DA2004/DA2005: Programming techniques

Lars Arvestad, Christian Helanow, Anders Mörtberg, Kristoffer Sahlin

Contents

Chapter 1

Getting started with Python

1.1 Programming and Python

These course notes are about *programming*, i.e., how to write code that a computer can execute to solve different computational problems. We will use the programming language *Python*, but many of the basic ideas are valid also for other programming languages (Java, JavaScript, C/C++, Ruby, Scala, Haskell, OCaml, ...). Python is very popular programming language (do a web search for "most popular programming languages"!) and has applications in just about any field. It is used in most companies, including IT giants like Instagram, Google, YouTube, Spotify, Amazon, et.c.

We will use version 3.x of Python. We also recommend installing *Anaconda*, a tool for managing packages and environments for Python. There are installation instructions on the course web, and it is not hard to find other instruction on the web, including on YouTube.

A *programming editor* or *coding editor* supports "syntax highlighting" and tools that make programming easier. An *IDE*, "Integrated Development Environment", supports large programming projects, but is harder to use. We will use the editor *Spyder* in this course, mainly because it is installed together with Anaconda, and is then setup nicely for Anaconda.

Other programming editors include:

- PyCharm Community Edition — "community edition" is free, good integration with Python.
- Visual Studio Code — Microsoft product that is a free download, has grown quite popular in recent years.
- Atom.io
- Sublime
- BBEdit — limited version is free
- For enthusiasts, we recommend two classics with a higher learning threshold:
  - Emacs
  - Vim

1.1.1 Programming in Python

When programming in Python, you write code in a file with the suffix `.py`, using your favorite editor (which in this course might be Spyder). Your code is then sent to a Python *interpreter* that reads and executes the code.

Starting the Spyder application from within Anaconda Navigator, you will get three panels: 'editor', 'help', and 'console'. In this text all code written in the editor has a gray background. We cad for example write these three code lines in Spyder's editor:

```
print(17)
print(1.1)
print('DA2005')
```

Pushing the green triangle up left, or perhaps using the keyboard shortcut "F5", will make Python execute the code in the console, where the output will appear. If you have not saved your code, there will first be a dialog suggesting you save the file. We suggest you save your code for this course in a new folder among your documents, perhaps named "DA2005". It is a good idea to structure this folder with different subfolders for the different assignments in the course.

As an exercise, we can name our first code file "f1.py" and if it contains the code above, you might see the following output (somewhat dependent on whether you are using MS Windows, Linux, or Mac):

```
runfile('/Users/arve/Documents/DA2005/f1.py',
    wdir='/Users/arve/Documents/DA2005/')
17
1.1
DA2005
```

Note: In this text, output from computations are written with a turquoise background.

1.2 Style

1.2.1 Representing code

This is a course in English, so of course that will be the main language. However, for coding in general, it is advised to always use English. This is common practice, in part because coding is often done in international environments. Many programmers prefer English also in very national environments since the programming language is in English.

When it is obvious that we are writing code in the editor and executed in the console of Spyder (or similar), we use gray and turquoise backgrounds, like this:

```
print("Welcome to DA2005")
print(2*2)
```

```
Welcome to DA2005
4
```

1.2.2 Assignments

For problems in this text, the symbol † mark exercises that we believe are harder.

1.2.3 Comments

It is important to document ones code with informative comments. In code, comments are started with the sign #. It is good practice to use comments to give context to what the code is supposed to do, and add information that is otherwise hard to deduce from the actual code. The main goal with comments is, of course, to make it easier for readers of the code to understand it. It is our own experience that it

can be very hard to come back to ones own code, so this potential future reader of the code might be yourself! Many, many, programmers have looked at their code some time later and asked "what was I thinking? What is happening here?!" So, write comments to help your future self!

An example:

```
print('This is printed') # This comment is not printed
```

```
This is printed
```

The following are examples of how to comment code:

```
print(17) # Prints the number 17
print(1.1) # Prints the number 1.1
print('DA2005') # Prints the course code for this course
```

One can note that these examples are not very good, because they are explaining what also a beginner might feel is the obvious outcome of the code. Good comments do not state the obvious, but explains *why* the code is written as it is.

1.3 Variables

Variables are used to store values of different sorts.

```
x = 17
y = 'DA2005'
z = 22/7
q = 22//7

print(x)
print(y)
print(z)
print(a)
print(x,y,z,a)
```

```
17
'DA2005'
3.14
3
17 DA2005 3.142857142857143 3
```

Note that we have to use `print` for the values to show up in the console.

The example above is using these types of values:

- integer, abbreviated to `int` in code
- floating point, abbreviated to `float` in code
- strings (for text)

These are called "primitive types" in Python, and we will come back to what that really means.

You can also assign several values at once:

```
x,y = 2,3
print(x)
print(y)
```

```
2
3
```

Note that if `x` and `y` has already been defined in the code (e.g., as `17` and `'DA2005'` earlier), then they will be overwritten with the new values `2` and `3`.

1.4 Working with strings and `print`

Strings can be written using single quotes `'` or double quotes `"`. This redundancy makes it easier to add single or double quotes in strings:

```
print("hej")
print('hej')
print("h'e'j")
print('h"e"j')
```

```
hej
hej
h'e'j
h"e"j
```

You can also use backslash, `\` , to "protect" the quotes:

```
print('h\'e\'j')
print("h\"e\"j")
```

```
h'e'j
h"e"j
```

Line breaks are written using `\n`:

```
print("h\ne\nj")
```

```
h
e
j
```

If you want to print several values with blank spaces in between, you simply call `print` with multiple parameters:

```
print('hello','how','are','you')
```

```
hello how are you
```

1.5 Arithmetic operations

Python supports the usual arithmetic operations:

- addition (+)

- subtraction (-)
- multiplication (*)
- division (/)
- integer division (//)
- exponentiation (**)
- modulo/remainder (%)

Try it!

```
2+3   # yields 5
2-3   # yields -1
2*3   # yields 6
2/3   # yields 0.6666666666666666
2//3  # yields 0
2**3  # yields 8
2%3   # yields 2
```

This also works for *floating point numbers*, the closes thing you get to the real numbers in a computer.

There are also operations on strings. For example, + concatenates strings:

```
'Mid' + 'summer'   # results in 'Midsummer'
```

If you want a blank space in between words, you have to add it yourself:

```
'hi' + ' ' + 'there'   # results in 'hi there'
```

The property that + can be used in different ways with different types of values makes it an *overloaded operator*.

The arithmetic operations work mostly as expected (e.g., associative rules are in effect, and parenthesis can be used), so `2 * 3 + 4` is the same as `(2 * 3) + 4`:

```
2 * 3 + 4    # yields 10
(2 * 3) + 4  # yields 10
2 * (3 + 4)  # yields 14
```

For numerical values, mixing types have (mostly) the expected effect:

```
2.0 ** 3 - 2.1312   # yields 5.8688
```

But working with strings gives operators novel meanings:

```
2 * 'two'           # yields 'twotwo'
```

And you cannot add a number to a string, nor multiply strings:

```
'hi' * 'there'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

```
2 + 'hej'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Note the informative error messages! A productive programmer studies and ponders error messages. They are sometimes pinpointing the problem for you, other times at least giving hints as to what the real problem is.

1.6 Input

We have seen that `print` can be used to output values. For reading from the keyboard, you use `input`. Using the statement `input()` alone simply waits for the user to write something and hit return, but if you give a string as parameter, then that is printed first:

```
name = input("What is your name?")
print("Hi " + name + "!")
```

Running this code in the console looks like:

```
What is your name?
Xena # typed by you!
Hi Xena!
```

1.7 Special words and names in Python

1.7.1 Reserved words

There are a number of reserved words, or *identifiers*, in Python:

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

These identifiers are part of the language. Using a reserved word as a variable identifier results in an error:

```
and = 12
```

```
File "/Users/kxs624/.spyder-py3/DA2005/vt21/f1.py", line 1
  and = 12
      ^
SyntaxError: invalid syntax
```

There are quite few restrictions on how to choose identifiers.

1.7.2 Identifiers

Identifiers are names for resources, such as variables. We will also use them for functions, classes, objects, etc.

Python has these rules for identifiers:

- May contain letters, digits and underscore (_).
- Cannot begin with a digit.

- Python is case sensitive; `Year` and `year` are not the same identifier.

Make sure to choose good identifiers to make the code easier to read! Naming a variable for someones age as `age` is much more informative than `x`.

1.7.3 Literals

A literal is a constant value in code. For example, `17`, `2.5`, and `'SU'` are literals.

1.8 Exercises

1. What happens if we run the code below? Why?

```
x,y = 2,3
x,y = y,x
print(x)
print(y)
```

2. Below is code to define the variables `x` and `y` with assigned values. Create and print a variable `z`with the value 100 by *combining* `x` and `y` arithmetically.

```
x,y = 4, -3
```

3. Let

```
s = 'hej'
```

   Write code to print `'hejhejhejhejhejhejhejhejhejhejhejhejhejhejhejhej'` (i.e., sixteen copies of `hej`).

4. Let

```
s = 'hej'
```

   Write code that results in `'hejhejhejhejhejhejhejhejhejhejhejhejhejhejhejhej'` by using the `+` operator as few times as possible.

5. Write code that asks the user for first and last name, stores them in variables, and then prints a greeting. Look at the section section 1.6 above for inspiration.

   Example session: `python  What is your first name?  What is your last name?  Hi Xena Warriorprincess!`

6. Write code that asks the user for his/her age and favorite number and store them in two variables. What happens if you add those values and prints the result? What did you expect and what can you conclude about `age` and `num`?

Chapter 2

Python fundamentals

2.1 Syntax and semantics

In programming, we often talk about *syntax* and *semantics*:

- *Syntax* deals with rules for what one *writes*
- *Semantics* deals with the *meaning* of what one writes

As an example, consider the following sentence:

> *Colorless green ideas sleep furiously.*

The sentence is syntactically correct: there is no *grammatical* error. But the sentence is not *meaningful*: the semantics is nonsense.

Syntax is especially important, and can be difficult to grasp, when one begins to learn to program. Just like with an ordinary language, one has to learn the rules for what is correct. In the case of Python, you need to learn for example what is a valid name for an identifier, where and how you should indent a block of code, how to write `if`-statements and `for`-loops (we will see these in later chapters), and so on. Every programming language has its own syntax, though there are often similarities. (Python, for example, has a lot in common with the language C.) You can get used to a language's syntax relatively quickly with practice, though, and then the focus shifts to semantics: what does a program do, and what do you want it to do?

If you see the label `SyntaxError` in an error message when you execute Python code, it means you have written something incorrectly and so Python cannot understand your code. In this case, you often get a fairly informative error message, as Python can point out the place where it got confused. For example, you might see the following message:

```
    3a = 2
     ^
SyntaxError: invalid syntax
```

The error message shows where Python found invalid syntax, here an invalid name (`3a`) for an identifier.

Semantic errors are harder to find, because Python itself does not know if something is wrong. The code is all valid: the result of a semantic error is that the program doesn't do what you expected. You can reduce the chance of a semantic error by writing well-organized, well-documented, and well-structured

code, that is to say "good" code. If it is clear what you *want* the code to do when someone (or you yourself) reads the code, it is much easier to find semantic errors.

2.2 Datatypes

Every value in Python has a type. We have already used different types when we for example assigned variable names to integers, floats, and strings. The most common types in Python we will use are:

- `str` – strings, that is letters (or symbols more generally)
  - Written in between `'` or `"`
- `int` – integers, of unlimited size
  - Written using digits `0`-`9`
  - In many other languages, `int` is the type of 32-bit integers, specifically integers in the range [-2^32, 2^31-1]
  - (before Python v3.0, `int` was the type of integers up to 32-bits, while unlimited-size integers were called `long`)
- `float` – floats, an approximation of a real number
  - Written using digits `0`-`9` and a decimal point `.`
  - Typically numbers in the range [-10^308, 10^308], with 15-digit precision
  - 64 bits, 11 för for the exponent e and 52 for the "mantissa" m
  - Details can be found by running `import sys` and looking at the object `sys.float_info`
  - Why isn't the result always exactly what you expect? Answer: Python represents floats using base 2 fractions. For details, see: https://docs.python.org/3/tutorial/floatingpoint.html
    In general, it is not possible to represent *all* real numbers *exactly* on a computer, no matter what representation one uses. *Some* real numbers can be represented exactly while others must be approximated: more bits results in a better approximation.
- `complex` – written `1+2j`
  - Represented by two floats
- `bool` – Boolean values (that is, 'true' or 'false'), represented by the reserved keywords `True` and `False`.
- `NoneType` – for the value `None`

In Python, you can see which type a variable or literal has by using `type`:

```
my_int = 1
my_float = 1.0
my_string = "1"
type(my_int)
type(my_float)
type(my_string)
type('hello')
```

```
int
float
str
str
```

2.2.1 Type coercion

You can convert a value of one type to another, but for each pair of types, there are different rules for whether one can convert between them and how it is done if so. Often, you don't have to think about it: it just works how you would intuitively expect. Such a type coercion is called *implicit*. For example, it is rarely a problem to mix different numerical types in an expression:

```
my_int = 1
my_float = 1.0
c = my_int + my_float
type(c)
```

```
float
```

Here the value in `my_int` is implicitly converted to a float which is then added to the float in `my_float`, and the result stored in `c` is another float.

As mentioned earlier, there are situations where such a coercion will not work. If, for example, you evaluated the expression `my_int + my_string`, you would get a `TypeError`. In such cases, you can make an *explicit coercion* by giving the value you want to convert as an argument to the type you want to convert it to. If we wanted to coerce a value stored in the variable `x`, for example, we could write:

```
str(x)
int(x)
float(x)
complex(x)
bool(x)
```

As a concrete example, we could write the following:

```
my_int = 1
my_string = "1"
c = my_int + int(my_string)
type(c)
```

```
int
```

Here, Python was able to convert the value in `my_string` to an integer to then add to `my_int`.

Warning: explicit coercion is not "magic" and will not always work:

```
int("hello!")
```

```
ValueError: invalid literal for int() with base 10: 'hello!'
```

Here, Python has no idea how to convert `'hello!'` to an integer.

Explicit conversion is often important when one uses `input()`, which reads input and returns a value of type `str`. In some cases one does not want the result as a string, as in the following:

```
age = int(input("Your age: "))
added_age = age + 10 # would not work if age was of type str
print(added_age)
```

Warning: Before v3, Python's `input()` tried to automatically interpret the type of its answer for you. This led to many bugs and security problems!

2.3 Boolean expressions in Python

Earlier, we saw examples of typical arithmetic operations (`+`, `-`,`*`, `/`, ...) that result in a value whose type depends on the types of the input arguments. Boolean expressions are expressions that result in a `bool`, that is `True` or `False`. Such expressions are very useful when one wants to check whether some condition is true or false.

It is worth pointing out that *most* values in Python are regarded as *true*. We can see this by using type coercion to `bool`:

```
bool(3)
bool(-427.2)
bool('hello')
```

```
True
True
True
```

Values that are regarded as *false* are typical "empty" values:

```
bool(0)
bool(0.0)
bool('')
bool(None)
```

```
False
False
False
False
```

2.3.1 Comparison operators

These operators compare their operands to check if various relations hold between them:

- Equality: `x == y`
- Inequality: `x != y`
- Strictly greater than: `x > y`
- Strictly less than: `x < y`
- Greater than or equal: `x >= y`
- Less than or equal: `x <= y`

These work as one would expect for numbers (experiment!), but like many operators in Python, you can also compare values other than `int`s and `float`s. For example:

```
print('hello shark' < 'hello shark')
print('hello shark' < 'hello shork')
print('hello shark' < 'hello')
print('hello shark' < 'Hello shork')
```

```
False
True
False
False
```

Google and see how Python compares strings!

2.3.2 Logical operators

Python includes the logical operations `and`, `or` och `not`.

- Are `x` *and* `y` true?: `x and y`
  `x and y` is `True` only if both `x` and `y` are `True`.
- Is `x` *or* `y` true?: `x or y`
  `x or y` is `True` only if at least one of `x` and `y` is `True`.
- Negate a value: `not x`
  `not x` is `True` if `x` is `False` and `False` otherwise.

These rules can be written out with truth tables. We can, for example, use logical operators to define an expression which is an exclusive or, `xor`, that is evaluates to true if exactly *one* of `x` or `y` is true:

```
x = True
y = False
xor_result = (not ((x and y) or (not x and not y)))
print(xor_result)
```

```
True
```

Python handles `and` and `or` specially if (both of) the operands are not of type `bool`: an `and`/`or` expression is not evaluated in this case to a `bool`. An `and`-expression returns the *second* operand if both operands are true, but returns the value of the first *false* operand if there is one. An `or`-expression returns the *first* value if the first operand is true, otherwise the value of the second operand. Some examples:

```
(42 and 'hello')
(42 and '')
(False and '')
(42.7 or 13)
('42' or None)
(False or 0)
```

```
'hello'
''
False
42.7
42
0
```

These evaluation rules can seem strange, but they have to do with how much of the expression needs to be evaluated to know the result of the whole expression. If the first operand to `and` is false, then it is not necessary to evaluate the second value; regardless of whether it evaluates to `True`, the result will be `False`. This time-saving technique is sometimes called “short circuiting”, or more generally “lazy evaluation” (because it can been seen as “laziness” to not evaluate everything). The purpose is of course to evaluate a program more quickly by not computing expressions that will not be needed.

2.4 Assignment operators

We have already seen that = is used for assignment. A convenient functionality in Python is the ability to combine these with arithmetic operators to get so-called "assignment operators":

```
x += 3    # x = x + 3
x -= 3    # x = x - 3
x *= 3    # x = x * 3
x /= 3    # x = x / 3
x %= 3    # x = x % 3
x //= 3   # x = x // 3
x **= 3   # x = x ** 3
```

2.5 Functions

As your program gets bigger and perhaps more complicated, you'll quickly realize the need to give some structure to your source code. You can start to do so with the help of *functions*. We have already used several of Python's built-in functions, for example `print`. Functions in programming have, as the name suggests, commonalities with mathematical functions: they take one or more arguments and give back a return value. In programing, however, a function does not *need* to take any arguments at all or to return a value, and it might do things *besides* return a value.

The point of functions is to:

- Organize code in easily handled and reusable pieces
- Avoid repetition of code
- Simplify advanced programs
- Increase *modularity*: a program divided up in small pieces is easier to understand and change than a big, monolithic program
- Make the program more readable:
  - Clarify dependencies in the code: "inputs" as arguments, the result after **`return`**. This is reminiscent of a function in mathematics.
  - Dividing code up makes it easier to document.
  - Dividing code up makes it easier to see that it does what it should (for example by testing, or even just by looking at it).

Functions are therefore a very important tool for writing good code.

Before you can use a function, you have to *define* it. This means using the reserved keyword **`def`** to give the function a name (identifier) and specify its arguments. Below is an example of a simple function `f` that takes a value `x` as an argument and adds the value to itself:

```
def f(x):
    return x + x
```

This function would be written in mathematics as $f(x) = x + x$. Once you have successfully defined a function, it can be called in the following way:

```
f(2.0)
```

```
4.0
```

Notice that Python is not concerned about what type the arguments have, as long as all the operators

and expressions in the function definition can be carried out. Above we applied `f` to a `float`, but we can try other types:

```
f(2)
f('hello')
```

```
4
'hellohello'
```

Just like mathematical functions, functions in Python can have multiple arguments:

```
def long_name(course_code, course_name):
    lname = course_code + course_name
    return lname
```

What is the result of `long_name('DA','2005')`?

Functions do not need to take any arguments at all:

```
def show_user_menu():
    print('Menu:')
    print('1. Add user')
    print('2. List users')
    print('3. Quit')
```

and can call other functions themselves:

```
def user_choice():
    show_user_menu()
    return input('Which assignment do you want to do? ')
```

2.5.1 About functions

The examples above introduced several new syntactic concepts in Python. A function has a definition (also called its "body") and a section (or "header").

- The header consists of the function name and a list of parameters (the function's arguments).
  - Syntax: `def functionname(argument1, argument2, ...):`
- The word parameter has two meanings.
  - Variables in the header are called *formal parameters*. Formal parameters are placeholders for the values that will be supplied when a function is called.
  - The values one gives when one calls a function are called *actual parameters*.
- The body has a documentation string (optional) followed by code, and is *indented*.

Indentation is important in Python! Python is an "indentation-sensitive" language, which is to say that incorrect indentation is a syntax error. Indention is what specifies *code blocks* that belong to a function body. You can use any size of indentation you want in Python (for example, two spaces or a tab), as long as you are consisent, but the *standard* indentation is four spaces.

That indentation should be four spaces is specified in Python's style guide, known as PEP-8, which also contains lots of other good information on how you can write good Python code.

Most modern program editors will insert four spaces when you press the Tab key. Be warned that older or more primitive editors might not handle the Tab key in a good way. The way the results look might

depend on how your editor is configured, so that your code looks different when opened in different editors or on different computers.

The function body often ends with the reserved keyword **`return`**. If **`return`** is missing, then the value `None` is returned. It is good to know that **`return`** can appear several times in one function, but if any **`return`** is executed then the function will always finish and return the value in the executed statement.

It is important to understand the difference between functions that return something and functions that only print something out. Consider the following code snippet:

```
def f1(x):
    return x + x

def f2(x):
    print(x + x)
```

The difference is that `f1` *returns* the value in `x + x` while `f2` only *prints* the value. If we now try to print the result of `f1(2)`, it works as expected:

```
print(f1(2))
```

```
4
```

but if we try to print the result of `f2(2)`, we get the following:

```
print(f2(2))
```

```
4
None
```

The reason is that `4` is first printed in the course of executing the function `f2`, but later when we print the result of `f2` we get `None`, which is in fact what `f2` returns! It is very important that if you want to write a function that *returns* something, you must use **`return`** and not `print`.

2.5.2 Multiple return values

What do you do if you want to return multiple values? There are several bad ways of doing it, but one good one: multiple return values.

```
def integer_div(nominator, denominator):
    q = nominator // denominator
    r = nominator % denominator
    return q, r
```

You can then call the function like so:

```
quotient, remainder = integer_div(17, 10)
```

2.5.3 Default values

It is common to write functions with parameters that are almost always set to the same value. Python lets you give such common values explicitly; these are called *default values*.

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

```
print(f())

print(f(3))
```

```
2
6
```

In the first printed line, we can see that `f` was called with `x=1`, even though we did not give it as an argument. When we call `f(3)` on the next line, `x` is set to `3` as normal.

2.5.4 Keyword parameters

Python also lets you specify parameters by names when you call a function. For example, `integer_div(nominator=17, denominator=10)` has the same effect as `integer_div(17, 10)`.

What happens when we try:

```
integer_div(denominator=10, nominator=17)
```

Python connects the identifiers among the formal parameters to the identifiers in the arguments, so the above results in precisely the same output.

An example of a function we have seen that has several keyword parameters is `print`:

- `sep=' '` — use a space as separator
- `end='\n'` — "backslash-n" means newline
- `file=sys.stdout` — send output to the terminal

Exempel:

```
print('hello','you',sep='\n')
```

```
hello
you
```

To see more information, run `help(print)` in the Spyder console (iPython).

2.5.5 Documentation of functions

There are two ways to document in Python:

- Documentation strings
  - Put at the top of a function body in between either `"""  """` or `'''  '''`
  - Primary documentation method.
  - Tied to the function. Try `help(print)` in the Python interpreter, or Ctrl-i in Spyder when the cursor is on a function call or Python instruction.
- Comments
  - Used to explain calculation steps, typically single lines or small blocks of code.
  - Comments are for understanding *how* a function works. You should be able to *use* a function without reading the comments.

Documentation strings are described in PEP-257:

> The docstring [...] should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called (all if

applicable). Optional arguments should be indicated. It should be documented whether keyword arguments are part of the interface.

An example of a single-line documentation string:

```
def square(a):
    '''Returns argument a squared.'''
    return a*a

print (square.__doc__)

help(square)
```

An example of a multi-line documentation string:

```
def square(x):
    """Description of square function

    Parameters:
    x (int): input number

    Returns:
    int:Square of x
    """

    return x*x

print(square.__doc__)

help(square)
```

A general principle for documentation is that you should explain what is not obvious from the identifier. If a function is called `compute_integral`, it is not necessary for the documentation string to say the same thing. Instead, it might be good to write which algorithm is used, what assumptions are made about the parameters, and what is guaranteed about the result.

2.6 Global and local variables

Variables in Python can be either *global* or *local*. These properties indicate where a variable is "visible" to code and where the variable can be modified. In the following example, `x` is a global variable, as it was defined on the top level (not within a function body).

```
x = "global"

def foo():
    print("x printed from inside foo:", x)

foo()
print("x printed from outside foo:", x)
```

The function cannot modify `x`: the value that `x` holds can only be read.

```
x = 42

def foo():
    x = x * 2
    print("x printed from inside foo:", x)
foo()
```

```
UnboundLocalError: local variable 'x' referenced before assignment
```

To be able to change the value in x, we need to first tell the function that it can change this global variable. We do this with the reserved keyword **global**.

```
x = 42

def foo():
    global x
    x = x * 2
    print(x)
foo()
```

Warning: global variables that are changed by functions in this way can easily lead to "spaghetti code". It can become extremely hard to keep track of where the value of x was changed! So be very careful if you use global variables in this way. There are, however, scenarios where they are very useful, for example as global counters. It is therefore very good to give global variables informative names, so that you do not change one by mistake!

Local variables are variables that are defined inside of a function:

```
def foo():
    y = "local"
    print(y)

foo()
```

The variable y is not visible outside the function, which is why it is called a *local* variable:

```
def foo():
    y = "local"

foo()
print(y)
```

```
NameError: name 'y' is not defined
```

Where x is visible is called its *scope*. If x is defined outside of the function and we later write over it within the function, its value does not change outside the function:

```
x = 5

def foo():
    x = 10
    print("local x:", x)
```

```
foo()
print("global x:", x)
```

The same thing happens with function arguments:

```
x = 5

def foo(x):
    print("local x:", x)

foo(12)
print("global x:", x)
```

2.7 Conditional statements

One often wants to execute a certain code block only if some expression is true or condition is met. This is done with so-called *conditional statements*, or "`if`"-statements. In python, we write these with the keywords **`if`**, **`elif`** (which is short for *else if*), and **`else`**.

```
answer = input('Write "done" if you are done: ')
if answer == 'done':
   quit()
print('Okay, I guess you were not finished yet.')
```

Obs: note the double equals sign! There is a difference between *comparison* (==) and *assignment* (=).

The above example could also be written:

```
answer = input('Write "done" if you are done: ')
if answer == 'done':
    quit()
else:
    print('Okay, I guess you were not finished yet.')
```

Note that the `print`-statement is indented by the same amount as `quit`. Just like for functions, it is the indented code block that is executed.

Indentation errors can be hard to see sometimes:

```
answer = input('Write "done" if you are done: ')
if answer == 'done':
   quit()
else:
    print('Okay, I guess you were not...')
   print('.. quite finished yet.') # indentation error
```

One more example with **`else`**:

```
namn = input('What is your name? ')
if namn == 'Anders':
   print('Hello Anders!')
else:
```

```
   print('Hello, whoever you are!')
```

You can use *nested* conditional statements, which is to say `if`-statements inside `if`-statements:

```
temp = int(input('What is the temperature? '))
if temp < -30:
   print('That is very cold!')
else:
   if temp < 0:
      print('That is pretty cold...')
   else:
      if temp == 0:
         print('Zero')
      else:
         if temp < 30:
            print('That is a comfortable temperature')
         else:
            print('Wow! Super warm')
```

In the above case, the code is not particularly readable. A better alternative would be to use **`elif`**:

```
temp = int(input('What is the temperature? '))
if temp < -30:
   print('That is very cold!')
elif temp < 0:
    print('That is pretty cold...')
elif temp == 0:
    print('Zero')
elif temp < 30:
    print('That is a comfortable temperature')
else:
    print('Wow! Super warm')
```

Readability is important! **`elif`** is strictly speaking unnecessary, as it can always be replaced with nested **`if-else`** statements, but it is very useful when it makes the code easier to read. This type of functionality is sometimes called *syntactic sugar*.

What is the difference in meaning between

```
x = int(input('Guess a number! '))
if x == 42:
    print('Correct!')
if x > 42:
    print('Too high...')
else:
    print('Too low...')
```

and

```
x = int(input('Guess a number! '))
if x == 42:
    print('Correct!')
```

```
elif x > 42:
    print('Too high...')
else:
    print('Too low...')
```

What happens if one enters 42?

Finally, we can make use of conditional statements to rewrite the `xor`-expression from Logical operators and define a function:

```
def xor(x, y):
    if x and y:
        return False
    elif not x and not y:
        return False
    else:
        return True

print(xor(True,True))
print(xor(True,False))
print(xor(False,True))
print(xor(False,False))
```

```
False
True
True
False
```

Which version is easier to read?

2.8 Exercises

1. Which alternative/alternatives should you keep in mind when choosing identifiers for variables and functions?

    a) Identifiers should be short
    b) Identifiers should be descriptive
    c) Identifiers should be unique within the function/module/program/script
    d) An identifier should be written with capitals
    e) None of the above

2. Write code that read an integer from the user and prints "Odd" if it is an odd number and "Even" if the number is even.

3. Make *truth tables* for the boolean operators `and` and `or` (i.e., replace the question marks with `True` or `False` in the table below).

x	y	x and y	x or y
True	True	?	?
True	False	?	?
False	True	?	?

x	y	x and y	x or y
False	False	?	?

4. Try, without running the code, to evaluate these four expressions for all possible values of `x`, `y` and `z`:

```
(not x) or (not y)
x and (y or z)
(x != z) and not y
x and z
```

   Then run the code to verify your answers.

5. Write a Python function that corresponds to the mathematical function $f(x, y) = x/y$. We want the function to notify the user if a number other than 0 is divided by 0. In this case, the function should print out `The result is infinite...` but *not* return any explicit value. If $0/0$ is called, the function should print `Indeterminate form...` and not return anything either.

6. Let:

```
x = True
y = False
```

   Which expressions below evaluate to `True`? Why?

```
(x and y) or (not x and not y)
(x or y) and (not x or not y)
x or y or not x or not y
not(not x and not (x and y))
```

7. Define `nand`, `nor`, and `xnor` as Python functions.

   Tip: https://en.wikipedia.org/wiki/Logic_gate#Symbols.

8. Test the functions `nand`, `nor`, and `xnor` with arguments that are *not* of the type `bool`. Are the results what you expected? If not, why?

Chapter 3

Lists and iteration

This chapter deals with:

- Lists
- Iteration with **`for`** and **`while`** loops

Lists are important in programming because they let us collect several elements together. Iteration deals with automatic repetition and is a fundamental concept in so-called *imperative* programming. Two important constructs we will discuss in this chapter are **`for`** and **`while`** loops. Lists and loops are especially useful together. For example, you can write a program that reads in numbers from the user, saves them in a list, and calculates the average of the inputs.

