A Guide to Advanced Programming in Python

Aarne Ranta

November 16, 2021

Contents

Chapter 1

Introduction

1.1 The aims of this guide and the course

This guide is written for the students of the course "Advanced Programming in Python" at Chalmers University of Technology. Its purpose is to give a unified view of the course and help your learning by showing the big picture and pointing you to further reading. It is *not* a complete description of the course material: we found it unnecessary, and even harmful, to repeat and digest all the material that you are expected to read. On the advanced level of programming, you should be able to read the original documentation, which is scattered all around the internet, and which may be difficult or even confusing. This guide will help you to get started, by giving pointers to material that we found relevant and which you *at least* should read while studying this course.

Despite its name, the course is not really what everyone would call "advanced". It is simply a second course in programming, after an introduction course. If we relate it to a common division of three levels - elementary, intermediate, and advanced - it should be called an *intermediate* course. Nevertheless, our ambition is to reach a level of *completeness*:

- **Complete knowledge of Python**: you will learn all constructs of the Python language, not only the parts covered by introduction courses.
- **Universal programming potential**: you will get the confidence that you can solve every programming task, as just a matter of having enough time to study the problem and work on it.

The latter goal is related to the idea of a computer as a **universal machine**,

which is able to perform all tasks that an algorithm can do. What is needed for this is, in addition to the peripheral devices, a programmer that can implement those algorithms on the machine. The goal of this course is to help you to become that programmer.

Really to become an advanced programmer, you will need to learn more things. This includes courses that are more specific than just programming: they will cover topics such as datastructures and algorithms, machine learning, and software engineering. Even more importantly, you will have to develop your skills by programming in practice.

1.2 Programmers of different kinds

The course has students from different backgrounds - computer science, data science, mathematics, physics, chemistry, design, engineering, ecology - and levels - bachelor, masters, PhD. Many of the fields have their own cultures of programming. It is helpful to identify some of them. The following three profiles are of course caricatures, but you can easily find code examples that satisfy all characteristics in each of them.

1.2.1 The Computer Scientist

This is the context in which the course is formally given. In this context, programming is a branch evolved from applied mathematics. The programmer is interested in the mathematical structure of programs and tries to find the most elegant data structures and algorithms. She also wants to make them as general and abstract as possible. The programs are typically not solutions to practical problems, but more on the level of **libraries** that can be used by others who are solving practical problems. The hard-core computer scientist programmer does not use libraries herself - instead, she tries to understand and build everything from the first principles.

Here are some typical features of the Computer Scientist's programs:

- they consist of functions whose size is just a few lines each,
- each function has a clear mathematical description,
- they are often close to pseudocode that could be published in a textbook,
- variable and function names are often short and similar to ones used in mathematics,

- the code avoids language-specific idioms,
- classes are used mainly to define data structures - not to structure entire programs,
- the code is accompanied by arguments about correctness and complexity,
- the code is not tested, just demoed with a couple of examples if not proved correct,
- there are very few comments in the code, since it is aimed to be self-explanatory,
- the programs are written by single persons and are self-contained.

With the growing needs of computer programs, Computer Scientist programmers are becoming more and more of a minority. They have, to some extent, made themselves superfluous by creating libraries that other programmers can rely on and do not need to understand the internals of. But of course, they are needed in the front line of research to invent new algorithms, new programming languages, and new kinds of computing devices. It is also common that theoretical Computer Science questions are used in interviews at tech companies.

1.2.2 The Software Engineer

This category contains the majority of programmers who have programming as their profession. They typically have a Computer Science background, but have "grown up" from that when getting encountered with the "real world". They have learned several programming languages and have lots of algorithms and data structures in their "toolbox". At the same time they know that, to solve a practical problem or to create end-user software, no single technique is sufficient in itself. They have no time to go into the internals of algorithms, but use whatever libraries and tools they - or their managers - have learned to trust. They often work in large groups and want to maintain a level of readability that enables others in the group take over their code.

Here are some typical features of the Software Engineer's programs:

- they are divided into modules and classes with complex hierarchies,
- each unit of the program is documented in detail with comments and often with diagrams,
- the names of variables, functions, and classes are long, descriptive, and systematic,

- the code is accompanied by a comprehensive set of tests,
- the code uses libraries whenever possible, avoiding to "reinvent the wheel",
- the code has a long lifetime and may have ancient layers that no-one dares to touch any more,
- the full software system may have code written in many languages and typically also contains configuration files and build scripts.

Competent software engineers on different domains of application are constantly and increasingly wanted by employers. Their competence may be based on academic education, but it is primarily a product of years of experience. It is proven, not by academic degrees or publications, but by a portfolio of software that the programmer has designed or contributed to. This experience often comes from enterprises, but it can also be built in open source projects where the programmer has made major contributions.

1.2.3 The Occasional Programmer

This is the category of programmers with no or little training in Computer Science. They can be teenagers or other hobby programmers, but also proficient scientist or engineers in other disciplines, who see programming as something that can be learned in an afternoon or two, when it is needed for some computations or experiments. For scientists, programming is a replacement of the earlier use of pencil and paper for similar tasks (now decades ago). Computers make it faster to apply mathematical formulas, and make it feasible to deal with much larger amounts of data than in earlier times, in particular if statistics or machine learning is involved.

Typical features of programs in this category are:

- the structure of the code is simple and linear - in Python, this means that it consists of top-level statements and global variables, avoiding functions and classes,
- the program is a single file that can be thousands of lines long,
- variable names are short and unsystematic,
- input and output is perfomed freely in different places of the code,
- as there are no return statements, the only way to combine the code with other programs is to pipe its output to them as a string,
- the code uses freely whatever libraries seem do the job,
- libraries are often imported in the middle of the file rather than in the beginning,

- the code is written by one person and is not intended even to be read by others,
- the code does one thing, aiming to do it with as little effort as possible,
- the code is only run once or a few times,
- well, it is often tested with the same input a large number of times, until it seems to do the job in the expected way,
- if a related task appeared again, the code is copied and patched to fit for that purpose.

This category of programmers is probably the largest, and includes both professionals and hobby programmers. In fact, since Python supports this style so well, it is occasionally used by all kinds of programmers. Programs in this category are often called **scripts** rather than "real programs". If you search the web for a Python solution to a particular problem, you are likely to find several examples of this kind of code: scripts with linear structure where large blocks are copies of each other and where input and output happen all over the place.

1.2.4 Programming in this course

The main part of this course belongs to the Software Engineer category:

- we want to solve substantial practical problems,
- we use proven techniques and libraries for this,
- we structure the code in a clear and reusable way,
- we document the code in a way that explains it to other programmers,
- we raise the level of reliability of the code by systematic testing.

As a support for this, we will also look into some Computer Science topics:

- the data model and semantics of Python,
- some fundamental algorithms and their theoretical aspects.

We will neither cover nor recommend the Occasional Programmer's style, not even for highly competent professionals in advanced scientific fields. You are probably right if you think that programming is easy compared with the deeper questions you deal with in your actual research. But even then, we believe that it is helpful to learn about the algorithms and data structures of the computer scientist, as well as the techniques and practices of the software engineer. This can make you both faster and more confident when writing code for your next experiment.

One can also compare programming with other hobbies: almost anyone can learn to cook food or play an instrument without any formal training

or even without reading books that teach these skills. But if you take your hobby seriously, it will give you satisfaction to know that you are doing it "the right way".

1.3 The Lab

Much of the course is centred around the **Lab**, a programming assignment whose goal is to build a "full-stack" web application. In the final demonstration of the application, the user can search for a route in the Gothenburg tram network and get it drawn on a map - in the way familiar from the numerous travel planning applications on the web. The Lab is described in detail in

https://github.com/aarneranta/chalmers-advanced-python/tree/main/labs

so let us here just explain the purpose of the lab in the context of the course.

The Lab is primarily an exercise in practical Software Engineering and serves the following learning outcomes:

- to write modular, well structured programs with reusable components,
- to use standard libraries and understand their documentation,
- to apply testing and version control practices,
- to document program code so that it can be understood by others,
- to get the basics of some useful techniques (graphs, visualization, web programming) that can be applied to numerous other tasks.

In addition, the Lab has some Computer Science aspects:

- to learn about graphs, which are a fundamental concept in many computing tasks,
- to widen your imagination so that you can search for solutions to practical problems from well-known general algorithms.

The standard Lab can be extended with a couple of related tasks, which use graphs for other things than route finding:

- map colouring: making sure that neighbouring countries have different colours,
- program analysis: finding dependencies between different parts of software,
- machine learning: finding clusters (tightly connected parts) in a network.

The basic lab is estimated to require up to two weeks of working time, whereas the extra labs might take a couple of days each. The actual time consumption is individual and depends greatly on your previous experiences.

Chapter 2

Learning the Python language

The goal of the first two lectures is to bring you to a level where they have seen "all of Python". This does not yet mean that you can use all constructs of Python efficiently. But you will have at least heard about things like dictionaries, slicing, comprehensions, and other special constructs that might not be familiar from your previous programming language. Those who have received their introduction in Python will see fewer new thing, but they should now get a more systematic view of how everything belongs together.

2.1 What do I learn when I learn a language

Let us assume that you have learned Java but do not know Python yet. Then certainly you do not need to learn everything that absolute beginners are taught when Python is their first programming language. You only need, so to say, learn the *differences* between Python and Java.

The situation resembles learning a *second language* when you already know your native language - for instance, learning English when you already know Swedish. It may have taken you seven years to learn to speak, read, and write Swedish. You have had to learn not only what words mean which things, but also the very idea that words can stand for things, and that written signs can represent sounds. While learning this, your brain and motoric skills have developed to use a language. If you start learning English at this point, most of this infrastructure is already in place. Therefore, it is not uncommon to learn a new language fairly well in less than a month, which is totally unconceivable when a child learns its first language.

So let us assume, analogously to Java programmers learning Python, that you are a native speaker of Swedish beginning to learn English. You know that the language consists of words, which are combined to phrases and sentences. What you need to learn is

- syntax: how phrases and sentences are formed,
- vocabulary: what words are used for what things.

In many cases, learning the syntax of a new language is a task that can be completed in a couple of weeks, since it will not be very different from your earlier languages. You will learn, for instance, that negation in English is formed in a different way from Swedish, by using auxiliary verbs (*I don't know* instead of *I know not*). It might take longer to use *actively* all parts of the syntax, but you will at least understand the structure of what you read and even if you need to look up words in a dictionary.

The vocabulary, in contrast to syntax, involves life-long learning. Even native speakers may occasionally need to look up a word in a dictionary.

Now, when learning Python on top of Java (or some other programming language), it is a good idea to start with the syntax - more precisely, how its syntax differs from Java, "how negation is expressed in Python" (yes, it uses the keyword `not` instead of the operator `!`). Most of this will be fairly obvious and follow a handful of general rules, while some things have no counterparts in Java. At the same time, some things available in Java are not available in Python, and for those things you will have to learn to think in a slightly different way.

Within a week or so - Python syntax is a lot easier than English - you will be able to understand the syntactic stucture of everything that is written in Python. There may be things you will never use in your own code, and will probably have a bias for Java-like expression - just like Swedish native speakers, even when fluent in English, can be biased to Swedish-like constructs.

In this course, we will actually try to help you become more *Pythonic*, since this will give you some satisfaction as well as credibility among *Pythonistas*. But I must confess that, since my own "native" programming language is not Python, I cannot guarantee always to be Pythonic myself.

What about the vocabulary of Python? There are 30 **reserved words**, and they are quickly learned as a part of the syntax. In addition, there are 71 **built-in functions**, many of which you will not need any time soon. But the main bulk of the vocabulary is in **library functions and classes**, and they involve life-long learning. The only viable approach is to learn to "look

up words in the dictionary", that is, search for class and function names in library documentation.

