Esta página se generó a partir de basics2.ipynb .

Open in Colab

Ejercicio 1 (enteros entre paréntesis)	Ejercicio 2 (listado de archivos)	Ejercicio 3 (rojo, verde, azul
Ejercicio 4 (frecuencias de palabras)	Ejercicio 5 (resumen)	Ejercicio 6 (recuento de arc
Ejercicio 7 (extensiones de archivo)	Ejercicio 8 (anteponer)	Ejercicio 9 (racional)
Ejercicio 10 (extraer números)		

Python (continúa)

Expresiones regulares

Ejemplos de

Ya hemos visto que podemos pedir de una cadena `str` si se comienza con un poco de subcadena de la siguiente manera: `str.startswith('Apple')` . Si quisiéramos saber si comienza con `"Apple"` o `"apple"` , tendríamos que llamar al `startswith` método dos veces. Las expresiones regulares ofrecen una solución más simple: . La notación entre corchetes es un ejemplo de la sintaxis especial de *las expresiones regulares* . En este caso, dice que cualquiera de los caracteres entre corchetes servirá: o . Las otras letras en actuarán normalmente. La cuerda se llama *patrón* . `re.match(r"[Aa]pple", str)` `"A"` `"a"` `"pple"` `r"[Aa]pple"`

Un ejemplo más complicado pregunta si la cadena `str` se inicia con cualquiera `apple` o `banana` (no importa si es la primera letra de capital o no): . En este ejemplo, vimos un nuevo carácter especial que denota una alternativa. A cada lado del carácter de barra tenemos un *subpatrón* . `re.match(r"[Aa]pple|[Bb]anana", str)` `|`

Un nombre de variable legal en Python comienza con una letra o un carácter de subrayado y los siguientes caracteres también pueden ser dígitos. Así nombres legales son, por ejemplo: `_hidden` , `L_value` , `A123_` . Pero el nombre `2abc` no es un nombre de variable válido. Vamos a ver

lo que sería el patrón de expresión regular para reconocer los nombres de variables válidos: `r"[A-Za-z_][A-Za-z_0-9]*\Z"`. Aquí hemos utilizado una forma rápida para rangos de caracteres: `A-Z`. Esto significa todos los caracteres desde `A` hasta `Z`.

El primer carácter del nombre de la variable se define entre los primeros corchetes. Los caracteres siguientes se definen en el segundo paréntesis. El carácter especial `*` significa que permitimos cualquier número (0,1,2,...) del subpatrón anterior. Por ejemplo el patrón `r"ba*"` permite cuerdas `"b"`, `"ba"`, `"baa"`, `"baaa"`, y así sucesivamente. La sintaxis especial `\Z` denota el final de la cadena. Sin él también lo aceptaríamos `abc-` como un nombre válido ya que la `match` función normalmente solo verifica que una cadena comience con un patrón.

Las notaciones especiales, como `\Z`, también causan problemas con el manejo de cadenas. Recuerde que normalmente en los literales de cadena tenemos una notación especial: `\n` significa nueva línea, `\t` significa tabulación, etc. Por lo tanto, tanto los literales de cadena como las expresiones regulares usan notaciones de aspecto similar, lo que puede crear una gran confusión. Esto se puede solucionar utilizando las denominadas *cadenas sin procesar* . Denotamos una cadena sin formato al tener una `r` letra antes de la primera comilla, por ejemplo `r"ab*\Z"`. Cuando se utilizan cadenas sin formato, la nueva línea (`\n`), tab (`\t`) y otras notaciones literales de cadenas especiales no se interpretan. ¡Siempre se deben usar cadenas sin procesar al definir patrones de expresión regular!

Patrones

A pattern represents a set of strings. This set can even be potentially infinite. They can be used to describe a set of strings that have some commonality; some regular structure. Regular expressions (RE) are a classical computer science topic. They are very common in programming tasks. Scripting languages, like Python, are very fluent in regular expressions. Very complex text processing can be achieved using regular expressions.