3.1 Lists

Python has support for lists of elements with different types. These are written in square brackets ("`[`" och "`]`"), and the elements of the list are separated with commas:

```
mylist = ["hej",2,"du"]
```

Python's `list` can be compared with *array* types in other languages, but it is a bit less efficient. However, Python's lists are easier to use. This is typical for scripting languages: data structures are simpler to use at the price of efficiency.

3.1.1 List indexing

You can access elements from a list by writing `mylist[i]` where `i` is an integer:

```
print(mylist[0])
print(mylist[1])
print(mylist[2])
```

```
hej
2
du
```

Important: lists are indexed beginning with `0`!

You can also use negative values to access an element from the end of the list.

```
print(mylist[-1])
print(mylist[-2])
print(mylist[-3])
```

```
du
2
hej
```

If you use an index that is too large, you'll get an error message:

```
print(mylist[3])
```

```
IndexError: list index out of range
```

Question: what happens if you index with a negative value that is too large (for example `mylist[-4]`)?

You can add elements to the end of a list using `append`.

```
mylist.append(True)
print(mylist)
```

```
['hej', 2, 'du', True]
```

You can easily assign a new value to a particular index in a list.

```
mylist[1] = 5
print(mylist)
```

```
['hej', 5, 'du', True]
```

We can even combine this with indexing from the end of the list:

```
mylist[-2] *= 3
print(mylist)
```

```
['hej', 5, 'dududu', True]
```

You can access all elements at indices in the range `i < j` using `mylist[i:j]`. This is called *slicing*.

```
print(mylist[0:1])
print(mylist[0:2])
print(mylist[1:3])
print(mylist[1:4])
```

```
['hej']
['hej', 5]
[5, 'dududu']
[5, 'dududu', True]
```

You can change the step size in the slicing to `k` by using `mylist[i:j:k]`:

```
print(mylist[0:4:2])
print(mylist[1:4:2])
```

```
['hej', 'dududu']
[5, True]
```

3.1.2 List functions

You can add lists and multiply them with a number to repeat them:

```
print(mylist + mylist)
print(2 * mylist)
```

```
['hej', 5, 'dududu', True, 'hej', 5, 'dududu', True]
['hej', 5, 'dududu', True, 'hej', 5, 'dududu', True]
```

The length of a list can be calculated with `len`:

```
print(len(mylist))
print(len(5 * mylist + mylist))
```

```
4
24
```

A very useful function is **`in`**, which tests if an element occurs in a list:

```
print(5 in mylist)
print(False in mylist)
```

```
True
False
```

Compare with $x \in S$ from mathematics, which says that x is in the set S. The function **`in`** also works on strings:

```
print("gg" in "eggs")
```

```
True
```

There is also **`not in`**, which corresponds to $x \notin S$ from mathematics:

```
print(5 not in mylist)
print(False not in mylist)
```

```
False
True
```

You can find the smallest and largest element of a list using `min` and `max`:

```
print(min([2,5,1,-2,100]))
print(max([2,5,1,-2,100]))
```

```
-2
100
```

You can find the first index at which an element occurs with the help of `index`:

```
print(mylist.index(True))
```

```
3
```

If you try to find the index of something that doesn't occur in the list, you'll get an error message:

```
print(mylist.index(42))
```

```
ValueError: 42 is not in list
```

You can count the number of times an elements occurs in a list with `count`:

```
mylist.append(5)
print(mylist.count(5))
print(mylist.count(42))
```

```
2
0
```

We can remove the element at a specific index or a whole slice with **`del`**:

```
print(mylist)
del mylist[2]
print(mylist)
del mylist[0:2]
print(mylist)
```

```
['hej', 5, 'dududu', True, 5]
['hej', 5, True, 5]
[True, 5]
```

Writing `del s[i:j]` has the same effect as writing `s[i:j] = []`. You can even write **`del`** `s[i:j:k]` to use a different step size.

We can also remove the element at index `i` and then return it using `pop`:

```
mylist = ['hej', 5, 'dududu', True, 5]
print(mylist)
x = mylist.pop(2)
print(mylist)
print(x)
```

```
['hej', 5, 'dududu', True, 5]
['hej', 5, True, 5]
dududu
```

This also works with negative values (that is, with indexing from the end). If no argument is given to `pop`, it behave likes `pop(-1)`: the last element is removed.

```
x = mylist.pop()
print(mylist)
print(x)
```

```
['hej', 5, True]
5
```

You can insert an element `x` at a given index `i` using `insert(i,x)`:

```
mylist.insert(1,"du")
print(mylist)
```

```
['hej', 'du', 5, True]
```

This has the same effect as writing `s[i:i] = [x]`.

To remove the first element that is equal to `x`, you can use `remove(x)`:

```
mylist.append(5)
print(mylist)
mylist.remove(5)
print(mylist)
```

```
['hej', 'du', 5, True, 5]
['hej', 'du', True, 5]
```

Finally, you can use `reverse` to reverse a list:

```
mylist.reverse()
print(mylist)
```

```
[5, True, 'du', 'hej']
```

3.1.2.1 Summary of useful list functions

- `mylist.append(x)`: add `x` to the end of `mylist`.
- `len(mylist)`: calculate the length of `mylist`.
- `x in mylist`: test if `x` occurs in `mylist` (returns a `bool`).
- `x not in mylist`: test if `x` doesn't occur in `mylist` (returns a `bool`).
- `min(mylist)` and `max(mylist)`: find the smallest, respectively largest element of `mylist`.
- `mylist.index(x)`: calculate the first index where `x` occurs `mylist`.
- `mylist.count(x)`: count how many times `x` occurs in `mylist`.
- `del mylist[i]`: remove the element at index `i` from `mylist` (without returning it).
- `mylist.pop(i)`: remove the element at index `i` from `mylist` (and return it).
- `mylist.insert(i,x)`: insert `x` at index `i` in `mylist`.
- `mylist.remove(x)`: remove the first occurrence of `x` from `mylist`.
- `mylist.reverse()`: reverse the order of elements in `mylist`.

For more list functions and details on lists, see: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

https://docs.python.org/3/library/stdtypes.html#lists

3.1.3 A practical function for building lists: `range`

One way to create lists of integers is to use `range`:

```
print(list(range(1,4)))
print(list(range(3)))      # same as range(0,3)
print(list(range(4,2)))
```

```
print(list(range(0,10,2)))
print(list(range(5,0,-1))) # iterate backwards
```

```
[1, 2, 3]
[0, 1, 2]
[]
[0, 2, 4, 6, 8]
[5, 4, 3, 2, 1]
```

Obs: we have to convert the result to a list with the help of `list(...)` because `range` does not directly return a list, but a "promise" of a list. That is, `range` returns an object that creates a list when you need it. This behavior is called *lazy evaluation* and is convenient when you do not need to create the whole list at once. This has the advantage that the whole list need not be saved in the computer's memory all at once, which could be slow for large lists. More on this later.

If you try to print `range(0,10)` without converting to a list, you'll get the following result:

```
print(range(0,10))
```

```
range(0, 10)
```

3.1.4 Characteristics of lists

Lists have many strengths, such as:

1. Simple and efficient to iterate over (more on this in the next section).
2. Simple to extend.
3. Immediate access to any particular index.
4. List elements can have different types.

Characteristic 3 is specific to Python. Other languages, for example C and Java, differentiate between *arrays* and *lists*. An array is like a vector and has a fixed length. In these languages, lists are instead a special data structure in which the elements are "linked" so that one can only immediately access the first (and possibly the last) element. Thus, one cannot automatically get immediate access to any element. In Python, though, one has direct access to all elements by list indexing, making these lists somewhat reminiscent of arrays.

A weakness of lists is that it is inefficient to *search* in them: the operator **`in`** becomes slow with large lists. Another weakness is that unexpected things can happen if one has a list containing elements of different types and tries to find the largest or smallest element (see exercise 2).

3.2 `for` loops

`for` is used to iterate over enumerations of different kinds, in particular lists. These enumerations are called *sequences* in Python. Even strings are sequences (of characters).

```
for x in [1, 2, 3]:
    print(x)
```

```
1
2
3
```

Obs: note that the `print` statement is indented! Everything that comes after **`for`** and is indented (by the same amount of spaces) will be run in the loop. As we saw in the previous chapter, Python is indentation-sensitive: it is important that you give all lines in a code block the same indentation, and this applies to **`for`** loops as well. This differentiates Python from languages such as C or Java, where one instead groups code blocks within curly braces ("{" and "}") and lines end with a semicolon ";". If something is indented incorrectly, Python will show the following error message:

```
IndentationError: expected an indented block
```

A code block can contain several instructions:

```
s = "A number: "

for x in [0,1,2]:
    print(x)
    s += str(x)

print(s)
```

```
0
1
2
A number: 012
```

We can also iterate over a list stored in a variale:

```
mylist = [112, 857, "hej"]

for x in mylist:
    print(x)
```

```
112
857
hej
```

As strings are also sequences, one can easily iterate over the letters in a string:

```
for c in 'DA2004':
    print(2*c)
```

```
DD
AA
22
00
00
44
```

3.2.1 A function that extracts the consonants from a string

Let's now implement a slightly more complicated function that extracts the consonants from a string. Before we begin writing code, we should think about how the function should work. Given a string `s`, it should iterate over all the letters in `s` and test if they are consonants. For each letter, if it is a consonant, then it should be added to a string with all the consonants we have found in `s`; otherwise, nothing should

happen. This can be implemented as follows:

```
def consonants(s):
    cs = "BCDFGHJKLMNPQRSTVWXZbcdfghjklmnpqrstvwxz"

    out = ""

    for c in s:
        if c in cs:
            out += c

    return out

print(consonants("Hey Anders!"))
```

```
Hndrs
```

Note how we defined a variable `out` which we used to keep track of the consonants we found in `s`. This is a very common way of programming and one we will see more of in the course.

3.2.2 Dividing up a string into words: `split`

We'll now try writing a function that takes a string and divides it up into pieces wherever it encounters a blank space. That is, we'll write a function `split(s)` that takes in a string `s` and returns all the words in the string. We can do this using two strings, one with the words we have found so far and one with all the characters in the word we are building up. For every character in `s`, we test if it is a space or not. If it is, we add our current word into the list; otherwise, we add the new character to our current word. Then we move on to the next character.

```
def split(s):
    out = []
    term = ""

    for c in s:
        if c == " ":
            out.append(term)
            term = ""
        else:
            term += c

    # add the final word unless it is empty
    if term != "":
        out.append(term)

    return out

print(split("Hey Anders! How are you?"))
```

```
['Hey', 'Anders!', 'How', 'are', 'you?']
```

Suppose now that we want to generalize this function to handle many different separators, with the

blank space as the default value. Then we can write the following instead:

```
def split(s,sep = " "):
    res = []
    term = ""
    for c in s:
        if c in sep:
            res.append(term)
            term = ""
        else:
            term += c

    if term != "":
        res.append(term)

    return res

print(split("Hey Anders! How are you?"))
print(split("Hey Anders! How are you?","!"))
```

```
['Hey', 'Anders!', 'How', 'are', 'you?']
['Hey Anders', ' How are you?']
```

Now we can call the function with one *or* two arguments.

Obs: There is already a version of `split` built into Python!

3.2.3 Control flow: `pass`, `continue`, `break` and `return`

We can control and stop iteration with **`pass`**, **`continue`**, **`break`** and **`return`**.

The simplest of these is **`pass`**, which does nothing.

```
for i in range(4):
    if i == 1:
        pass
        print("passed 1")
    else:
        print(i)
```

```
0
passed 1
2
3
```

Obs: **`pass`** can seem useless at first, as it doesn't do anything, but a common use is as a placeholder when you've started outlining the structure of your code but haven't filled in the details yet. For example, you can define a function whose body is just **`pass`** or have an **`if`** statement where you haven't filled in either case yet. In this way, you can test all the other code in the program now, then replace the **`pass`**es when you are sure that everything else works.

A **`continue`** instruction ends the current iteration of the loop and moves on to the next:

```
for i in range(4):
    if i == 1:
        continue
        print("passed 1")
    else:
        print(i)
```

```
0
2
3
```

A **break** instruction ends the loop entirely:

```
for i in range(4):
    if i == 2:
        break
    else:
        print(i)
```

```
0
1
```

Finally, you can use **return** in a loop to return from a function, as always. For example:

```
def first_even_number(l):
    for number in l:
        if number % 2 == 0:
            return number

print(first_even_number([1,45,24,9]))
```

```
24
```

3.2.4 Python's for is different

It is good to be aware that that Python's **for** loop is more general than in most other languages. For example, **for** in other languages can often ony iterate over integer values. Python's version is more "modern": it works for any data structure that supports iteration. It is possible, and not so hard, to write your own iterable datatypes for special purposes. So far we have seen that you can iterate over lists and strings; we will see more examples in the future.

3.2.5 A common problem with lists in Python: mutability

A common mistake made with lists in Python, one we will see several times in the course, is accidentally changing a list while iterating over it. This is a classic source of bugs, both for beginners and more experienced programmers.

Suppose we have the following program, which we expect to take in a list and copy each word in it with more than 6 letters to the front of the list.

```
animals = ["cat", "dog", "tortoise", "rabbit"]

for x in animals:
```

```
    if len(x) > 6:
        animals.insert(0, x)

print(animals)
```

But if we run this code, Python just loops forever! To stop the loop, you can press `CTRL-c`. The problem is that we changed the list `animals` at the same time that we looped over it. There is a logic to Python's behavior in such a case, but it is difficult to follow and rarely matches your intentions.

To avoid such problems, one should instead iterate over a *copy* of the list. We can do this by using `.copy()`:

```
animals = ["cat", "dog", "tortoise", "rabbit"]

for x in animals.copy():
    if len(x) > 6:
        animals.insert(0, x)

print(animals)
```

```
['tortoise', 'cat', 'dog', 'tortoise', 'rabbit']
```

Here we use `animals.copy()` instead of `animals` so that we get a copy of the list and avoid the endless loop.

This kind of problem often comes up with one programs with lists without being careful to make copies in the right places. It can lead to very mysterious bugs like the infinite loop above. What makes these issues come up is that Python's lists are *mutable* and many list operations work by *mutating* lists. We will return to mutability later in the course, but for the moment we just need to remember that we may need to use `.copy()` on a list variable to avoid accidentally changing it when we don't want to.

3.3 `while` loops

Sometimes there is no obvious structure to iterate over, but we still want to run a code block several times. For this, we have **`while`** loops. A **`while`** loop lets us run a block of code until some boolean condition is satisfied. Consider the following example:

```
i = 10

while i > 0:
   print(i)
   i -= 1

print("Done!")
```

```
10
9
8
7
6
5
```

```
4
3
2
1
Done!
```

A common use for **while** loops is to loop until the code gets proper input from a user. The program below will read in input from the user until they enter `exit`:

```
loop = True

while loop:
    print("Write 'exit' to quit")
    answer = input("Write something: ")
    if answer == "exit":
        print("Goodbye!")
        loop = False
    else:
        print("Let us keep going!")
```

```
Write 'exit' to quit
Write something: hey
Let us keep going!
Write 'exit' to quit
Write something: ho
Let us keep going!
Write 'exit' to quit
Write something: exit
Goodbye!
```

3.3.1 `while` loops over lists

Like many other datatypes, lists can be implicitly coerced to booleans. The empty list corresponds to `False`, while every other list corresponds to `True`.

```
list = []

if list:
   print("Not empty!")
else:
   print("Empty!")

list2 = ["Hey"]

if list2:
   print("Not empty!")
else:
   print("Empty!")
```

```
Empty!
Not empty!
```

This capacity enables the following *idiom*, where one loops over a list and removes its elements:

```
mylist = ["Ho","ho","ho!"]

while mylist:
    x = mylist.pop()
    print(x)

print(mylist)
```

```
ho!
ho
Ho
[]
```

When the loop is over, the list is empty, so one need not compare the list with `[]` explicitly. This is also an example where we change a list while looping over it.

3.3.2 Remove spaces from the end of a string

With the help of a **while** loop and slicing, we can remove the blank spaces from the end of a string:

```
def remove_trailing_space(s):
    i = len(s)
    while i > 0 and s[i-1] == " ":
        i -= 1
    return s[0:i]

print(remove_trailing_space("SU  "))
```

```
SU
```

We begin with a counter `i` which is set to the length of the list. Then we count the number of characters that are blank spaces starting from the end of list, then return a slice of the original list that leaves out these blank spaces.

3.3.3 Greatest common divisor: `GCD`

The *greatest common divisor* (GCD) of two numbers is the largest integer that divides both of them. For example, 3 is the GCD of 15 and 12.a

Euclid's algorithm for calculating the GCD of two numbers `a` and `b` works as follows:

- Suppose `a >= b`,
- take `r` to be the remainder of the integer division `a // b`,
- if `r == 0`, then `GCD(a, b) = b`,
- otherwise, we can use the fact that `GCD(a,b) = GCD(b, r)`.

Note that this is a iteration with a conditional, but without any clear structure to iterate over. It is therefore a good case for a **while** loop:

```
def GCD(a,b):
    if b > a:
        print("a must be bigger than b in GCD(a,b)")
        return None

    r = a % b
    while r != 0:
        a = b
        b = r
        r = a % b
    return b

print("GCD: " + str(GCD(15,12)) + "\n")
print("GCD: " + str(GCD(12,15)))
```

```
GCD: 3

b must be bigger than a in GCD(a,b)
GCD: None
```

3.4 Exercises

1. What is the result of `[1,2] + ['a', 'b']`?
2. What happens if we use `min`/`max` on lists with elements of different types?
3. Complete the function

   ```
   def vowels(s):
       vs = "AEIOUYaeiouy"
       pass
   ```

   that takes in a string `s` and returns its vowels.
4. Modify the function in exercise 3 so that it instead returns the consonants in `s`, but do so wihout changing the string `vs`. Then write a function `vowels_or_consonants` taking an extra parameter that determines whether vowels or consonants should be returned. The parameter should have a default value so that vowels ar returned if the programmer calls the function with only one argument.
5. What happens if you leave out the code block beginning with `if term != ""` in the function `split(s)` above? Find an example of a string `s` such that the output is incorrect if it is missing. Furthermore, how does `split` behave on strings with several spaces in a row? Write a correct version of the function.
6. Test the following code snippet:

   ```
   list = [1,2,3]
   list2 = list
   list2.reverse()
   print(list)
   ```

Does it work as you expected? Why not? How can you fix it?

7. Rewrite the code

```
mylist = ["Ho","ho","ho!"]

while mylist:
    x = mylist.pop()
    print(x)

print(mylist)
```

so that the list `mylist` is not empty at the end.

Tip: use `.copy()`. How can you do it without using copy?

8. Try using **`continue`**, **`break`** and **`pass`** in the **`while`** loop below. What happens?

```
n = 10
while n > 0:
     n -= 1
     if n == 7:
         # test continue, break or pass here
     print(n)
```

9. Write a **`while`** loop that reads in numbers from the user and saves them in a list. It should continue to read in until the user enters `0`; then the program should end the loop and print out the list.

10. Write the function

```
def naiveGCD(a,b):
    pass
```

that calculates the GCD of `a` and `b` by testing all numbers between `1` and `min(a,b)` to see which is the biggest number that divides both.

11. Write the function `passcode()` that reads in one digit (`1`–`9`) at a time from the user. After each digit is read, the function should test if the last four digits entered matches the correct code. If the correct code is entered, the function should print `Door unlocked!` and return `True`. Choose the correct passcode yourself, for example `1337`.

Tip: What loop should we use?

Chapter 4

Dictionaries, file handling, and more on loops

A "data structure" is a type for storing data. There are many kinds of data structures; we have seen a couple in the course already, among them lists. In this chapter, we will introduce dictionaries, which are like lists that can be indexed by types other than integers. Another way to save data is to write it to a file on the computer. We will also see how to do that in this chapter. Finally, we will look at more we can do with loops.

4.1 Dictionaries

The dictionary is a data structure that has become a "workhorse" for scripting languages. They are easy to use and useful in many situations. Dictionaries are much like lists, but instead of being indexed by integers, they are indexed by *keys*. You can think of a dictionary as a collection of *key-value pairs*. In Python, the type of dictionaries is called `dict`.

There are two typical ways to create a dictionary in Python. For example, you can construct a dictionary that has keys `'DA2004'` and `'MM2001'` with respective values `7.5` and `30` like so:

```
hp1 = dict(DA2004=7.5, MM2001=30)
hp2 = {'DA2004': 7.5, 'MM2001': 30}
```

Both methods give the same result, and `hp1 == hp2`. Note that the first uses the same syntax as calling a function with keyword parameters `DA2004` and `MM2001`, while the second uses a *string* for each key with its value separated by a colon.

To access a value, we use `[]`-syntax with a key:

```
print('Number of ECTS points for DA2004: ' + str(hp1['DA2004']))
```

```
Number of ECTS points for DA2004: 7.5
```

Note that the key argument can be an expression. That expression is first evaluated, and the result is the key then looked up in the dictionary. The same principle applies to the `{}` syntax for creating dictionaries. In both `hp1` and `hp2`, the keys have type `str`. An example that illustrates this point:

```
DA2004 = 33
MM2001 = 'hello'
hp3 = {DA2004: 7.5, MM2001: 30}
print('Number of ECTS points for DA2004: ' + str(hp3[DA2004]))
print('Number of ECTS points for DA2004: ' + str(hp3[33]))
print('Number of ECTS points for MM2001: ' + str(hp3[MM2001]))
print('Number of ECTS points for MM2001: ' + str(hp3['hello']))
```

```
Number of ECTS points for DA2004: 7.5
Number of ECTS points for DA2004: 7.5
Number of ECTS points for MM2001: 30
Number of ECTS points for MM2001: 30
```

Here a dictionary is created with keys `33` and `'hello'`, the results of evaluating the expressions `DA2004` and `'hello'` respectively. Likewise, using either `DA2004` or `33` as an argument to a lookup has the same result. If you tried to use a *string* of *value* `'MM2001'`, on the other hand, you would get an error, as that key is not in the dictionary:

```
print('Number of ECTS points for MM2001: ' + str(hp3['MM2001']))
```

```
KeyError: 'MM2001'
```

In Python, all objects that cannot be changed can be used as keys. Dictionaries are implemented with a technique called hash tables, which involves converting the keys to integers (by "hashing" them, which is to say chopping up and blending their bytes into integers) used for lookups. If the key can change, be "mutated", then there is a risk of losing access to entries you created. In general, the rule is that *immutable* values, for example strings, numbers, and tuples, can be used as keys. Mutable types like lists are excluded.

When you use `dict` to create a dictionary, the keyword parameters are always read as strings.

For more detailed documentation of dictionaries in Python, see:

https://docs.python.org/3/tutorial/datastructures.html#dictionaries

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

4.1.1 Important characteristics

- A lookup is cheap (that is, fast).
- It is easy to change the contents.
- Dictionaries are easy to iterate over (but you cannot add new entries to a dictionary during iteration).

4.1.2 Other names

- Hash table
- Hash map

In "traditional" languages (C, C++, Java, etc.), dictionaries do not play such a central role as in scripting languages, which make dictionaries especially easy to use.

4.1.3 More examples

An empty dictionary can be written as `{}`, which is syntactic sugar for `dict()`.

```
empty = {}
print(empty)
print(dict())
```

```
{}
{}
```

We can also create dictionaries in the following way:

```
test1 = {'hopp': 4127, 'foo': 4098, 'hej': 4139, 12: 'hej'}
test2 = dict([('hej', 4139), ('hopp', 4127), ('foo', 4098)])

print(test1)
print(test2)
```

```
{'hopp': 4127, 'foo': 4098, 'hej': 4139, 12: 'hej'}
{'hej': 4139, 'hopp': 4127, 'foo': 4098}
```

Note that the keys in a dictionary need not all have the same type!

We can update dictionaries in a way similar to lists, adding or updating key-value pairs with `[-]` and deleting them with **`del`**:

```
hp = {}
hp['DA2004'] = 7
print(hp['DA2004'])

hp['DA2004'] += 0.5
print(hp['DA2004'])

hp['MM2001'] = 30
hp['UG1001'] = 1
print(hp)

del hp['UG1001']
print(hp)
```

```
7
7.5
{'DA2004': 7.5, 'MM2001': 30, 'UG1001': 1}
{'DA2004': 7.5, 'MM2001': 30}
```

4.1.4 Iterating over dictionaries

We can use **`for`** to iterate over keys:

```
for key in hp:
    print(key)
```

```
DA2004
MM2001
```

We can also be more explicit by using `.keys()`:

```
for key in hp.keys():
    print(key)
```

```
DA2004
MM2001
```

or iterate over values instead using `.values()`:

```
for val in hp.values():
    print(val)
```

```
7.5
30
```

We can also extract all key-value pairs with `.items()`:

```
for key, val in hp.items():
    print('Number of ECTS points for ' + key + ': ' + str(val))
```

```
Number of ECTS points for DA2004: 7.5
Number of ECTS points for MM2001: 30
```

All of these functions return iterable objects with elements consisting of keys, values, or key-value pairs.

4.1.5 Functions for dictionaries

We can manipulate dictionaries with functions similar to those we had for lists:

```
print(len(hp))
hp['foo'] = 20.0
print(len(hp))

print('DA2004' in hp)
print('20.0' in hp)
```

```
2
3
True
False
```

One more useful function is `.get()`, which takes two arguments: a key and a value to be returned if the key is *not* in the dictionary. If the key *is* found, its associated value is returned.

```
print(hp.get('DA2004', 42)) # key exists
print(hp.get('bar', 42)) # key does not exist
```

```
7.5
42
```

4.2 File handling

Files are data stored on the hard disk: text files, pictures, video, and so on. We access files by interacting with the operating system. This can be done directly from Python.

4.2.1 Reading from files

To read from a file, for example some `data.txt`, we first create a "handle". We do this using the function `open()`, which takes two required arguments: the name of the file as a string and a string that specifies if the file will be readable, writable, etc.—see the help/documentation string for more information.

```
h = open('data.txt', 'r')   # 'r' for "read"
```

To read in all of the file's contents, we then write:

```
h.read() # whole file in string
```

To instead get a list of strings where each list element is a line from the file:

```
h.readlines() # whole file as list of strings
```

In many cases, these functions work poorly with large files. Problems arise when we read in a file that is larger than the computer's RAM.

To only read a single line from `h`, we use:

```
h.readline()
```

Note that the contents of a file are read through a handle *one time*: if we have already used `read` or `readlines` with `h`, `readline` will not do anything, as the handle/file pointer has already come to the end of the file.

4.2.2 Writing to files

If we want to write to a file `out_data.txt`, we must again first create a handle, this time with `w` for *write*.

```
output = open('out_data.txt', "w")
```

This either *creates* or *empties* `out_data.txt`. **Warning:** Be aware that everything in the file will disappear if it already exists! To now write to `output` (that is, the file `out_data.txt`), we use `write`:

```
output.write("This is my first line")
output.write("\n") # start a new line
output.write("This is my second line")
```

4.2.3 Closing files

When we are finished with a file, we should close it:

```
h.close()
output.close()
```

It is important to close files:

- The number of open files is limited in an operating system— don't be wasteful with them!

- Another program might want to write to the file—close your handle other programs don't need to wait!
- You can lose data if you don't close the file.

When a program finishes or is terminated, all its handles are closed automatically.

4.2.4 An example

Consider a file `people.txt`with comma-separated information: `last name,first name,birth year`.

```
Svensson,Lisa,1815
Olsson,Erik,1901
Mörtberg,Anders,1986
```

The following program creates a file `modern.txt` containing the full name of each person from the 1900's:

```
h_in = open('people.txt', 'r')
h_out = open('modern.txt', 'w')
for n in h_in:    # loop through the file, row by row
    lname, fname, year = n.split(',')
    if int(year) > 1900:
        h_out.write(fname + ' ' + lname + '\n')
h_out.close()
h_in.close()
```

If we now open the file `modern.txt`, we find it contains

```
Erik Olsson
Anders Mörtberg
```

4.2.5 Important idiom: `with`

Here we have an alternative notation to open/close:

```
with open('ut.txt', 'w') as out:
     out.write("hej hopp")
```

This little example corresponds to

```
out = open('ut.txt', 'w')
out.write("hej hopp")
out.close()
```

It is good practice to use 'with'—this is idiomatic Python. Some advantages:

- Easy to read
- Encourages good code structure
- Considered to simplify error handling
- The file is closed when execution leaves the `with`-block

In Python, `with` is called a *context manager*. It can also be used with things other than files.

The example above can rewritten using `with` as follows:

```
with open('people.txt', 'r') as h_in, open('modern.txt', 'w') as h_out:
    for n in h_in:
        lname,fname,year = n.split(',')
        if int(year) > 1900:
            h_out.write(fname + ' ' + lname + '\n')
```

4.2.6 What to keep in mind when handling files

- Reading and writing files is slow.
  - Avoid reading a file more than once.
  - Use files as temporary storage only if you must.
- Some files are big. Avoid reading in everything if you don't need it.
- Avoid hard-coding file paths.
- A good rule of thumb: read from `stdin` and write to `stdout`. This is Unix tradition.
- Temporary files should have unique and temporary names
  - Use the module `tempfile` (https://docs.python.org/3/library/tempfile.html for safety and convenience!

4.3 More on loops: nesting

The syntax for a **for**-loop allows the indented "body" to contain one or more statements. We can naturally combine *multiple* loops:

```
for i in range(3):
    for j in range(3):
        print(i * j)
```

```
0
0
0
0
1
2
0
2
4
```

Note that we increase the indentation within each loop. Pay careful attention to the indentation, as it determines the structure of the program and how it will be executed! An example:

```
print("---------")
for i in range(1,4):
    for j in range(1,4):
        print(i, j, i * j, sep="   ")
    print("---------")
```

```
---------
1   1   1
1   2   2
1   3   3
```

```
---------
2   1   2
2   2   4
2   3   6
---------
3   1   3
3   2   6
3   3   9
---------
```

Note that `print("---------")` is only run once in each pass through the outer loop. What would happen if we instead had it indented to align with the previous line `print(i, j, i * j, sep=" ")`?

We can also restrict to only compute even products with the help of `if`-statements in loops. Note the indentation levels of the two different **`if`**-statements.

```
for i in range(1,8):
    if i % 2 == 1:
        continue
    for j in range(1,8):
        if j % 2 == 1:
            continue
        print(i * j)
```

```
4
8
12
8
16
24
12
24
36
```

4.4 Exercises

1. Given the two lists below

   ```
   a = ['a','b','c']
   b = [1, 2, 3]
   ```

   create a dictionary `{'a': 1 ,'b': 2,'c': 3}` using a loop.

   Tip: There are several ways to do this. One way is to use `range` to get an index that you can then use to access elements of `a` and `b`.

2. Write code that takes all pairs `(x,y)` of numbers between `2` and `20` such that `x != y` and prints out their greatest common divisor (GCD).

   Tip 1: Use nested loops and `range`.