2.2 Some characteristics of Python syntax

2.2.1 Modules, classes, functions, statements

Assuming that your first language is Java, you have probably started with a "Hello World" program like this one:

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

In order to run it, you have first compiled it to Java bytecode with the `javac` program, and then executed the resulting binary with the `java` program:

```
$ javac Hello.java
$ java Hello
Hello World
```

If you mechanically convert your Java program into Python code, you will get

```
class Hello:
    def main():
        print("Hello World")

Hello.main()
```

In order to run this, you just need to run `python` (often named `python3` to distinguish it from earlier versions):

```
$ python3 hello.py
Hello World
```

In other words, there is no separate step of compilation, but you can "run your program directly". (In reality, there is a compilation phase to Python's bytecode, but it happens behind the scenes.) Thus running the code looks simpler, but the code itself is about as complicated as in Java.

However, the code shown above is what you end up with if you directly convert your Java thinking into Python. The "normal" Hello World program in Python is much simpler:

```
print("Hello World")
```

This program consists of a single statement - no class, no function is needed. There is also an intermediate way to write the same program, which contains a function but no class:

```
def main():
    print("Hello World")

main()
```

If your background is in C rather than Java, this will probably be the most natural way to write the Hello World program.

The Hello World examples illustrate four different levels in which programs are structured in both Java and Python, and also in many other languages:

- modules (typically, files),
- classes,
- functions (or "methods"),
- statements.

In Java, these levels are stricty nested: statements reside inside in functions, functions inside classes, and classes are the top-level structure of modules. In Python, the same strict hierarchy *can* be followed, but this is not compulsory. It is common that a top-level module is a mixture of classes, functions, and statements. An extreme case is a module consisting only of statements: this is the dominating style in what we called "occasional programming" and has given Python the label **scripting language**.

Even though we will discourage code consisting of top-level statements, we have to add that Python in fact does not even force any nesting of the levels: class and function definitions are themselves statements, and can appear anywhere inside other statements, such as the bodies of `while` loops. Hence, if you want to write ten times "Hello World", you can do it as follows:

```
i = 0

while i < 10:
    class Hello:
        def main():
            print("Hello World")
    Hello.main()
    i += 1
```

This, of course, is *not* good use of the freedom you get in Python. It is overly complicated to write, as well as overly expensive to run. But we will later see situations where functions inside functions, or even classes inside functions, are a good way to write programs.

2.2.2 The layout syntax

Both Java and C programmers are used to terminating statements with semicolons (`;`) and enclosing blocks of code in curly brackets ({}). This is not the case in Python. Instead,

- statements are terminated by newlines,
- blocks are marked by adding indentation.

As a general rule,

- a line ending with a colon (`:`) marks the beginning of a new block, expecting added indentation on the next line,
- these lines start with a keyword such as `class`, `def`, `while`,
- a block ends when indentation comes back to some of the earlier levels.

A slight modification in these rules is that it is possible to divide a statement, or lines expecting a new block to start, into several lines, if additional indentation is used. But such examples are rare and typically appear only if the lines contain expressions that are two long to fit into the recommended 79 characters per line.

And yes, indeed, Python comes with recommended lengths for lines and indentations:

- lines should not be longer than 79 characters,
- each indentation level should be 4 spaces (no tabs recommended),
- each class and function definition should be separated by 2 blank lines.

Such rules may sound pedantic at first, but if you follow them, they will soon become a second nature and make your Python code easier to read.

2.2.3 Dynamic typing

Another thing that you can notice as a Java programmer is the lack of **types** in the Python code:

- function definitions do not indicate argument or return types,
- variable types are not declared.

The lack of types in Python source code is sometimes taken to mean that "Python has no types". But this is not true: Python does not have types of **expressions**, but it does have types of their **values**, i.e. objects that are created at **run time**, when the program is executed. Hence the correct term to use is **dynamic typing**.

Dynamic typing implies that there is no **static type checking** of your source code. An expression - such as a variable or a function call - can have different types in different places of the code, and even at different runs of the program. It can then happen that no possible type is found, which leads to a **run-time type error**.

A **statically typed language** such as Java can find type errors at **compile time** and prevent the execution of the program. This is not provided by Python: it can in fact happen that a program is run 1000 times without failurs, but after that a type error is encountered when the execution, for the first time, enters a rarely used branch of some conditional. Thus it is difficult ever to be sure that your Python program is failure-free.

What is more, the lack of static checking does not only concern types, but also function and variable names. If a name in a rare branch has a typo, making it into a name that is not in scope, you will only notice this when some run of the program enters this branch.

Some later extensions of Python have made it possible to annotate functions with type information, and there are even programs that perform static type checking. But this is not yet standard, and we will not use it very much during the course. Moreover, static type annotations are deemed to be incomplete, because Python is intended to support a high degree of **polymorphism**. This means that functions can be applied to many different types, and it is difficult to specify in advance which types these exactly are. Such information is in library documentation given in natural language.

2.3 A grammar of Python

This section is something you can just glance through and use later as reference. The main purpose at this point is to show that there is not so much language-wise you need to learn: just a couple of pages. We will explain most of the things in more detail when going through the tutorial.

It is a condensed and somewhat simplified version of the official grammar in

https://docs.python.org/3/reference/grammar.html

Figure 2.1 shows the syntax of Python in the BNF notation (Backus-Naur form). It covers slightly less than the full grammar, but its shortness is mainly due to its higher level of abstraction. In particular, it does not indicate precedence levels.

Here are the **reserved words**, each of which appears in syntax:

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

The **literals** are infinite classes of "words", the most important of which are

- **strings**: `'single quotes' "double quotes" """many lines between groups of three quotes"""`,
- **integers**: any number of digits `1234567890`,
- **floats**: digits with decimal point `3.14 .005` possibly with exponent `2.9979e8 6.626e-34`,

Comments are lines starting with `#`. A comment can also terminate a non-comment line, but it is recommended to use entire lines. Multi-line comments can be given as string literals in triple quotes `"""`.

To conclude the explanation of words, the following **built-in functions** will appear throughout the course:

- `abs(x)`, absolute value of a number,
- `bool(x)`, conversion from various types to booleans,
- `chr(n)`, conversion from numeric code to character,
- `dict(c?)`, creation of or conversion to a dictionary,

```
<stm> ::= <decorator>* class <name> (<name>,*)?: <block>
      |   <decorator>* def <name> (<arg>,*): <block>
      |   import <name> <asname>?
      |   from <name> import <imports>
      |   <exp>,* = <exp>,*
      |   <exp> <assignop> <exp>
      |   for <name> in <exp>: <block>
      |   <exp>
      |   return <exp>,*
      |   yield <exp>,*
      |   if <exp>: <block> <elses>?
      |   while <exp>: <block>
      |   pass
      |   break
      |   continue
      |   try: <block> <except>* <elses> <finally>?
      |   assert <exp> ,<exp>?
      |   raise <name>
      |   with <exp> as <name>: <block>
<decorator> ::= @ <exp>
<asname>    ::= as <name>
imports     ::= * | <name>,*
<elses>     ::= <elif>* else: <block>
<elif>      ::= elif exp: <block>
<except>    ::= except <name>: <block>
<finally>   ::= finally: <block>
<block>     ::= <stm> <stm>*
<exp> ::= <exp> <op> <exp>
      |   <name>.?<name>(<arg>,*)
      |   <literal>
      |   <name>
      |   ( <exp>,* )
      |   [ <exp>,* ]
      |   { <exp>,* }
      |   <exp>[exp]
      |   <exp>[<slice>,*]
      |   lambda <name>*: <exp>
      |   { <keyvalue>,* }
      |   ( <exp> for <name> in <exp> <cond>? )
      |   [ <exp> for <name> in <exp> <cond>? ]
      |   { <exp> for <name> in <exp> <cond>? }
      |   - <exp>
      |   not <exp>
<keyvalue> ::= <exp>: <exp>
<arg>      ::= <name>
           |   <name> = <exp>
   |   *<name>
   |   **<name>
<cond> ::= if <exp>
<op>   ::= + | - | * | ** | / | // | % | @
       |   == | > | >= | < | <= | != | in | not in | and | or
<assignop> ::= += | -= | *=
<slice> ::= <exp>? :<exp>? <step>?
<step>  ::= :<exp>?
```

Figure 2.1: The syntax of most of Python.

- `dir(m)`, listing of contents in a module,
- `eval(s)`, evaluation of a string as an expression,
- `exec(s)`, execution of a string as a statement,
- `float(x)`, conversion to a float,
- `help(f)`, documentation of a function, class, or module,
- `input(p='')`, receiving input from prompt p,
- `init(x)`, conversion to integer,
- `len(c)`, length of a collection,
- `list(x?)`, creation of or conversion to list,
- `max(c)`, maximum of a collection,
- `min(c)`, minimum of a collection,
- `next(g)`, receive the next object from a generator,
- `open(f)`, open a file,
- `ord(c)`, numeric code of a character,
- `print(x*)`, print a sequence of objects,
- `range(m=0, n)`, range of integers from m to n-1,
- `reversed(s)`, reverse of a sequence,
- `round(d, p=0)`, round a float to p decimals,
- `set(c?)`, creation of or conversion to a set,
- `sorted(s)`, sorted sequence,
- `str(x)`, conversion to a string,
- `sum(c)`, sum of a collection,
- `super()`, superclass of a class,
- `tuple(c?)`, create or convert to a tuple,
- `type(x)`, type of an object.

The tutorial will cover many of these in more detail. A full list, with full details, can be found in

https://docs.python.org/3/library/functions.html

In addition to the built-in functions, there are several **methods for built-in classes** that are used all the time. Here are some:

- strings: `split()`, `join()`, `format()`, `index()`,
- lists: `append()`, `sort()`, `reverse()`, `pop()`,
- dictionaries: `get()`, `keys()`, `values()`, `items()`,
- sets: `add()`, `remove()`.

The **on-line help** in the Python shell is always available for more information. For instance, the command

```
help(str)
```

gives a detailed list of all methods available for strings.

Chapter 3

Diving into the official tutorial

This chapter corresponds to Lectures 1 and 2.

We will go through the official Python tutorial in

https://docs.python.org/3/tutorial/index.html

by showing some highlights from it. The tutorial is written by the creator of Python, Guido van Rossum. It covers practically everything in the Python language, and does it in a way that makes you hear van Rossum's line of thought when designing the language. Hence it is recommended reading, from the beginning to the end. However, it is not a tutorial for beginner programmers, but assumes - just like we are doing here - that you already know the basic concepts from some other context.

During the first two lectures, we will go through the tutorial, focusing on the things highlighted here, and demonstrating them with live coding. So once again: the sections below are just pointers to further reading and practice.

3.1 Tutorial 1: Whetting your appetite

There is one point we want to raise: the chapter says that "Python also offers much more error checking than C". This is not completely true, because C has static type checking and Python does not. In C - as in many other languages - static type checking is needed to support compilation to bare machine code (as opposed to virtual code in the case of Python). In processors such as Intel and ARM, different instructions and registers are used for

different types of objects, e.g. for integers and floating point numbers. To make an optimal use of the machine, one has to know the types of expressions at compile time.

Static typing and compilation to machine code is what makes languages like C to enable more efficient programs than Python. Much of this is, however, helped by enabling Python code to use modules written in C via foreign code interfaces - which is also mentioned in this chapter.

3.2 Tutorial 2: Using the Python interpreter

This chapter explains the different ways of invoking the Python interpreter. The main options are

- running the entire program from the operating system shell (in a way familiar from Java and C),
- opening the Python shell and importing the module from there (familiar from e.g. Lisp, Haskell, and Prolog).

From the operating system shell

You can write

```
$ python3 hello.py
```

where the source file name is given, or

```
$ python3 -m hello
```

where the module name (without extension `.py`) is given. We will return to the use of command line arguments (`sys.argv[]`) when discussing the `sys` library.

Inside the Python shell

First start the Python shell from the operating system, then `import` the module from there.

```
$ python3
[welcome message displayed]
>>> import hello
```

You will then be able to execute statements that refer to functions and classes defined in `hello.py`:

```
>>> hello.hello()
Hello World
```

You can of course execute statements that do not refer to the imported module. This you can do even without importing any module:

```
$ python3

>>> print(2+2)
4
```

What is more, you can evaluate expressions in the shell, so that their values are shown, without the need of `print()` around.

```
>>> 2+2
4
```

This is a difference between writing code in the shell and in Python files: if you write just `2+2` on a line in a Python file, nothing is shown about it when you run the module. To see the value, you must make the expression into a statement by using `print()`.

