A Guide to Advanced Programming in Python

Aarne Ranta

October 28, 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 statics 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 code can be very long, as there is no attempt to abstraction or avoidance of repetition,
- 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

Lectures 1 and 2: 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 Diving into the official tutorial

In this section, we will show some highlights of the official Python tutorial in

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

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.

2.3.1 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.

2.3.2 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.

2.3.3 Tutorial 3: An informal introduction to Python

3.1.1 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
  ```

3.1.2 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.

3.1.3 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.

3.2 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.

2.3.4 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

```
def 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.

2.3.5 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.

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, in virtue of **keys**.

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}
```

THIS IS THE END OF LECTURE 1.