   Tip 2: Use the function `GCD` from the previous chapter.

The code should for example write out the following:

```
x y GCD
2 3 1
2 4 2
2 5 1
2 6 2
2 7 1
2 8 2
# etc...
```

3. In exercise 2, it is unnecessary to print the GCD of, for example, both `x,y = 2,3` and `x,y = 3,2`. Rewrite the code so that `x` is always less than `y` in the printed lines.

   Tip: In the inner `for`-loop, what happens if we iterate starting from the current value of `x`?

4. In chapter 3, we worked with the function `vowels`. Use a dictionary that has the vowels in `vs` as keys and numbers as values (you can choose the values yourself). Modify the function so that it instead replaces each vowel with the corresponding number.

5. In chapter 3, we worked with the function `split` (given below). Use the structure of this function as a guide to help you write a function that takes a string and associates each word with a number representing where the word appears for the last time in the string. The function should return a dictionary with words as keys and numbers as values.

```
def split(s):
    out = []
    term = ""
    for c in s:
        if c == " ":
            out.append(term)
            term = ""
        else:
            term += c

    # add the final word unless it is empty
    if term != "":
        out.append(term)

    return out
```

   The function should produce results like so:

```
print(annotate_word("hej du hej hej du"))
```

```
{'hej': 4, 'du': 5}
```

6. How can we easily get `annotate_word` from exercise 5 to instead return a dictionary with numbers as keys and words as values?

7. Modify `annotate_word` from exercise 5 so that not only the last occurrence but *all* occurrences are stored in the dictionary. What data structure should we have as *values* in our data structure?

Chapter 5

Error handling/exceptions and list comprehensions

Things go wrong sometimes, in programming as in life, and we have to handle it. When it comes to programming, it is typically the status of a function we are interested in: was a computation successful, or was there an error? How do we continue after an error?

Until now, we have defined functions that return a result (value), while instructive messages are perhaps written to the user with `print`. Alternatively, a program might simply crash or get stuck in an endless loop.

In this chapter we will see how code, upon an error, can call (or "raise") an exception. These can be handled using **error handling**, which in Python is expressed using the `try` and `except` keyboards.

At the end of the chapter, we will have a look at another concept: list comprehensions. These provide a convenient way of creating and modifying lists, often much more compact than writing a loop.

5.1 Error handling and exceptions

A naive way of handling errors in a program is to return some kind of error code in the form of a number or string. We will first take a look at how that can play out, but quickly move on to see how errors can be handled more effectively with the help of exceptions.

5.1.1 Error handling via returned error codes

Using special return values for error handling leads to complex programs that tend to be hard to read. Many function calls must be followed by `if`-statements testing whether an error has occurred that needs to be dealt with. If an error has occurred, it can be tricky to return to the right place in the program from which to continue. Experience shows that programmers end up avoiding the increased complexity by ignoring returned error codes, which leads to unstable programs that can crash. This type of error handling is often found in C, Fortran, and other older programming languages.

An example of this old-fashioned technique follows below, where we call a function that modifies a list. Any return value other than 0 signals that an error has occurred.

```
def divide_list(l,x):
    if type(x) != int or type(x) != float:
        return 11
    elif x == 0:
        return -1
    else:
        # check that elements in list are integers
        if not all([ type(item) == int for item in l]):
            return 12
        for i in range(len(l)):
            l[i] = l[i]/x # we are modifying the list items here
        return 0

return_code = divide_list(l,x)
if return_code == -1:
    # do something
elif return_code == 11:
    # do something else...
elif return_code == 12:
    # do something else...
elif return_code == 0:
    # good!
```

This code is fairly messy and unnecessarily complicated. Many modern languages therefore support specific tools to handle errors via so-called *exceptions*.

5.1.2 Exceptions

Python makes use of the reserved keywords **`try`** and **`except`** to pair up "unsafe" code that might raise an exception with code to be run in that case.

```
try:
   # unsafe code
   ...
except:
   # action
   ...

# Rest of the code
...
```

The keyword **`try`** opens a block of code regarded as unsafe (`# unsafe code` above), while **`except`** encloses the measures to be taken if an error arises. Just as with function definitions, conditionals, and loops, the indentation is essential. The block belonging to **`except`** (`#action`) is executed only if an exception is raised in the unsafe passage within the `try`-block: if no error has occurred, the execution continues below without running the code in the **`except`**-block. If an error occurs, then the code in the **`except`**-block is run. It might end the program or handle it in another way—perhaps even ignore it, though one should not do so carelessly.

5.1.2.1 Example: reading an integer from the user

Consider the following code snippet:

```
while True:
    answer = input("Write a number (write 0 to quit): ")
    x = int(answer)
    if x == 0:
        print("Bye bye!")
        break
    else:
        print("The square of the number is: " + str(x ** 2))
```

What happens if the user enters something other than a number?

```
Write a number (write 0 to quit): hej
```

```
ValueError: invalid literal for int() with base 10: 'hej'
```

As we've seen earlier, this results from the fact that the value of `input` always has the type `str` and that it is not always possible to explicitly coerce from `str` to `int`. Here, `ValueError` is an exception: there is something wrong with the value of the argument (`'hej'`) to `int`. Using **`try`**/**`except`**, we can handle the error in some appropriate way:

```
while True:
    try:
        answer = input("Write a number (write 0 to quit): ")
        x = int(answer)
        if x == 0:
            print("Bye bye!")
            break
        else:
            print("The square of the number is: " + str(x ** 2))
    except ValueError:
        print("You must write a number!")
```

In the code above, any exception of the kind `ValueError` raised in the **`try`**-block will be caught. If some other kind of error is raised, the program will terminate and display the error as before. We can also catch *all* errors by using an unqualified **`except`**:

```
while True:
    try:
        answer = input("Write a number (write 0 to quit): ")
        x = int(answer)
        if x == 0:
            print("Bye bye!")
            break
        else:
            print("The square of the number is: " + str(x ** 2))
    except:
        print("You must write a number!")
```

In order that the code be predictable and clear, however, one should write as little code within the

`try`-block as possible. For example, we can move the `if`-statement in the code above out of the `try`-block like so, making it clearer where we expect an exception might be raised:

```
while True:
    answer = input("Write a number (write 0 to quit): ")
    try:
        x = int(answer)
    except ValueError:
        print("You must write a number!")
        continue

    if x == 0:
        print("Bye bye!")
        break
    else:
        print("The square of the number is: " + str(x ** 2))
```

5.1.2.2 Example: calculating the lengths of lines in a file

Suppose we want to calculate the length of each line in a file:

```
def len_file(filename):
    with open(filename, 'r') as h:
        for s in h:
            print(len(s))

len_file("foo.txt")
```

If the file `foo.txt` does not exist, we receive the following error:

```
FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'
```

It is therefore better to write:

```
def len_file(filename):
    try:
        with open(filename, 'r') as h:
            for s in h:
                print(len(s))
    except FileNotFoundError:
        print("Could not open file " + filename + " for reading.")

len_file("foo.txt")
```

5.1.2.3 Different actions for different exceptions

There are many things that can go wrong. Depending on which exception is raised, it could be appropriate to take different actions. In the example below, we respond differently depending on whether something goes wrong with file reading, division by zero, or coercing a string read from a file to an integer. Note that the **`try-except`** need not be placed exactly where the error occurs.

```
def divide_by_elems(filename,x):
    quotients = []
    with open(filename, 'r') as h:
        for n in h:
            frac = x / int(n)
            quotients.append(frac)
    return quotients


try:
    data = divide_by_elems('numbers.txt', 2)
except IOError:
    print("divide_by_elems: A file-related problem occurred.")
except ZeroDivisionError:
    print("divide_by_elems: Division by zero.")
except ValueError:
    print("divide_by_elems: Character could not be converted to int.")

print(data)
```

For example, if the code above were run with a file `numbers.txt` containing the following:

```
1
2
0
4
3
0
```

then an exception would be generated and caught. Which? Can you make changes so that each of the other exceptions is raised?

5.1.2.4 Exceptions for control flow

We can also use exceptions for control flow. Suppose we want to calculate the quotients of the elements at each index of two lists (and ignore all elements at the end of the longer list), but not crash if there is a division by zero. To do so, we can write the following:

```
def compute_ratios(xs,ys):
    ratios = []
    for i in range(min(len(xs),len(ys))):
        try:
            ratios.append(xs[i]/ys[i])
        except ZeroDivisionError:
            ratios.append(float('NaN')) # NaN = Not a Number
    return ratios

print(compute_ratios([2,3,4],[2,0,5]))
print(compute_ratios([2,3,4,0],[2,0,5]))
```

In this case, we could have equally well used an `if`-statement and tested if `ys[i] == 0`. We'll say more

on this later.

5.1.3 General advice on exceptions

It is considered idiomatic Python to make use of `try-except`. It is not unusual when reading Python code to see *EAFP*, which stands for *it's Easier to Ask Forgiveness than Permission.* This is contrasted with *Look Before You Leap (LBYL).* This does not mean that you should *overuse* `try-except`: raised exceptions can sometimes give the clearest information. Not every exception in code is meant to be handled. The best time to use `try-except` is when you have an unsafe passage and a clear idea of what should be done in case of error.

So what is an unsafe passage in a program? Some typical cases are:

- All file handling
- Communication with the user
- Code that mentions insecurity in its documentation ("throws exception if…")
- Code you discover is unsafe (for example, implements an unstable algorithm)

There are several common actions to take after catching an exception:

- **Print an error message and exit:** used for example if data is missing or the error is so serious that there is no reasonable way to continue.
- **Print a warning, ignore the error, and continue:** if the program has received strange input data, but can ignore it and continue computation anyway.
- **Handle the error and continue:** works well for some foreseeable errors. For example, if you get bad input from a user, you can simply ask for new input.
- **Create a new exception:** This is appropriate if you have code where several errors can occur, but these can be organized into more informative groups. If an algorithm is unstable, `ArithmeticError` might be a better error than `ZeroDivisionError`: perhaps the user of the algorithm knows about the algorithm's general instability, but won't understand what it has to do with division by zero.

There are also actions that you should avoid:

- **Don't ignore the error!** The worst thing you can put in an `except` block is `pass`. It then becomes very difficult for the user understand what went wrong and where it happened. Python would have helpfully raised an exception, but the programmer decided to suppress it. The error may cause problems down the line, and it may be difficult to trace them back to the source.
- For **serious** errors, it is *not enough* to print an error message and continue. It isn't useful to let a program continue if the error is so serious that you know it will cause another error later. Some errors should simply end the program.

5.1.4 Managing your own uncertainty

It is not always necessarily a bad thing for code to raise an error. It is difficult, if not completely impossible, to for example

- Predict the format of *all* possible inputs.
- Ensure that all users (including yourself) call functions correctly.

Python includes many specific exceptions (see the documentation for a detailed accounting):

- `Exception`
- `ArithmeticError`
- `IOError`

- `IndexError`
- `MemoryError`
- `ZeroDivisionError`
- and so on.

In Python, an exception is not just a text string or error code; exceptions are a class of their own. (We will talk about classes in more detail in a later chapter.) This makes it possible to easily create your own exceptions based on the existing ones.

You can raise an exception with the reserved keyword **`raise`**. (In Java or C++, the analogous operator is `throw`.) For example, if we want to get the first element of a list but throw an error if the list is empty, we can write:

```
def head(xs):
   if not xs:
      raise Exception("head: input list is empty")
   else:
      return xs[0]

try:
    print(head([]))
except Exception as e:
    print(str(e))

print(head([]))
```

```
head: input list is empty
# specific information about where the exception was caught
Exception: head: input list is empty
```

Note that the code continues after the *first* time `head` is called in the **`try`**-**`except`** block: only the *string* with the error message is printed. We can "save" the exception in a variable that we can later use (for information, for example, or if we want to re-raise the exception after say saving something to a file) by writing **`except`** `Exception as e`; here `e` is an identifier to which we assign the caught exception. The string with the error message can then be obtained by writing `str(e)`. The *second* time `head` is called above, the exception occurring in the function is raised without being caught. The same information is printed again, but now it is the raised `Exception` that delivers the message, rather than the code in the **`except`**-block, and the program exits immediately after.

The keyword **`raise`** is useful for writing more detailed error messages, or when an error occurs that isn't covered by Python's existing error handling. An example of such an error is when an algorithm that we have written must maintain a value greater than `0.01` in a parameter `h` if the algorithm is to work. (Think for example of a step length on the `x`-axis for finding zeros of a polynomial.) We can write something like

```
def my_algorithm(x, h):
    while True:
        if h < 0.01:
            raise ArithmeticError('Stepsize h should not be smaller than 0.01.\
                                   The value of x was:', h)

        # check polynomial root
```

```
        ...

        # update h
        ...
```

5.1.5 How you deal with exceptions depends on what you're writing

There are different scenarios where we might want to use exceptions differently.

- **Application programming:** If you write a program that uses code from different modules, both your own and others', then you will encounter functions and methods that generate exceptions. It then becomes important to use `try-except` to deal with exceptions when they arise. The program should not exit abruptly because of a problem that could be foreseen and perhaps easily dealt with. You also want to avoid forcing a user to understand a Python error message.

  Some examples:

  - If an exception occurs because of bad input, then you should tell the user in a clear way. Letting through an error message from an exception is not user-friendly, because the cause of the problem is often not at all clear from the exception that was raised.
  - If an exception occurs because of, say, numeric uncertainty or uncertainty in the environment (for example, because a file with some name already exists), then you should probably deal with this somehow rather than throwing an exception in the user's face.

- **Support programming:** It is very common to write separate code as support for the program you actually want to write. If your goal is to write your own version of Matlab, then you'll need many functions that perform calculations; if you want to analyse data stored in a complicated format, it might be good to have a module that just handles the data so that you can later focus on the calculations. These support routines can run into problems that must be reported to the user (that is, the main program), and raising an exception with `raise` is an appropriate way to do so.

Python will report *that* an error has occurred. As the programmer, you can report *why* an error has occurred.

5.1.6 Common missteps

5.1.6.1 Use `if`-statements instead of exceptions for control flow

It can be tempting to use `except` for control flow, but often it is better to instead use an `if`-statement. For example, consider this code:

```
def elem_access(xs,i):
    try:
        if i >= len(xs):
            raise IndexError
    except IndexError:
        print("Index is out of range.")
    return xs[i]
```

There is clearly no point in first raising an exception and then immediately catching it. Another mistake in this example is that if there is an error, the program will *continue* after printing an error message, here trying to access `xs[i]` even though `i` is too large. If you catch an exception, then you should handle

the problem in some way. It is often appropriate to end the program by printing an error message and calling `quit()`.

5.1.6.2 Overuse

Don't use exceptions to patch up code that isn't behaving as you expected.

In the example below, we create a new dictionary with keys `e` taken from a list `selected` by calling a function that uses the information in another dictionary `data`. Clearly the programmer has noticed that not all values in `selected` exist in `data` and therefore tried to compensate by catching the `KeyError`'s that arise:

```
data = {1 : "Make bed",
        2 : "Shower",
        4 : "Eat breakfast"}
selected = [1,3,2]

d = {}
for e in selected:
    try:
        d[e] = perform_task(data[e])
    except KeyError:
        print()      # Bad line of code!
```

Instead, one should try to eliminate the underlying cause of the error.

5.2 List comprehensions

List comprehension is a tool for creating lists that is also popular in functional programming languages and often comes up in functional-like programming in Python. List comprehensions are useful for initializing and operating on lists. A benefit, which perhaps lies behind their popularity, is that they can be very clear thanks to their close similarity to mathematical notation. For example, consider the set of squares of the natural numbers up to `10`:

```
[x**2 for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can also filter out elements: not all need be added to the list. For example, we can create a list of all odd numbers up to `20` like so:

```
[x for x in range(20) if x % 2 == 1]
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

(We can shorten this expression a bit; can you see how?)

We can also call functions that convert elements according to some scheme:

```
def transform(x):
    if x % 2 == 0:
        return "Even"
    if x % 3 == 0:
```

```
        return True
    else:
        return x
print([transform(x) for x in range(20)])
```

```
['Even', 1, 'Even', True, 'Even', 5, 'Even', 7, 'Even', True]
```

Summary: a list comprehension has three parts:

- the result part, an expression that gives the elements in the result;
- the generator part, where `for` is used to feed values to the result part;
- the predicate part, which determines which values from the generator will be used.

Here is a more complicated example with a nested loop:

```
[(a,b) for a in range(2,8) for b in range(2,8) if a % b == 0]
```

```
[(2, 2), (3, 3), (4, 2), (4, 4), (5, 5), (6, 2), (6, 3), (6, 6), (7, 7)]
```

The above is convenient for getting all combinations of elements in two lists. Note that the output is a list of **tuples**, that is, pairs of elements.

A useful function is `zip`, which takes two lists of the same length and makes a list of tuples pairing together the two elements at each index. This function is already available in Python, but we could easily implement it ourselves using list comprehensions.

```
def zip(xs,ys):
    if len(xs) != len(ys):
        print("zip unequal lengths!")
        return []
    return [ (xs[i],ys[i]) for i in range(len(xs)) ]
```

5.2.1 `dict`-comprehensions

List comprehension is the traditional technique, but in Python these have been generalized to dictionaries (and sets, which we will introduce in a later chapter). In short, we can create dictionaries using `dict`-comprehensions. In this case, we must create both keys and values:

```
my_dict1 = {x: x ** 2 for x in (2, 4, 6)}
my_dict2 = {str(x): x ** 2 for x in range(3)}
my_list = ['1', '42', '13']
my_dict3 = {'key_' + x: int(x) for x in my_list}

print(my_dict1)
print(my_dict2)
print(my_dict3)
```

```
{2: 4, 4: 16, 6: 36}
{'0': 0, '1': 1, '2': 4}
{'key_1': 1, 'key_42': 42, 'key_13': 13}
```

5.3 Exercises

1. What happens if we call the function `divide_by_elements` (the function given in section 5.1.2.3) with a string as the argument to `x`? If the string can be converted to an integer, let us say the function should continue without crashing; otherwise it should return `None`. Achieve this behavior with the help of **`try`** and **`except`**.

2. Rewrite the function `divide_by_elems` so that it does not break after the first error, but continues to divide the number `x` with lines that come after. For example, for the file `numbers.txt` as above, the function should return the list

   ```
   [2.0, 1.0, 0.5, 0.6666666666666666]
   ```

   Tip: Replace the line

   ```
   with open(filename, 'r') as h:
   ```

   with

   ```
   h = open(filename, 'r')
   ```

   That is, do without the **`with`** functionality. This will make the restructuring a bit simpler. You can close the file with `h.close()`.

3. Begin with the following list:

   ```
   xs = [2,-167,12,5676,-14,-7, 12,12,-1]
   ```

   Create, with the help of a list comprehension, a list `ys` that contains only the positive numbers from `xs`.

4. Using a list comprehension, create a list `ys` in which all the numbers in the list `xs` from exercise 3 have been converted to positive if they are negative (for example `-167` becomes `167`).

5. Begin with the following function:

   ```
   def is_prime(x):
       if x >= 2:
           for y in range(2,x):
               if not (x % y):
                   return False
       else:
           return False
       return True
   ```

   Create a list with all the prime numbers under 100 using a list comprehension.

6. Begin with the following list:

   ```
   xs = [2,2,7,9]
   ```

   Using a list comprehension, create a list `ys` containing the results of multiplying every possible pair of elements taken from `xs`.

7. Using a list comprehension, create a list `ys` containing the results of multiplying every possible pair of elements taken from `xs` *except* the results of multiplying each element with itself.

Obs: The number `2` at index `0` in `xs` should not be treated as the "same" as the `2` at index `1`. Thus, you should not test whether the numbers are the same, but rather whether the indices are the same.

Tip: use `range`.

8. Rewrite Lab 1 using exceptions so that the program does not crash on improper input.

Chapter 6

Sequences and generators

6.1 Sequences

Python has support for a number of different types of "sequences", which generally support the same kinds of operations. Examples of such operations include:

- `+`: concatenation/union
- `*`: repetition
- `in` and `not in`: test if a value occurs/does not occur as an element in the sequence
- `len()`: length of the sequence/how many elements it has
- `min()` and `max()`: minimum and maximum element in the sequence (if the element type is compatible with < and >)
- `[:]`: slicing for accessing elements
- and more…

You probably recognize many of the operations above from `list`. Indeed, lists are a sequence type in Python. The other types we will have a look at in this chapter are tuples, `range`, and strings, all of which we have previously encountered at one point or another.

Sequence types can be divided into two subcategories: *mutable* and *immutable*. By mutability we mean that a created object can be *changed* after being created. Conversely, immutable objects are constant. How an object is used and which side effects operations have often depend on whether an object is mutable or not.

For more detailed information, see the Python documentation on sequence types.

6.1.1 Lists

Lists are a type of sequences that are *mutable*. This is an important characteristic of lists: we can update and insert values in a list without making a copy of it. We have already seen many examples of what we can do with lists, but to emphasize the effects of mutability of lists, consider the following function definition and code:

```
def sum_of_increment(l, x):
    for i in range(len(l)):
        l[i] += x
```

```
    ret_sum = sum(l)
    return ret_sum

mylist = [3, 3, 4, 5]
print("mylist is:", mylist)
s = sum_of_increment(mylist, 2)
print("mylist after first call:", mylist)
print("First call returns:", s)
s = sum_of_increment(mylist, 2)
print("mylist after second call:", mylist)
print("Second call returns:", s)
```

```
mylist is: [3, 3, 4, 5]
mylist after first call: [5, 5, 6, 7]
First call returns: 23
mylist after second call: [7, 7, 8, 9]
Second call returns: 31
```

The result of the function call is different every time, even though the function is called with the same variables every time! This results from the fact that `mylist` is modified inside of the function, which is possible because lists are mutable. It is also important here that Python does not *copy* list arguments (here `l`) to a called function (here `sum_of_increment`). This is sometimes what you want, but you should also be careful to avoid unintended consequences.

For more on lists and the functions that the `list` type supports, see the Python documentation and more on lists.

6.1.2 Tuples

Tuples are similar to lists, but they are *immutable*, so we cannot simply update a tuple. We have seen tuples before, even if we haven't explicitly created variables of tuple type: multiple return values from functions are in fact tuples of multiple elements. The syntax for tuples is very similar to that for lists, the difference being that we use `()` to create a tuple instead of `[]`.

```
mytuple = (3, 3, 4, 5)
myothertuple = 3, 3, 4, 5 # equivalent syntax
print(mytuple)
print(mytuple == myothertuple)
print(mytuple[0])
print(mytuple[3:1:-1])
print(len(mytuple))
print(3 in mytuple)

# function returning multiple values
def integer_div(nominator, denominator):
    q = nominator // denominator
    r = nominator % denominator
    return q, r

divs = integer_div(17, 10)
```

```
print(type(divs))

# tuples are immutable
mytuple[0] = 42
```

```
(3, 3, 4, 5)
True
3
(5, 4)
4
True
<class 'tuple'>
TypeError: 'tuple' object does not support item assignment
```

Notice that slicing a tuple gives back a tuple: slicing any sequence type gives back an element of the same type.

If we tried to rerun the example from the section Lists with the function `sum_of_increment` defined in the same way but applied to a tuple, we would get an error:

```
mytuple = (3, 3, 4, 5)
s = sum_of_increment(mytuple, 2)
```

```
TypeError: 'tuple' object does not support item assignment
```

Just like lists, tuples can contain elements of different types, and we can nest them:

```
t = (12345, 54321, 'hello!')
mytuple = (t, (1, 2, 3, 4, 5))
print(mytuple)
```

Tuples can even contain *mutable* objects.

```
t2 = ([1, 2, 3], [3, 2, 1])
print(t2)
```

This last example `t2` *cannot* be used as a key in a dictionary. Compare for example:

```
d = {t: 'hello'} # works
d2 = {t2: 'goodbye'} # TypeError: unhashable type: 'list'
```

We said in a previous chapter that key values must be constant. As we see in this case, it is not enough that a tuple itself is immutable: if it contains mutable objects (lists), it cannot be used as a key.

```
# t2[0] = [1, 2, 4] # throws TypeError, immutable
print(t2)
t2[0][0] = 42
print(t2)
```

```
([1, 2, 3], [3, 2, 1])
([42, 2, 3], [3, 2, 1])
```

6.1.3 range

Python's range is also an *immutable* sequence type. It behaves like a tuple but lacks support for certain operations, for example + and *.

```
myrange = range(10)
print(myrange[3])
print(myrange[2:9:3])
print(4 in myrange)
print(42 in myrange)
myrange + myrange
```

```
3
range(2, 9, 3)
True
False
TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

Again, slicing here gives back another object of range type.

6.1.4 Strings

Strings are also an example of a sequence type, again *immutable*. Like lists, we have already made frequent use of strings. There are very many useful string functions, most with names that clearly explain what they do. The following lines give just a selection of commonly used string functions:

```
my_string = "Hello world! This is fun"
print(my_string.index("o"))
print(my_string.count("l"))
print(my_string.upper())
print(my_string.lower())
print(my_string.capitalize())
print(my_string[3:7])
print(my_string[3:7:2])
print(my_string[::-1])
print(my_string.startswith("Hello"))
print(my_string.endswith("asdfasdfasdf"))
print(my_string.split())
print(my_string.split("i"))
```

```
4
3
HELLO WORLD! THIS IS FUN
hello world! this is fun
Hello world! this is fun
lo w
l
nuf si sihT !dlrow olleH
True
False
['Hello', 'world!', 'This', 'is', 'fun']
['Hello world! Th', 's ', 's fun']
```

```
H
```

Remember that in Spyder (in the console, for example), you can easily see which functions are associated with strings (or a given object in general) by using `dir()`. For example, one can call:

```
dir(my_string)
```

This can give a little *too much* information. For a more interactive menu of functions, you can enter `my_string.` (notice the period) in the console and press `TAB`.

For more on strings and more string functions, see the Python documentation.

6.2 Generators

Generators in Python are iterable objects that return a value to the user after each iteration. What the value is depends on how a generator is defined. We have encountered several other kinds of iterable object before; for example, all the sequence types we just saw in Sequences are iterable.

The biggest difference between generators and sequence types (besides `range`) is when calculation of elements happens. Calculation can either happen *immediately* when one creates an object, or not until the value of the element *is needed*, for example when one executes a loop. Generators are an example of a type where the value of an element is calculated first when it is needed. This is called *lazy evaluation*.

6.2.1 Lazy and strict evaluation

If all the values in an object are calculated when the object is created, we call this *strict* or *eager evaluation*. The opposite is *lazy evaluation*, where the values are calculated when needed. An example with sequence types that shows the difference:

```
mylist = [0, 1, 2, 3, 4, 5]
myrange = range(6)
print("mylist: ", mylist)
print("myrange: ", myrange)
```

```
mylist:  [0, 1, 2, 3, 4, 5]
myrange:  range(0, 6)
```

Here we have created a list `mylist`, which is an example of strict evaluation: all the values that the list contains are here calculated and stored in memory, as the printed output shows. The printing of `range`, on the other hand, shows only that it is an object of type `range` created with arguments `(0, 6)`. This is an example of lazy evaluation: the values that `myrange` contains are *calculated* only when one for example runs `myrange[0]`. Lazy evaluation is useful for iteration, where it is much more *memory-efficient* than iterating over an object that is stored in memory all at once.

The `range` type is a bit of a special case compared with the other sequence types, precisely because it is lazily evaluated. It exists because it is such a common idiom to use a sequence of increasing numbers. If we want to use lazy evaluation for more complicated structures, then we turn to generators.

6.2.2 Generator functions

In Python, we create a generating function in a very similar way to how we create an ordinary function. When an ordinary function is called, all the code in the function is executed until a `return`-statement is

encountered, whereupon a value (possibly) is returned to the user and the function ceases to execute. A generator function returns a value with **`yield`**, but the function's *state is remembered*. The next time the function is called, it continues from where it left off until the *next* time a **`yield`**-statement is encountered, whereupon it returns the value and again "pauses".

Here we use "function call" a little loosely. In fact, the generator function is called only *one time* and gives back a generator. It is this generator that executes the code in the generator function when we iterate over it.

An example will be more clear:

```
def inf_gen(start=0, step_size=1):
    """Infinitely generate ascending numbers"""
    while True:
        yield start
        start += step_size
```

This generator function counts up endlessly: it is an "infinite `range`". If we had tried to create a list like this, we would quickly see that it is hard to create an object with *infinitely* many elements: we would have run out of memory on our computer. But if we represent it as a generator, we have no problem creating such a thing or iterating over it. To access the underlying generator we can iterate over, we call the generator function:

```
my_inf_gen = inf_gen(10, 1)
print(my_inf_gen)
print(next(my_inf_gen))
print(next(my_inf_gen))
print(next(my_inf_gen))
```

```
<generator object inf_gen at 0x7f1d7855be40>
10
11
12
```

Here we use the `next` function with the generator `my_inf_gen` as an argument to step to the next value. It is in fact `next` that is implicitly called when we iterate over `my_inf_gen` by for example writing **`for`** `i` **`in`** `my_inf_gen: print(i)`. We cannot do that here, because it would continue infinitely, but test it for yourself to see what happens. (To exit from an infinitely looping program, you can press `CTRL+c`.)

Generators do not, of course, need to be infinitely iterable as in the above example. As you can see, you can decide for yourself how complicated a generator will be with more or less the same syntax as for an ordinary function. The iteration over a generator ends when execution reaches the end of a function body without encountering a **`yield`**. Just as one can have several **`return`**-statements in a function, so can one have several **`yield`**-statements in a generator function. The difference is that function execution continues from where it finished at the last **`yield`**. An important detail—and difference from `range`—is that we can only iterate over a given generator *once*.

We illustrate the concepts above with a simple generator function:

```
def my_generator(stop=5):
    i = 0
    j = 0
```

```
    while i + j < stop:
        print("i: ", i, "j:", j)
        i += 1
        yield i + j
        print("i: ", i, "j:", j)
        i += 1
        j += 1
        yield i + j
        print("i: ", i, "j:", j)

it = 1
gen = my_generator()
for i in gen:
    print("Iteration: ", it, "gen returns: ", i)
    it += 1

# loop a second time
for i in gen:
    print("Will this be printed???")
```

```
i:  0 j: 0
Iteration:  1 gen returns:  1
i:  1 j: 0
Iteration:  2 gen returns:  3
i:  2 j: 1
i:  2 j: 1
Iteration:  3 gen returns:  4
i:  3 j: 1
Iteration:  4 gen returns:  6
i:  4 j: 2
```

6.2.3 Generator expressions

Generators can also be written using syntax similar to list comprehension. In this case, no list is created; instead, we get a generator which we can then iterate over. Below we create a list and a generator using comprehension.