The shell has two kinds of prompts:

- `>>>`, the **primary prompt**, corresponding to no indentation in a Python file,
- `...`, the **secondary prompt**, expecting added indentation.

The secondary prompt is needed when you for instance want to run a `for` loop in the shell:

```
>>> for i in range(2):
...     print("Yes!")
...
Yes!
Yes!
```

You must remember to add the indentation and keep it constant; and empty line will get you back to level 0. To my personal taste, it is quite awkward to write multiline statements in a shell, and therefore I prefer writing them inside functions in files and just calling these functions from the shell.

If you `import hello` from the Python shell, you can access the functions in it with the prefix `hello.`. You can avoid this by writing, instead,

```
>>> from hello import *
```

Then all names defined on the top level of the file become available without a prefix. This is handy if you open the file in the purpose of testing its functions in different combinations.

The effect of both ways of importing a module is that the statements in it are executed in the order in which they appear. This can be disturbing, if you are for instance opening a module in order to test the functions in it one by one. You can avoid most of the disturbance by not including `print()` statements on the top level but only inside functions - which is a good practice anyway.

3.3 Tutorial 3: An informal introduction to Python

Numbers

Here are some points of interest, potentially surprising:

- Integers have arbitrary size, whereas floats have limited precision. Therefor, int to float conversion can be lossy:
  ```
  >>> int(float(12345678901234567890))
  12345678901234567168
  ```
  (Thus the type names `int` and `float` are themselves used as conversion functions.)
- The **floor division** operator `//` returns the largest smaller integer:
  ```
  >>> 20 // 7
  2

  >>> -20 // 7
  -3
  ```

Strings

Points of interest:

- Single and double quotes mean the same; an advantage is that escapes of quotes are seldom needed:

```
'"Yes," they said.'
```

- Evaluating a string literal shows quotes (usually converted to single), printing it drops the quotes:

```
>>> 'hello'
'hello'

>>> "hello"
'hello'

>>> 'hello' == "hello"
True

>>> print('hello')
hello
```

  Notice also that there is a type difference (`NoneType` is user for expressions that return no useful value; `type()` returns the type):

```
>>> type('hello')
<class 'str'>

>>> type(print('hello'))
hello
<class 'NoneType'>
```

- **Raw strings** prefixed with `r` treat backslashes literally:

```
>>> print('\tmp\name')
        mp
ame
>>> print(r'\tmp\name')
\tmp\name
```

in normal string literals, \t is tab and \n is newline, as expected.

- Triple quotes """ enclose **multiple line string literals**. This is also the way to create multiline comments - or comment out blocks of code.

- Two or more string literals are glued together:

```
>>> 'Py' 'thon'
'Python'
```

- You can concatenate strings with + and multiply by *:

```
>>> name = 'Joe'

>>> 'hey ' + name
'hey Joe'

>>> 6*name
'JoeJoeJoeJoeJoeJoe'
```

- Notice that print() takes many arguments, converts all to strings, and adds spaces:

```
>>> print(name,  23, 'years')
Joe 23 years
```

  But + requires explicit conversions and spaces:

```
>>> name + 23 + 'years'
TypeError: can only concatenate str (not "int") to str

>>> name + str(23) + 'years'
'Joe23years'
```

- The **index** notation [] returns characters in given positions, starting from 0. A negative index counts backwards from the last character, -1.

```
>>> 'hello'[0]
'h'

>>> 'hello'[4]
'o'

>>> 'hello'[-1]
'o'
```

Notice that there is no special type of characters: a character is simply a one-character string.

- The **slice** notation generalizes from the index notation to substrings.

```
>>> 'hello'[1:4]
'ell'
```

Notice that the first index is included, the second one is not: slices is like a semi-open intervals in mathematics, in this case `[1,4[`.

- The start and end indices in slices are optional:

```
>>> 'hello'[:2]
'he'

>>> 'hello'[2:]
'llo'
```

- Adding a third argument indicates a **step**

```
>>> '0123456789'[0:9:2]
'02468'
```

Quiz: you can create the reverse of a string by using negative indices and steps. How?

The final point in this section is that strings are **immutable**. Mutability is a general topic, which we will cover when comparing strings with lists.

Lists

Points of interest:

- Lists can be given with the bracket notation, and indexed and sliced just like strings:

```
>>> digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> digits[6]
6
>>> digits[3:6]
[3, 4, 5]
```

- One can assign values to indices, as lists are **mutable**

```
>>> digits[4] = 'four'
>>> digits
[0, 1, 2, 3, 'four', 5, 6, 7, 8, 9]
```

  Notice here also that a list can contain elements of different types.

- You can also assign lists to slices - even lists of a different size:

```
>>> digits[3:5] = ['III','IV']
>>> digits
[0, 1, 2, 'III', 'IV', 5, 6, 7, 8, 9]

>>> digits[3:5] = []
>>> digits
[0, 1, 2, 5, 6, 7, 8, 9]
```

Unlike lists, strings are not mutable. To create a "mutable string", you can make a copy that is converted to a list, and **join** the result back to a string:

```
>>> s = 'python'
>>> ls = list(s)
>>> ls
['p', 'y', 't', 'h', 'o', 'n']
```

```
>>> ls[0] = 'P'
>>> ls
['P', 'y', 't', 'h', 'o', 'n']

>>> ''.join(ls)
'Python'
```

This is a useful technique when lots of changes are performed on a long string. Using lists instead of strings avoids creating new copies of the string.

First steps towards programming

The Fibonacci example

```
a, b = 0, 1

while a < 10:
     print(a)
     a, b = b, a+b
```

shows a `while` loop, which is probably familiar, but also two examples of a **multiple assignment** - assignment to several variables simultaneously. A simple standard example of multiple assignment is swapping the values of two variables:

```
a, b = b, a
```

Without multiple assignments, doing this would require a temporary third variable.
THIS IS THE POINT REACHED AT LECTURE 1.

3.4 Tutorial 4: More control flow tools

Loops and conditionals

Many of the statement forms for control flow are familiar from other langusages, with just minor differences:

- `while` loops (above, from Tutorial 3.2),
- `if-elif-else` statements, which can have any number of `elif` branches,
- `for` loops, which can iterate over any **iterable** type,
- `break` statements, which interrupt a loop and go back to the previous block level,
- `continue` statements, which interrupt a loop and go to the next round of iteration.

Some statement forms are less familiar:

- `pass` statements, which do nothing - commonly used as place-holders for what would be an empty block in a language using brackets,
- `match` statements, starting from Python 3.10, which enable structural pattern matching familiar from functional languages such as Haskell.

Of the familiar statements, `for` loops provide a prime example of what is considered "Pythonic", that is, coding style that maximally uses the possibilities of Python. Here is an example: a `for` loop that simply prints each character of a string:

```
for c in s:
    print(c)
```

A less Pythonic way go get the same output is to loop over an index that ranges from 0 to the length of the string:

```
for i in range(len(s))
    print(s[i])
```

This style is considered overly complicated, except if you really need to access the integer index. The `range()` function, by the way, creates sequences of integers,

- the sequence 0,1,2,3,4 for `range(5)`
- the sequence 2,3,4 for `range(2,5)`

An example where looping over an indes is needed is the following: if you want to change a list, for instance increment each element in it by one, you might try

```
ds = [1, 2, 3, 4]

for d in ds:
    d += 1
```

But this will *not* change the list `ds`. What it does is just to give new values to the variable `d`. The proper way to increment each element in the list is to assign to each element of it:

```
for i in range(len(ds)):
    ds[i] += 1
```

The function `len()` returns the length of a list or a string.

One more thing: you might expect that the variable bound in the `for` loop has its scope limited to that loop. But in fact, the variable continues to be alive, with the last value assigned to it in the loop. Hence for instance `i` has the value 3 after the loop just shown.

The `match` statement (Tutorial 4.6) is a novelty in Python 3.10. It will look familiar to functional programmers - and it actually feels like the missing piece to some of us! - but we will not cover it here, since it is not yet available in all Python installations. What is more, it breaks the idea of data hiding in object-oriented programming, if used in a way that makes class-internal variables visible.

Function definitions

Function definitions, as observed before, have no type information attached to the parameters or to the return values. This makes it possible to define functions that return different types of values, or nothing at all (which in Python, however, is treated as value `None`), in different branches of a conditional. An example is the following function that find real roots for quadratic equations

```
import math

def quadratic_eq(a, b, c):

    discr = b**2 - 4*a*c
```

```
if discr < 0:
    print("no roots")
elif discr == 0:
    return -b/(2*a)
else:
    rdiscr = math.sqrt(discr)
    return [(-b - rdiscr)/(2*a),
            (-b + rdiscr)/(2*a)]
```

A problem with this style is that it is difficult for another function to take the output of this function as input. For instance, it cannot loop over the roots, because the result is iterable (a list) only if there are two roots. In this case, it would of course be easy to make each of the three branches return a list.

The number of arguments to a function can be varied by using **optional arguments** that have, if not given, **default values**:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
```

The first argument is a **positional** arguments, which is compulsory to give. The other arguments can be given either by position or by using the variable name, then known as **keyword arguments**. The order of keyword arguments is not significant, but they must come after the positional arguments:

```
ask_ok('> ', 3)    # third arg 'Please try again'
ask_ok('> ', reminder='Come on!')  # second arg 4
ask_ok('> ', reminder='Come on!', retries=3)
```

Yet another specialty of Python is the use of **packed arguments**, marked by

- `*` for any list (or tuple) of positional arguments,
- `**` for any list of keyword arguments

which must appear in this order, as in the Tutorial example

```
def cheeseshop(kind, *arguments, **keywords):
```

The markers `/` and `*` used as in

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

are yet another way specify how arguments can be passed to a function. Tutorial Section 4.8.3.5 (in version 3.10) gives guidance for how to use this mechanism in the intended way.

Lambda expressions

Lambda expressions create **anonymous functions** - functions that are created without defining them with `def` statements. A common use is in keyword arguments, for instance, in sorting:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

An alternative to this would be

```
def snd(pair):
    return pair[1]

pairs.sort(key=snd)
```

The point with using lambda is that a line of code is saved and no name is needed for this function that is used only once. A limitation is that the definition of the lambda function must be single expression, hence it cannot contain statements.

Documentation strings and function annotations

A string literal immediately after a function header (the `def` line), as in

```
def quadratic_eq(a,b,c):
    "solving quadratic equations"
    # etc
```

has a special status as a **documentation string**. It gives the value of the "invisible" `__doc__` variable, but also the answer to the `help()` function:

```
>>> quadratic_eq.__doc__
'solving quadratic equations'

>>> help(quadratic_eq)

Help on function quadratic_eq in module mymath:
```

```
quadratic_eq(a, b, c)
    solving quadratic equations
```

Including documentation strings is a good practice, in particular in library functions. They can consist of many lines if triple quotes are used.

Functions can also be equipped with **annotations** (Tutorial 4.8.8) that give type information. They can be useful for documentation, but, unlike in e.g. Java, they are not checked by the compiler, and their expressivity is limited. For example, if a function has variable argument and return types, there is no standard way to express this.

Coding style

We have already mentioned then "Pythonic" recommendations to use 4 spaces for indentation 2 empty lines between functions, and maximum line length 79 characters. Some others are mentioned here, and we will try to follow them, in particular,

- spaces after commas, e.g. `f(x, y)`,
- `lowercase_with_underscores` for function names.

In many other languages, `camelCase` is preferred for function names, which makes it natural to continue this convention in Python. However, this will soon lead to mixtures of the two conventions when libraries are used, which gives a messy impression.

3.5 Tutorial 5: Data structures

This section covers different types of **collections**:

- **lists**, e.g. `[1, 4, 9]`
- **dictionaries**, e.g. {`'computer': 'dator', 'language': 'språk'`}
- **tuples**, e.g. `'computer', 'dator'` (parentheses not needed!)
- **sets**, e.g. {`1, 4, 9`}

To these, one could also add two kinds of sequences that we have already seen:

- **strings**, e.g. `hello`
- **ranges**, e.g. `range(1,101)`

These six types have a lot in common, but also differences that make them usable for different purposes. Therefore it is often necessary to convert between

them, and we will cover some of the main ways to do so. The conversions sometimes work as expected, but sometimes lead to loss of information:

```
>>> dict([(1,2)])
{1: 2}

>>> list({1: 2})
[1]
```

The fundamental explanation of all differences lies in a handful of hidden methods in class definitions, which we will cover in Lecture 4 on the data model of Python. The visible part of these concepts can be seen in type error messages, such as:

```
>>> s = "hello"
>>> s[3] = 'c'
TypeError: 'str' object does not support item assignment
```

We will go through most of the methods listed in Tutorial 5.1, not only for lists but also for the other data structures, to get a feeling of which of them work for which.

Notice that all of these collections are **objects** of **classes**, which have **methods** applied by using a special syntax - for instance,

```
xs.sort()
```

which is really a shorthand for a function application

```
list.sort(xs)
```

In other words, the object `xs` is really the first argument of the function `sort()`from the class `list`. Normally, the first kind of **object-oriented** notation is recommended for these functions.

Comprehensions

List comprehensions are introduced in 5.1.3, and provide a powerful, highly Pythonic way to create lists from given ones. What gives even more power is that comprehensions work for other collection types as well, and even for mixtures of them. Thus we have **dictionary comprehensions**,

```
>>> {n: n**3 for n in range(7)}
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216, 7: 343}
```

and **set comprehensions**

```
>>> {n % 3 for n in range(100)}
{0, 1, 2}
```

Booleans

There are a couple of special things about booleans:

- Keywords `and`, `or`, `not` are used for boolean operators.
- Comparison operators can be chained: `a < b <= c` means the same as `a < b and b <= c`.
- Many types can be cast into booleans: `0`, `''`, `[]`, {} all count as `False` when used in conditions.
- Boolean `or` can return values in these other types:

  ```
  ```>>> {} or 0 or 'this'
  'this'
  >>> {} or 0
  0
  ```

  Finally, let us tell you a secret that is not covered by the tutorial: Python has a **conditional expressions**, which have a slightly surprising syntax:

  ```
  >>> x = 10
  >>> 'big' if x > 10 else 'small'
  'small'
  ```

  Thus the "true" value comes first, the condition in the middle, and the "false" value last.
  ```

A summary of collection types

type	notation	indexing	assignment	mutable	order	reps
`str`	`'hello'`	`s[2]`	-	no	yes	yes
`list`	`[1, 2, 1]`	`l[2]`	`l[2] = 8`	yes	yes	yes
`tuple`	`1, 2, 1`	`t[2]`	-	no	yes	yes
`dict`	`{'x': 12}`	`d['x']`	`d['x'] = 9`	yes	no	no
`set`	`{1, 2}`	-	-	yes	no	no
`range`	`range(1, 7)`	`r[2]`	-	no	yes	no

The difference between lists and sets is the familiar one from mathematics: lists care about the order of elements, and remember repetitions of elements ("reps" in the table), whereas sets store every element just ones and does not specify their order. Hence the order in which a set is converted to a set can be different from the order in which the set was defined:

```
>>> s = {4, 1, 2}
>>> list(s)
[1, 2, 4]
```

Dictionaries function as sets in these respects. But unlike sets, they allow indexing via **keys** (the things on the left sides of the colons).

The ordering is illustrated by how equality works. The following tests show that order matters for lists but not for sets:

```
>>> [2, 3] == [3, 2]
False

>>> {2, 3} == {3, 2}
True
```

What about dictionaries? A new implementation in Python 3.6 is reported to preserve the order in which the items are introduced [1]. However, this does not change the mathematical property that order is irrelevant (tested in 3.9.7):

```
>>> {1: 2, 2: 3} == {2: 3, 1: 2}
True
```

[1] see https://docs.python.org/3.6/whatsnew/3.6.html

Mutability and **item assignment** go hand in hand, except for sets, where indexing makes no sense. But sets are still mutable, by the `add()` and `remove()` methods:

```
>>> s = {4, 1, 2}
>>> s.add(8)
>>> s
{8, 1, 2, 4}
>>> s.remove(2)
>>> s
{8, 1, 4}
```

3.6 Tutorial 6: Modules

3.6.1 Importing

Modules can be **imported** from both the Python shell and from Python files in many different ways:

- `import math` makes `math.cos(), math.sin()` available
- `import math as m` makes `m.cos(), math.sin()` available
- `from math import cos` makes `cos()` available (but not `sin()`)
- `from math import cos, sin` makes `cos(), sin()` available
- `from math import *` makes `cos(), sin(), tan()`,... available

The last way of importing everything from a module without a prefix may be tempting, since it saves typing, but it also opens the way for hard to detect name clashes. It is mainly used in the Python shell when testing the functions of your own modules that are under development.

3.6.2 Running a main function

When a Python file is executed from the operating system or imported in the shell, all its top-level statements (i.e. ones not enclosed in functions or classes) are executed, e.g. all print statements. But nothing else is. Hence, even if a function has the name `main()`, it is not executed, if it is only defined but not called.

Calling `main()`, however, is not always desirable either. Hence the common idiom is to call it conditionally. The condition is that the module is executed as a main module, e.g. from the operating system shell. Then Python gives is the name `__main__`, and the conditional call can be written

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

Now, since the function name `main` has no special status, you can often see code where the same condition is prefixed to other kinds of code. This is of course fine - but it can still add clarity to the code if there is a function that has the name `main` and is used in the way specified.

3.6.3 Listing names with dir()

The built-in function `dir()` shows the names currently in scope

- `dir(m)` lists names in module `m`,
- `dir()` lists the names you have defined in the current session.

So it is a good way quicly to look up names. As we have seen, `help()` can then give more information about a name, especially if it has been given a document string.

3.6.4 The rest of Tutorial 6

There are some more details about

- the module search path,
- compilation to `.pyc` files,
- the `sys` module,
- packages

But we will save these details to later occasions, where we will need them.

3.7 Tutorial 7: Input and Output

3.7.1 String formatting

Assume you want to report the populations of countries, e.g.

```
country = 'Sweden'
population = 10_402_070
```

(yes, you can group digits in number literals with underscores!) The string you want to produce is

```
'the population of Sweden is 10402070'
```

This string can be produced in many ways:

- by using `+` between the parts, converted to strings if necessary:

  ```
  'the population of ' + country + ' is ' + str(population)
  ```

- with an **f-string** (string prefixed with `f`):

  ```
  f'the population of {country} is {population}'
  ```

- with the `format()` method:

  ```
  'the population of {} is {}'.format(country, population)
  ```

- with the `%` operator ("printf-style", inspired by the C language):

  ```
  'the population of %(c)s is %(p)d' % {'c': country, 'p': population}
  ```

The printf-style `%` operator is considered deprecated, but it is common in older Python code, so one has to be able to read it. It has a wide range of options about argument types and rounding.

F-strings are the most recent method and recommended since Python 3.6. Both f-strings and the `format()` method provide a wide range of possibilities, described in

https://docs.python.org/3/library/string.html#formatstrings

What can be included in the {} parts is therefore called a "Mini-Language". There are some differences between how arguments are given to f-strings and the `format()` method. One place to look is

https://www.python.org/dev/peps/pep-0498/

PEP is a series of **Python Enhancement Proposals**, where each new feature of Python is discussed before it is released. It is the ultimate source of explanations and design choices of those features.

The formatting "mini-language" has its own syntax, which is more compressed and therefore often harder to read than normal Python syntax. For example, argument types are one-letter symbols similar to the old printf style: `d` means decimal integers. It can often be avoided by modifying the arguments themselves with usual Python operators.

Here is an example: we want to print the populations of many countries in a table of fixed-width columns, where the country names are left-justified and their populations right-justified:

```
Iceland                      371580
Sweden                     10402070
```

We can do this with the mini-language syntax, where justification is indicated by the < and > signs:

```
f'{country:<24} {population:>12}'
```

But we can also use the normal string methods `ljust()` and `rjust()`:

```
f'{country.ljust(24)} {str(population).rjust(12)}'
```

The mini-language method is more compact, but needs the mastery of a new "language", and is also harder to read for those who do not know the language. The normal method needs more writing, and can be harder to read just because the code becomes longer. But it gives the full power of Python's string operations. The mini-language has more limited functionalities, but has on the other hand made some commonly used patterns readily available.

3.7.2 Reading and writing files

When you **open a file** in the **reading mode**, you get an **iterator** over the lines of the file. You can then read the lines one by one with the `next()` function:

```
>>> countries = open('countries.tsv', 'r')

>>> next(countries)
'Afghanistan\t36643815\n'

>>> next(countries)
'Albania\t3020209\n'
```

The file is read **lazily**: its contents are not stored in the memory, but read one by one and then forgotten. When you have reached the end of the file, `next()` gives an error.

You should always **close** a file when you do not need it any more:

```
countries.close()
```

A modern and recommended way is to read the file inside a `with` statement. This is the syntax used for **context managers**, of which opening and closing files is an example. Here is an example where each line of the file is printed in a formatted version:

```
with open('countries.tsv', 'r') as inf:
    for line in inf:
        data = line.split('\t')
        country, population = data[0].strip(), data[1].strip()
        print(f'{country:<24} {population:>12}')
```

(`split()` and `strip()` are standard methods for strings, see doc).

Now, this can give unwanted results because some country names are too long:

```
Saint Lucia                    178844
Saint Vincent and the Grenadines       109897
Samoa                          196440
```

Quiz (to be solved during the lecture if we have time): instead of 24, use the maximum length of country names.

If you want to **write** the result into another file, one way is to redirect the output in the operating system call of Python:

```
$ python3 print_pop.py >formatted-countries.txt
```

From inside Python, you can do this by opening a file in the `w` mode,

```
with open('formatted-countries.txt', 'w') as outf:
    outf.write(CONTENTS)  # a string that you have constructed
```

3.7.3 The rest of Tutorial 7

- some more methods on file objects: read if you need them
- the JSON format: to be covered in more detail in Lecture 3 (next chapter in this document)

3.8 Tutorial 8: Errors and Exceptions

The internals of error handling presuppose knowledge of classes, so we leave them to lectures 5 and 6. The most important features of useing error handling are given in an example in Tutorial 8.7:

```
def divide(x, y):
   try:
       result = x / y
   except ZeroDivisionError:
       print("division by zero!")
   else:
       print("result is", result)
   finally:
       print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
```

```
>>> divide(2, 0)
division by zero!
executing finally clause

>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'‘‘‘
```

3.9 Tutorial 9: Classes

The topic of classes will be discussed more thoroughly in Lectures 5 and 6 (Chapter 6 in this document). Here we will just mention some highlights.

3.9.1 Namespaces and scopes

The Tutorial chapter 9 starts with a discussion of **namespaces**, which are mappings of names (variables, functions, classes, modules) to objects. Namespaces are a feature of the runtime state of the program built in the computer's memory. In the source code, the related term **scope** means a part of the code where a name space is directly accessible, i.e. names can be used without qualifiers (e.g. `pi` instead of `math.pi`).

There is an unlimited hierarchy of namespaces and scopes. In an order opposite to the list in Tutorial 9.2, we have

- built-in names
- names in the current module (functions, classes, global variables)
- names in the current function (parameters and other local variables)
- names in function inside that function
- and so on, because functions can be nested

The notions of **global** and **nonlocal** variable are a bit tricky, in particular since the status can be modified by keywords. The code example in 9.2.1 is worth studying: the effect "After global assignment" can be surprising.

3.9.2 Class definitions

The format of class definitions is extremely liberal: it can consist of any statements, including just `pass`.

