.
- Understand abstraction: the idea of capturing a computation, burying it inside a procedure so that you now have a contract with the computer that you need not worry about the underlying mechanics/procedures as long as it delivers the answer it says it should.
- Be able to use classes and methods as a way to modularise systems and to capture combinations of data and functions that operate on said data in an elegant way.
- Be exposed to classes of algorithms (search and sort) and their complexity.