```
big_list = [x**2 for x in range(10**8)]
big_generator = (x**2 for x in range(10**8))
print("Size of big_list in memory:", big_list.__sizeof__())
print("Size of big_generator in memory:", big_generator.__sizeof__())
```

```
Size of big_list in memory: 859724448
Size of big_generator in memory: 96
```

As wee can see, the list comprehension has generated the whole list in the computer's memory. This illustrates the importance of using generators when we want to iterate over many values.

6.2.4 Example: a generator for all prime numbers

Generators are convenient for several purposes, among them representing infinite sequences.

Suppose we want to calculate all the prime factors of a number `x`. We can create a generator with *all* the prime numbers and see how many times each prime divides `x`:

```
def is_prime(x):
    for t in range(2, x):
        if x % t == 0:
            return False
    return True


def prime_divs(x):
    primes = (i for i in inf_gen(2) if is_prime(i))
    ps = []
    for p in primes:
        if p > x:
            return ps
        else:
            while x % p == 0:
                ps.append(p)
                x //= p

print(prime_divs(2))
print(prime_divs(3))
print(prime_divs(24))
print(prime_divs(25))
print(prime_divs(232))
```

```
[2]
[3]
[2, 2, 2, 3]
[5, 5]
[2, 2, 2, 29]
```

6.3 Exercises

1. We saw that the function `sum_of_increment(l, x)` changed our list when we called it several times. We also saw that we could not call `sum_of_increment` with a tuple. Can you draw from this a conclusion about when it can be good to use tuples?
2. Write a function `is_palindrome` that takes a string as input and returns `True` if the string is a palindrome (that is, if it is the same forwards and backwards) and returns the reversed string if it is not a palindrome.

   Tip: use string-slicing.
3. Does the `is_palindrome` you wrote in exercise 2 also work for lists and tuples? Why?
4. Write a function `occurrences` that calculates how many times letters occur in a string. The

function should return a dictionary with the letters as its keys and the number of times each letter occurs in the string as the corresponding values. The function should not differentiate between upper- and lowercase, but count these as the same (for a given letter). All characters that are *not* letters should be counted under the key `'non_alphas'`, and characters that don't occur in the string should not appear in the dictionary.

Tip: make use of the built-in string functions. See for example the Python documentation for string methods.

Example:

```
s = 'Hello my very happy friend! Is the sun shining?!'
uo = occurrences(s)
print(s)
for k, v in uo.items():
    print(k, ":", v)

s = 'JjJJJjj hHhh !/:;(+[//=--*])'
uo = occurrences(s)
print("\n" + s)
for k, v in uo.items():
    print(k, ":", v)
```

```
Hello my very happy friend! Is the sun shining?!
h : 4
e : 4
l : 2
o : 1
non_alphas : 11
m : 1
y : 3
v : 1
r : 2
a : 1
p : 2
f : 1
i : 4
n : 4
d : 1
s : 3
t : 1
u : 1
g : 1

JjJJJjj hHhh !/:;(+[//=--*])
j : 7
non_alphas : 17
h : 4
```

5. Use `occurrences(s)` above to write a program that reads in a file and prints out each letter paired with the number of times it occurs. To test it, you might use a file `palindrome.txt` with the

following contents:

```
not palindrome line
palindrome line enil emordnilap
no way
hello ! olleh
Tomorrow will be a glorious day!!!
oooooooooooooooooooooooooo
rust is also a programming language
ooooooooooooioooooooooooo
```

Tip: read in the whole file as a string.

6. Write a generator function `fibonacci(stop=x)` that generates the Fibonacci numbers (see https://en.wikipedia.org/wiki/Fibonacci_number) less than or equal to `x`.

   Tip: Adapt the structure of the function `my_generator` from above.

7. In the first line in the function `prime_divs` in Example: a generator for all prime numbers, we use a generator expression to create a generator "filtering" just the prime numbers from an existing generator. To define a filtered generator, we write:

   ```
   filter_gen = (i for i in iterable if function(i))
   ```

   Here `function` is a (user-defined) function that gives back `True` or `False` when `i` is given as an argument, while `iterable` is an iterable object (for example a generator, as we had in `prime_divs`). All `i` such that `function(i) == True` will become a part of `filter_gen`. Python has already implemented this in the function `filter`, which you can read the documentation for. A similar function is `map` (see the documentation) where the generator that is created contains the *values* that a function `f` *returns* when applied to all elements in an interable object.

   Write your own function `map_gen` that accomplishes this by taking an *arbitrary* **function** and an iterable object as arguments and returning a generator.

   Example:

   ```
   gen = map_gen(int, '123456')
   sum(gen)
   ```

   ```
   21
   ```

8. Write a function `palindrome_rows` that takes a filename as an argument and checks every row in the file to see if it is a palindrome or not. `palindrome_rows` should create a generator that one can iterate over when the function is called.

   Tip: keep in mind that lines are read *including* the newline symbol (`\n`). Modify `is_palindrome` from exercise 2 so that it only returns `True` or `False` based on whether the string is a palindrome (that is, doesn't return the original string in reverse if it *isn't* a palindrome).

   An example with the file `palindrome.txt` from above:

   ```
   p_rows = palindrome_rows('palindrome.txt')
   i = 1
   for p in p_rows:
       print("Row", i, "palindrome?", p)
       i += 1
   ```

```
Row 1 palindrome? False
Row 2 palindrome? True
Row 3 palindrome? False
Row 4 palindrome? True
Row 5 palindrome? False
Row 6 palindrome? False
Row 7 palindrome? False
Row 8 palindrome? True
```

9. You "know" that in the file `palindrome.txt` there hides a *single* capitalized word, but you do not know which one it is. You are extremely irritated by this! Use `open` together with string functions to find out which word it is *without using a loop*. You can assume that "words" are separated by spaces.

   Tip: one of the `strip` functions, combined with indexing, can be of use here.

Chapter 7

Modules, libraries, and program structure

7.1 Modules

Many programming languages come with a "module system", Python among them. These are useful when one writes larger programs and libraries of one's own. They can be used for:

- Code reuse – importing code that other programmers (or you yourself) have written;
- Structure – it is hard to work with a lot of code in one file, so spread it over several files instead;
- Collaboration – it is much easier to write code together as a team when it is divided up so that not everyone is making changes in the same file.

With the help of modules, you can begin to understand a large project as composed of independent units. Ideally, you do not need to understand all the details of a module: it is enough to understand its *interface*. The interface consists of the functions and variables intended to be visible to other programmers who use the module.

Large programs quickly become impossible to fully understand at a glance. Well-chosen module structure is thus essential for grasping both your own and others' programs.

A technical advantage of modules is that they help you to manage *namespaces*. Often interacting programs will want to use the same identifiers for functions and variables (for example `sort`). Modules conceal implementations as well as variable and function names.

Principles:

- Every Python file is a module.
- Anything to be used from another file must first be *imported*.

7.1.1 Importing modules

We can import modules in a number of different ways:

- `import modulename`: Allows functions and variables to be used with dot notation; in other words, the contents of `modulename` are imported "qualified".

```
import math
math.ceil(3.5)  # Round up
ceil(4.4)  # Error!
```

- `from modulename import f1, f2, ...`: Allow use of `f1` and `f2` (but no others) in your program without dot notation.

```
from math import ceil
ceil(3.5)
```

- `from modulename import *`: Make everything from `modulename` accessible without dot notation.

  Warning: this is considered to reduce readability and makes it difficult to tell which parts of a module your program depends on.

  Obs: identifiers that begin with _ are not imported. These are considered "private" to the module.

  Note: some modules assume that they will be imported using `*`, as they work by overwriting certain built-in functions.

- `import modulename as X`: Use modules with long names more easily with dot notation. For example:

```
import math as M
M.ceil(3.4)     # Useful for modules with long names
```

- `from modulename import X as Y`: For avoiding naming collisions. For example:

```
from math import gcd as libgcd

def gcd(a,b):
    r = a % b
    if r == 0:
        return b
    else:
        return gcd(b, r)

for a in range(2,10):
    for b in range(2,a):
        print(gcd(a,b) == libgcd(a,b))
```

- Modules can be grouped in *packages*, which can be understood as hierarchically arranged groups of modules.

```
import os.path as OS
if OS.isfile('data.txt'):
    ...
```

7.1.2 Writing your own modules

Suppose we have the following Python code in a file `circle.py`:

```
pi = 3.14
```

```
def area(radius):
    return pi * (radius ** 2)

def circumference(radius):
    return 2 * pi * radius
```

We can then import it in another file:

```
import circle

print(circle.pi)
print(circle.area(2))

import circle as C

print(C.pi == circle.pi)

tau = C.pi * 2

def circumference(radius):
    return tau * radius

print(C.circumference(2) == circumference(2))
```

Note: the file `circle.py` must be somewhere Python can find it when it looks for modules to import. The first place Python looks is the folder where the program/`.py`-file you are running is stored; the easiest option is therefore to keep your module file (here `circle.py`) in the *same* folder as the main program/file that imports the module.

7.1.3 `main()`-functions

A common idiom:

```
print("Hello")

def main():
    print("python main function")

if __name__ == '__main__':
    main()

print("__name__ value: ", __name__)
```

Here, `main` is called if and only if one runs this program as a standalone file. If the file is used as a module instead, then `main` is not run. You can thereby write modules that contain example programs and/or test code that is not run when the module is `imported`. The reason that `main` is not run if the file is imported as a module is that the variable `__name__` is set to the module name in that case, so the body of the `if`-statement is not run.

7.2 Popular libraries and modules

Python comes with batteries included: there are heaps of good and useful modules that come as part of the Python implementation. These modules are called Python's standard library. For a long list and documentation, see https://docs.python.org/3/library/.

- `math` – Mathematics: trigonometry, logarithms, et cetera.
- `sys` – Advanced file handling and more.
- `os` – Helps avoid depending on specifics of Win/Mac/Linux (for example avoid problems with "`/`" in filenames in Linux/Mac vs. "`\`" in filenames for Windows).
- `argparse` – Used to access parameters when one starts a program from the command line.
- `itertools` – Useful for advanced and fast iteration.
- `functools` – Several higher-order functions.
- `datetime` – Date and time types.
- `pickle` – Serializing of Python objects.
- `csv` – It is harder than you think to read a comma-separated file!
- `json` – Save data in JSON format so that you can easily read it in again.
- `matplotlib` – Visualize data.
- `numpy` and `scipy` – For calculations.
- ...: Investigate for yourself!

7.2.1 JSON and serialization

Problem: it is a pain to write and read files! Especially when it comes to large data structures (think: nested lists and/or dictionaries).

The solution: serialization.

Idea: dump data in a file in a standard format that can easily be read back in again.

There are standard Python-specific modules for this (`marshal` and `pickle`).

A good *portable* solution: JSON – JavaScript Object Notation

```
import json
legends = [ ['Lovelace', 'Ada', 1815]
          , ['Babbage', 'Charles', 1791]
          , ['Beurling', 'Arne', 1905]]

# Create a data file:
with open('legends.json', 'w') as outfile:
    outfile.write(json.dumps(legends))

# Later... write out birth years:
with open('legends.json', 'r') as data_h:
    external_data = json.load(data_h)
```

```
    for x in external_data:
        print(x[2])            # Print birth year
```

7.2.2 NumPy: for vectors and matrices

numpy (https://www.numpy.org/) is a package for scientific computation in Python. It comes with very efficient storage of vectors and matrices using "arrays". Python's `list` supports typical array operations, but also much more. That flexibility comes at the cost of efficiency. Arrays in numpy are closer to the computer's own representation, which makes calculations more efficient (better use of memory and computation time). Efficiency comes at the cost of flexibility, but in many applications it is well worth the tradeoffs.

One can create vectors and matrices using numpy arrays:

```
import numpy as np        # It is idiomatic Python to import numpy as np

a = np.array([1,2,3,4])   # From an ordinary list
b = np.ones((2,2))        # 2x2 unit matrix---note the tuple!
c = np.zeros((10,10))     # 10x10 zero matrix
d = np.linspace(0, 1, 16) # 16 elements from 0 to 1, floats!
e = np.reshape(d, (4,4))  # Creates a 4x4 matrix with numbers from 0 to 1

print(a)
print(b)
print(c)
print(d)
print(e)
```

```
[1 2 3 4]
[[1. 1.]
 [1. 1.]]
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[0.         0.06666667 0.13333333 0.2        0.26666667 0.33333333
 0.4        0.46666667 0.53333333 0.6        0.66666667 0.73333333
 0.8        0.86666667 0.93333333 1.        ]
[[0.         0.06666667 0.13333333 0.2       ]
 [0.26666667 0.33333333 0.4        0.46666667]
 [0.53333333 0.6        0.66666667 0.73333333]
 [0.8        0.86666667 0.93333333 1.        ]]
```

Note that `np` is an established alias, and `import` is most often used as in this example, including in the

numpy documentation.

Tip: try for example `help(np.linspace)` to see how it works.

We naturally get access to various operations on matrices, for example matrix multiplication:

```
m = np.array([[1,2], [3,4]])
v = np.array([1, 0.5])
product = np.matmul(m, v)
```

```
[[2. 5.]]
```

numpy generously regards an array as a vector in an appropriate sense. Here, the above v is treated as a column vector; if we wrote `np.matmul(v,m)` instead, it would be treated as a row vector.

There is a special data structure for matrices defined in numpy, but it is avoided today as there are plans to remove it. The current recommendation is to always use `np.array`.

numpy is a part of scipy (https://www.scipy.org/), which contains a number of different packages for doing scientific computation and handling "big" data with Python.

7.3 Rules of thumb for program structure and good code

It is hard to write good code. By "good" code, we mean code that is easy to read, understand, use, and maintain. By writing good code, we can also avoid introducing bugs. It is difficult to say exactly what makes code good, but hopefully you have gotten some experience of both good and less good code by reviewing each other's code. Many programmers have different definitions of what makes code good, in the same way that "good art" and "good literature" are very subjective.

You can find many Python-specific tips at https://docs.python-guide.org/writing/style/. There is also a style guide, with the mysterious name PEP-8, which was written by Guido van Rossum, the creator of Python. When you start programming in another language, it is a good idea to find out which conventions its programmers follow—it differs from language to language.

There are tools for automatic review of code. One of these is https://www.pythonchecker.com/, which runs on the web. There are also IDE's with similar support and libraries such as `pylint` that you can run in a terminal window.

7.3.1 Programming guidelines

Here is a list (in no specific order) of things that we think make code better:

- Avoid long lines; less than 79 characters per line is good. If a line is too long, it becomes hard to read. PEP-8 specifies this limit.
- One unit of code per line. The following code is bad:

  ```
  print('one'); print('two')

  if x == 1: print('one')

  if <complex comparison> and <other complex comparison>:
      # do something
  ```

  The following code is much better:

```
print('one')
print('two')

if x == 1:
    print('one')

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

- Strive to write short functions. The majority of functions should fit in half a laptop screen with font size 12. In some cases, they can of course be longer, but think carefully whether it is possible (or would be clearer) to factor out the code into smaller functions first. When a function is very long, it is hard to get a sense of everything it does at a glance.
- Divide up code into functions that do **one** thing and do it well. This makes it much easier to read, test, and understand the code.

  Example: "read in several datasets, do a calculation with each one, and write out a summary" should be divided up like so:

  - One function for reading in datasets.
  - One function that does the calculation on a dataset.
  - A main function that calls the two other functions and writes out the summary.
- Functions should return something. That way it is easier to test them and to understand what they do or are expected to do. There are situations where a function doesn't *need* to return anything (printing, list manipulation, and so on), but the default expectation should be that a function returns something. A common beginner mistake is to use global variables to handle data flow in a program and thus "not need" return values. Among beginners, a lack of return values can also be an indication of bad program structure.
- Every function should have a documentation string.
- Separate **calculation** from **interaction**: functions that do calculations should not use `print`/`input`. Functions that handle interaction should not do complex calculations. By interaction, we mean simple `print` statements as well as questions to the user. You will often want to reuse calculating functions without any associated interactive components.
- Think about how a program should be structured and which abstractions it should use *before* you start to code. Group functionality and attributes in classes[1]—so as to maximize abstraction and it will be easier to change your code later.
- Divide up code into small units, that is, into appropriate functions and classes. Code is called *modular* when every logical component of the code can be studied as its own little module which is then combined with other modules to create complex programs. This makes the program much more easily understandable.
- Do **not** return values of type `int` or `bool` as strings, for example `'2'` or `'True'`. This makes it harder to test and use functions later. It is also easy to introduce bugs in this way. For example:

[1] Classes will be introduced in chapter 9.

```
def f(x):
    return '2'

if f(1) == 2:
    print("This is not printed!")
else:
    print("This is printed!")
```

- Do not return "magic" values to indicate errors or problems. Use exceptions instead. For example: "`return 1000000`" is a bad way to signal "does not converge". **Exception:** `None` can be an acceptable magic value.
- Multiple return values are practical! Better than strings that contain several values.
- Variables/constants are better than hard-coded literals. For example, if we write a graphics program and want a window with size `600 x 600`, then it is better to first write

  ```
  width , height  = 600 , 600
  ```

  and then then use `width`/`height` instead of repeating `600` everywhere. This makes it easier to change the program in the future if we want the window to have a different dimension. It is also easier for someone reading code to understand our intention when we use `width` as opposed to some arbitrary number.
- Avoid global variables, as these often lead to "spaghetti code": messy code where completely separate parts of the program can affect each other. A reader should be able to understand one function without looking at other parts of a program. It is especially dangerous to manipulate global variables in functions (by using the keyword **`global`**).
- Include tests to both test and specify how a program is expected to behave. This is easier to do if the program is divided into smaller functions that do **one** thing and **return** it.
- Simple code is easier to read. You should therefore avoid writing unnecessarily complicated code. For example, "`if b == True:`" is more complicated than just "`if b:`". It is like Swedish: it is bad style to staple many small words together. In the same way that simple sentences are easier to read, simple code is easier to understand.

  Here is an example of an unnecessarily complicated program:

  ```
  def make_complex(args):
      x, y = args
      d = dict()
      d['x'] = x
      d['y'] = y
      return d
  ```

  Much better:

  ```
  def make_complex(x,y):
      return {'x': x, 'y': y}
  ```

  A good trick is to always read through your code several times and consider how it might be simplified.

- Be careful with mutable datatypes—many bugs arise from lists being accidentally emptied by say `pop()`! It is especially dangerous to store mutable objects in global variables or class attributes.
- Divide up larger projects in logical units in different files/folders.
- It should be clear from the identifier what a variable contains or what a function is meant to do. For example, if you have the line

  ```
  x = 17  # critical value, abort if exceeding
  ```

  then it is naturally better to replace `x` with something else, for example:

  ```
  critical_value = 17  # abort if exceeding
  ```

  One more example: instead of

  ```
  nc = 1.7       # normalization constant
  ```

  it is better to write:

  ```
  normalization_constant = 1.7
  ```

  Now no commentary is necessary.
- There are situations where function names can be uninformative: for example, the purpose of `main` is clear by convention.
- Avoid generic identifiers. Use identifiers that are descriptive.
  - *Bad*: `start`, `compute`, `f`, `foo`
  - *Good*: `remove_outliers`, `newton_raphson`, `compute_integral`
- Code changes. If a function's purpose changes, then you should change its name! It is confusing if your function is called `print_integral` but doesn't print anything.
- Comments are good, but not a replacement for simple code with well-chosen identifiers. If your code needs comments to be understandable, then you are probably doing something complicated. Good uses of comments are to explain why you are using a specific data structure (for example, `set` instead of `list`) or how a function is expected to be used.

  Comments that explain nothing more than what the code already says should be avoided. For example, the following comment is completely redundant:

  ```
  rows = []       # Create an empty list
  ```

  Anyone who knows a little Python already understands that this line creates an empty list, so why write it in a comment?

7.3.2 Signs of not-so-good code

- "Copy-paste code": it is easy to introduce bugs when you copy lines and paste them again elsewhere. The program also gets hard to read if it gets too long. Can you introduce a loop or function instead?
- Is it hard to come up with good tests? Then you haven't divided up your program into small parts that do **one** specific thing. Rethink your design and introduce better functions and abstractions.
- Deep indentation: if you have deeply nested `if`-statements, the code becomes hard to read and should be divided up into smaller parts. A common cause of deep indentation is that you have

several different logical cases; you should then create functions for the different cases. In this way, the case analysis becomes clearer, with the different function calls indicating what happens in each case.

Here is an example of "bad" code:

```
# Print the prime numbers from a collatz sequence
# for 100 consecutive starting numbers
for i in range(1,100):
    n = i
    primes = []
    while n != 1:
        is_p = True
        for x in range(2, n):
            if not (n % x):
                is_p = False
        if is_p:
            primes.append(n)
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n + 1
    print("Start number :", i, "Primes in Collatz sequence: ", primes)
```

This code block is indented to four levels. It isn't terrible, but we can structure it better:

```
def collatz_primes(n):
    """Computes prime numbers from
       the collatz sequence for an integer n.
    Parameters:
    n (int): input number
    Returns:
    list: list of prime numbers in collatz sequence for n
    """
    primes = []
    while n != 1:
        is_p = True
        for x in range(2, n):
            if not (n % x):
                is_p = False
        if is_p:
            primes.append(n)
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n + 1
    return primes

for i in range(1,100):
    primes = collatz_primes(i)
    print("Start number :", i, "Primes in Collatz sequence: ", primes)
```

We have broken out a special function, `collatz_primes(n)`, which is used in a **`for`**-loop, and the function has only three indentation levels (the **`while`**-loop, the **`for`**-loop, and the **`if`**-statement).

We can go a step further and move the update of the variable `n` and primality test into their functions:

```
def collatz_update(n):
    if n % 2 == 0:
        return n//2
    else:
        return 3*n + 1

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True

    for y in range(2,n):
        if not (n % y):
            return False
    return True

def collatz_primes(n):
    """Computes prime numbers from
       the collatz sequence for an integer n.
    Parameters:
    n (int): input number
    Returns:
    list: list of prime numbers in collatz sequence for n
    """
    primes = []
    while n != 1:
        if is_prime(n):
            primes.append(n)
        n = collatz_update(n)
    return primes

for i in range(1,100):
    primes = collatz_primes(i)
    print("Start number :", i, "Primes in Collatz sequence: ", primes)
```

Which version do you think is easiest to read and understand?

- Do your functions that do calculations lack parameters and **`return`**-statements? This will likely cause you trouble. The purpose of a function is usually to take inputs and compute something. The computation need not be mathematical—it can simply be manipulation of data. It is less common that a function lacks parameters entirely, but there are of course exceptions, for example a function that prints a menu or initializes a data structure. For beginners, however, it is often tempting to store data in one or more global variables that functions later work with. Such a setup makes it difficult to see how functions depend on each other; you need to read functions carefully to

identify the flow of data through the program. And you cannot use the function in another module later on if it doesn't take its parameters explicitly!

The code below works even if we put `is_prime` in another module `m` (that is, another file) and call it with `m.is_prime(x)`:

```
def is_prime(x):
    if x >= 2:
        for y in range(2,x):
            if not (x % y):
                return False
    else:
        return False
    return True

def main(x):
    print("The number is prime:", is_prime(x))

x = int(input("Provide a number to analyze: \n"))
main(x)
```

The code below is more poorly written and works only if we have the global variable `x` in the same file. This makes it impossible to usefully import the function in another file.

```
def is_prime():
    if x >= 2:
        for y in range(2,x):
            if not (x % y):
                return False
    else:
        return False
    return True

def main(x):
    print("The number is prime:", is_prime())

x = int(input("Provide a number to analyze: \n"))
main(x)
```

7.4 Exercises

1. Import the library `random` (see https://docs.python.org/3/library/random.html) and use it to generate 10 random *integers* between 1 and 100 and 10 random *floats* between 1 and 100.

   Tip: Read the documentation. Which functions are the most appropriate in this case?

2. Consider the code below, which reads in an integer and decides if it is a perfect number or a prime number. Rewrite the code so that it has better structure.

   Tip: Divide up the code into functions. Which parts of the code can naturally be made into functions? What should the functions take as parameters and return so that they can be useful

outside this module? You might want to have a main function `main(x)` that takes the number `x` as a parameter.

```
# checks if a number is (1) perfect (2) prime
x = int(input("Provide a number to analyze: \n"))
sum = 0
for i in range(1, x):
    if (x % i == 0):
        sum = sum + i

if x >= 2:
    prime = True
    for y in range(2,x):
        if not (x % y):
            prime = False
else:
    prime = False

if sum == x:
    print("The number is perfect")
else:
    print("The number is not perfect")
if prime:
    print("The number is prime")
else:
    print("The number is not prime")
```

3. Adjust your restructured program from Exercise 2 to let the user decide whether to choose a number themselves or let the program choose one at random. Where in the program should we put this new code?

4. The code below reads in a DNA-string (for example `'AGCTAGCGGTAGC'`) and first looks for the most commonly occurring substring of length `k`. Then the program prints out the pairwise distances between copies of this string. For example, for `'AGCTAGCGGTAGC'` and `k = 3`, the code below returns `[4, 6]`: there are 4 and then 6 nucleotides separating the start positions of the three occurrences of the most common substring, `'AGC'`.

   Exercise: Rewrite the program to be more readable. Which parts can be made into sensible functions? Make sure they both take appropriate parameters and return appropriate data structures. It should be possible to call any of the functions individually from another module.

```
# Given a sequence as input find distances between the copies of
# the most frequent substring

seq = input("Enter a dna string: ")
k = 3
substrings = {}
for i in range(len(seq) - k + 1):
    substring = seq[i: i+k]
    if substring in substrings:
        substrings[substring] += 1
```

```
    else:
        substrings[substring] = 1

max_count = 0
most_freq_substring = ""
for substring, count in substrings.items():
    if count > max_count:
        most_freq_substring = substring
        max_count = count


positions = []
for i in range(len(seq) - k + 1):
    substring = seq[i: i+k]
    if substring == most_freq_substring:
        positions.append(i)

distances = []
for p1,p2 in zip(positions[:-1], positions[1:]):
    distances.append(p2 - p1)

print(distances)
```

5. † You have five numbers in a list `[13, 24, 42, 66, 78]` and want to calculate all the ways that a number can be written as a "direct concatenation" of two elements from the list; by a "direct concatenation" of two integers we mean for example `13, 78 -> 1378` and `66, 42 -> 6642`. That is, want to look at all permutations of two elements from the list and combine each permutation to make a number.

   Tip: Look in the standard module `itertools` for permutations. To concatenate the integers, try first explicitly coercing the numbers to `str`, concatenating them with the string function `.join()`, and finally coercing them back to `int`.

   The output, if all the numbers are saved in a list that is then printed, should be:

```
[1324, 1342, 1366, 1378, 2413, 2442, 2466, 2478, 4213, 4224, 4266, 4278, 6613,
 6624, 6642, 6678, 7813, 7824, 7842, 7866]
```

6. Rewrite lab 1 (temperature conversion) so that the functions are in a separate module with the name `temp_functions.py` and document each function with a documentation string (see chapter 2). The main program should import the functions in `temp_functions.py` and handle errors with **`try`**/**`except`**.

Chapter 8

Functional programming

In this chapter, we look at *functional programming* in Python. This is a way of programming based on calling, combining, and modifying functions. In this way we can do much of what we have already seen in other chapters, but now working entirely with functions. For example, we can replace loops with *recursive* functions: functions that call themselves.

8.1 Recursion

Functions can call themselves and are then called *recursive*. There are many examples of this in mathematics, but recursion can also be useful in programming; it often produces elegant and short programs. It can, however, be hard to begin "thinking recursively" and see how to solve different problems with recursion. A good way to learn is to take a program you have already written with loops and rewrite it to use recursion instead.

8.1.1 Factorial

A classic example of a recursive function from mathematics in the factorial. This is written $n!$ and is calculated by taking the product of all numbers from 1 to n. For example:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

This can be described recursively, that is in terms of itself, like so:

$$\begin{aligned} 5! &= 5 * 4! \\ &= 5 * 4 * 3! \\ &= 5 * 4 * 3 * 2! \\ &= 5 * 4 * 3 * 2 * 1! \\ &= 5 * 4 * 3 * 2 * 1 * 0! \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 120 \end{aligned}$$

This is a recursive calculation with the *base case* $0! = 1$. A mathematician might write the general definition as:

$$n! = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

In Python, we can write it like so:

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

The case `n == 0` is called a *base case*. This is where the recursion ends and is wound back up. The other case is called the *recursive step* and involves making a "smaller" call, where the problem in some sense has been made smaller.

A recursive function is called exactly like any other function:

```
print(fac(5))
```

```
120
```

8.1.2 The Fibonacci numbers

Every course that covers recursion must use the Fibonacci numbers as an example. This is a sequence of numbers that begins with:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The nth Fibonacci-number, $F(n)$, is the sum of the two previous Fibonacci numbers. The sequence has two base cases, $F(0) = 0$ and $F(1) = 1$. We can thus write

$$F(n) = \begin{cases} 0 & \text{if } n \text{ is } 0 \\ 1 & \text{if } n \text{ is } 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

This is clearly a recursive definition. We can implement it like so:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

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

```
0
1
```

```
1
2
3
5
8
13
21
34
```

You can compare this implementation of a function that calculates Fibonacci numbers with the one from exercise 6 in chapter 6, which uses a generator.

8.1.3 Greatest common divisor, recursively

Recall Euclid's algorithm for calculating the GCD ("Greatest Common Divisor") of two numbers `a` and `b` from section 3.3.3:

- Assume `a >= b`,
- let `r` be the remainder of the integer division `a // b`,
- if `r == 0`, then `GCD(a, b) = b`,
- otherwise use that `GCD(a,b) = GCD(b, r)`.

A simple iterative implementation:

```
def gcd(a,b):
    if b > a:
        print("a must be bigger than b in gcd(a,b)")
        return None

    r = a % b
    while r != 0:
        a = b
        b = r
        r = a % b
    return b

print("GCD: " + str(gcd(15,12)) + "\n")
print("GCD: " + str(gcd(12,15)))
```

```
GCD: 3

a must be bigger than b in gcd(a,b)
GCD: None
```

It is not so easy to see how this implementation follows the algorithm. A recursive implementation is much closer:

```
def gcd(a,b):
    if b > a:
        print("a must be bigger than b in gcd(a,b)")
        return None
```

```
    r = a % b
    if r == 0:
        return b
    else:
        return gcd(b,r)

print("GCD: " + str(gcd(15,12)) + "\n")
print("GCD: " + str(gcd(12,15)))
```

```
GCD: 3

a must be bigger than b in gcd(a,b)
GCD: None
```

8.1.4 Loops from recursion

We saw above an example where we converted a program we had already written with loops to use recursion. This is always possible, both for **for** and **while** loops.