```
class <name>:
    <statements>
```

These statements define their own namespace, outside which the names must be qualified by the class name. These names are called **attributes**.

New **instances** of the class are created by using the class name as a function: "'<classname>() The instances can introduce each of their own attributes: there is no requirement that all instances of a class have the same attributes. This makes classes *extremely flexible* in Python.

However, there is a more restricted way to use classes, which is closer to the usual ideas of object-oriented design. Here is an example from two-dimensional graphics, with a typical structure and some terminology:

```
class Point:
    "points in a two-dimensional space"

    # creating a point
    def __init__(self, x, y):
        "create a point by giving x and y coordinates"
        # the instance variables _x and _y are "private"
        self._x = x
        self._y = y

    # public getters of the two coordinates
    def get_x(self):
        return self._x

    def get_y(self):
        return self._x

    # public setters of the two coordinates
    def set_x(self, x):
        self._x = x
```

```
    def set_y(self, y):
        self._y = y

    # any number of other methods
    def move(self, dx, dy):
        self.set_x(self.get_x() + dx)
        self.set_y(self.get_y() + dy)
```

A class can **extend** other classes and thereby **inherit** its attributes:

```
class ColoredPoint(Point):
    def __init__(self, x, y, color='black'):
        "give coordinates, and optionally color, default black"
        super().__init__(x, y)
        self._color = color

    def get_color(self):
        return self._color

    def set_color(self, color):
        self._color = color
```

The following session shows how instances can be created and accessed:

```
>>> p = ColoredPoint(3, 5)

>>> p.get_x()
3

>>> p.move(1, 10)
>>> p.get_x()
4

>>> p.get_color()
'black'

>>> p.set_color('red')
>>> p.get_color()
'red'
```

3.10 Tutorial 10 and 11: the Standard Library

In the labs and exercises of this course, we will need at least the following standard libraries and functions:

- `sys`, system: `argv` command line arguments
- `re`, regular expressions: methods for analysing strings
- `math`, mathematical functions: `sin()`, `cos()`, `pi`
- `random`: `choice()`, `randrange()`
- `statistics`: `mean()`, `median()`
- `urllib`, accessing the internet: `urlopen()`
- `timeit`, measuring the time to perform a computation: `timeit()`
- `collections`: `deque()`
- `json`, to structure and store data in files
- `csv`, to read and write files with comma-separated data (or with some other delimiter)
- `xml.etree.ElementTree`, to read and write XML data (in an extralab)
- `ast`, Abstract Syntax Trees, to analyse Python code (in an extralab)

We will also use libraries from other sources. Those will probably need to be installed with the `pip` program, for instance,

```
$ pip3 install graphviz
```

for the library we will use for the visualization of graphs.

3.11 Tutorial 12: Virtual Environments

This topic will be discussed in Lecture 9 on development frameworks.

3.12 Tutorial 13: What Now?

The most thorough and widely usable documents are

- the Python Language Reference, https://docs.python.org/3/reference/index.html#refere index
- the Python Standard Library, https://docs.python.org/3/library/index.html#library- index

Chapter 4

Storing and retrieving information

This chapter corresponds to Lecture 3.

4.1 Databases

A **database** is an integral part of many software systems. In general terms, it is a collection of data which is not a part of the program code, but resides in separate files or even on different servers around the Internet.

A programming language like Python would of course be able to express a database as a part of program code, by using dictionaries and lists. But this is normally not done. Instead, Python provides functions for **retrieving** data from a database, **storing** new data, and **updating** old data. Dictionaries and lists are used at runtime when the data is processed. The files can use textual formats such as JSON, CSV, and TSV, which are in focus in this chapter. In large databases, compressed binary data is common, but outside our focus.

In this chapter, we will look at two kinds of data: tabular and hierarchic (tree-like). We will look at how data is stored in files and how it is used in Python. Both of these tasks will then be trained in Lab 1. The lab will moreover emphasize a fundamental design principle: avoiding **redundancy**: one should "say each thing only once."

One aspect of databases that we do not address here is **persistent storage**. We will just read the database from file to memory and manipulate it

in memory, as a Python object. The data becomes persistent when we write it back to a file, but there is no guarantee of persistence at the intermediate stages: if the computer is turned off before saving to a file, the data is lost.

4.2 Tabular data

A simple example of a database is a **table**, which consists of a number of **rows** that contain pieces of information in different **columns**. Such a table can be stored as a text file, where each line contains a row, and the columns are separated by a **delimiter** such as a comma or a tabulator. **Spreadsheet** programs such as Excel can both read and write tables in such formats; Excel is indeed perhaps the most widely used technology for tabular databases, although not officially recognized as databases. A more advanced technology is **relational databases**, also known as **SQL** databases. While Python - no surprise! - supports them as well, they are beyond the scope here.

4.2.1 Reading tabular data from files

Here is an example: the beginning of a table with information about the countries of the world:

country	**capital**	**area**	**population**	**continent**	**currency**
Afghanistan	Kabul	652230	36643815	Asia	afghani
Albania	Tirana	28748	3020209	Europe	lek
Algeria	Algiers	2381741	41318142	Africa	dinar
Andorra	Andorra la Vella	468	76177	Europe	euro

The data could be stored in a **CSV file** (Comma-Separated Values), which uses the comma as delimiter,

```
Afghanistan,Kabul,652230,36643815,Asia,afghani
```

or a **TSV file** (Tab-Separated Values), which uses the tabulator `\t`,

```
Afghanistan\tKabul\t652230\t36643815\tAsia\tafghani
```

A complete such file can be found in

https://github.com/aarneranta/chalmers-advanced-python/blob/main/exercises/ex02/countries.

as material for the exercise session following this lecture.

Files of these forms can correspondingly be read into Python by using the `split()` method: for CSV,

```
with FILE.csv as file:
    data = []
    for line in file:
        data.append(line.split(','))
```

This simple piece of code is, however, too brittle. Special attention must be paid to situations where some field itself contains the delimiter character or a newline. The Python standard library `csv`

https://docs.python.org/3/library/csv.html

provides functions for dealing with such situations, with freely chosen delimiters (comma is just the default) and "dialects" such as Excel. Here is how to read a TSV file:

```
with open('FILE.tsv') as file:
    rows = csv.reader(file, delimiter='\t')
    data = [row for row in rows]
```

4.2.2 Processing tabular data in Python

Reading a file, either with the the above `split()` method or with the `csv` library, results in a list of lists of strings, such as

```
[
  ['country', 'capital', 'area', 'population', 'continent', 'currency'],
  ['Afghanistan', 'Kabul', '652230', '36643815', 'Asia', 'afghani'],
  ['Albania', 'Tirana', '28748', '3020209', 'Europe', 'lek'],
  ['Algeria', 'Algiers', '2381741', '41318142', 'Africa', 'dinar']
]
```

The first line is used as the **header** that gives the **labels**, also known as **attributes**, of each column. A simple conversion (left as exercise!) converts this to a list of dictionaries, where the keys are these labels:

```
[
  {'country': 'Afghanistan', 'capital': 'Kabul', 'area': '652230',
   'population': '36643815', 'continent': 'Asia', 'currency': 'afghani'},
  {'country': 'Albania', 'capital': 'Tirana', 'area': '28748',
   'population': '3020209', 'continent': 'Europe', 'currency': 'lek'},
  {'country': 'Algeria', 'capital': 'Algiers', 'area': '2381741',
   'population': '41318142', 'continent': 'Africa', 'currency': 'dinar'}
]
```

In many applications, one would like to see the country names as keys. For this purpose, one can convert this list into a dictionary of dictionaries:

```
{
 'Afghanistan': {'capital': 'Kabul', 'area': 652230, 'population': 36643815,
   'continent': 'Asia', 'currency': 'afghani'},
 'Albania': {'capital': 'Tirana', 'area': 28748, 'population': 3020209,
   'continent': 'Europe', 'currency': 'lek'},
 'Algeria': {'capital': 'Algiers', 'area': 2381741, 'population': 41318142,
   'continent': 'Africa', 'currency': 'dinar'}
}
```

Notice that we have also converted numerical values from strings to integers. Producing this format is also left as an exercise!

The last dictionary is perhaps the most natural form for a database of countries. Having countrynames as dictionary keys makes it fast to look up information about each country.

```
>>> cdict['Sweden']['population']
10379295
```

One can also make other **database queries**, not quite as fast, but still in a natural way, by using comprehensions: "'[c for c in cdict if cdict[c]['currency'] == 'euro'] returns the list of Euro countries. Finally, one can perform **aggregations** such as maximum, minimum, and average:

- *Which country has the largest population in Africa?*

```
max([c for c in cdict if cdict[c]['continent']=='Africa'],
    key=lambda c: cdict[c]['population'])
```

- *What is the average area of countries in Europe?*
- *How many countries are there in South America?*

Here are more complex ones:

- *Which continent has the smallest total population?*
- *List all continents and their total populations in the descending order of population.*

They require what in the database world is called **grouping**: we group the data along a new key, which is 'continent' in this case. This can be done by building an auxiliary dictionary with just continents and their populations. Some more queries are given as exercises.

4.2.3 Saving data in JSON

Storing data in TSV or CSV files is not ideal, because

- these formats are not specified precisely but have different "dialects",
- reading the data in desired format (such as the country dictionary) is complicated.

A better solution is to use JSON (JavaScript Object Notation), which is a standard data interchange format on the web. The Python standard library `json` provides functions for reading and writing it, in a way that is much simpler and less error-prone than `csv`:

https://docs.python.org/3/library/json.html

Provided that you have converted the original data into a dictionary as shown above, you can print it into a JSON file as follows:

```
with open('full-countries.json', 'w') as jfile:
    json.dump(cdict, jfile, indent=4)
```

The resulting file looks as follows:

```
{
    "Afghanistan": {
        "capital": "Kabul",
```

```
        "area": 652230,
        "population": 36643815,
        "continent": "Asia",
        "currency": "Afghan afghani"
    },
    # etc
}
```

It can easily be read back into a Python dictionary:

```
with open('full-countries.json', 'r') as file:
   jdict = json.load(file)
```

4.3 The JSON format

JSON is more expressive than tabular data. It was designed to store datastructures of the JavaScript language, but it also has a close correspondance with Python structures. The following table defines a mapping between JSON and Python objects.

JSON	**Python**
object	dict
array	list
string ("foo")	str ('foo')
number (int)	int
number (real)	float
true	True
false	False
null	NONE

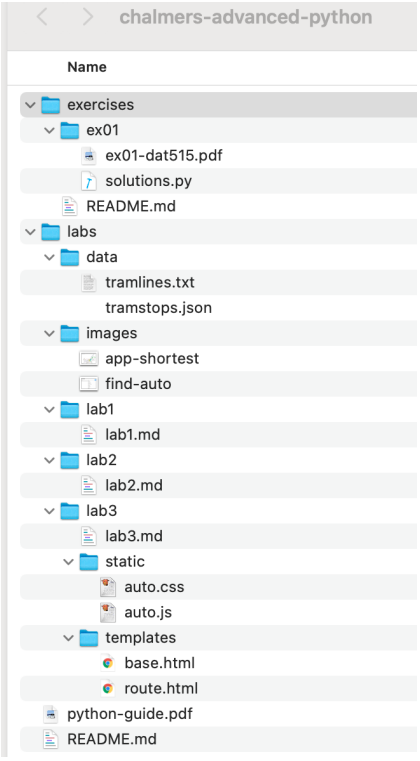
This table is a part of the full documentation in

https://docs.python.org/3/library/json.html

4.3.1 Hierarchic data: trees

Hierarchic data, such as **trees**, can be naturally represented in JSON. An example of trees is the **directory structure** of **file systems**. Here is the current structure of the course GitHub in

https://github.com/aarneranta/chalmers-advanced-python



A simple JSON encoding is to represent

- each directory name as a key with a subtree as value,
- each file name as a key with null as value.

Then the `exercises` subdirectory is represented as