In the GCD-example, it was a **while** loop that was replaced with a recursive definition. Below, it is a simple function that uses a **for** loop to calculate the sum of all numbers up to and including x:

```
def f(x):
    s = 0

    for i in range(x+1):
        s += i;

    return s
```

We can write this recursively in the following way:

```
def f_rec(x):
    if x == 0:
        return 0
    else:
        return x + f_rec(x-1)
```

The idea here is that we call f_rec recursively with smaller and smaller numbers until to get to the base case when x is 0. This represents the loop in f. We can test that the two functions return the same results:

```
for i in range(5):
    print("f(" + str(i) + ") = " + str(f(i)))
    print("f_rec(" + str(i) + ") = " + str(f_rec(i)))
```

```
f(0) = 0
f_rec(0) = 0
f(1) = 1
f_rec(1) = 1
f(2) = 3
f_rec(2) = 3
```

```
f(3) = 6
f_rec(3) = 6
f(4) = 10
f_rec(4) = 10
```

8.1.5 Recursion over lists

We have seen how we can write recursive functions over numbers. We can do the same with other datatypes. The general idea is that we should have one or more base cases, while the recursive case should reduce the size of the input in some way so that we eventually get to a base case. For numbers, we often use `0` as a base case and shrink the input when we make a recursive call. If we want to write recursive functions over lists, we instead often use `[]` as the base case and remove an element from either the beginning or end in the recursive call.

We can in this way write our own `len` function:

```
def length(xs):
    # obs: lists behave as boolean values ([] is False)
    if xs:
        xs.pop()
        return (1 + length(xs))
    else:
        return 0

print(length([1,341,23,1,0,2]))
```

```
6
```

Warning: this function changes `xs`! Test the following:

```
mylist = [1,2,3,6,7]

print(mylist)

print(length(mylist))

print(mylist)
```

```
[1, 2, 3, 6, 7]
5
[]
```

As `length` has called `pop` repeatedly, `mylist` has been emptied out! To avoid this, we must call `length` with a *copy* of `mylist`:

```
mylist = [1,2,3,6,7]

print(mylist)

print(length(mylist.copy()))

print(mylist)
```

```
[1, 2, 3, 6, 7]
5
[1, 2, 3, 6, 7]
```

The sum of all elements in a list can be calculated in the following way.

```
def sum(xs):
    if xs:
        x = xs.pop()  # pop returns the element
        return (x + sum(xs))
    else:
        return 0

print(sum([1,3,5,-1,0,2]))
```

```
10
```

And the product like so:

```
def prod(xs):
    if xs:
        x = xs.pop()  # pop returns the element
        return (x * prod(xs))
    else:
        return 1

print(prod([1,3,5,-1,2]))
```

```
-30
```

8.1.5.1 Sorting lists

One good and fast sorting algorithm is *quicksort*. It works like so:

1. Choose an element in the list, called the "pivot".
2. Divide up the list into two sublists: one with all the elements smaller than the pivot and one with all the elements larger than the pivot.
3. Sort the sublists by recursive calls. The base case is the empty list, which is already sorted.
4. Combine the lists with the pivot element in the middle.

To implement this in python, we first write two helper functions for getting all elements smaller and larger than a given element:

```
# All elements less than or equal to v in xs
def smallereq(xs,v):
    res = []
    while xs:
        x = xs.pop()
        if x <= v:
            res.append(x)
    return res
```

```
# All elements greater than v in xs
def greater(xs,v):
    res = []
    while xs:
        x = xs.pop()
        if x > v:
            res.append(x)
    return res

print(smallereq([2,4,5,6,7,8],5))
print(greater([2,4,5,6,7,8],5))
```

```
[5, 4, 2]
[8, 7, 6]
```

We can then implement `quicksort` as follows:

```
def quicksort(xs):
    if xs == []:
        return xs

    p = xs.pop()
    s = quicksort(smallereq(xs.copy(),p))
    g = quicksort(greater(xs.copy(),p))
    return (s + [p] + g)

print(quicksort([8,3,3,2,4,5,8,5,6,7,8]))
```

```
[2, 3, 3, 4, 5, 5, 6, 7, 8, 8, 8]
```

We can instead write `smallereq` and `greater` directly with list comprehensions for an even shorter implementation:

```
def quicksort(xs):
    if xs == []:
        return xs
    else:
        p = xs.pop()
        s = quicksort([ x for x in xs if x <= p ])
        g = quicksort([ x for x in xs if x > p ])
        return (s + [p] + g)

print(quicksort([8,3,3,2,4,5,8,5,6,7,8]))
```

```
[2, 3, 3, 4, 5, 5, 6, 7, 8, 8, 8]
```

8.1.6 Tail recursion

A danger with writing recursive functions is that they can be much slower compared with iterative code written with loops. For example, if we expand the definition of `fib(5)` by hand, we see the following:

```
fib(5) = fib(4) + fib(3)
       = (fib(3) + fib(2)) + (fib(2) + fib(1))
       = ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
       = (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0)))
         + ((fib(1) + fib(0)) + fib(1))
       = (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
       = 5
```

The problem is that we calculate the same number many times, which makes for very slow code. If we test the following, we can see it takes longer and longer to print each result:

```
for x in range(35):
    print(fib(x))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
```

The last number here takes several seconds to print.

We can get better performance by writing a smarter version of `fib` that uses *tail recursion*:

```
def fib_rec(n,a=0,b=1):
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fib_rec(n-1,b,a+b)
```

The idea here is that `n` is a counter for the index of the Fibonacci number we are currently calculating, `a` is the value 2 steps earlier in the sequence, and `b` is the value 1 step earlier. In this way, we need only one recursive call and the code becomes much faster. For example, the following result is printed instantaneously:

```
print(fib_rec(100))
```

```
354224848179261915075
```

In calculating `fib_rec(5)` (that is, `fib_rec(5,0,1)`), the function works like so:

```
fib_rec(5) = fib_rec(5,0,1)
           = fib_rec(4,1,1)
           = fib_rec(3,1,2)
           = fib_rec(2,2,3)
           = fib_rec(1,3,5)
           = 5
```

We have thus avoided the "combinatorial explosion", making `fib_rec` much faster than `fib`. We can even try with larger numbers:

```
print(fib_rec(1000))
```

```
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

If we want to try *very* large numbers, we need to change a setting first. Python has a built-in limit for how many recursive calls are permitted:

```
import sys

sys.setrecursionlimit(100000)

print(fib_rec(10000))
```

The calculation of this very large number completes in no time, even though the result has more than `2000` digits. So recursive functions can be very fast if we write them properly.

However, Python is not always so fast when it comes to executing functions written using recursion. It is often better to write functions using `for` or `while` if one needs fast code. Other languages do more to optimize recursive functions so that they become tail recursive, but in Python we must do this by hand.

8.1.7 Discussion of recursion

- Principle: break up a problem instance into smaller instances and solve these.
- Recursion is "getting help from a (invisible) friend".
- Recursion is an efficient algorithmic technique.
  - It is, however, not Python's strongest side. Many programming languages internally convert recursive code to while loops (possible when the recursive call is the last step in the function, "tail recursion"), which makes for faster and more memory-efficient execution.
- It is a frequently useful technique in programming with data structures (for example search trees).

Principles of function calls:

- When a function call is made, the call parameters are stored in an area of memory called the *stack*.
- When the function begins executing, the first thing it does it grab the parameters from the stack and store them in local variables.
- When `return` is executed, the return value is placed on the stack.
- The code that made the function call can then grab the result from the stack when the function is finished.

These principles make recursion calls simple from the computer's perspective; they are just like ordinary function calls.

8.2 Anonymous and higher-order functions

When we've talked about variables and values, we have thus far primarily talked about integers, floats, strings, lists, and so on. There are good reasons to add *functions* to our collection of values we work with, which entails the possibility of creating new functions when we need them, using functions as arguments, returning functions, and applying operations to functions. When a programming language has these capabilities, we say that functions are treated as *first class citizens* (or *first class objects* or *first class functions*).

Old languages in the same tradition as Python, so-called *imperative languages*, were bad at handling functions in this way, but functions have been first class citizens in Lisp and other *functional programming* languages since the 50's. Today this is a common feature supported by languages of all kinds, among them Python.

8.2.1 Anonymous functions

The concept of *anonymous functions* via *lambda-expressions* appears in many languages today. We say that the functions are *anonymous* because we do not give them a name, simply define them. Consider this code snippet:

```
double = lambda x: 2 * x

print(double(3))
```

```
6
```

We assign to `double` a function that takes an argument, `x`, and calculates `2 * x`. We can write a function that takes two arguments like so:

```
f = lambda x, y: 2 * x + y

print(f(3,4))
```

```
10
```

This is equivalent to the following definition:

```
def f(x,y):
   return 2 * x + y
```

One should not however see lambda-expressions simply as an alternative to ordinary function definitions. They are rather *complementary* to **def**; their foremost application is for constructing small, temporary functions for use with *higher-order functions*.

The reason anonymous functions are introduced with **lambda** is that the concept originates with the so-called λ-calculus. This is a model of computation invented by the logician Alonzo Church in the 1930's (http://en.wikipedia.org/wiki/Lambda_calculus). In this model, all expressions are built up from functions and variables (so for example, even integers are represented as functions); it is in fact possible to represent all computations possible on a Turing machine using only lambda-expressions (and vice versa).

Remark: A Turing machine is a mathematical model/concept of computation, used for understanding algorithms and their limits. Computers that are used today are not exactly built according to the Turing machine concept. However, one can still speak of rules such as those making up a programming language being *Turing complete*, meaning that they suffice to represent any computation possible on a Turing machine.

8.2.2 Higher-order functions

A *higher-order function* is a function that takes a function as an argument. This is a deeper concept than it might seem at first: higher-order functions are an effective way of generalizing functionality. By identifying common patterns in calculations and encapsulating them in higher-order functions, we can simplify, shorten, and possibly even speed up our code. To get a sense of what we mean, consider these examples of computations we might want to do:

1. Take a list of numbers and return a list of their squares;
2. Take a list of non-empty lists and return a list with the first element from each list.
3. Take a list of strings and calculate their lengths.

These three computation have something in common: we want to do something to every element of a list. It is not hard to write a **for**-loop to do this, but the function `map` is more compact, is recognizable to other programmers, and is already available in Python (see also exercise 7 in chapter 6). The first argument to `map` is a function that is then applied to all the elements in a list. This function can either be defined using **lambda** or be any named (built-in or **def**ined) Python function.

The exercises above are then easy to carry out with short and simple expressions:

```
# 1
print(list(map(lambda x: x**2, [1,2,3,4])))

# 2
```

```
print(list(map(lambda t: t[0], [[1,1], [0,2,1], [3,1,53,2]])))

# 3
print(list(map(len, ["hej","hopp"])))
```

```
[1, 4, 9, 16]
[1, 0, 3]
[3, 4]
```

Obs: `map` returns a generator, so we must use `list` to get a list out.

8.2.2.1 Not just for lists

Notice also that `map` and similar functions are not limited to lists, but can be applied to any *iterable* object in Python. This includes for example strings, tuples, and dictionaries, but can also include objects given iterable functionality by the programmer themself.

To calculate the length of every line in the file `people.txt` with the contents:

```
Anders
Christian
Kristoffer
Lars
```

we can run:

```
print(list(map(len, open('people.txt'))))
```

```
[7, 10, 11, 5]
```

Note that the newline symbol `\n` is counted in the length of each line.

8.2.2.2 Higher order function: `filter`

A second pattern we often see in algorithms is the singling out of elements from a collection that satisfy some criterion. For this, there is the function `filter`. It takes a function—let's call it `pred`—and an iterable data structure and returns a generator (that you can iterate over or coerce to a list with `list`) with the elements for which `pred` returns `True` (or any value that `bool` coerces to `True`). For example:

```
print(list(filter(lambda x: x % 2, range(10))))

print(list(filter(lambda x: x, [[1], [2,3], [], [], [4,5,6]])))
```

```
[1, 3, 5, 7, 9]
[[1], [2, 3], [4, 5, 6]]
```

8.2.3 Your own higher order functions

The functions `map` and `filter` have no special status in Python; they are simply good tools that many find useful. It is not so hard to write these functions yourself. On the other hand, you might find your own generalizable patterns in your programs.

For example, consider taking the sum or product of elements in a list, which we might do using slicing and recursion:

```
def sum_rec(xs):
    if xs == []:
        return 0
    else:
        return (xs[0] + sum_rec(xs[1:]))

def prod_rec(xs):
    if xs == []:
        return 1
    else:
        return (xs[0] * prod_rec(xs[1:]))

print(sum_rec([1,2,3,4]))
print(prod_rec([1,2,3,4]))
```

```
10
24
```

The only difference between these is that one uses `+` and `0`, while the other uses `*` and `1`. As such, we can generalize to a function that takes a (binary) function `f` and an element `x`:

```
def foldr(xs,f,x):
    if xs == []:
        return x
    else:
        return f(xs[0],foldr(xs[1:],f,x))

sum = lambda xs: foldr(xs,lambda x,y: x + y,0)
prod = lambda xs: foldr(xs,lambda x,y: x * y,1)

print(sum([1,2,3,4]))
print(prod([1,2,3,4]))
```

```
10
24
```

We can now also easily write a function that uses exponentiation as its operation:

```
exps = lambda xs: foldr(xs,lambda x,y: x ** y,1)

print(exps([2,3,4]))
```

```
2417851639229258349412352
```

We can even write a function for combining a list of lists by concatenation:

```
join = lambda xs: foldr(xs,lambda x,y: x + y,[])

print(join([[1], [2,3], [], [], [4,5,6]]))
```

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

Higher order functions are, in short, a powerful way to write very general code and minimize code

duplication!

Obs: there is a function called `reduce` that functions exactly like our `foldr`, just with the arguments in a different order. It belongs to the library `functools` and can be imported like so:

```
import functools

print(functools.reduce(lambda s, x: s+x, [1,2,3], 0))
```

```
6
```

8.3 Exercises

1. Write a recursive function `rec_prod_m(n,m)` that takes two numbers `n` and `m` and calculates `n * (n-m) * (n-2*m) * ...`, continuing until `n-k*m` becomes negative.

   Tip: you can come up with this function by modifying the function `fac` a bit.

2. What happens if we make a change to `return 0` in the base case of `prod`? Why?

3. Define five functions `f1, ..., f5` using `lambda` that perform addition, subtraction, multiplication, division, and exponentiation respectively. For example, `f1(4,7)` should return 11. Use these functions to write the expressions

   ```
   (6*(2+3)/5)**2
   (10 - 2**3)*5
   ```

4. Use `map` and `lambda` to write a function `get_evens(xs)` that takes a list of integers `xs` and converts even numbers to `True` and odd numbers to `False`.

5. Study the code below. How many calls to `f` are made in total if we call `f` with 3? How many if we call it with 4? Why?

   ```
   def f(n):
       if n <= 1:
           return 1
       else:
           return f(n-1) + f(n-2) + f(n-3)
   ```

6. Rewrite the function `collatz(n)` (see chapter 7) as a recursive function.

7. Use the function `foldr` to write a function `joinstrings` that concatenates a list of strings.

8. Write `fac` and `fib` using loops instead of recursion.

9. What goes wrong if we don't copy `xs` in `quicksort`?

10. † This is (the top of) Pascal's triangle:

    ```
        1
       1 1
      1 2 1
     1 3 3 1
    1 4 6 4 1
    ```

and so on… Each number in the triangle is the sum of the two numbers above it; for example, the leftmost 4 is the sum for 1 and 3. Write a function `pascal(n)` that returns the nth row. It should work like so:

```
for i in range(1,8):
   print(pascal(i))
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
```

Chapter 9

Object orientation 1: Classes

9.1 Object orientation

Object-oriented programming (abbreviated OO) is a programming paradigm or programming style in which data and data structures are organized together with the functions that apply to them. In Python, everything is in principle implemented as an object; this applies to integers, floats, strings, lists, and dictionaries as well as functions. What do we mean by tying data structures together with the "functions that apply to them"? Take for example lists:

```
l = [2,7,2]
print(l.append(1))
```

```
[2, 7, 2, 1]
```

In object-oriented terms, we have in the first row created an object `l` of type `list`. On the second row, we have used the function `append`, which is *specific to* the datatype `list`. In object-oriented programming, functions that are tied to an object are called *methods*. They are called using dot notation, `object.method(<parameters>)`. We will soon take a look at the anatomy of a method. Here is another example with strings:

```
s = "Hej"
print(s.lower())
```

```
hej
```

Here we create an object `s` of type `str` and use the method `lower`, specific to the datatype `str`.

9.1.1 Why object orientation?

Object-oriented programming is well-suited to the task when we want to:

- combine together a lot of interrelated data (as belonging to the same object);
- be able to define our own datatypes.

For example, say that we want to keep the data associated to a car (perhaps for a computer game) in the memory of a program. This might include tire size, color, number of doors, current gas level, speed,

angle of the front wheels, and more. It would be convenient to have all the data for a given car "in one place", so that given a car-object b, we could access and change all its *attributes* like so:

```
b.tire_width()
b.color()
b.increase_speed(2)
```

It is natural to think of objects as "physical things" (for example cars), but there are also natural objects that we cannot get our hands on. We will soon see how we can implement courses (for example DA2004) as objects, organizing student information, results, and so on. Finally, we will also see how we can implement our own data structures to represent sets of integers.

Two benefits of writing object-oriented code are that it:

1. encourages good structure by organizing data and related functionality in the same place;
2. is good for abstraction, as it makes it easier to hide implementation details.

9.1.2 How are objects created?

So, how do we write object-oriented code? To create our own objects of a given kind (for example, a car or a course), we first need to specify a description of the objects' data, methods, and general structure. Such a template is called a *class* in object-oriented terminology. A class packages data and functionality; you can interpret "class" as short for "classification". A class is comparable with a type, and once we define a class, we can create new *instances* of that class. For example, `[1,2,3]` is an instance of the class `list`. Every instance can have different *attributes* (variables) coupled to it for representing its current state. Instances can also have *methods* (functions defined in the class) for modifying their state (for example `xs.pop()`).

In this course, we have already used many examples of classes: integers, lists, dictionaries, and more. Now we will learn to write our own.

Here is a minimal example, an empty class:

```
class Course:
    pass
```

The convention is that class names begin with capital letters. Once we have declared a class, we can *instantiate* it, which creates an *object* of that class. We can later tie different attributes to the object.

```
k = Course()       # An object is created, an instance of the class Course
k.code = "DA2004"  # An attribute is assigned to the object
k2 = Course()      # A new object is created
print(k.code)
```

```
DA2004
```

If we instead try to access the code attribute of the object k2, we will get an error message: we have not ascribed any such attribute to k2:

```
print(k2.code) # This object does not have an attribute "code"
```

```
AttributeError: 'Course' object has no attribute 'code'
```

Here, code is a variable that belongs to the object/instance k. We will often use the term "instance attribute" for this type of variable.

9.1.3 Methods

A *method* is a function defined in a class. In the specification of `Course` below, each of the `def`-declarations defines a method. Methods follow the same conventions as ordinary functions: lowercase letters and words separated with _.

```
class Course:

    def __init__(self, code, name, year=2020):       # Constructor
        self.participants = []
        self.code = code
        self.name = name
        self.year = year

    def number_participants(self):
        return len(self.participants)

    def new_participant(self, name, email):
        self.participants.append((name, email))
```

The first method, `__init__`, has a special status and is called a *constructor*. A constructor initiates a new object. It is called only upon instantiation, that is, when we create an object. In general, special methods used by the Python system have names of the form `__fkn__`. The constructor is always called `__init__`, so when we run

```
da2004 = Course("DA2004", "Programmeringsteknik")
```

the constructor method `Course.__init__` is called and the result is stored in the variable `da2004`. The object in `da2004` is then an *instance* of the `Course` class. It is only `__init__` that is called upon instantiation. The other methods are accessible for use when we call them.

We can modify an object by calling its methods, which we do using dot notation (exactly as we use `xs.pop()` and other list methods to modify lists).

```
da2004.new_participant('Johan Jansson', 'j.jansson@exempel.se')
print(da2004.number_participants())
```

```
1
```

9.1.4 Self

What is the expression `self` that appears before all attributes and as parameters to methods? The variable `self` in the `Course` class represents the object from which the method was called. (It is not necessary to call this argument `self`, but it is convention among Python programmers to do so.) In the example above, the argument `self` in `new_participant` is instantiated with the object `da2004`. This can be hard to grasp when you are first learning about classes and object orientation. For illustration, you can imagine that the line

```
da2004.new_participant('Johan Jansson', 'j.jansson@exempel.se')
```

really corresponds to:

```
__course_object__.new_participant(da2004, 'Johan Jansson', 'j.jansson@exempel.se')
```

Obs: This is not correct syntax in Python, just an illustration of the concept that the method `new_participant` takes the object `da2004` as a parameter so that it can modify it.

9.1.5 Class and instance attributes

Attributes are another way of name for variables that are specific to an object or class. There are two different types of attributes: instance *attributes* and *class attributes*. In general, *instance attributes* are used to hold data specific to an instance (that is, specific to an object), while *class attributes* are used for attributes and methods shared by all instances of the class:

```
class Course:

    university = 'Stockholm University'  # class attribute shared by all instances
                                         # (1) Occurs outside of constructor
                                         # (2) Has no 'self', i.e., specific object,
                                         # associated to it

    def __init__(self, code, name, year=2020):
        # instance attributes below, unique to each instance
        self.participants = []
        self.code = code
        self.name = name
        self.year = year
```

An example use:

```
da2004 = Course('DA2004', 'Programmeringsteknik')
da2005 = Course('DA2005', 'Programmeringsteknik (online version)', year = 2021)

print(da2004.university)                          # shared by all courses
print(da2005.university)                          # shared by all courses
print(da2004.code, da2004.year, da2004.name)      # unique to da2004
print(da2005.code, da2005.year, da2005.name)      # unique to da2005
print(Course.university)                          # you can also access a class
                                                  # attribute from the class itself
```

```
Stockholm University
Stockholm University
DA2004 2020 Programmeringsteknik
DA2005 2021 Programmeringsteknik (online version)
Stockholm University
```

We can also overwrite existing attributes like so:

```
da2004.year = 2021
print(da2004.year)
```

```
2021
```

In the same way we overwrote the instance attribute `year` in `da2004` but not in `da2005`, we can also overwrite methods in an object. Remember that functions have no special status in Python; they are just like other objects.

Warning: Mutable data can have surprising effects when used in class attributes!

For example, if we have defined the list `participants` as a class variable, it will be shared by all courses:

```
class Course:

    university = 'Stockholm University'   # class attribute shared by all instances
    participants = []                     # class attribute shared by all instances

    def __init__(self, code, name, year=2020):
        # instance attributes below, unique to each instance
        self.code = code
        self.name = name
        self.year = year

    def new_participant(self, name, email):
        self.participants.append((name, email))

d = Course('da2004', 'Programmeringsteknik')
e = Course('da2005', 'Programmeringsteknik (online version)')

d.new_participant('Johan Jansson', 'j.jansson@exempel.se')
print(d.participants)                 # shared by all courses
print(e.participants)                 # shared by all courses
```

```
[('Johan Jansson', 'j.jansson@exempel.se')]
[('Johan Jansson', 'j.jansson@exempel.se')]
```

9.1.6 Private and public methods and attributes

In object-oriented programming, we often distinguish between *private* and *public* methods and attributes. The public methods constitute the *interface* for a class and are the methods that users of the class will use. The private methods and attributes are for internal use. In many programming languages, there is support for controlling access to methods and attributes so that one doesn't rely on something outside the program's interface by mistake. In this way, one achieves *abstraction* with the help of object orientation: https://en.wikipedia.org/wiki/Abstraction_(computer_science)#Abstraction_in_object_oriented_programming.

Python focuses on simplicity and instead has a convention: identifiers for attributes/methods that begin with an underscore _ are *regarded as private*. They should be therefore only be used internally, although the language itself does not prevent external use.

Below, we have added a private method `_check_duplicate` in our course example. It is common that students register themselves with different email-addresses. This method checks whether an identical name occurs in the participant list when we add a participant and prints a warning in that case. We can then take appropriate action, for example with some method `remove_participant` for removing duplicated participants.

```
import warnings  # included in Python's standard library

class Course:
```

```
    def __init__(self, code, name, year=2020):       # Constructor
        self.participants = []
        self.code = code
        self.name = name
        self.year = year

    def _check_duplicate(self, name, email):
        for p in self.participants:
            if name == p[0]: # check if identical name
                # raise warning and print message if identical name:
                # uses string formatting, see e.g.:
                # https://www.w3schools.com/python/ref_string_format.asp
                warnings.warn(
                    "Warning: Name already exists under entry: {0}".format(p))

    def number_participants(self):
        return len(self.participants)

    def new_participant(self, name, email):
        self._check_duplicate(self, name, email)
        self.participants.append((name, email))
```

Now let's see what happens if we register students with the same name:

```
da2004 = Course('DA2004', 'Programmeringsteknik')
da2004.new_participant('Johan Jansson', 'j.jansson@exempel.se')
da2004.new_participant('Johan Jansson', 'j.jansson@gmail.com')
```

The third line will generate a "warning" printed like so:

```
UserWarning: Warning: Name already exists under entry:
    ('Johan Jansson', 'j.jansson@exempel.se')
```

In this example, the method `_check_duplicate` is used for internal purposes in the class `Course`. It is not expected that the user of the class will call this method: the method is not part of the *interface* and so does not increase the "size" (complexity of use) of our class. At the same time, it performs a useful function "under the hood" when we add a participant. Note that even if we mark that a method is private with _, it is still possible to call it explicitly like so:

```
da2004._check_duplicate('Johan Jansson', 'j.jansson@exempel.se')
```

9.1.7 Vocabulary list

- *Object*: an element of some type with various attributes and methods.
- *Class*: a type paired with attributes and methods (for example `list`, `dict`, `Course` from the example above, etc.).
- *Instantiation*: when an object is created from a given class.
- *Method*: a function **def**ined in a class (for example `pop`, `insert`).
- *Constructor*: the method for instantiating an object of the class (for example `dict()`). Every class has a constructor `__init__` that is either implicitly or explicitly defined.

- *Class attribute*: a variable defined outside of the constructor or methods of a class, shared by all instances of the class.
- *Instance attribute*: a variable defined in the constructor for the class or set with `self` in the class. Belongs to a specific instance of the class.

9.2 Modularity via object orientation

In the `Course` example above, we used a list `participants` of tuples to represent the list of participants in a course. A more object-oriented solution would be to have a separate class for participants:

```
class Participant:

    def __init__(self, name, email):
        self._name = name
        self._email = email

    def name(self):
        return self._name

    def email(self):
        return self._email
```

We must now modify the methods of `Course` so that they work with participants represented with `Participant` instead of tuples of strings. For example:

```
class Course:

    # old code unchanged, except for _check_duplicate which
    # has to be modified to take a Participant

    def new_participant(self, p):
        self._check_duplicate(self, p)
        self.participants.append(p)
```

We must now create participants like so:

```
d = Participant("Johan Jansson", 'j.jansson@exempel.se')
da2004.new_participant(d)
da2004.new_participant(Participant("Foo Bar", "foo@bar.com"))
```

It might seem strange to rewrite our code like this, but the advantage is that it will be much simpler to change this code in the future. For example, suppose we want to separate the first and last names of participates. It's then enough to write:

```
class Participant:

    def __init__(self, fname, lname, email):       # Changed!
        self._fname = fname                         # Changed!
        self._lname = lname                         # Changed!
        self._email = email
```

```
    def name(self):
        return self._fname + " " + self._lname     # Changed!

    def email(self):
        return self._email
```

The code for `Course` doesn't need to be changed all, because it only uses `Participants`'s interface. In this way, object orientation produces more modular code: every class can be seen as a little module in itself, and internal modifications do not effect other code in the program so long as the interface is unchanged.

9.3 One more example of OO: sets of numbers

Python has a type for *sets*. These are like dictionaries, but the keys are not associated with any values. For documentation, se:

https://docs.python.org/3/tutorial/datastructures.html#sets

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

The syntax for creating a set (`set`) is similar to that for lists, but uses `{}`. Some examples:

```
basket = set(['apple', 'banana', 'orange', 'pear'])
# the above is equivalent to
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)
print('orange' in basket)
print('crabgrass' in basket)

a = set([1,2,5])
b = set([3,5])
print(a)
print(a - b)  # difference
print(a | b)  # union
print(a & b)  # intersection
print(a ^ b)  # numbers in a or b but not both
```

```
{'pear', 'apple', 'orange', 'banana'}
True
False
{1, 2, 5}
{1, 2}
{1, 2, 3, 5}
{5}
{1, 2, 3}
```

9.3.1 Our own implementation

Here we consider a larger example taken from the course book (Introduction to Computation and Programming using Python by John V Guttag, page 111), where we implement our own set-datatype

`IntSet` using lists. We can represent sets of integers as lists with the invariant that they contain no duplicates.

```
class IntSet:
    """An IntSet is a set of integers"""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self,e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)

    def member(self,e):
        """Assumes e is an integer.
           Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def delete(self,e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def get_members(self):
        """Returns a list containing the elements of self.
           Nothing can be assumed about the order of the elements"""
        return self.vals
```

We maintain the *invariant* that instances of `IntSet` contain no duplicates in all our functions.

A few tests:

```
x = IntSet()
print(x)
x.insert(2)
x.insert(3)
x.insert(2)
print(x)
print(x.member(5))
print(x.member(3))
x.delete(2)
print(x.get_members())
```

```
<__main__.IntSet object at 0x7febb7002040>
<__main__.IntSet object at 0x7febb7002040>
False
```

```
True
[3]
```

The two first `print` calls print information on the object: where it is defined and which memory address it is stored at. If we run this code snippet again, the object will probably be in a different place in the memory.

We can also add methods `__str__(self)` and `__len__(self)` which make it so that we can use Python's syntax `str()` and `len()`. See https://docs.python.org/3/reference/datamodel.html#object.__str__ and https://docs.python.org/3/reference/datamodel.html#object.__len__. We can thereby make it so that sets are printed more nicely than they are above. Here is an example of how a `__str__` method might look:

```
    def __str__(self):
        result = ''
        for e in self.vals:
            result += str(e) + ','
        return '{' + result[:-1] + '}'
```

Now this code is called when we convert a set to a string (such as when we `print` it):

```
x = IntSet()
print(x)
x.insert(2)
x.insert(3)
x.insert(2)
print(x)
```

```
{}
{2,3}
```

Suppose we want to be able to compare `IntSet` objects using <=, with `x <= y` taken to mean "x is a subset of y". We can do this by adding another method to the `IntSet` class:

```
    def __le__(self,other):
        for x in self.vals:
            if not other.member(x):
                return False
        return True
```

Some tests:

```
print(x <= x)
empty = IntSet()
print(empty <= x)
print(x <= empty)
```

```
True
True
False
```

In this way, we can extend the other standard Python operations (+, <, =, …) to our own classes; this is called *operator overloading*. We can also overload other operators such as **`in`** and **`and`**. For documentation of the method names used for each standard operation, see https://docs.python.org/3/library/operator.

html. For example, we can renme the method `member` in our `IntSet` class to `__contains__`. We can then use the operator `in` and write `5 in x` instead of `x.member(5)`.

9.4 Exercises

1. Implement the method `__str__` so that we can use `str()` to print an appropriate string in `Participant` and `Course`. It is up to you how the string looks, but the name and email should appear in `Participant`, likewise the course code and name in `Course`.

2. Add the attribute `teacher` of type `bool` to the class `Participant`. Print the teacher's name in some special way, so that one can see who it is from the `Participant` list.

3. Add methods so that the operators == and != are defined for the class `IntSet`. The operator == should return `True` if the sets contain the same elements, otherwise `False`. The operator != is the negation (logical opposite) of ==.

   Tip: read the documentation https://docs.python.org/3/library/operator.html.

4. Implement union, difference, and intersection methods for `IntSet`. You can implement intersection however you like, but union should be implemented by overloading + and difference by overloading -. Each method should return a new `IntSet` object. For example:

```
x = IntSet()
x.insert(2)
x.insert(3)
x.insert(1)

y = IntSet()
y.insert(2)
y.insert(3)
y.insert(4)
print(x == y, y != x)
print(x.intersection(y))
print(x + y, y - x)
```

```
False True
{2,3}
{4,2,3,1} {4}
```

5. Implement a class `Dog` with functionality making the followng possible.

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.add_trick('shake hands')
print(d.name, d.tricks)
print(e.name, e.tricks)
```

```
Fido ['roll over', 'shake hands']
Buddy ['play dead']
```

6. Implement the methods `__str__` and `__gt__` in the class `Dog` so that we can write `str(d)` and

compare `d > e` (with > based on how many tricks they know). For example, given the code above, we should have:

```
print(str(d))
print(d > e)
```

```
Fido knows: roll over, shake hands
True
```

Chapter 10

Object orientation 2: Inheritance

Inheritance, a structured way of reusing code and functionality in already-defined classes by *adding* or *changing* parts of a class, is a central principle in object orientation (OO). Often, there is an existing class that almost has the attributes and methods you want, and it is then a good idea to *inherit* these from that class.

As a simple example, suppose we have a class `A` whose functionality we want to reuse with certain parts changed:

```
class A:
    """This is a base class
    """
    class_integer = 10

    def __init__(self):
        self.vals = [5]

    def power_sum(self):
        return sum([i**2 for i in self.vals]) + A.class_integer

class B(A):
    """This is a subclass
    """
    def __init__(self, x, y):
        self.vals = [x, y]

    def print_result(self):
        print(self.power_sum())
```

Here `A` is a *base class* for `B` and `B` is a *subclass* of `A`. We can also say that `A` is a *superclass* of `B`. In several programming languages, the terms *parent class* and *child class* are used for base classes and subclasses respectively.

Nose that the **`class`** definition of `B` takes `A` as an argument, which indicates that `B` inherits from `A`. In this situation, *inherit* means that `A`'s functionality will *automatically* be implemented in `B`, so long as we

do not actively redefine anything. In this case, the class constructor for `B` (`__init__`) has been redefined to take *two* arguments (`x` and `y`) instead of *one*. Besides this, `B` reuses the functionality from `A`; in the example above, there is no need to redefine `power_sum` in `B`.

The use of these classes is now fairly straightforward:

```
a = A()
print(a.power_sum())   # method defined in A

b = B(10, 20)
print(b.power_sum())   # calling method in B inherited from A
b.print_result()       # uses method specific to B
```

```
35
510
510
```

Obs: in Python, instance attributes are not inherited automatically! This is not the case in many other programming languages.

```
class C(A):
    def __init__(self, y):
        self.vals.append(y)

c = C(10)
```

```
AttributeError: 'C' object has no attribute 'vals'
```

To ensure that `C` also inherits the instances attributes of `A`, we can change the constructor slightly and use `super` so that the initializationion function from the base class is also called:

```
class C(A):
    def __init__(self, y):
        super().__init__()  # call constuctor of super class
        self.vals.append(y)

class D(B):
    def __init__(self, y):
        super().__init__(5, 10) # call constuctor of super class
        self.vals.append(y)

c = C(10)
d = D(10)
print(c.power_sum())
d.print_result()
c.print_result()       # raises an Error
```

```
135
235
AttributeError: 'C' object has no attribute 'print_result'
```

Here, we get an error at the end because `print_result()` is not defined for `C`, which inherits from `A`. Class `D`, on the other hand, inherits the method from `B`.

10.1 Class diagrams

A class diagram visualizes the relationships between classes. It contains no information on specific objects. Using a class diagram, we can indicate that a class inherits from another (the relation "is-a") and that a class contains objects from another class ("aggregation" and "composition").

There are many ways to draw class diagrams; often informal notation is used. There are professional tools, often built on the industry standard UML, that generate class diagrams from code. For example, fig. 10.1 was generated using the Python library `pyreverse` (which is a part of the code analysis tool `pylint`).

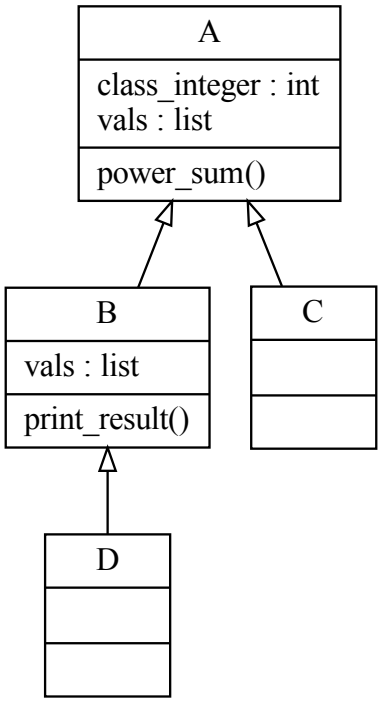

Figure 10.1: Class diagram for `A`, `B`, `C` and `D`

An *object diagram* is similar to a class diagram, but displays the *state* in a program at a *certain* point in time: for example, an object diagram of the above code would show that `a.vals` contains `[5]`, `b.vals` contains `[10, 20]`, `c.vals` contains `[5, 10]`, and `d.vals` contains `[5, 10, 10]`. The class diagram in fig. 10.1 displays the relations between classes *independent* of any particular point in time.

10.2 The substitution principle

It is admittedly difficult to choose classes well. Many books have been written on the subject with different strategies for deciding what class structure one should have. This can quickly become very complicated. A good rule of thumb, known as Liskov's substitution principle after MIT professor and Turing prize winner Barbara Liskov, is:

- Any property satisfied by objects in one class should also hold of objects in its subclasses.

Keeping to this principle helps to ensure *good design*. It might seem like an obvious principle, but it is easy to find yourself wanting to remove functionality in subclasses, and there are many examples that go against the principle.

10.3 Functions that give information on classes

Two useful built-in functions for dealing with classes are `isinstance` and `issubclass`. The first function tests if a given *object* is an *instance* of a class, while the second tests if one *class* is a subclass of another *class*, i.e. if one class inherits from another.

If we start with the class definitions (`A`, `B`, `C`, `D`) and instances (`a`, `b`, `c`, `d`) from above, then we can for example write:

```
isinstance(a, A) # a is an instance of the class A
isinstance(d, A) # d is an instance a class that inherits from A
isinstance(d, C) # d is not an instance of a class that inherits from C

issubclass(A, A)
issubclass(D, A)
issubclass(D, C)
```

```
True
True
False

True
True
False
```

Notice that these functions do not only check if one class is *directly* the base class of another, but looks farther back in the chain of inheritance.

In programming languages that support multiple inheritance (such as Python—more on that in chapter 11), such a "chain of inheritance" is called *method resolution order*, usually abbreviated MRO. MRO specifies the order in which a given class searches its base classes (if there are multiple) for methods. In Python, classes support a method `.mro` that shows the class's MRO. For example:

```
A.mro()
D.mro()
```

```
[__main__.A, object]
[__main__.D, __main__.B, __main__.A, object]
```

Here, we can see that `D` inherits from `B`—the first class definition after `D` where Python looks to see if

a method is defined for `D`—and then from `A`. The final element in the list, `object`, is the *fundamental foundation* for types in Python; it is therefore the last in the list, and both `A` and `D` have it as a part of their MRO's. One often hears that *"Everything is an object in Python"*:

```
isinstance(a, object)
isinstance(print, object) # function print is an object
issubclass(str, object)
issubclass(int, object)
```

```
True
True
True
True
```

10.4 Example: Geometric figures

As an example of inheritance, we now implement classes representing geometric objects (figures) that collect common/typical operations that one might want to carry out with such figures. Our intention with this example is to:

- illustrate the value of inheritance;
- illustrate *method overriding/overloading*;
- illustrate the value of *encapsulation*, the packaging of data attributes with the methods that act on that data;
- draw class diagrams.

We want to write code that implements Python objects representing different geometric figures composed of (straight) sides. To follow Liskov's substitution principle, we'll start with a base class that represents the most general case: any polygon. The subclasses will then add data and functionality appropriate to more specific types of geometric figure.

We begin by defining the base class `Polygon`, which is meant to cover the general case within certain limits.

```
class Polygon:
    def __init__(self, n_sides, n_unique_sides=None):
        self.n_sides = n_sides
        if n_unique_sides:
            self.n_unique_sides = n_unique_sides
        else:
            self.n_unique_sides = n_sides
        if self.n_sides % self.n_unique_sides:
            raise ValueError(
                "The total number of sides must be an integer"
                " multiple of the number of unique sides")
        self.sides = None
        self.__name__ = 'Polygon'

    # "Private" methods
    def _check_sides(self):
        if (not self.sides or
```

```
            not all(map(lambda x: x > 0, self.sides))):
            print("Invalid side lengths")
            return False
        else:
            return True

    def _get_side(self, side_no):
        s = float(input("Enter side " + str(side_no + 1) + ": "))
        return s

    # "Public" methods
    def input_sides(self):
        print("Please enter the " + str(self.n_unique_sides) +
              " unique side lengths for a " + self.__name__)
        unique_sides = [self._get_side(i) for i in
                        range(self.n_unique_sides)]
        self.sides = unique_sides * (self.n_sides
                                     // self.n_unique_sides)
        if not self._check_sides():
            print("Please provide positive side lengths")
            self.input_sides()

    def display_sides(self):
        if self._check_sides():
            for i in range(self.n_sides):
                print("Length of side", i + 1, ":", self.sides[i])
```

The class constructor takes two arguments, a required argument `n_sides` specifying how many sides the polygon has and a keyword argument `n_unique_sides` with the default value `None`. We want to make it possible to specify that a polygon has a *number* of sides `n_sides`, but only a certain number of unique side lengths; for example, `n_unique_sides=1` should mean that all sides of the polygon have the same length.

In the constructor, we set the instance attributes to the values provided in the parameters. If `n_unique_sides` has not been specified, then the corresponding instance attribute takes the same value as `n_sides`: each side has its own length. We want, however, to limit how a user can specify the number of unique sides of a polygon: the total number of sides must be an integer multiple of the number of unique sides. We check that this condition is satisfied by checking if the condition `n_sides % n_unique_sides` evaluates to `True`, that is if there is a remainder after the division (a number other than `0`), raising a `ValueError` if this is so—recall that all numbers different from `0` are considered `True` in Python.

The main method in `Polygon` is `input_sides`, which when called first prints a message and then, using a list comprehension, reads `n_unique_sides`-many side lengths from the user. We read in the *individual* side length using the private method `_get_sides`. As we have already ensured in the constructor that `n_unique_sides` divides `n_sides`, we can multiply this list of (unique) side lengths by `n_sides // n_unique_sides` to get a list containing *all* the side lengths (so `len(sides) == n_sides`). If there are fewer unique sides than the total number of sides, then we get a list with the unique lengths repeated an appropriate number of times. Finally, we check that all the side lengths are acceptable with the private method `_check_sides`, which, using `map` and an anonymous function (`lambda`), checks that all side

lengths are positive. If this is not so, an error message is printed. The first part of the **if**-statement condition in `_check_sides` checks that `sides` is set: if `sides == None`, then the error is issued *regardless* of whether the other condition is satisfied. We check this first because `map` will raise an exception if `sides == None`, as we cannot iterate over `None`. By evaluating `not sides` first, we jump into the `if` block if `sides == None` without ever executing `map`; if `sides` is set, then we assume it is a list and so `map` can iterate over it.

The method `display_sides` prints out all side lengths if `sides` contains valid side lengths.

Now we have a base class. Using inheritance, we can easily define subclasses that represent specific polygons. For example, we can define a triangle type like so:

```
class Triangle(Polygon):
    def __init__(self):
        super().__init__(3)
        self.__name__ = 'Triangle'

    # method specific to triangles
    def get_area(self):
        a, b, c = sorted(self.sides)
        if c > (a + b):
            print("Non-valid side lengths for a triangle.")
            return
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        return (s * (s - a) * (s - b) * (s - c))**0.5
```

Here we can see that `Triangle` inherits from `Polygon`, and that in `Triangle`'s constructor we call the *base class*'s constructor with one argument (`3`). By calling the constructor explicitly with `super`, we instantiate `Polygon` with `n_sides=3,n_unique_sides=None`, and get that an instance of `Triangle` inherits both the instance attributes and methods from `Polygon`. We override the instance attribute `__name__` so that the name for instances of this class is `Triangle`. We *extend* the base class's functionality by adding a method `get_area` that calculates a triangle's area if it has valid side lengths.

We can define a class for rectangles in a similar way. Rectangles have four sides, but only two *unique* side lengths. We achieve this easily by defining a class `Rectangle` that calls the base class's constructor with arguments that specify exactly this: we have 4 sides but only 2 unique side lengths.

```
class Rectangle(Polygon):
    def __init__(self):
        super().__init__(4, n_unique_sides=2)
        self.__name__ = 'Rectangle'

    def get_area(self):
        return self.sides[0] * self.sides[1]

    # method specific for rectangles
    def get_diagonal(self):
        return (self.sides[0]**2 + self.sides[1]**2)**0.5
```

`Rectangle` also defines methods specific to rectangles: `get_area` and `get_diagonal`.

Finally, we define a special case of a rectangle, namely a `Square`. A square is a rectangle, so both the

area and diagonal can be calculated in the same way as in `Rectangle`. It is therefore natural to inherit the functionality of `Rectangle`.

```
class Square(Rectangle):
    def __init__(self):
        super(Rectangle, self).__init__(4, n_unique_sides=1)
        self.__name__ = 'Square'

    def get_corner_coordinates(self):
        s = self.sides[0]/2
        return ((-s, -s), (-s, s), (s, s), (s, -s))
```

The difference between a square and a rectangle is that there is only *one unique* side length. In this case, we *cannot* call the base class's constructor in `Square` to inherit instance attributes, because the base class `Rectangle`'s constructor does not take any arguments, instead always setting `n_sides = 4` and `n_unique_sides = 2` implicitly. We instead want to call the constructor for `Rectangle`'s base class, `Polygon`. We can do so by calling `super` with the arguments `Rectangle` and `self`, which specifies that we want to call the initialization function for the *base class of* `Rectangle` (that is, `Polygon`) but apply it to the instance `self` (of `Square`). We still inherit from `Rectangle` (for example, the methods `get_area` and `get_diagonal`), but the constructor is run as `Polygon(4, n_unique_sides=1)` in order to reduce the number of unique side lengths to *one* while still having four sides in total.

We can now try using the classes we've defined:

```
t = Triangle()
t.input_sides()
print('The area of the triangle is:', t.get_area())
t.display_sides()

r = Rectangle()
r.input_sides()
print('The area of the rectangle is:', r.get_area())
print('The diagonal of the rectangle is:', r.get_diagonal())

s = Square()
s.input_sides()
print('The area of the square is:', s.get_area())
print('The diagonal of the square is:', s.get_diagonal())
print('The coordinates of the square corners are:\n',
      s.get_corner_coordinates())
```

```
Please enter the 3 unique side lengths for a Triangle
Enter side 1 : 1
Enter side 2 : 2.5
Enter side 3 : 2
The area of the triangle is: 0.9499177595981665
Length of side 1 : 1.0
Length of side 2 : 2.5
Length of side 3 : 2.0
Please enter the 2 unique side lengths for a Rectangle
Enter side 1 : -1
```

```
Enter side 2 : 2
Invalid side lengths.
Please provide positive side lengths.
Please enter the 2 unique side lengths for a Rectangle
Enter side 1 : 1.2
Enter side 2 : 1.9
The area of the rectangle is: 2.28
The diagonal of the rectangle is: 2.247220505424423
Please enter the 1 unique side lengths for a Square
Enter side 1 : 6
The area of the square is: 36.0
The diagonal of the square is: 8.48528137423857
The coordinates of the square corners are:
 ((-3.0, -3.0), (-3.0, 3.0), (3.0, 3.0), (3.0, -3.0))
```

10.4.1 Implementing comparisons

As we did in chapter 9, we can implement comparisons for our own types (classes) using special private methods. It is especially convenient if there is a general base class where some logical comparison between objects is possible, so that we can automatically inherit comparison operators on any subclasses.

Suppose, for example, that we want to implement <, >, and == based on the area of an object. For the geometric figures `Triangle`, `Rectangle`, and `Square`, we can use `get_area` to calculate the area of an object. We can thus add the functions below to `Triangle` and `Rectangle`:

```
def __lt__(self, other):
    return self.get_area() < other.get_area()

def __gt__(self, other):
    return self.get_area() > other.get_area()

def __eq__(self, other):
    return self.get_area() == other.get_area()
```

Notice that we do *not* need to add this to `Square`, as this class inherits from `Rectangle`.

In fig. 10.2, we illustrate the class diagram for this example.

10.5 Example: Monster Go

The following slightly larger example is inspired by Pokemon Go. This example is a little contrived, but touches on important points that illustrate the value of object-oriented programming.

The goal of this example is to:

- illustrate the value of inheritance;
- illustrate the value of abstraction: hiding implementation details.

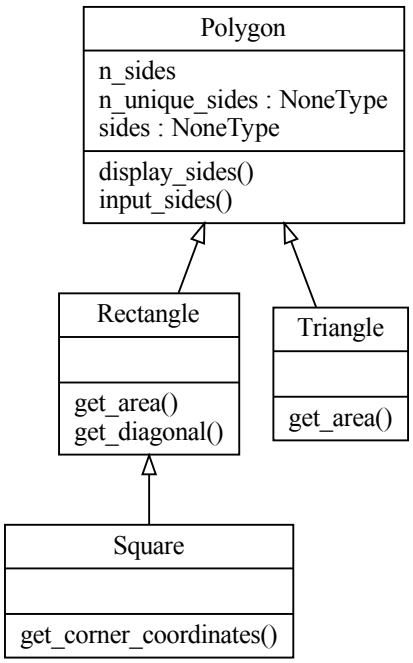

Figure 10.2: Class diagram for geometric figures.

10.5.1 A base class: ProgMon

The class `ProgMon` contains attributes and methods that will be common to all objects. We try here to follow Liskov's substitution principle.

In this example, we will make frequent use of private methods and attributes, both those used by Python that begin and end with `__` (such as `__str__`) and those that are private for our implementation and begin with `_` (such as the instance attribute `_attack` below).

```
class ProgMon():
    def __init__(self):
        self._attack = 0.0
        self._defense = 0.0
        self._caffeinated = False
        self._unit_testing = False

    def __str__(self):
        return "<ProgMon object>"

    def get_attack(self):
        if self._caffeinated:
            return 2 * self._attack
        else:
            return self._attack

    def get_defense(self):
        if self._unit_testing:
            return 2 * self._defense
        else:
            return self._defense

    def fight(self, other_progmon):
        '''
        Attack is the best form of defense!
        Returns (True, False) if self wins, (False, True) if other_progmon
        wins and (False, False) otherwise.
        '''
        if self.get_attack() > other_progmon.get_defense():
            return (True, False)
        elif other_progmon.get_attack() > self.get_defense():
            return (False, True)
        else:
            return (False, False)
```

10.5.2 Three subclasses

In the subclasses `Hacker`, `Newbie`, and `Guru`, we add specialization: data and behavior specific to each class. We do by changing the values of the private attibutes and defining `__str__`.

```
class Hacker(ProgMon):
    def __init__(self):
```

```
        super().__init__()  # Call the superclass constructor!
        self._attack = 0.5
        self._defense = 0.25

    def __str__(self):
        return "<Hacker A=" + str(self._attack) + ">"

class Newbie(ProgMon):
    def __init__(self):
        super().__init__()
        self._attack = 0.15
        self._defense = 0.1

    def __str__(self):
        return "<Newbie>"

class Guru(ProgMon):
    def __init__(self):
        super().__init__()
        self._attack = 1.0
        self._defense = 1.0
```

Try it out with:

```
h = Hacker()
n = Newbie()
g = Guru()
n.fight(h)
g.fight(h)
```

```
(False, True)
(True, False)
(True, False)
```

Obs: Common functionality in the base class helps us avoid duplication of code.

10.5.3 Adjustments

Suppose we don't like the idea that a `Guru` tries to win fights. We can implement a "Don't win, don't lose" outcome for a `Guru` that initiates a "fight" by changing our definition of `Guru`:

```
class Guru(ProgMon):
    def __init__(self):
        super().__init__()
        self._attack = 1.0
        self._defense = 1.0

    # method overloading
    def fight(self, other_progmon):
        '''
        Don't win, don't lose.
```

```
        '''
        # same type of return value as in the super class
        return (False, False)
```

Try testing this definition: `g.fight(h)` ger `(False, False)`.

Obs: we can change the behavior of objects in a subclass by redefining a method in the subclass.

10.5.4 Abstraction protects us from details

The base class `ProgMon` defines an *interface* that provides a useful abstraction: you do not need to know any *details* about the base class or its subclasses if you use the interface.

10.5.5 Example: a hacker dojo

Suppose that we create a class for a "Hacker Dojo", where ProgMons gather to train. One should be able to challenge a Hacker Dojo, with victory conditional on winning a fight against each member. A loss against a single member means losing the challenge.

```
class HackerDojo():
    def __init__(self):
        self._members = []

    def add_member(self, m):
        self._members.append(m)

    def challenge(self, pm):
        for monster in self._members:
            win, loose = pm.fight(monster)
            if loose:
                # Lost against one member, challenge failed.
                return False
        # Won against all members of the dojo
        return True
```

Testing it out:

```
dojo = HackerDojo()
dojo.add_member(h)
dojo.add_member(n)
dojo.challenge(g)
dojo.challenge(n)
```

```
True
False
```

Obs: the only thing we use in this implementation is the interface from `ProgMon`. We get the slightly different specialization for `Guru` automatically. Or rather, we need not worry about such details.

10.5.6 Changing details

One of the big benefits of object orientation (though this is not unique to OO) is abstraction. By putting together good interfaces (that is, well-chosen public methods), we "future-proof" a program: it becomes easy to change details in one part of the program without affecting other parts. This is enormously important in large projects, which might have thousands or even millions of lines of code.

Suppose we realize that `_attack` is not a good attribute. Strength should instead be determined by experience, level, and other attributes specific to the subclasses.

- What changes do we need to make, and which classes will be affected?
- Is `get_attack` a good method?

In fig. 10.3, we illustrate this example with a class diagram.

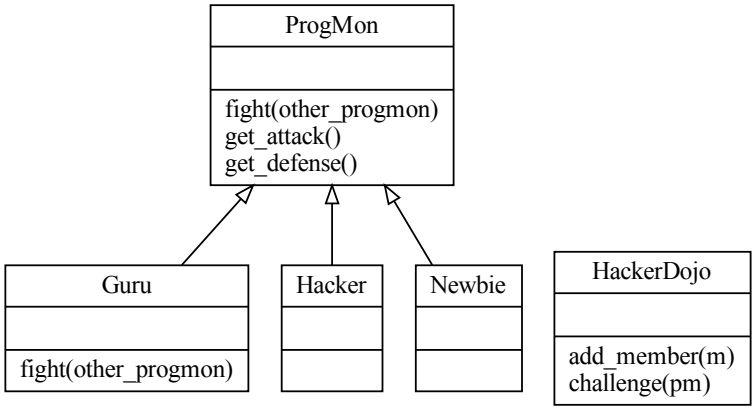

Figure 10.3: Class diagram for Monster Go.

10.6 Exercises

1. Create a class `Equilateral` that inherits from `Triangle` in the example above and overrides `get_area`.

   Tip: The area of an equilateral triangle is $\sqrt{3}a^2/4$, where a is the side length. You can import the function `sqrt` from the standard module `math`.

2. Create a class `Student` that inherits from `ProgMon` (including instance attributes), but adds an instance attribute `is_learning` of `bool` and overrides the methods `get_attack` and `get_defense` in `ProgMon`. The class `Student`'s methods `get_attack` and `get_defense` should use the corresponding `ProgMon` methods, but multiply the strength by two if the `Student` is learning, that is, if `is_learning` is `True`. You can decide what values the instance attributes `_attack` and `_defense` should be for this class yourself.

3. In exercise 5 and 6 in chapter 9, we implemented a class `Dog`. Implement a class `Tournament` which

1. ranks dogs by how many tricks they can do, and
2. prints out the result.

Tip: take inspiration from the way `HackerDojo` relates to `ProgMon`.

4. Define a base class `Vehicle` that contains the instance attributes `wheels`, `wings`, and `sound`. In addition, `Vehicle`'s method `__str__` should return a string with information on the vehicle (how many wheels, etc., it has). Create three subclasses `Car`, `Motorcycle`, and `Plane` that all inherit from `Vehicle` but each have the appropriate number of wheels/wings and make the "right" sound.

   Example usage and expected output:

```
car = Car()
mc = Motorcycle()
plane = Plane()
print(car)
print(mc)
print(plane)
```

```
A Car has:
4 wheels
and makes a AAARRRRRRRR! sound

A Motorcycle has:
2 wheels
and makes a VROOM VROOM! sound

A Plane has:
2 wings
and makes a WHOOSH! sound
```

5. Write your own string class that "supports" subtraction and division. By *subtraction* of strings, we mean that in the string being subtracted from (to the left of `-`), the first occurrence *from the right* of the string being subtracted (right of `-`) should be removed. By *division*, we mean that *all* occurrences of the string to the right of the `/` operator should be removed from the main string.

   Tip: create a subclass `MyString` of `str` and read up on which private methods define `-` and `/`.

   Exempel:

```
s1 = MyString("This is my own string!")
s2 = MyString("My, oh my, oh Oh.... oh")
subtract = MyString('my')
div = MyString('oh')
print(s1 - subtract)
print(s1 / div)
print(s2 - subtract)
print(s2 / div)
print(s1 - 'hello')
print(s1 / 'hello')
```

```
This is  own string!
This is my own string!
```

```
My, oh , oh Oh.... oh
My,  my,  Oh....
This is my own string!
This is my own string!
```

Chapter 11

Object orientation 3: more on inheritance

11.1 Different types of inheritance

In the previous chapter, we saw different examples of inheritance in Python. Different forms of inheritance have different names in object-oriented programming:

1. *single inheritance*: class `B` inherits only from class `A`;
2. *multilevel inheritence*: class `B` inherits from `A` and class `C` inherits in turn from class `B`;
3. *hierarchical inheritance*: class `A` is a superclass to `B`, `C`, `D`, ...;
4. *multiple inheritance*: class `C` inherits from *both* `A` and `B` (but these do not inherit from each other);
5. *hybrid inheritance*: a mix of two or more types of inheritance.

We saw how we can illustrate different types of inheritance using class diagrams. The previous chapter contained examples of simple, multilevel, hierarchical, and hybrid inheritance; now we look at multiple inheritance.

Not all OO programming languages permit multiple inheritance, but Python does. Consider the following example:

```
class A:
   pass    # some code

class B:
   pass    # some more code

class C(A, B):
   pass    # even more code
```

We can illustrate using a class diagram:

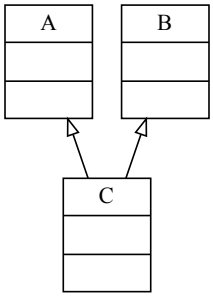

Figure 11.1: Class diagram for `A`, `B` and `C`

Here `C` has two superclasses, `A` and `B`, which do not inherit from each other in either direction. This may seem unproblematic, and you might wonder why not all languages support it. But in complete generality, it is easy for multiple inheritance to cause chaos as a result of the so-called *diamond problem*.

11.1.1 The diamond problem

Consider the following code:

```
class A:
    def f(self):
        print("f of A called")

class B(A):
    def f(self):
        print("f of B called")

class C(A):
    def f(self):
        print("f of C called")

class D(B,C):
    pass

d = D()
d.f()
```

This is a typical diamond problem. If we draw out the class diagram, we can indeed see a diamond:

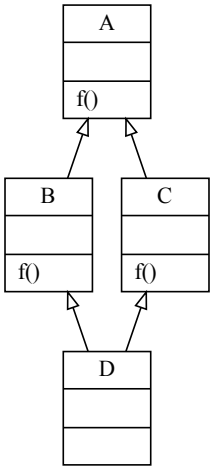

Figure 11.2: Class diagram for `A`, `B`, `C` and `D`

If we run the code above, we get

```
f of B called
```

But if we instead run the following code, where the inheritance has been changed to `D(C,B)`, we get:

```
class A:
    def f(self):
        print("f of A called")

class B(A):
    def f(self):
        print("f of B called")

class C(A):
    def f(self):
        print("f of C called")

class D(C,B):
    pass

d = D()
d.f()
```

```
f of C called
```

So the output changes when one inherits in a different order! This is an example of a "diamond problem" that can occur with multiple inheritance. You can easily imagine how code can become incomprehensible when there are many classes with many methods and attributes inheriting from each other in this way.

This type of problem is what causes chaos in code with multiple inheritance, and it is for this reason that many languages forbid it.

11.2 Example: Algebraic expressions

Now, let's look at another example of inheritance. The idea with this example is to illustrate two things:

1. how we can use classes to *structure* data;
2. how we can use recursion in combination with classes.

This combination is very powerful and helps us solve many different types of programming problems in a simple way. We'll begin by writing a program for representing and modifying algebraic expressions. These expressions can contain numbers, variables, `+`, `*`, and *matched* parentheses. Two examples of algebraic expressions are `x * y + 7` and `x * (y + 7)`.

How might we represent these in Python? A naive solution might be to simply use strings: represent `x * y + 7` by the string `"x * y + 7"`. But suppose we want to replace `x` with `3` and `y` with `2` and then calculate the value of the expression… this will be *very* complicated with strings!

Another problem with strings is handling parentheses. According to convention `*` binds more tightly than `+`, so `x * y + 7` should be interpreted as `(x * y) + 7` and not `x * (y + 7)`. If we are not careful, we'll get the wrong answer when we evaluate expressions. A much more convenient approach is to use **trees**. In this way, we can represent algebraic expressions in two dimensions instead of one!

A tree is a data structure where each element is either a leaf or a node with a number of branches, each of which leads to a *subtree*. So for example, `(x * y) + 7` can be represented unambiguously as a tree with a topmost node `+` that has `x * y` and `7` as its two subtrees. The expression `x * y` is itself represented as a tree, this time with node `*` on top and `x` and `y` as (single-leaf) subtrees. We can draw this tree like so:

```
      +
     / \
    *   7
   / \
  x   y
```

Obs: computer scientists draw trees upside-down as compared to actual trees!