```
{"chalmers-advanced-python": {
    "exercises": {
        "ex01": {
            "ex01-dat515.pdf": null,
            "solutions.py": null
     },
        "README.md": null
    }
  }
}
```

Constructing the tree for the whole directory in the picture is left as an exercise. The best way to do this is to automate it by writing a function

which, for any directory, constructs a dictionary from which the JSON object can be dumped, by using

- `os.listdir(dir)` to list the contents of a directory,
- `os.path.isdir(file)` to test if a file is a directory,
- `os.path.join(path, file)` to add a path to a file name.

from the Operating System library `os`,

https://docs.python.org/3/library/os.html

This is left as exercise.

Trees are also a special case of **graphs**, which will have a central role in Labs 2 and 3.

4.4 Lab 1 introduction

We will go through the specification in

https://github.com/aarneranta/chalmers-advanced-python/tree/main/labs/lab1

4.5 Testing in Lab 1

As a part of the labs, you are expected to maintain a **test suite** and run it continuously on your solution. In Lab 1, we will start by a simple usage of Python's `unittest` library,

https://docs.python.org/3/library/unittest.html

In later labs, we will introduce some more testing techniques, but the tests created for each lab will follow throughout the later labs. This is a part of the **continuous integration** practice, together with version control and automatic build procedures.

Chapter 5

Graphs and graph algorithms

This chapter corresponds to Lecture 4. We will cover most of the material that will be needed for the first part of Lab 2: the file 'graphs.py'. The Lab specification is in

https://github.com/aarneranta/chalmers-advanced-python/blob/main/labs/lab2/lab2.md

We will illustrate the presentation with the tram network graph built in Lab 1, but also with other - mostly smaller, but also some larger - graphs.

5.1 What is a graph

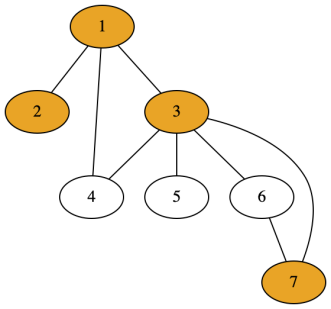
We will start with the usual definition of graphs in discrete mathematics and computer science, as explained for instance in

https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

A graph consists of

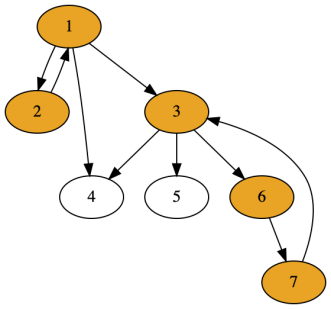
- a set of **vertices**, also known as **nodes**
- a set of **edges**, also known as **links**, that connect pairs of vertices

The following picture shows a simple graph with 7 vertices and 8 edges:



The coloured vertices mark the **shortest path** from vertex 2 to vertex 7. Finding shortest paths is one of the most well-known **graph algorithms**, used in numerous route-finding systems and also in our Lab 2. The shortest path in this simple case is the traversal that goes through the minimum number of vertices. This is why for instance the transition from 3 to 7 is by the direct edge, not via vertex 6.

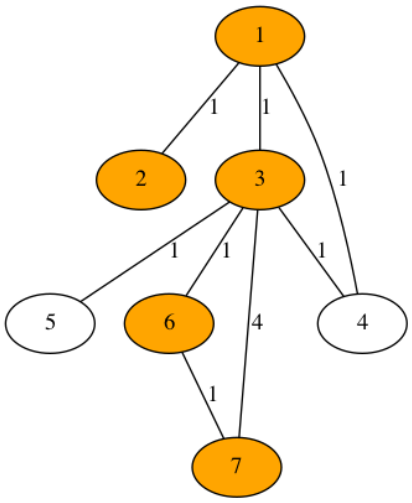
The above picture shows an **undirected graph**, which means that transitions are possible in both directions at every edge. In a **directed graph**, edges are one-way streets, which in diagrams are marked by arrow heads. The following graph has the same vertices and edges as before, but now directed.



Notice that the shortest path from 2 to 7 is now different, since the direct line from 3 to 7 is no more possible. Also notice that we have 2 edges connecting 1 and 2, to mark that both directions are possible.

The shortest path solution can also vary because of **weights** associated with edges, also known as **costs**. Imagine, for instance, that the direct way

from 3 to 7 is with a superfast special train that costs 4 times as much as a usual ticket. If we use price as **cost function**, the shortest path algorithm now gives us the **cheapest path**. In the following picture, we have marked the prices as weights of edges. Since the price is used as cost function, the shortest path from 3 to 7 is different from the the first graph where, technically, the cost function is a constant function.



5.2 Graphs for transport networks

In Labs 2 and 3, we are mostly working with undirected graphs, because it is a feature of Gothenburg tram lines that they can always be travelled in both directions. However, since this is not a universal feature of transport networks, our datatypes and algorithms will also cover directed graphs. If properly designed, this generality comes with very little overhead in the code - and such design is, indeed, one of the wider learning outcomes of this course.

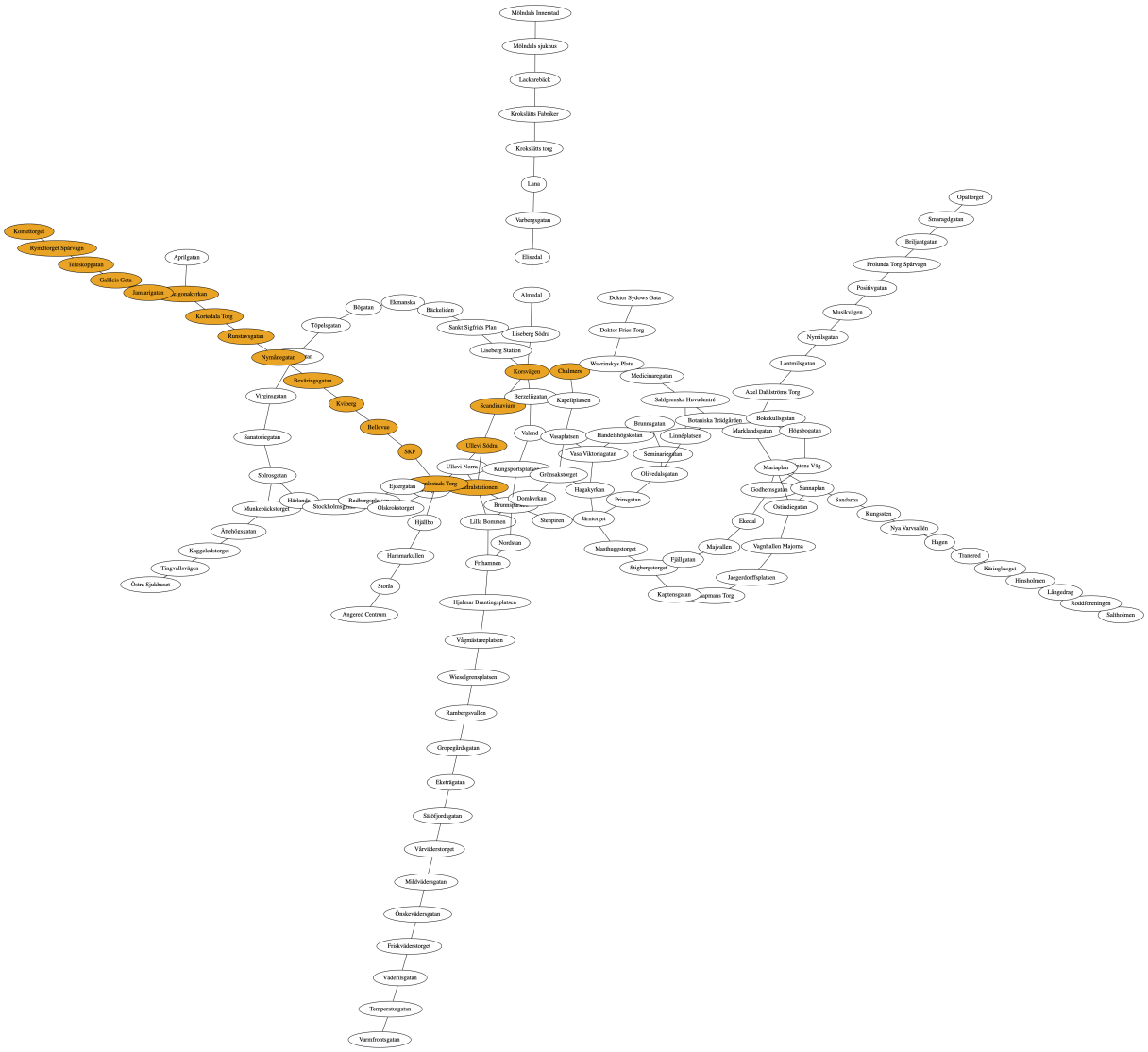
Weights and cost functions will, however, play an important role for us. In general, the **best path** in a network can depend on many different things:

- time,
- geographical distance,
- price,
- number of changes,
- preference to travel via a certain place or route,

- preference to use a certain mode of transport (tram, bus, ferry) or provider (train company, airline).

Full-scale travel planning applications can optimize for all these features, and many more. Also our solution in Lab 2 will be ready for this. But the final app in Lab 3 will only consider time, distance, and number of changes.

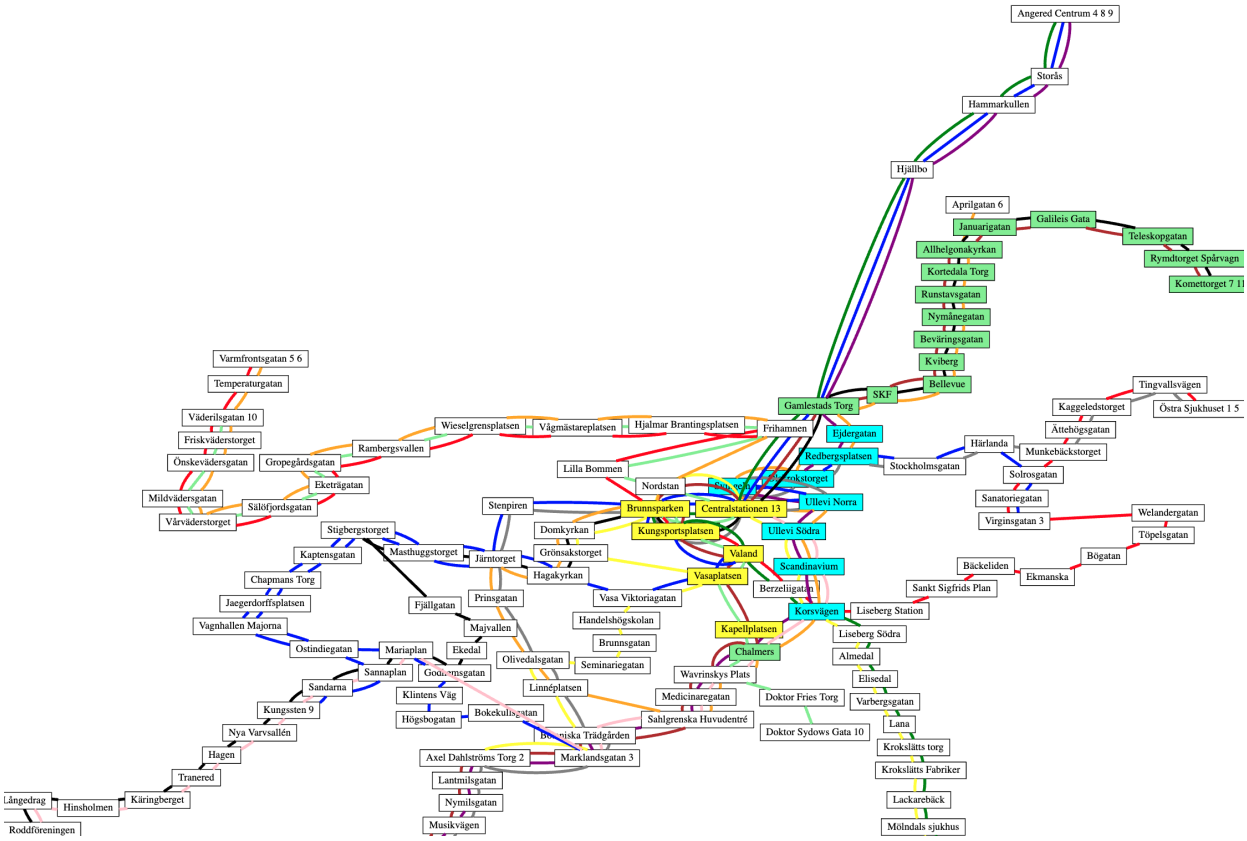
The following picture shows the Gothenburg tram network as a graph built from the data from Lab 1, with a shortest path from Chalmers to Komettorget.



This is the level of visualization we will reach in Lab 2. In Lab 3, visualization is a central topic, and we will be able also to show

- stops in realistic geographical locations,
- colours of tram lines,
- different colours for shortest time and shortest distance, if different.

The following picture is a screenshot form the app, also shown in Lab 3 specification.



5.3 Representations of graphs

A graph is, mathematically, given by

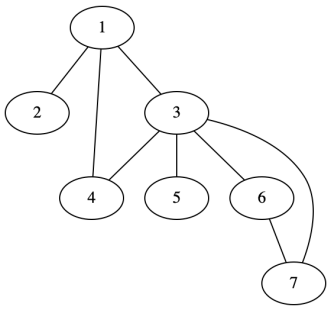
- a set of vertices,
- a set of edges, represented by pairs of vertices.

The set of vertices might look redundant, since it can usually be derived from the set of edges. However, since graphs can also contain isolated vertices (with no edges), it is necessary to include that set. (If storage space is to be optimized, it is of course enough to store the isolated vertices, but this is not what we are going to do.)

In undirected graphs, edges `(a,b)` and `(b,a)` are considered equal, and need not be repeated. In directed graphs, each of the pairs must be stored separately, if both directions work. (Alternatively, one can save space by

using a special format or separate set for bidirectional edges, but this is again something we will not consider.)

A straightforward representation of our first example graph,



is thus with two lists - in Python notation,

```
vertices = [1, 2, 3, 4, 5, 6, 7]
edges = [(1,2),(1,3),(1,4),(3,4),(3,5),(3,6),(3,7),(6,7)]
```

An alternative to this representation is **adjacency list**: a dictionary which to every vertex associates a list of its **neighbours** - that is, the vertices to which it has an edge.