The expression `x * (y + 7)`, on the other hand, is represented by the following tree:

```
     *
    / \
   x   +
      / \
     y   7
```

This type of tree is called **binary**: all nodes have either two or zero subtrees. Let's now represent algebraic expressions with trees in Python! We'll start with an empty superclass for expressions which will have 3 subclasses (one for numbers, one for variables, and one for `+`):

```
class Expr:
    pass

class Constant(Expr):
```

```
    def __init__(self, value):
        self._value = value

    def __str__(self):
        return str(self._value)

class Var(Expr):

    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class Plus(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

    def __str__(self):
        return str(self._left) + " + " + str(self._right)

print(Plus(Constant(3),Var("x")))
```

```
3 + x
```

Observe that the call to `str` in `__str__` for `Plus` is *recursive*: we call the `str` method on the left and right subtrees. What functions are then run depends on the forms of these subtrees.

Let's now add `*`:

```
class Times(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

    def __str__(self):
        return str(self._left) + " * " + str(self._right)

print(Plus(Times(Var("x"),Var("y")),Constant(7)))
```

```
x * y + 7
```

But this definition is exactly the same as `Plus`, aside from the fact that we print `*` instead of `+`. Copying and pasting code is dangerous—it is easy to introduce bugs! A better solution is to instead introduce a new class for binary operations from which `Plus` and `Times` can inherit:

```
class BinOp(Expr):
```

```
    def __init__(self, left, right):
        self._left = left
        self._right = right

class Plus(BinOp):

    def __str__(self):
        return str(self._left) + " + " + str(self._right)

class Times(BinOp):

    def __str__(self):
        return str(self._left) + " * " + str(self._right)

print(Plus(Times(Var("x"),Var("y")),Constant(7)))
```

```
x * y + 7
```

This seems like it works well, but it goes wrong if we try:

```
print(Times(Var("x"),Plus(Var("y"),Constant(7))))
```

```
x * y + 7
```

We get the exact same string as above! The problem is of course that we forgot to insert parentheses. A naive solution to this problem is to always insert parentheses around the subexpressions in + and *:

```
class BinOp(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

class Plus(BinOp):

    def __str__(self):
        return "(" + str(self._left) + ") + (" + str(self._right) + ")"

class Times(BinOp):

    def __str__(self):
        return "(" + str(self._left) + ") * (" + str(self._right) + ")"

print(Plus(Times(Var("x"),Var("y")),Constant(7)))

print(Times(Var("x"),Plus(Var("y"),Constant(7))))
```

```
((x) * (y)) + (7)
(x) * ((y) + (7))
```

These are correct, but filled with unnecessary parentheses! What we'd actually like to see is:

```
print(Plus(Times(Var("x"),Var("y")),Constant(7)))

print(Times(Var("x"),Plus(Var("y"),Constant(7))))
```

```
x * y + 7
x * (y + 7)
```

How can we do it?

Solution: decorate each expression with how tightly its outermost operator binds. With this extra information, we can decide whether or not to print parentheses around each subexpression. We do this by adding a class variable `prec` (for *precedence*) to the `Expr` class and then overriding it in `Plus` and `Times`. We then have a helper function `parens` that decides whether to print parentheses around an expression.

```
class Expr:

    # default precedence very high so that we don't put parantheses
    prec = 1000

class Constant(Expr):

    def __init__(self, value):
        self._value = value

    def __str__(self):
        return str(self._value)

class Var(Expr):

    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class BinOp(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

# Function for inserting parentheses around a string if precedence p1
# is smaller than precedence p2
def parens(p1,p2,s):
    if p1 < p2:
        return "(" + s + ")"
    else:
        return s
```

```
class Plus(BinOp):

    prec = 1

    def __str__(self):
        s1 = parens(self._left.prec,self.prec,str(self._left))
        s2 = parens(self._right.prec,self.prec,str(self._right))
        return s1 + " + " + s2

class Times(BinOp):

    prec = 2

    def __str__(self):
        s1 = parens(self._left.prec,self.prec,str(self._left))
        s2 = parens(self._right.prec,self.prec,str(self._right))
        return s1 + " * " + s2


print(Plus(Times(Var("x"),Var("y")),Constant(7)))

print(Times(Var("x"),Plus(Var("y"),Constant(7))))

print(Times(Var("x"),Times(Var("y"),Plus(Constant(3),Var("z")))))
```

```
x * y + 7
x * (y + 7)
x * y * (3 + z)
```

11.3 Example: Binary trees

We can generalize the datatype for algebraic expressions to arbitrary binary trees. In this way, we can write our functions once and for all, instead of needing to replicate them for every type of binary tree we end up using.

A binary tree is either a node (with a value and left and right subtrees) or a leaf (with just a value). On top of these, we want the following functionality:

- `__str__`: coerce to a string.
- `member(x)`: test if `x` appears as a value in the tree.
- `map_tree(f)`: apply the function `f` to all the values in the tree (note that this is a higher-order function).
- `linearize()`: convert the tree to a list.

In this example, the main class `BinTree` is empty, but it is still good to have around in case we one day decide to add something, like we did with `prec` for algebraic expressions.

```
class BinTree:
    pass

class Branch(BinTree):

    def __init__(self, node, left, right):
        self._node = node
        self._left = left
        self._right = right

    def __str__(self):
        n , l , r = str(self._node) , str(self._left) , str(self._right)
        return "(" + n + "," + l + "," + r + ")"

    def member(self, x):
        return self._node == x or self._left.member(x) or self._right.member(x)

    def map_tree(self,f):
        return Branch(f(self._node),self._left.map_tree(f),self._right.map_tree(f))

    def linearize(self):
        return self._left.linearize() + [self._node] + self._right.linearize()

class Leaf(BinTree):

    def __init__(self, val):
        self._val = val

    def __str__(self):
        return "(" + str(self._val) + ")"

    def member(self, x):
        return self._val == x

    def map_tree(self, f):
        return Leaf(f(self._val))

    def linearize(self):
        return [self._val]

t = Branch(-32,Leaf(2),Branch(1,Branch(23,Leaf(4),Leaf(-2)),Leaf(12)))

print(t)
print(t.member(12))
print(t.member(-123))

print(t.map_tree(lambda x: 0))
print(t.map_tree(lambda x: x ** 2))
```

```
print(t.linearize())
```

```
(-32,(2),(1,(23,(4),(-2)),(12)))
True
False
(0,(0),(0,(0,(0),(0)),(0)))
(1024,(4),(1,(529,(16),(4)),(144)))
[2, -32, 4, 23, -2, 1, 12]
```

Note that all the functions that we defined are recursive. This example shows how we can combine classes and recursion in a simple way to represent and modify structured data.

11.4 Exercises

1. Divide up the examples from the previous chapter by whether they exhibit simple, multilevel, hierarchical, or hybrid inheritance.
2. Move the `__str__` method for `Plus` and `Times` to `BinOp` in an appropriate way so that we avoid the code duplications. The examples should still print the same results.
3. Add a class `Exp` for exponentiation of algebraic expressions. It should bind more tightly than both `+` and `*` and be printed as `**` (as in Python's syntax).
4. Add a function `big_constant_sum(n,e)` where `n` is a positive integer and `e` is an algebraic expression. The function should create an expression where `e` is added to itself `n` times. If `n` is `0`, it should return the constant `0`. In other words, the function should return the sum

$$\sum_{i=0}^{n-1} e$$

   Tip: use recursion on `n`.
5. Add a function `big_sum(n,e)` where `n` is a positive integer and `e` is a function from integers to algebraic expressions. The function should create an expression that represents

$$\sum_{i=0}^{n} e_i$$

6. Define the evaluation of algebraic expression given a dictionary `d` from variables to numbers. To do this, add a method `evaluate(d)` to `Constant`, `Var`, `Plus`, `Times`, and `Exp`. You can assume that all variables in the expression are defined in `d`.
7. † Implement rose trees. This is a type of tree where every node holds a value and a list of more rose trees. A leaf with value `5` is represented as `RoseTree(5,[])`. Implement a class for rose trees that has the same methods as `BinTree`.

Chapter 12

Defensive programming

Anyone who has learned to program has, of course, also been forced to learn to hunt for bugs. All programmers make mistakes, accidentally rely on incorrect assumptions, and misunderstand documentation. Errors might cause a program to crash right at the moment where a mistake is made, but they can also lie undetected until much later. If a division by zero occurs somewhere in a program, it might only be a symptom of an earlier mistake. Perhaps a variable was set to 0 at an inappropriate moment, but where and why? It can take a lot of work to track down the source of an error. The *defensive programming* style aims in part to make errors apparent quickly, near the actual mistake, to avoid tiresome debugging sessions, and in part to make your assumptions clear in your code.

Defensive programming means expecting that functions will be used in the wrong way, algorithms will be implemented incorrectly, unexpected combinations of variable values will occur, and generally that oddities of all kinds will arise during program development. By *coding in* your expectations for values of variables by using `assert` throughout your program, you can discover errors earlier and closer to the source, and thereby simplify debugging.

12.1 Assert

The statement `assert condition` indicates the the expression `condition` is expected to be true. If the expression is true, nothing happens, but if it is false the program crashes with an exception, an `AssertionError`. This type of instruction exists in many languages and is an aid to debugging, development, and maintenance of programs.

Some examples:

```
assert len(data_list) > 0
assert x != 0
assert x > 0 and y > 0
```

These lines specify three possible demands on the state of a program. The instruction `assert` can be read as “it must be the case that…”.

12.1.1 Interruption and error messages

Suppose we have written a division function like so:

```
def divide(x,y):
    assert y != 0
    return x / y
```

If we now call `divide` with `0` as the second argument

```
print(divide(10,2))
print(divide(10,0))
```

then an error message is printed:

```
Traceback (most recent call last):
  File "f12.py", line 6, in <module>
    print(divide(10,0))
  File "f12.py", line 2, in divide
    assert y != 0
AssertionError
```

Note that the exception `AssertionError` is not expected to be caught with **`except`**, though Python technically allows this; we use **`assert`** to detect serious programming errors, not to handle runtime errors (for example, that some file is missing, that there is not enough memory, that one has recursed too deeply, etc.).

12.1.2 In generality

In general, one can write

```
assert condition [, comment_string]
```

where `condition` is a Boolean expression and `comment_string` is an optional explanatory message. We can add messages to the code above like so

```
assert len(data_list) > 0, 'User submitted empty data'
assert x != 0, 'x==0 must be avoided early in the application!'
assert x > 0 and y > 0, 'Both x and y have to be positive!'
```

to help us understand any assertion error that occurs.

For example:

```
def divide(x,y):
    assert y != 0, "Second argument cannot be 0 when dividing"
    return x / y

print(divide(10,0))
```

gives

```
Traceback (most recent call last):
  File "f12.py", line 6, in <module>
    print(divide(10,0))
  File "f12.py", line 2, in divide
    assert y != 0, "Second argument cannot be 0 when dividing"
AssertionError: Second argument cannot be 0 when dividing
```

12.1.3 Rules of thumb for `assert`

`assert` is implemented with exceptions, but its purpose is to express assumptions in code, not to signal exceptional situations that should be handled in some way.

- **`assert`** is for problems that arise during coding/development.
- **`raise`** and **`try`**/**`except`** are for problems that arise at runtime.

We can say that **`assert`** is used to flag *unexpected* situations in code, **`raise`** is used to signal errors that might have been expected, and **`try`**/**`except`** are used to capture more-or-less expected errors. A program should not depend on the **`assert`** statements it uses, but work just as well with them removed. In general, including **`assert`**s makes a program slower, because there is more code to execute, but you should not therefore be reluctant to use **`assert`** in your code. If speed is important, you can start your program with `python -O kod.py` instead of `python kod.py`; the flag `-O` deactivates all **`assert`**-instructions.

12.1.3.1 Example 1: `remove_trailing_spaces`

Below, we have an example of **`assert`** in `remove_trailing_spaces`:

```
def remove_trailing_spaces(s, spaces=" \t"):
    assert len(spaces) > 0
    i = len(s)
    while i > 0 and s[i-1] in spaces:
        i -= 1
    return s[0:i]
```

12.1.3.2 Example 2: `newton_raphson`

Next, we have an example of **`assert`** in a function for calculating the roots of a numerical function with the help of the Newton-Raphson algorithm:

```
def newton_raphson(f, f_prime, x, precision):
    assert isinstance(x, float)
    assert isinstance(f(x), float)
    assert isinstance(f_prime(x), float)
    assert  0.0 < precision < 1.0, 'precision must be in the interval (0,1)'

    ready = False
    while not ready:
        x_next = x - f(x)/f_prime(x)      # Main line of code to be repeated
        ready = abs(x-x_next) < precision # Test for stopping criterion
        x = x_next                        # Preparing the next iteration

    return x_next
```

As you can see, we can test the type of an object using `isinstance`. The function `type`, which returns the object's type, is also useful, but `isinstance` is more general: `isinstance` can take a tuple of classses as its second argument and returns `True` if the first argument is an instance of *one* of the classes *or* one of their subclasses.

Notice also that **`assert`** can be seen as extra documention of the assumptions at play in the function or program.

12.1.4 Conditions and invariants

We can divide up uses of `assert` into three categories:

- **preconditions**: expressions that should be true when a function call is executed;
- **postconditions**: expressions that should be true when a function returns;
- **loop invariants**: expressions that should be true in a loop.

12.1.4.1 Example 3

Suppose we have a database that connects unique identifiers (represented as `int`s) with names (represented as `str`s). To make it efficient to look up someone using *either* their identifier or their name, we'll have two dictionaries, one from identifiers to names and one from names to identifiers. For this to work properly, the keys in one dictionary must be the same as the values in the other and vice versa.

```
class MyDB:

    def __init__(self):
      self._id2name_map = {}
      self._name2id_map = {}

    def add(self, id, name):
      self._name2id_map[name] = id
      self._id2name_map[id] = name

    def by_name(self, name):
      id = self._name2id_map[name]
      return id
```

One way this could go wrong would be if the user simply overwrote one of `_id2name_map` or `_name2id_map`; another would be if we later added some method with a bug that broke this invariant for the class. We can add an `assert` in `by_name` where we double-check that the invariant has not been broken:

```
class MyDB:

    def __init__(self):
      self._id2name_map = {}
      self._name2id_map = {}

    def add(self, id, name):
      self._name2id_map[name] = id
      self._id2name_map[id] = name

    def by_name(self, name):
      id = self._name2id_map[name]
      assert self._id2name_map[id] == name
      return id
```

If we now add a buggy method or a user uses the class in the wrong way, then `by_name` will generate an `AssertionError`. Another invariant is that an ID should be an `int` and a name a `str`. In many programming languages with stricter type systems, we can specify this in the signature of the class

or constructor. In Python, we can achieve something like the same effect by adding an `assert` using `isinstance`:

```
class MyDB:

    def __init__(self):
      self._id2name_map = {}
      self._name2id_map = {}

    def add(self, id, name):
      assert isinstance(id, int), ("id is not an integer: " + str(id))
      assert isinstance(name, str), ("name is not a string: " + str(name))
      self._name2id_map[name] = id
      self._id2name_map[id] = name

    def by_name(self, name):
      id = self._name2id_map[name]
      assert self._id2name_map[id] == name
      return id


db = MyDB()
db.add(2,2)
db.add("hej","du")
```

12.2 Testing

It is difficult to guarantee that one's code is correct. You can reason mathematically about code and algorithms, but even then it is easy to make a mistake. No matter how much you plan, think, and review, it is good to also carefully test your code. But how can you best do so?

Experience shows it is neither sufficient nor practical to test if a whole program, the *product* itself, works as it should. If we want to discover problems early and protect yourself against errors caused by changes in code, then we need to test the program's functions and methods individually.

A typical workflow is to (1) write a little code, perhaps a function, (2) verify that the code works, and then continue to write more code. It can be simple to verify the code: choose some parameters to your function and see the result is what you expected. One scenario that is even more typical in practice is the following:

1. Make a change in existing code, to add new functionality.
2. Verify the new functionality.
3. Verify the old functionality.

Point 3 is often forgotten, and can also be a little difficult to get a handle on. We of course have our focus on the new functionality, not the old, and how was it we tested the old code originally? When you make changes in some function `fun`, do you remember to also test other parts of your code that depend on `fun`? As time goes by and a project changes, it becomes all the more difficult to verify old functionality. If you work with others' code, it is of course even harder to determine how to test the existing functions. So what can one do?

What we need is *systematic* and *automatic* testing: to write code that runs a suite of tests. This methodolgy is called *unit testing*. Today, there is enough support for such testing in typical programming languages that it is often simple to include tests in your project. In this section, we look at severl different ways of working with tests.

You can read more at: https://realpython.com/python-testing/

12.2.1 Testing with `assert`

We can test code both manually and automatically. For simple manual testing, we can use **`assert`**.

In lab 2, we wrote functions manipulating and evaluating list representations of polynomials. One part of the exercise was to test these functions by manually inspecting the output of a given input. A better way would have been to include the following file in the instructions for lab 2:

```
from labb2 import *

p = [2,0,1]
q = [-2,1,0,0,1]
p0 = [2,0,1,0]
q0 = [0,0,0]

assert (poly_to_string(p) == '2 + x^2')
assert (poly_to_string(q) == '-2 + x + x^4')
assert (poly_to_string([]) == '0')
assert (poly_to_string([0,0,0]) == '0')
assert (drop_zeroes(p0) == [2, 0, 1])
assert (drop_zeroes(q0) == [])
assert (eq_poly(p,p0))
assert (not eq_poly(q,p0))
assert (eq_poly(q0,[]))
assert (eval_poly(p,0) == 2)
assert (eval_poly(p,1) == 3)
assert (eval_poly(p,2) == 6)
assert (eval_poly(q,2) == 16)
assert (eval_poly(q,-2) == 12)
assert (eq_poly(add_poly(p,q),add_poly(q,p)))
assert (eq_poly(sub_poly(p,p),[]))
assert (eq_poly(sub_poly(p,neg_poly(q)),add_poly(p,q)))
assert (not eq_poly(add_poly(p,p),[]))
assert (eq_poly(sub_poly(p,q),[4, -1, 1, 0, -1]))
assert (eval_poly(add_poly(p,q),12) == eval_poly(p,12) + eval_poly(q,12))
```

This file will only run to completion if all the tests pass! We do not need to include the tests or the polynomials we only use for testing (`p`, `q`, `p0`, and `q0`) in the main file `lab2.py`.

In this way, we can use **`assert`** to test and specify our programs. One problem, however, is that we only see the results of the first **`assert`** that does not hold; it might be more useful to see *all* the tests that are failing. A better way to test more complex programs is with *unit testing*.

12.2.2 Unit testing with `unittest`

The module `unittest` has been included in Python's standard library since version 2.1 and is used in many projects, both commercial and open-source. To use `unittest`, we

1. group all tests in classes where the tests are methods;
2. use specific assert-methods from `unittest.TestCase` instead of the built-in **`assert`** expression.

To rewrite the lab 2 example above so that it uses `unittest`, we do the following:

- import `unittest`;
- create a class `TestPoly` that inherits from the `TestCase` class;
- replace each `assert` with a method in this class using `self.assertEqual()`;
- change the file so that `unittest.main()` is run when it loads;
- create a class `TestPoly` that inherits from the `TestCase` class.

The file might then look like so:

```
import unittest

import labb2

p = [2,0,1]
q = [-2,1,0,0,1]
p0 = [2,0,1,0]
q0 = [0,0,0]

class TestPoly(unittest.TestCase):

    def test_poly_to_string_p(self):
        self.assertEqual(labb2.poly_to_string(p),'2 + x^2', \
                                        "Should be 2 + x^2")

    def test_poly_to_string_q(self):
        self.assertEqual(labb2.poly_to_string(q),'-2 + x + x^4',\
                                        "Should be -2 + x + x^4")

    def test_poly_to_string_empty(self):
        self.assertEqual(labb2.poly_to_string([]),'0', "Should be 0")

    def test_poly_to_string_empty2(self):
        self.assertEqual(labb2.poly_to_string([0,0,0]),'0', "Should be 0")

    def test_drop_zeroes_p0(self):
        self.assertEqual(labb2.drop_zeroes(p0),[2, 0, 1])

    def test_drop_zeroes_q0(self):
        self.assertEqual(labb2.drop_zeroes(q0),[])

    def test_eq_poly_pp0(self):
```

```
        self.assertEqual(labb2.eq_poly(p,p0),True)

    def test_eq_poly_qp0(self):
        self.assertNotEqual(labb2.eq_poly(q,p0),True)

    def test_eq_poly_q0(self):
        self.assertEqual(labb2.eq_poly(q0,[]),True)

    # ...

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

If we run this, we'll get:

```
.........
----------------------------------------------------------------------
Ran 9 tests in 0.001s

OK
```

If we have a bug, for example if `poly_to_string` turns out to always return the empty string, we'll instead get:

```
.....FFFF
======================================================================
FAIL: test_poly_to_string_empty (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 19, in test_poly_to_string_empty
    self.assertEqual(poly_to_string([]),'0', "Should be 0")
AssertionError: '' != '0'
+ 0 : Should be 0

======================================================================
FAIL: test_poly_to_string_empty2 (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 22, in test_poly_to_string_empty2
    self.assertEqual(poly_to_string([0,0,0]),'0', "Should be 0")
AssertionError: '' != '0'
+ 0 : Should be 0

======================================================================
FAIL: test_poly_to_string_p (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 13, in test_poly_to_string_p
    self.assertEqual(poly_to_string(p),'2 + x^2', "Should be 2 + x^2")
AssertionError: '' != '2 + x^2'
+ 2 + x^2 : Should be 2 + x^2
```

```
======================================================================
FAIL: test_poly_to_string_q (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 16, in test_poly_to_string_q
    self.assertEqual(poly_to_string(q),'-2 + x + x^4', "Should be -2 + x + x^4")
AssertionError: '' != '-2 + x + x^4'
+ -2 + x + x^4 : Should be -2 + x + x^4

----------------------------------------------------------------------
Ran 9 tests in 0.002s

FAILED (failures=4)
```

We thus see that 4 of the tests failed, and we can begin to debug our code. By writing good tests that we run every time we make a change, we can be sure that our code continues to work as we originally expected despite the changes. This is extremely useful in large project where one has many collaborators. Many systems for collaborative programming (for example Github) have support for running this type of automatic testing every time someone has a change they want to make to the codebase; this development technique is called *continuous integration*.

12.2.3 Random testing (à la QuickCheck):

It is difficult to write good tests by hand. An alternative that can be very convenient is to use a library that generates tests automatically, that is by so-called "random testing". For this type of testing, one writes a specification—for example `add_poly(p,q) == add_poly(q,p)`—and the library then generates random inputs (polynomials, in this case) to test that the specification is satisfied.

Python has support for this via the hypothesis library.

12.3 Exercises

1. The Jaccard index (J) is a measure of how similar two sets `A` and `B`, defined as

$$J := \frac{|A \cap B|}{|A \cup B|}$$

   where $\cap$ and $\cup$ are *intersection* and *union* of the sets `A` and `B` (see Wikipedia). Implement this formula using a Python function `jaccard_index(l1,l2)` that takes two lists `l1` and `l2` as arguments.

   Tip: Convert the lists to sets and use set methods. For example, the function should give the following results:

```
print(jaccard_index(['dog', 'cat', 'mouse'], ['dog', 'cat', 'mouse']))
print(jaccard_index(['dog', 'cat'], ['dog', 'cat', 'mouse']))
print(jaccard_index(['dog', 'cat'], ['mouse', 'rat']))
```

```
1.0
0.6666666666666666
0.0
```

2. Use the module `unittest` to write unit tests for your function from Exercise 1.

3. A limitation of the Jaccard-index is that it does not notice repeated elements. For example, the two calls below give the same Jaccard-indices as above, although we might want to describe these two *multisets* (sets where the same element can appear multiple times) as "more different".

```
print(jaccard_index(['dog', 'dog', 'dog', 'cat', 'mouse'],
                    ['dog', 'cat', 'cat', 'cat', 'cat',  'mouse']))
print(jaccard_index(['dog', 'cat','cat', 'cat'],
                    ['dog', 'cat', 'mouse', 'mouse']))
```

```
1.0
0.6666666666666666
```

Implement code for computing a *weighted* Jaccard-index. Let A and B be two *multisets* and let A_z and B_z denote how many times an element z occurs in A and B respectively. Then we define the weighted Jaccard-index J_w as

$$J_w := \frac{\sum_{z \in (A \cup B)} \min(A_z, B_z)}{\sum_{z \in (A \cup B)} \max(A_z, B_z)}$$

For example:

```
print(weighted_jaccard_index(['dog', 'dog', 'dog', 'cat', 'mouse'],
                             ['dog', 'cat', 'cat', 'cat', 'cat', 'mouse']))
print(weighted_jaccard_index(['dog', 'cat', 'cat', 'cat'],
                             ['dog', 'cat', 'mouse', 'mouse']))
```

```
0.375
0.3333333333333333
```

When you implement this, keep our tips on structure and code from our section on "good code" in mind. For example, separate code in small functions that do one thing and avoid long one-line expressions. Think about how the code should be structured before beginning.

4. Use the module `unittest` to write unit tests for your functions from Exercise 3.

Appendix A

Sample solutions

A.1 Chapter 1

1. x and y switch values
2. For example `(x - y - y)**2`
3. `s*16`
4. ```
   s = 'hej'
   s2 = s+s
   s4 = s2+s2
   s8 = s4+s4
   s16= s8+s8
   print(s16)
   ```
5. ```
   name = input("What is your first name?")
   surname = input("What is your last name?")
   print('Hi', name, surname + '!')
   ```
6. The numbers are concatenated because they are stored as strings. To actually add the numbers, you have to convert the strings to integers.

   ```
   age = input("Age?")
   nr = input("Favorite number?")
   print(int(age)+ int(nr))
   ```

A.2 Chapter 2

1. b, c

2. ```
   num = int(input("Enter an integer: "))
   if num % 2:
       print("Number is odd!")
   else:
       print("Number is even!")
   ```

3. | x | y | x and y | x or y |
   |---|---|---|---|
   | True | True | True | True |
   | True | False | False | True |
   | False | True | False | True |
   | False | False | False | False |

4. An example:

   ```
   x = True
   y = False
   z = False
   ```

   gives

   ```
   True
   False
   True
   False
   ```

5. ```
   def division(x, y):
       if x == 0 and y == 0:
           print("Indeterminate form....")
       elif y == 0:
           print("The result is infinite...")
       else:
           return x/y
   ```

6. ```
   False
   True
   True
   True
   ```

7. ```
   def nand(x, y):
       return not (x and y)

   def nor(x, y):
       return (not x) and (not y)
   ```

```
def xnor(x, y):
    return ((not x and not y) or (x and y))

# alternativ xnor
def xnor(x, y):
    return (x==y)
```

8. If the functions are defined as above, then we'll get for example:

```
xnor(47, 33)
```

```
33
```

Because both `47` and `33` are true, you might have expected the result to be `True`, but the functions in the definition of `xnor` can return other values when their arguments are not of type `bool`.

A.3 Chapter 3

1. The list `[1, 2, 'a', 'b']`
2. It is not possible for example to compare strings and floats or ints.

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

3.
```
def vowels(s):
    vs = "AEIOUYaeiouy"
    out = ""
    for v in s:
        if v in vs:
            out += v
    return out
```

4.
```
def vowels_or_cons(s, save='v'):
    vs = "AEIOUYaeiouy"
    out = ""
    if save != 'c' and save != 'v':
        print("illegal argument to save")
        # Here we should raise error. More about error control later
        return

    for v in s:
        if save == 'v' and v in vs:
            out += v
        elif save == 'c' and v not in vs:
            out += v

    return out
```

5. If we leave out the block

```
if term != "":
    out.append(term)
```

then the last word will not be added if the string does not end with a space. If we leave out `if term != ""` but leave `out.append(term)` at indentation level 1, then an empty string will be added to the end if the string ends with a space. Note also that our split only handles one space; it will give an incorrect result if there are several spaces in a row. We can fix this by testing if term is empty as below:

```
def split(s):
    out = []
    term = ""
    for c in s:
        if c == " ":
            # Do not add term if empty (End up here when consecutive spaces)
            if term != '':
                out.append(term)
```

```
                term = ""
            else:
                term += c

    # add the final word unless it is empty
    #if term != "":
    out.append(term)

    return out
```

6. The problem is that we did not use `.copy()` on `list` when we create `list2`. If we do not do this, `list` and `list2` will simply be different names for the same list, so if we change one then the other will change as well. Test the following instead:

```
list = [1,2,3]
list2 = list.copy()
list2.reverse()
print(list)
```

7. Alternative 1:

```
mylist = ["Ho","ho","ho!"]
m = mylist.copy()
while m:
    x = m.pop()
    print(x)

print(mylist)
```

Alternative 2:

```
mylist = ["Ho","ho","ho!"]
n = len(mylist)
while n > 0:
    print(mylist[n-1])
    n -= 1

print(mylist)
```

Alternative 3:

```
mylist = ["Ho","ho","ho!"]
for i in mylist[::-1]:
    print(i)

print(mylist)
```

8. With **break**:

```
10
9
```

With **continue**, everything but 7 will be printed. With **pass**, all the numbers will be printed.

9. For example:

```
mylist = []
while True:
    ans = int(input("Enter a single digit (quit with 0): "))
    # if 0 is *not* to be included in the list.
    if ans == 0:
        break
    else:
        mylist.append(ans)
    print(mylist)
```

Another option is to use the `:=` operator, which is a relatively new addition to Python (version ≥ `3.8`, so use this operator with care). The `:=` operator evaluates *and* assigns in an `if` statement, so we can avoid one line in the code block:

```
mylist = []
while ans := int(input("Enter a single digit (quit with 0): ")):
    mylist.append(ans)
print(mylist)
```

Here `ans` is first assigned the input value and the `if` condition is evaluated *afterwards*. Because `0` coerces to `False` (`bool(0) == False`), we will exit the `while`-loop when the user enters `0`.

10. For example:

```
def naiveGCD(a,b):
    for i in range(min(a,b), 0, -1):
        if a % i == 0 and b % i == 0:
            return i

a = 12
b = 3
print(naiveGCD(a,b))
```

or:

```
def naiveGCD(a,b):
    i = min(a,b)
    while i > 0:
        if a % i == 0 and b % i == 0:
            return i
        i -= 1
```

11. For example:

```
def portkod(pw ='1337'):
    curr = ""
    while True:
        n = input("Enter digit:")
        # make sure our string is only 4 digits long
        if len(curr) >= 4:
            curr = curr[1:]
```

```
            print(curr)
        curr += n # add new digit
        if curr == pw:
            print("door unlocked")
        break

portkod(pw ='1337')
```

```
def portkod(pw ='1337'):
    curr = ""
    while True:
    n = input("Enter digit:")
        # make sure our string is only 4 digits long
        if len(curr) >= 4:
            curr = curr[1:]
            print(curr)
        curr += n # add new digit
        if curr == pw:
            print("door unlocked")
        break

portkod(pw ='1337')
```

A.4 Chapter 4

1. For example:

```
a = ['a','b','c']
b = [1, 2, 3]

# alt 1
d = {}
for i in range(len(a)):
    d[a[i]] = b[i]

# alt 2
d = {a[i] : b[i] for i in range(len(a)) }
```

2.
```
def naiveGCD(a,b):
    for i in range(min(a,b), 0, -1):
        if a % i == 0 and b % i == 0:
            return i

print("x, y, GCD")
for x in range(2, 21):
    for y in range(2, 21):
        gcd = naiveGCD(x, y)
        print(x, y, gcd)
```

3. Replace `for y in range(2, 21):` with `for y in range(x, 21):`

4.
```
def vowels(s):
    vs = "AEIOUYaeiouy"
    d = {vs[i] : i for i in range(len(vs))}
    out = ""

    for v in s:
        if v in d:
            out += str(d[v])
        else:
            out += v
    return out

print(vowels("Hej"))
```

5.
```
def annotate_word(s):
    out = {}
    term = ""
    cnt = 1
    for c in s:
        if c == " ":
            out[term] = cnt
```

```
            term = ""
            cnt +=1
        else:
            term += c

    # add the final word unless it is empty
    if term != "":
        out[term] = cnt
    return out
```

6. Alternative 1: Write `out[cnt] = term` instead of `out[term] = cnt` Alternative 2: Add the following line at the end of the function.

```
out_reversed = { v : k for k, v in out.items()}
```

7. We can have lists as values:

```
def annotate_word(s):
    out = {}
    term = ""
    cnt = 1
    for c in s:
        if c == " ":
            if term in out:
                out[term].append(cnt)
            else:
                out[term] = [cnt]
            term = ""
            cnt +=1
        else:
            term += c

    # add the final word unless it is empty
    if term != "":
        if term in out:
            out[term].append(cnt)
        else:
            out[term] = [cnt]
    return out
```

A.5 Chapter 5

1. We get a `TypeError`. We can catch it like so:

```
def divide_by_elems(filename,x):
    try:
        x = int(x)
    except TypeError:
        print("divide_by_elems:",
              "Second argument cannot be interpreted as an integer.")
        return

    # ... rest of function here
```

2. For example:

```
def divide_by_elems(filename,x):
    try:
        x = int(x)
    except TypeError:
        print("divide_by_elems:",
              "Second argument cannot be interpreted as an integer.")
        return
    divs = []

    try:
        h = open(filename, 'r')
    except IOError:
        print("divide_by_elems: A file-related problem occured.")

    for n in h:
        try:
            frac = x / int(n)
            divs.append(frac)
     except ZeroDivisionError:
            print("divide_by_elems: Division by zero.")
        except ValueError:
            print("At least one line contains characters that",
                  " cannot be converted with int().")
    h.close() #  close the file
    return divs
data = divide_by_elems('numbers.txt', 2)
print(data)
```

3.
```
ys = [i for i in xs if i > 0]
```

4.
```
ys = [i if i > 0 else -i for i in xs]
```

   or

```
[abs(i) for i in xs]
```

5. ```
   [p for p in range(2,100) if is_prime(p)]
   ```

6. For example:

   ```
   [x1*x2 for x1 in xs for x2 in xs ]
   ```

7. ```
   [x[i]*x[j] for i in range(len(x)) for j in range(len(x)) if i != j]
   ```

8.
   -

A.6 Chapter 6

1. If for example we don't intend to change the information in a data structure, then we should save the information in a tuple. Then Python will let us know when the program tries to change it instead of quietly making the change. This is very important when we begin to use others' code, in which case we don't have a full view of how their functions and modules work.

2. For example:

```
def is_palindrome(s):
    backward_s = s[::-1]
    if s == backward_s:
        return True
    else:
        return backward_s
```

3. Yes, the notation `s[::-1]` works for all sequence types in Python.

4. For example:

```
def occurrences(s):
    """Returns number of occurrences of letters in string.
    """
    s = s.lower()
    d = {}
    for c in s:
        if not c.isalpha():
            c = 'non_alphas'
        if c not in d:
            d[c] = 0
        d[c] += 1
    return d
```

5. The function `occurences` works on strings, so if we read in the whole file as *one string*, we can use it directly. We can read in the whole file with the `.read()` method. This includes the character for a new line (`\n`), so if we don't want to include these in the calculation, we should take them out with the string function `replace`.

```
with open('palindrome.txt', 'r') as fp:
    # reads contents of entire file in one go.
    # Note: memory consuming if a large file.
    file_content = fp.read()

# if newline characters, \n, are to be counted
chars_with_newline = occurrences(file_content)

# if newline characters are to be skipped, modify the string by
# replacing the '\n' character with the empty string ''
file_content_no_newline = file_content.replace('\n', '')
chars_without_newline = occurrences(file_content_no_newline)

print(chars_with_newline)
```

```
print(chars_without_newline)
```

6. For example:

```
def fibonacci(stop=55):
    i = 0
    yield i
    j = 1
    yield j
    while i + j < stop:
        yield i + j
        i, j = j, i + j
```

7. For example:

```
def map_gen(f, it):
    """Applies a function to the elements of an iterable and returns a
    generator.
    """
    return (f(i) for i in it)
```

8. For example:

```
def is_string_palindrome(s):
    """Checks if string is a palindrome
    """
    return s == s[::-1]


def palindrome_rows(filename):
    """Checks if each row of file is a palindrome string
    """
    return (is_string_palindrome(row.strip('\n'))
            for row in open(filename, 'r'))


# OR

def palindrome_rows(filename):
    with open(filename, 'r') as fp:
        for row in fp:
            yield is_string_palindrome(row.strip('\n'))
```

9. For example:

```
file_content = open('palindrome.txt').read()
capitalized_word = file_content.lstrip(file_content.lower()).split()[0]
```

A.7 Chapter 7

1. For example:

```
import random
random_integers = [random.randint(1, 100) for i in range(10)]
random_floats = [random.uniform(1, 100) for i in range(10)]
```

2. For example:

```
def is_perfect(x):
    sum = 0
    for i in range(1, x):
        if(x % i == 0):
            sum = sum + i
    if sum == x:
        return True
    else:
        return False

def is_prime(x):
    if x >= 2:
        for y in range(2,x):
            if not ( x % y ):
                return False
    else:
        return False
    return True

def main(x):
    print("The number is prime:", is_prime(x))
    print("The number is prime:", is_perfect(x))

x = int(input("Provide a number to analyze: \n"))
main(x)
```

3. We can add something like

```
choice = input(
    "Type 'G' to manually enter number or 'R'"
    "to randomly generate number: \n")
if choice == "G":
    x = int(input("Provide a number to analyze: \n"))
elif choice == "R":
    x = random.randint(1, 100)
else:
    print("You failed")
```

The important of the `random` module can be at the top of the file, and our new code should sit below the functions `is_perfect(x)` and `is_prime(x)`. We can have it in `main` or separately; it depends on how we want to be able to use `main`. Should it include the new functionality or not? Normally, functions one might want to import should not include interactive functionality (communication

with the user).

4. We divide up the code so that every function takes parameters and returns. We have removed all globally defined variables (`seq`, `k`, `substrings`) so that we can reuse each individual function in other programs.

```
# Given a sequence as input find distances
# between the copies of the most frequent substring

def get_substrings(seq, k):
    substrings = {}
    for i in range(len(seq) - k + 1):
        substring = seq[i: i+k]
        if substring in substrings:
            substrings[substring] += 1
        else:
            substrings[substring] = 1
    return substrings

def find_most_freq_substring(substrings):
    max_count = 0
    most_freq_substring = ""
    for substring, count in substrings.items():
        if count > max_count:
            most_freq_substring = substring
            max_count = count
    return most_freq_substring

def find_positions(query_substring, seq):
    positions = []
    k = len(query_substring)
    for i in range(len(seq) - k + 1):
        substring = seq[i: i+k]
        if substring == query_substring:
            positions.append(i)
    return positions

def get_distances(positions):
    distances = []
    for p1,p2 in zip(positions[:-1], positions[1:]):
        distances.append(p2 - p1)
    return distances


## Driver code
def main():
    seq = input("Enter a dna string: ")
    k = 3
    substrings = get_substrings(seq, k)
    max_substring = find_most_freq_substring(substrings)
```

```
    positions = find_positions(max_substring, seq)
    distances = get_distances(positions)
    print(distances)
```

5. For example:

```
import itertools
numbers = [13, 24, 42, 66, 78]
string_numbers = [str(i) for i in numbers]
# permutations
perm = itertools.permutations(string_numbers, r=2)
results = [int(''.join(i)) for i in perm]
```

6. •

A.8 Chapter 8

1. FOr example:

```
def rec_prod_m(n,m):
    if n <= 0:
        return 1
    else:
        return n * rec_prod_m(n-m, m)
```

2. `prod([])` will return `0`, which will make it so that all other function calls on line 4 of the function will be multiplied with `0`. For example, when `rec_prod_m(1,1)` is called, the expression in the `return` statement will become `1 * rec_prod(0,1)`, which will in turn become `1 * 0`.

3. For example:

```
f1 = lambda x, y: x + y
f2 = lambda x, y: x - y
f3 = lambda x, y: x * y
f4 = lambda x, y: x / y
f5 = lambda x, y: x ** y

print((6*(2+3)/5)**2)
print(f5(f3(f4(f1(2,3),5),6),2))

print((10 - 2**3)*5)
print(f3(f2(10,f5(2,3)),5))
```

4. For example:

```
def get_evens(xs):
    return list(map(lambda x: x % 2 == 0, xs))

l = [1,4,2,3,4,5]
print(get_evens(l))
```

5. 7 function calls will be made in total for 3 (including the call `f(3)`) and 13 calls for 4. For `f(3)`:

```
f(3) = f(2) + f(1) + f(0)
     = (f(1) + f(0) + f(-1)) + 1 + 1
     = 1 + 1 + 1 + 1 + 1
     = 5
```

   and for `f(4)`:

```
f(4) = f(3) + f(2) + f(1)
```

   We know that `f(3)` will have 7 calls, `f(2)` will have 4, and `f(1)` will have 1, so in total we get `1 + 7 + 4 + 1 = 13` calls.

6. For example, any of the following definitions:

```
def collatz_rec(n):
    if n == 1:
        return []
```

```
    if n % 2 == 0:
        return [n] + collatz_rec(n//2)
    else:
        return [n] + collatz_rec(3*n+1)


def collatz_rec2(l):
    if l[-1] == 1:
        l.pop(-1)
        return l
    if l[-1] % 2 == 0:
        l.append(l[-1]//2)
    else:
        l.append(3*l[-1]+1)
    return collatz_rec2(l)


def collatz_rec3(n):
    while n != 1:
        return [n] + collatz_rec3(n//2 if not n % 2 else 3*n+1)
    return []


def collatz_rec4(n, collatz_seq):
    if n == 1:
        return collatz_seq
    else:
        collatz_seq.append(n)
        n = n//2 if not n % 2 else 3*n+1
        return collatz_rec4(n, collatz_seq)
```

These functions work in slightly different ways but give the same results. To calculate the Collatz sequence for `10`, we would call the functions in the following way:

```
collatz_rec(10)
collatz_rec2([10])
collatz_rec3(10)
collatz_rec4(10,[])
```

7. For example:

```
joinstrings = lambda xs: foldr(xs,lambda x,y: x + y,"")
```

8. For example:

```
def fac(n):
    ret = 1
    for i in range(1, n + 1):
        ret *= i
    return ret
```

```
def fib(n):
    i, j = 0, 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    k = 1
    while k < n:
        i, j = j, i + j
        k += 1
    return j
```

9. If `xs` is not copied, every function with `smallereq()` (and `greater()`) can potentially modify the list by removing all the elements (via `.pop()`). This means that after calling `smallereq`, there will be only the empty list to give to `greater` in a recursive call.

10. For example:

```
def pascal(n):
    if n == 1:
        return [1]
    else:
        p_line = pascal(n-1)
        line = [ p_line[i] + p_line[i+1] for i in range(len(p_line)-1)]
        line.insert(0,1)
        line.append(1)
        return line
```

A.9 Chapter 9

1. For example:

```
    def __str__(self):
        return "<" + self._fname + " " + self._lname
                + ", " + self._email + ">"
```

```
    def __str__(self):
        return "<Course: " + self._code + " " + self._name + ">"
```

2. For example:

```
class Participant:

    # Changed!
    def __init__(self, fname, lname, email, is_teacher = False):
        self._fname = fname                         # Changed!
        self._lname = lname                         # Changed!
        self._email = email
        self.is_teacher = is_teacher                # Changed!

    def name(self):
        return self._fname + " " + self._lname      # Changed!

    def email(self):
        return self._email

    def __str__(self):
        if self.is_teacher:
            return "<" + self._fname + " " + self._lname
                + ", " + self._email + "(Teacher) >"
        else:
            return "<" + self._fname + " " + self._lname
                + ", " + self._email + ">"
```

3. For example:

```
class IntSet:
    """An IntSet is a set of integers"""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self,e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)

    def member(self,e):
        """Assumes e is an integer.
```

```
        Returns True if e is in self, and False otherwise"""
    return e in self.vals

def delete(self,e):
    """Assumes e is an integer and removes e from self
       Raises ValueError if e is not in self"""
    try:
        self.vals.remove(e)
    except:
        raise ValueError(str(e) + ' not found')

def get_members(self):
    """Returns a list containing the elements of self.
       Nothing can be assumed about the order of the elements"""
    return self.vals

def __eq__(self, other_intset):
    for e in self.vals:
        if e not in other_intset.vals:
            return False
    for e in other_intset.vals:
        if e not in self.vals:
            return False
    return True

    # eller:
    # return (all([e in other_intset for e in self.vals ] ) and
    #         all([e in self.vals for e in other_intset  ] ))

    # eller:
    # return sorted(self.vals) == sorted(other_intset)

def __neq__(self, other_intset):

    return not self.__eq__()

def intersection(self, other_intset):
    i_new = IntSet()
    i_new.vals = [e for e in self.vals if e in other_intset.vals]
    return i_new

def __add__(self, other_intset):
    i_new = IntSet()
    i_new.vals =  ([e for e in other_intset.vals if e not in self.vals ]
                   + self.vals)
    return i_new

def __sub__(self, other_intset):
```

```
        i_new = IntSet()
        i_new.vals = [e for e in self.vals if e not in other_intset.vals ]
        return i_new
```

4. See the example to Exercise 3.

5. For example:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)
```

6. For example:

```
class Dog:
    kind = 'canine'
    # class variable shared by all instances
    def __init__(self, name):
        self.name = name
        self.tricks = []
        # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

    def __str__(self):
        return self.name + " knows: " + ','.join(self.tricks)

    def __gt__(self, other_dog):
        if len(self.tricks) > len(other_dog.tricks):
            return True
        else:
            return False
```

A.10 Chapter 10

1. For example:

```
from math import sqrt

class Equilateral(Triangle):
    def __init__(self):
        super().__init__()
        self.__name__ = 'Equilateral Triangle'

    def get_area(self):
        a, b, c = self.sides
        # check if all sides are of equal length
        if not a == b == c:
            errmsg = ("An ", self.__name__,
                      "cannot have different side lengths!")
            raise ValueError(errmsg)
        # calculate area
        return sqrt(3) * a**2 / 4
```

2. For example:

```
class Student(ProgMon):
    """A Pokemon Go Monster Student.... person.
    """
    def __init__(self, is_learning,
                 is_caffeinated=True,
                 is_unit_testing=False):
        super().__init__()
        self._attack = 0.05
        self._defense = 0.1
        self.is_learning = is_learning
        # convenience attribute to use for the multiplication of the
        # base class ProgMon's get_* functions.
        self._learning_multiplier = 2 if is_learning else 1

    def __str__(self):
        return "<Student>"

    def get_attack(self):
        return self._learning_multiplier * ProgMon.get_attack(self)

    def get_defense(self):
        return self._learning_multiplier * ProgMon.get_defense(self)
```

3.
```
class Tournament():
    """Documentation for Tournament

    """
```

```
    # Note: either an immutable type should be used as a default argument
    # for participants (like an empty tuple) or like below, None.
    # an alternative would be to have the call signature as:
    # __init__(self, participants=())
    # and cast participats to a new list in the constructor, i.e.
    # self.participants = list(participants)
    def __init__(self, participants=None):
        if not participants:
            self.participants = []
        else:
            self.participants = participants

    def add_participant(self, participant):
        self.participants.append(participant)

    def start_tournament(self):
        """Decide who the winner of the Tournament is, and rank the rest.

        Based on the number of tricks (i.e. size of the list contianing tricks)
        """
        # sort the list of participants based on how many tricks they
        # know (key is number of tricks for a Dog). Since `sorted` is
        # by default ascending, we reverse the list.
        # Sorted can make use of comparison operators such as `__lt__`, `__leq__`,
        # and more (in this case it finds the __gt__ method from class Dog()
        self.ranking = sorted(self.participants, reverse = True)
        self.winner = self.ranking[0] if self.ranking else None

    def print_results(self):
        print("Tournament Results\n" + "-"*18)
        print(self.winner.name, "is the winner!\n" +
              "Tricks:\n\t", self.winner.tricks)
        print("\nTournament runner-ups:")
        for i, d in enumerate(self.ranking[1:], start=2):
            print(str(i) + " place:", d.name)
```

Note: in the constructor for `Tournament`, we use a default value `None` which has an immutable type. Generally, it is important that the default value to functions and classes are of immutable types. The default value for a keyword parameters is set when a function/class is *defined*, and these arguments are *not* redefined every time a function is called. A mutable type as default value (for example `participants=[]`) would be set *once*. Suppose that we create an object `t = Tournament()`. Every time `participants` (or `self.participants` for that matter) is modified in the object `t`, the default value will be modified. This is not what we want: if one creates a new tournament with `t2 = Tournament()`, then the default value to `participants` in `t2` will not be the empty list. With the constructor above (and specifically its immutable default value), this is avoided, and a new instance of `Tournament` will always be empty (if one does not specify `participants` as an argument). See the secxtion on lists in Chapter 6.

4. ```
class Vehicle():
    def __init__(self, wheels=0, wings=0, sound=''):
        self.wheels = wheels
        self.wings = wings
        self.sound = sound
        self.__name__ = 'Abstract vehicle'

    def __str__(self):
        msg = 'A ' + self.__name__ + ' has:\n'
        msg_add = ''
        if self.wheels:
            msg_add += str(self.wheels) + ' wheels\n'
        if self.wings:
            msg_add += str(self.wings) + ' wings\n'
        if self.sound:
            if msg_add:
                msg_add += 'and '
            msg_add += 'makes a ' + self.sound + ' sound\n'
        return msg + msg_add

class Car(Vehicle):
    def __init__(self):
        super().__init__(wheels=4, sound='AAARRRRRRRR!')
        self.__name__ = 'Car'

class Motorcycle(Vehicle):
    def __init__(self):
        super().__init__(wheels=2, sound='VROOM VROOM!')
        self.__name__ = 'Motorcycle'

class Plane(Vehicle):
    def __init__(self):
        super().__init__(wings=2, sound='WHOOSH!')
        self.__name__ = 'Plane'
```

5. ```
class MyString(str):
    def __init__(self, args):
        super().__init__()

    def __sub__(self, other):
        return ''.join(self.rsplit(other))

    def __truediv__(self, other):
        return ''.join(self.split(other))
```

A.11 Chapter 11

1. …
2. For example:

```
class BinOp(Expr):
      def __init__(self, left, right):
              self._left = left
                      self._right = right

      def __str__(self):
          s1 = parens(self._left.prec,self.prec,str(self._left))
          s2 = parens(self._right.prec,self.prec,str(self._right))
          return s1 + self.op + s2

class Plus(BinOp):
      prec = 1
      op = " + "

class Times(BinOp):
      prec = 2
      op = " * "
```

3. Combined with the answer to the previous exercise, we just need to write:

```
class Exp(BinOp):
      prec = 3
      op = " ** "

print(Exp(Times(Var("x"),Var("y")),Constant(7)))
print(Times(Var("x"),Exp(Var("y"),Constant(7))))
print(Times(Var("x"),Exp(Var("y"),Plus(Constant(3),Var("z")))))
```

```
(x * y) ** 7
x * y ** 7
x * y ** (3 + z)
```

4. For example:

```
# using recursion
def big_constant_sum(n,e):
    if n == 0:
       return Constant(0)
    elif n == 1:
         return e
    else:
         return Plus(e,big_constant_sum(n-1,e))

# OR
# using iteration
def big_constant_sum_it(n,e):
```

```
    if n == 0:
       return Constant(0)
    elif n == 1:
         return e
    else:
        expr_so_far = e
        for i in range(n-1):
            expr_so_far = Plus(expr_so_far,e)
    return expr_so_far


print(big_constant_sum(5,Var("x")))
# print(big_constant_sum_it(5,Var("x")))
```

```
x + x + x + x + x
```

5. For example:

```
def big_sum(n,e):
    if n == 0:
        return e(0)
    else:
        return Plus(e(n),big_sum(n-1,e))

# tests
print(big_sum(5,lambda n: Var("x") if n == 1 else Var("y")))

def foo(n):
    if n == 0:
         return Constant(10)
    elif n == 1:
         return Var("x")
    elif n == 2:
         return Times(Var("z"),Var("y"))
    else:
         return Constant(n)

print(big_sum(10,foo))
```

```
y + y + y + y + x + y
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + z * y + x + 10
```

6. The changes we need to make are:

```
class Constant(Expr):
      def __init__(self, value):
          self._value = value

      def __str__(self):
          return str(self._value)
```

```
      def evaluate(self,d):
          return self._value

class Var(Expr):
      def __init__(self, name):
          self._name = name

      def __str__(self):
          return self._name

      def evaluate(self,d):
          return d[self._name]

# BinOp code unchanged...

class Plus(BinOp):
      prec = 1
      op = " + "

      def evaluate(self,d):
          return self._left.evaluate(d) + self._right.evaluate(d)

class Times(BinOp):
      prec = 2
      op = " * "

      def evaluate(self,d):
          return self._left.evaluate(d) * self._right.evaluate(d)

class Exp(BinOp):
      prec = 3
      op = " ** "

      def evaluate(self,d):
          return self._left.evaluate(d) ** self._right.evaluate(d)

vals = { 'x' : 2, 'y' : 3, 'z' : 5 }

# x * y + 7
e1 = Plus(Times(Var("x"),Var("y")),Constant(7))
# x * (y + 7)
e2 = Times(Var("x"),Plus(Var("y"),Constant(7)))
# x * y ** (3 + z)
e3 = Times(Var("x"),Exp(Var("y"),Plus(Constant(3),Var("z"))))

print(e1.evaluate(vals))
print(e2.evaluate(vals))
print(e3.evaluate(vals))
```

```
13
20
13122
```

7. For example:

```
# helper function to print lists the way we want
def to_string(xs):
    if xs == []:
        return '[]'
    elif len(xs) == 1:
        return '[' + str(xs[0]) + ']'
    else:
        out = ''
        for x in xs[:-1]:
            out += str(x) + ','
        return '[' + out + str(xs[-1]) + ']'

class RoseTree:
    def __init__(self,v,ts):
        self._val = v
        self._subtrees = ts

    def __str__(self):
        return ("(" + str(self._val) + "," +
                to_string([ str(x) for x in self._subtrees ]) + ")")

    def member(self,x):
        return (self._val == x or
                any (map(lambda v: v.member(x),self._subtrees)))

    def map_tree(self,f):
        return RoseTree(f(self._val),map(lambda v:v.map_tree(f),self._subtrees))

    def linearize(self):
        return ([self._val] +
                [ x for l in map(lambda v: v.linearize(),self._subtrees)
                    for x in l ])

# tests
t = RoseTree(-32,[RoseTree(2,[])
                 ,RoseTree(1,[RoseTree(23,[RoseTree(4,[])
                                          ,RoseTree(-2,[])])
                              ,RoseTree(12,[])])])
print(t)
print(t.member(12))
print(t.member(-123))
print(t.map_tree(lambda x: 0))
print(t.map_tree(lambda x: x ** 2))
```

```
print(t.linearize())
```

```
(-32,[(2,[]),(1,[(23,[(4,[]),(-2,[])]),(12,[])])])
True
False
(0,[(0,[]),(0,[(0,[(0,[]),(0,[])]),(0,[])])])
(1024,[(4,[]),(1,[(529,[(16,[]),(4,[])]),(144,[])])])
[-32, 2, 1, 23, 4, -2, 12]
```

A.12 Chapter 12

1. For example:

```
def jaccard_index(l1, l2):
    s1 = set(l1)
    s2 = set(l2)
    return float(len(s1.intersection(s2)) / len(s1.union(s2)))
```

2. Consider testing different special cases: empty lists, empty intersection, identical sets, incorrect arguments. When one tests functions, it is not only important to see that they are correct, but that they also raise errors when expect. Otherwise, when we use our functions in larger programs, we can introduce bugs such that the code runs without error on input data where there *should* be an error.

   If you have written your solutions in a file named `kap12.py`, create another file (say `test_kap12.py`), which could for example look like so:

```
import unittest
import kap12

class TestJaccard(unittest.TestCase):
    def test_identical_jaccard_index(self):
        l1 = [2,0,1]
        l2 = [2,1,0,0,1]
        self.assertEqual(kap12.jaccard_index(l1,l2),1.0, "Should be 1.0")

    def test_disjoint_jaccard_index(self):
        l1 = [1,2,3]
        l2 = [0,0,0]
        self.assertEqual(kap12.jaccard_index(l1,l2),0, "Should be 0")

    def test_empty_jaccard_index(self):
        l1 = [1,2,3]
        l2 = []
        self.assertEqual(kap12.jaccard_index(l1,l2),0, "Should be 0")

    def test_wrong_argument_jaccard_index(self):
        # check that jaccard_index fails with TypeError when
        # argument is not iterable
        with self.assertRaises(TypeError):
            l1 = [1,2,3]
            l2 = 1
            kap12.jaccard_index(l1,l2)

    def test_empty_lists_jaccard_index(self):
        # check that jaccard_index fails with TypeError when
        # argument is not iterable
        with self.assertRaises(ZeroDivisionError):
            l1 = []
            l2 = []
```

```
            kap12.jaccard_index(l1,l2)

unittest.main()
```

3. For example:

```
def make_weighted_operation(l1, l2, f):
    # returns total number of elements shared
    return sum([f(l1.count(item), l2.count(item)) for item in set(l1 + l2)])


def weighted_jaccard_index(l1, l2):
    # for intersection
    count_intersect = make_weighted_operation(l1, l2, min)
    # for union
    count_union = make_weighted_operation(l1, l2, max)
    return float(count_intersect) / count_union
```

The code above is however computationally expensive, as the method count must step through the whole list to count the number of occurences of the element `item` for each unique `item`. For a total number of elements n and m unique `items`, there will then be $m \cdot n$ iterations. The solution below saves the weights in a dictionary and thus needs only a few loops (7 function calls) all of size n. Under the following code, we have illustrated the time used by these two different solutions.

```
def make_weighted_set(l):
    w_set = {}
    for element in l:
         if element not in w_set:
             w_set[element] = 1
         else:
             w_set[element] += 1
    return w_set

def weighted_jaccard_index_faster(list1, list2):

    w_set1 = make_weighted_set(list1)
    w_set2 = make_weighted_set(list2)

    key_union = set(list(w_set1.keys()) + list(w_set2.keys()))
    min_total = 0
    max_total = 0
    for key in key_union:
        # get retrieves the element if present, otherwise returns the value
        # that is specified in the second argument (in this case 0)
        min_total += min(w_set1.get(key,0),w_set2.get(key,0))
        max_total += max(w_set1.get(key,0),w_set2.get(key,0))

    return min_total/max_total
```

```
from time import time
import random
l1 = [random.randint(1, 1000) for i in range(100000)]
l2 = [random.randint(1, 1000) for i in range(100000)]

t_start = time()
print(weighted_jaccard_index(l1, l2))
print("Elapsed:", time() - t_start)

t_start = time()
print(weighted_jaccard_index_faster(l1, l2))
print("Elapsed:", time() - t_start)
```

gives

```
0.8919685933213508
Elapsed: 4.24 seconds
0.8919685933213508
Elapsed: 0.02 seconds
```

```
from time import time
import random
l1 = [random.randint(1, 10000) for i in range(1000000)]
l2 = [random.randint(1, 10000) for i in range(1000000)]

t_start = time()
print(weighted_jaccard_index(l1, l2))
print("Elapsed:", time() - t_start)

t_start = time()
print(weighted_jaccard_index_faster(l1, l2))
print("Elapsed:", time() - t_start)
```

gives

```
Elapsed: 459.42 seconds
0.8938228234057563
Elapsed: 0.28 seconds
```

4. The test file can have the same structure as in Exercise 2, but remember to test all the functions separately. For example, for the first solution above:

```
import unittest
import kap12


class TestWeightedJaccard(unittest.TestCase):
    def test_identical_wjaccard_index(self):
        l1 = [2, 0, 1]
        l2 = [2, 1, 0, 0, 1]
        self.assertEqual(kap12.weighted_jaccard_index(
```

```
            l1, l2), 3/5, "Should be 3/5")

    def test_disjoint_wjaccard_index(self):
        l1 = [1, 2, 3]
        l2 = [0, 0, 0]
        self.assertEqual(kap12.weighted_jaccard_index(
            l1, l2), 0, "Should be 0")

    def test_empty_wjaccard_index(self):
        l1 = [1, 2, 3]
        l2 = []
        self.assertEqual(kap12.weighted_jaccard_index(
            l1, l2), 0, "Should be 0")

    def test_wrong_argument_wjaccard_index(self):
        # check that jaccard_index fails with TypeError when
        # argument is not iterable
        with self.assertRaises(TypeError):
            l1 = [1, 2, 3]
            l2 = 1
            kap12.weighted_jaccard_index(l1, l2)

    def test_empty_lists_wjaccard_index(self):
        # check that jaccard_index fails with TypeError when
        # argument is not iterable
        with self.assertRaises(ZeroDivisionError):
            l1 = []
            l2 = []
            kap12.weighted_jaccard_index(l1, l2)

    def test_make_weighted_operation_min(self):
        l1 = [1, 2, 3]
        l2 = [0, 0, 0]
        f = lambda x, y: min(x, y)
        self.assertEqual(kap12.make_weighted_operation(
            l1, l2, f), 0, "Should be 0")

    def test_make_weighted_operation_max(self):
        l1 = [1, 2, 3]
        l2 = [0, 0, 0]
        f = lambda x, y: max(x, y)
        self.assertEqual(kap12.make_weighted_operation(
            l1, l2, f), 6, "Should be 6")

    def test_make_weighted_operation_empty(self):
        l1 = []
        l2 = []
        f = lambda x, y: max(x, y)
```

```
        self.assertEqual(kap12.make_weighted_operation(
            l1, l2, f), 0, "Should be 6")

unittest.main()
```