```
adjlist = {
   1: [2, 3, 4],
   2: [1],
   3: [1, 4, 5, 6, 7],
   4: [1, 3],
   5: [3],
   6: [3, 7],
   7: [3, 6]
   }
```

In this representation, we do not need to store vertices separately, since isolated vertices have simply an empty list associated. Notice also that edges are given in both directions even for undirected graphs. This would not be necessary for data representation, but makes it faster to look up the neighbours of each vertex.

Yet another representation is **adjacency matrix**, which is a square matrix with rows and columns for every vertex. Every cell (i,j) has value 1 if there is an edge from i to j, otherwise 0. The simplest Python representation is a list of lists:

```
[
  [0, 1, 1, 1, 0, 0, 0],
  [1, 0, 0, 0, 0, 0, 0],
  [1, 0, 0, 1, 1, 1, 1],
  [1, 0, 1, 0, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 1],
  [0, 0, 1, 0, 0, 1, 0]
]
```

together with the list of vertices, which gives the order in which the rows and columns represent the vertices. Adjacency matrices are easy to generalize to weighted graphs: store the weight of the edge instead of 1, and `None` instead of 0. They are not optimal for **sparse graphs**, for which they contain mostly zeros. Therefore we will not use them in our lab, but we will consider them in exercises.

The three representations of graphs mentioned above can be converted to each other, which will be covered in Lab2 and in the exercises. They also enable the most important **graph operations**, which are needed when graphs are used in computing tasks.

- looking up neighbours,
- looking up **predecessors** and **successors** (for directed graphs),
- adding and removing edges and vertices,
- getting and setting weights of edges,
- getting and setting **values** of vertices, which means extra information associated with them (for instance, the geographic locations of tram stops).

Now it is time to set this all in practice and look at the implementation details.

5.4 Implementing graphs in Python

This section is partly a repetition from Lab2 spefication. We will define a class, to be named `Graph`, to store graphs. The class will have an internal storage of vertices and edges, using internal (hidden) instance variables. As the case is in Python, we cannot really make these variables "private", but we can start their names with underscores as conventional. More important than naming is to make sure that the hidden variables are never needed when graphs are used, but that enough "public" methods are provided by the class.

We will go into more details about class definitions in next lecture. Here it is enough to show the shape of the expected class definition:

```
class Graph:
  def __init__(self, start=None, values = None, directed=False):
      self._adjlist = {}
      if values is None:
          values = {}
      self._valuelist = values
      self._isdirected = directed

      # plus some code for building a graph from a 'start' object
      # such as a list of edges

  # here are some of the public methods to implement
  def vertices(self):
  def edges(self):
  def neighbours(self,v):
  def add_edge(self,a,b):
  def add_vertex(self,a):
  def is_directed(self):
  def get_vertex_value(self, v):
  def set_vertex_value(self, v, x):
```

We have made some design choices that could be made differently, as you will notice if you look at graph implementations in other sources:

- We choose to store the graph internally as an adjacency dictionary.
- We use the same class for directed and undirected graphs.

- We include the values of vertices in the basic Graph class.

  Important notice: we use ‘None‘ as the default value for both the ‘start‘ and ‘values‘ arguments. A more natural choice would be an empty list or dictionary. However, since these are mutable objects, and since the evaluation of default arguments is performed only once, changes in these objects in any instance of the ‘Graph‘ class would also affect all other class instances. This is the behaviour that the official Python tutorial, section 4.8.1, warns about.

  A **weighted graph** is a subclass, which just adds weights of edges (such as transition times between tram stops):

```
class WeightedGraph(Graph):

  def set_weight(self, a, b, w):
  def get_weight(self, a, b):
```

  All of this will be your job in Lab 2 to implement. During the lecture, we will do some live coding that uses my own implementation to show examples of how the methods are used.

5.5 Graph algorithms: search and connectedness

One of the learning outcomes in this course is to convert pseudocode to actual Python implementations. This ability will be tested in Lab 2 with a shortest path algorithms; see next section. We will prepare for it here by a slightly simpler algorithm: **breadth-first search**.

The word "search" refers to one of the applications of the algorithm: go through the vertices of a graph (following edges) until you find a vertex with a specified property. A special case is searching for the uniformly false property. The algorithm can then be made to return the sequence of vertices traversed. If this sequence contains all the vertices of the graph, the graph is **connected**. Otherwise, the graph has disconnected parts, such as isolated nodes. Another example is if we put together the tram networks of Gothenburg and Norrköping: the result will still be a graph, but it will consist of two disconnected parts.

We will implement the pseudocode from

https://en.wikipedia.org/wiki/Breadth-first_search

The same article also shows an animation that you can watch. The following code listing shows the Wikipedia pseudocode side by side with a Python implemenration directly translated from it.

```
 1  procedure BFS(G, root) is
 2      let Q be a queue
 3      label root as explored
 4      Q.enqueue(root)
 5      while Q is not empty do
 6          v := Q.dequeue()
 7          if v is the goal then
 8              return v
 9          for all edges from v to w in G.adjacentEdges(v) do
10              if w is not labeled as explored then
11                  label w as explored
12                  Q.enqueue(w)
```

```
def BFS(G, node, goal=lambda n: False):
    Q = deque()
    explored = [node]
    Q.append(node)
    while Q:
        v = Q.popleft()
        if goal(v):
            return v
        for w in G.successors(v):
            if w not in explored:
                explored.append(w)
                Q.append(w)
    return explored
```

The main difference is that we have added the `goal` function as argument and a `return` statement at the end.

The pseudocode uses the data structure of **queues**, which we in Python import from the standard library `collections.deque`

https://docs.python.org/3/library/collections.html

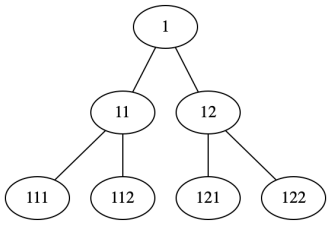
The main thing we will need from this library is getting the first element of a list, by the method `popleft()`. In practice, when the graph is small, we could just use ordinary lists and the method `pop(0)`. But knowing when to use queues as opposed to lists is a basic skill that advanced programmers should master:

- lists are used for **stacks**, with the LIFO principle: last in, first out;
- queues are used the FIFO principle: first in, first out.

An example of where stacks are used - and slightly easier to understand - is **depth-first search**, which is left as an exercise following the pseudocode from

https://en.wikipedia.org/wiki/Depth-first_search

The difference between the two traversals can be clearly seen from the following example:



The nodes are visited in the following order:

```
DEPTH-FIRST:   [1, 12, 122, 121, 11, 112, 111]
BREADTH-FIRST: [1, 11, 12, 111, 112, 121, 122]
```

Thus depth-first follows each branch in the tree till the end before starting with the next branch. Breadth-first proceeds "layer by layer", traversing all intermediate neighbours before going to the neighbours below them.
THIS IS THE POINT WE REACHED AT LECTURE 4

5.6 Graph algorithms: shortest path

The algorithm to be implemented in Lab 2 is the shortest path algorithm of Dijkstra (named after the Dutch computer scientist E.W. Dijkstra, who invented this algorithm in 1956). Therefore it is commonly known as **Dijkstra's algorithm**.

The implementation is suggested to follow the pseudocode from

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

The same page also gives animations and explanations, which we will look at during the lecture if time permits.

More details on what is expected can be found in Lab 2 specification. And much more details and depth about graph and other algorithms (including stacks and queues) can be found from data structure courses and books. The following free on-line books are used at the Chalmers course:

https://chalmersgu-data-structure-courses.github.io/OpenDSA/Published/ChalmersGU
DSABook/html/index.html

https://runestone.academy/runestone/books/published/pythonds3/index.html

You can also take a look at the `networkx` library,

https://networkx.org/

which is designed to work for large and complicated graphs of various kinds. You could actually implement most of Lab 2 by just using `networkx` - but this time, we do not allow this, because we expect you to work out the basics yourself. However, we will encourage you to try this as an exercise.

5.7 Baseline visualization

We will use the `graphviz` library,

https://graphviz.readthedocs.io/en/stable/api.html

Not very much will be needed at this point: the main things are

- the class `graphviz.Graph`,
- its method `node(v)` to add a vertex,
- its method `edge(v, w)` to add an edge,
- its method `render()` to draw the graph.

And yes, the `Graph` class has the same name as our class, but this is no problem unless we import `*` from `graphviz`. The basic algorithm in visualization is simply:

1. create an instance of `graphviz.Graph`,
2. loop over the vertices of your graph applying `node()` to them,
3. loop over the edges of your graph applying `edge()` to them,
4. `render()` the result

We will come back to visualization in Chapter 7 (Lecture 7), and introduce more functionalities of `graphviz` but also take a look at other visualization libraries.

5.8 Other examples of graphs

5.8.1 Knowledge graphs from WikiData

WikiData is a **knowledge base** that is increasingly used for storing facts reported in the Wikipedia:

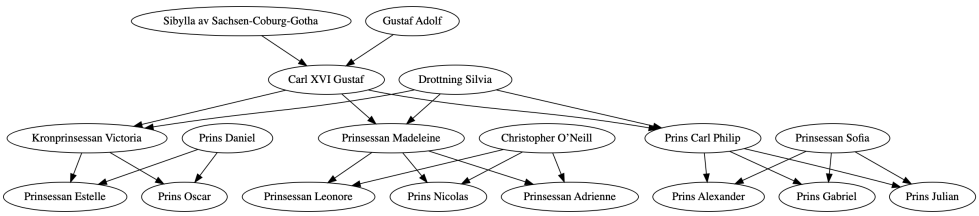
https://www.wikidata.org/wiki/Wikidata:Main_Page

A direct way to use WikiData is to send **SPARQL queries**, using the query language SPARQL (SPARQL Protocol and RDF Query Language). SPARQL has some similarities with SQL, but it is designed for **graph databases** rather than relational ones.

Graphs in WikiData consist of **RDF triples**, (Resource Description Framework), which are meant to represent the subject-predicate-object relations. Here are examples from the Swedish Royal Family:

- CarlGustaf-father-Victoria
- Silvia-mother-Victoria

Such triples can obviously be used for building graphs, where subjects and objects are vertices and predicates are labelled edges.



The query for retrieving this data can be found in

https://w.wiki/4BX9

A special feature of knowledge graphs is that there can be more than one edge between two vertices, representing different relations. For instance, a person can be the mother of another one, and at the same time her teacher. How to represent these situations will be left as an exercise in object-oriented design.

5.8.2 Maps and graph colouring

Another example from WikiData is built on the relation "country X is a neighbour of country Y".

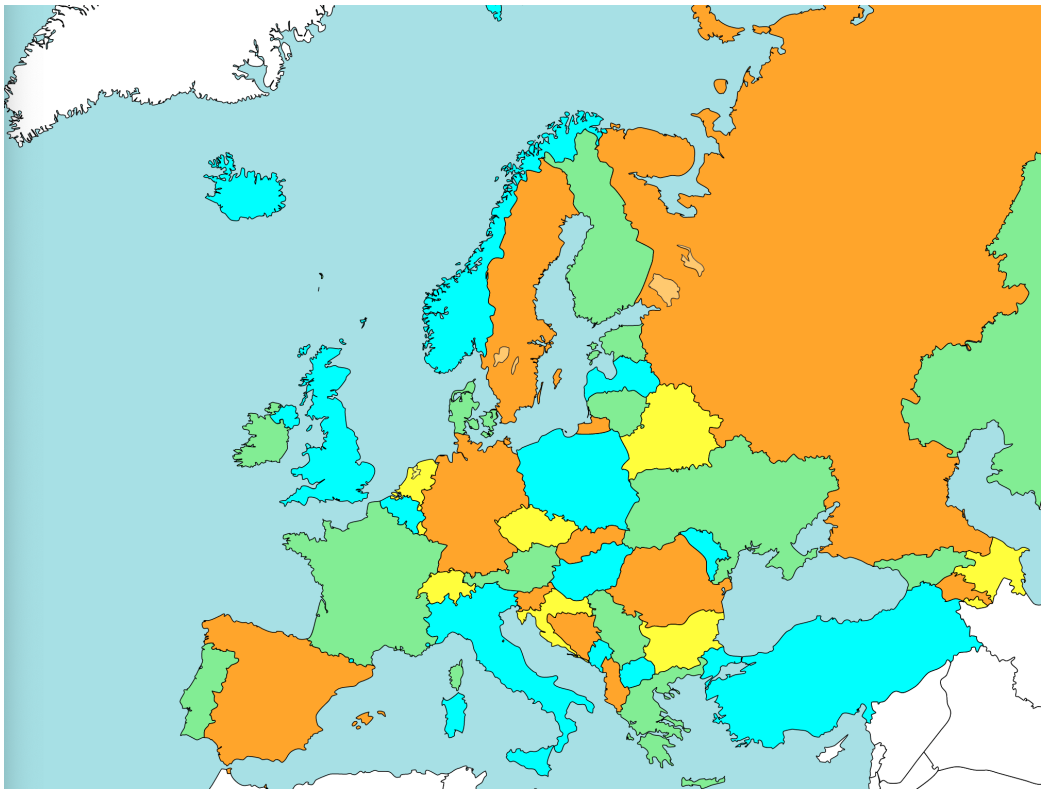
https://w.wiki/4EuG

This graph can be used as input for a **graph colouring algorithm**, which enables always selecting different coulours for countries that are neighbours. The famous **four-colour theorem**

https://en.wikipedia.org/wiki/Four_color_theorem

states that **planary graphs** - of which two-dimensional maps are an example - it are always possible to colour by just four different colours.

Graph colouring will be specified as one of the optional extra labs of this course. An example is a map of Europe coloured by using the algorithm:



Deciding if a graph is planar is an interesting field of algorithms in itself.

5.8.3 Social networks

Social networks defined by "person X is connected to person Y" is a field where very large graphs can easily be found. Interesting questions about such graphs are

- how connected the graph is,
- how many connections persons have,
- what **clusters** there are, i.e. subgraphs of closely related vertices.

An anomymized example can be found in

https://snap.stanford.edu/data/ego-Facebook.html

We will use this as one of the examples for another extra lab, **clustering**.

5.9 Testing graphs with randomly generated data

Graph algorithms are very subtle, and it is easy to forget testing them with all relevant kinds of cases. Therefore, we will in Lab 2 introduce a technique known as **property-based testing**. This technique uses randomly generated data, covering a large number of test cases. It is made available by the `hypothesis` library,

https://hypothesis.works/ (main page)

https://hypothesis.readthedocs.io/ (documentation)

5.9.1 Property-based testing: first example

The Hypothesis documentation starts with a quick start example, which we have modified just a bit:

```
from hypothesis import given, strategies as st

@given(st.integers(), st.integers())
def test_ints_are_commutative(x, y):
    assert x - y == y - x

test_ints_are_commutative()
```

When we run this (written in the file `htest.py`), we get

```
$ python3 htest.py
Falsifying example: test_ints_are_commutative(
    x=0, y=1,
)
```

The function name indicates that we wanted to test commutativity, but the code indicates that it is the communtativity of subtraction - which obviously should fail! Hypothesis has found the simplest possible counterexample. Trying out the same with `+` instead of `-`, no counterexamples are found.

Now, we are interested in more subtle properties, having to do with our own functions rather than `+` or `-` of integers. But first we must look a bit closer to the simple example code, since it uses a Python construct that we have not seen before: a **decorator**

5.9.2 Decorators

A decorator in Python is a "function returning another function", as explained in

https://docs.python.org/3/glossary.html#term-decorator

The decorator function, prefixed with the `@` symbol, is written on the line before the definition of the decorated function. Decorators are **syntactic sugar**, which means that they can always be converted to "normal" syntax and thereby avoided:

```
@dec                                    def f(...):
def f(...):         CONVERTS TO             ...
    ...                                 f = dec(f)
```

The decorator syntax is used when it is considered more readable, which is often the case in libraries. Hypothesis is the first example we see at this course. In the example above, converting the decorator to normal syntax would give us

```
def test_ints_are_commutative(x, y):
    assert x - y == y - x

test_ints_are_commutative = given(st.integers(),
           st.integers())(test_ints_are_commutative)
```

If you are used to the decorator syntax, it will certainly look more readable! However, if you have not seen it before, it may look like magic.

5.9.3 Strategies for graphs

Random values are generated by `strategies`, which have methods for many datatypes of Python, as explained in

https://hypothesis.readthedocs.io/en/latest/data.html

Here is a simple way to create a strategy for graphs:

```
# generate small integers, 0...10
smallints = st.integers(min_value=0, max_value=10)

# generate pairs of small integers
twoints = st.tuples(smallints, smallints)

# generate lists of pairs of small integers
# where x != y for each pair (x, y)
st_edge_list = st.lists(twoints,
  unique_by=(lambda x: x[0], lambda x: x[1]))
```

The generation would also work without the `min`, `max`, and `unique` arguments, but it would produce more uninteresting example, for instance, very large integers as vertices.

As an example, let us test whether depth first and depth first search always give the same results:

```
@given(st_edge_list)
def test_searches(eds):
    G = Graph()
    for (a,b) in eds:
        G.add_edge(a, b)
    root = eds[0][0]
    assert breadth_first(G, root) == depth_first(G, root)
```

As expected, this is not the case:

```
Falsifying example: test_searches(
    eds=[(0, 2), (1, 0), (2, 1)],
)
```

The actual results can be seen by adding suitable print statements:

```
BREADTH FIRST [0, 2, 1]
DEPTH FIRST [0, 1, 2]
```

However, what is actually more interesting to test is if the two search methods find the same **sets** of nodes, even if the order differs. Changing the assertion and testing again should not show any counterexamples:

```
assert set(breadth_first(G, root)) == set(depth_first(G, root))
```

The `hypothesis` library provides many more ways to define, combine, and restrict strategies, but you can get started with these simple ones to generate random graphs.

Chapter 6

Object-oriented design

This chapter corresponds to lectures 5 (second part) and 6.

Python supports object-oriented program design by enabling the definition of classes as well as **class hierarchies** using **inheritance**, which can be **multiple** like in C++ but unlike Java. Classes also support **polymorphism** by **method overriding**, but one should at the same time point out that Python is intrinsically more polymorphic than many other languages because of dynamic typing.

Python does *not* support all aspects of object-oriented programming: in particular, **information hiding** is not forced by the language but depends totally on the programmer's discipline in following certain conventions. Neither does Python *force* object-oriented programming like Java does, but permits a mixture of functions and classes like C++.

However, and very importantly, Python *is* inherently object-oriented due to its **data model**, which is entirely based on objects. Even basic types such as `int` are actually types of objects in this data model. This gives Python a special flavour that is rarely seen in elementary programming but exploited heavily in more advanced Python.

We will therefore start this chapter with an overview of Python's data model and how it is accessed by programmers. After that, we will look at object-oriented concepts proper, focusing on the role of data abstraction and code reuse, which are essential in all program design. Finally, links to traditional object-oriented design will be given by **UML models** and **design patterns**.

6.1 The data model of Python

The primary source for this section is Chapter 3 of the Python reference manual,

https://docs.python.org/3/reference/datamodel.html

Just like when reading the Tutorial, we will go through the standard document itself and just list here some highlights to which special attention should be paid.

6.1.1 Operator overloading

Of special interest are the **special method names** explained in Section 3.3 of the reference manual. They make it possible to define standard operations, such as arithmetic, comparison, and iteration, for any class. This is known as **operator overloading**.

The special names have the format where a double underscore `__` appears on both sides. These methods are therefore sometimes called **dunder methods** ("double under"). In the normal usage of classes, the programmer never sees them, but only their effects on standard Python operators such as `==` and `+`.

For many Python programmers, defining dunder methods is seldom a relevant task. The only exception is `__init__()`, which defines how new instances of a class are built. But dunder methods definitely belong to the toolbox of advanced programmers, and knowing about them gives a deeper understanding of how Python works.

Again with the official reference, Section 3.3, as our main source, let us list some examples of dunder methods and what their effects might be for graphs:

```
__repr__(), exact representation string:
  >>> G = Graph([(1, 2), (1, 3), (2, 4)])
  >>> print(G.__repr__())
  Graph({1: [2, 3], 2: [1, 4], 3: [1], 4: [2]})

__str__(), conversion to string:
  >>> print(G)
  {1: [2, 3], 2: [1, 4], 3: [1], 4: [2]}
```

```
__eq__(H), equality:
  >>> H = Graph([(3, 1), (4, 2), (1, 2)])
  >>> print(G == H)
  True

__len__(), length:
  >>> print(len(G))
  4

__getitem__(), get item by key:
  >>> print(G[1])
  [2, 3]

__setitem__(), set value of key:
  >>> G[1] = [4, 5]
  >>> print(G)
  {2: [4], 3: [], 4: [2, 1], 1: [4, 5], 5: [1]}

__iter__(), iterate over elements:
  >>> for v in G: print(G[v])
  [4]
  []
  [2, 1]
  [4, 5]
  [1]

__add__(H), addition:
  >>> print(G + H)
  {2: [4, 1], 4: [2, 1], 1: [4, 5, 3, 2], 5: [1], 3: [1]}

__neg__(), unary negation:
  >>> G = Graph([(1, 2), (1, 3), (2, 4)], directed=True)
  >>> print((-G).edges())
  [(2, 1), (3, 1), (4, 2)]
  >>> print(-G == G)
  False
```

The definitions of these methods is not always obvious. If there is no natural choice, it is usually better not to define them. Let us consider just one example: unary negation:

```
def __neg__(self):
    if self.is_directed():
        H = Graph(directed=True)
        for (a, b) in self.edges():
            H.add_edge(b, a)
        return H
    else:
        raise TypeError('negation not defined for undirected graphs')
```

The idea is that only directed graphs can be negated, and this is done by reversing all the edges. For undirected graphs, a `TypeError` is raised, similar to what you get when for instance trying to negate a string or a list.

This definition does make some sense, but it is by no means obvious. It could be compared to defining the negation of a list as its reverse. Since this is not provided by standard Python, we might prefer not to define negation for graphs either.

Some other dunder methods will be given as an an exercise.

6.2 How to define classes

6.2.1 Abstraction

6.2.2 Inheritance

Diamond properties: https://www.python.org/download/releases/2.3/mro/

6.2.3 Polymorphism

6.2.4 Ad hoc uses of classes

Class vs. dictionary vs. labelled tuple.

6.3 UML, Unified Modelling Language

6.4 Design Patterns

Chapter 7

Visualization

Lecture 7.

Chapter 8

Web programming

Lecture 8.