In patterns, normal characters (letters, numbers) just represent themselves, unless preceded by a backslash, which may trigger some special meaning. Punctuation characters have special meaning, unless preceded by backslash (`\`), which deprives their special meaning. Use `\\` to represent a backslash character without any special meaning. In the following slides we will go through some of the more common RE notations.

```
. Matches any character
[...] Matches any character contained within the brackets
[^...] Matches any character not appearing after the hat (^)
^ Matches the start of the string
$ Matches the end of the string
* Matches zero or more previous RE
+ Matches one or more previous RE
{m,n} Matches m to n occurences of previous RE
? Matches zero or one occurences of previous RE
```

We have already seen that a `|` character denotes alternatives. For example, the pattern `r"Get (on|off|ready)"` matches the following strings: `"Get on"`, `"Get off"`, `"Get ready"`. We can use parentheses to create groupings inside a pattern: `r"(ab)+"` will match the strings `"ab"`, `"abab"`, `"ababab"`, and so on. These groups are also given a reference number starting from 1. We can refer to groups using backreferences: `\number`. For example, we can find separated patterns that get repeated: `r"([a-z]{3,}) \1 \1"`. This will recognise, for example, the following strings: `"aca aca aca"`, `"turn turn turn"`. But not the strings `"aca aba aca"` or `"ac ac ac"`.

In the following, note that a hat (^) as the first character inside brackets will create a complement set of characters:

```
`\d` same as `[0-9]`, matches a digit
`\D` same as `[^0-9]`, matches anything but a digit
`\s` matches a whitespace character (space, newline, tab, ... )
`\S` matches a nonwhitespace character
`\w` same as `[a-zA-Z0-9_]`, matches one alphanumeric character
`\W` matches one non-alphanumeric character
```

Using the above notation we can now shorten our previous variable name example to `r'[a-zA-Z_]\w*\Z'`

The patterns `\A`, `\b`, `\B`, and `\Z` will all match an empty string, but in specific places. The patterns `\A` and `\Z` will recognise the beginning and end of the string, respectively. Note that the patterns `^` and `$` can in some cases match also after a newline and before a newline, correspondingly. So, `\A` is distinct from `^`, and `\Z` is distinct from `$`. The pattern `\b` matches at the start or end of a word. The pattern `\B` does the reverse.

Match and search functions

We have so far only used the `re.match` function which tries to find a match at the beginning of a string The function `re.search` allows to match any substring of a string. Example: `re.search(r'\bback\b', s)` will match strings `"back"`, `"a back, is a body part"`, `"get back"`. But it will not match the strings `"backspace"` or `"comeback"`.

The function `re.search` finds only the first occurence. We can use the `re.findall` function to find all occurences. Let's say we want to find all present participle words in a string `s`. The present participle words have ending `'ing'`. The function call would look like this: `re.findall(r'\w+ing\b', s)`. Let's try running this:

```
[1]: import re
```

```
s = "Doing things, going home, staying awake, sleeping later"
re.findall(r'\w+ing\b', s)
```

```
[1]: ['Doing', 'going', 'staying', 'sleeping']
```

Let's say we want to pick up all the integers from a string. We can try that with the following function call: `re.findall(r'[+-]?\d+', s)`. An example run:

```
[2]: re.findall(r'[+-]?\d+', "23 + -24 = -1")
```

```
[2]: ['23', '-24', '-1']
```

Suppose we are given a string of if/then sentences, and we would like to extract the conditions from these sentences. Let's try the following function call: `re.findall(r'[Ii]f (.*), then', s)`. An example run:

```
[3]: s = ("If I'm not in a hurry, then I should stay. " +
         "On the other hand, if I leave, then I can sleep.")
     re.findall(r'[Ii]f (.*), then', s)
```

```
[3]: ['I'm not in a hurry, then I should stay. On the other hand, if I leave']
```

But I wanted a result: `["I'm not in a hurry", 'I leave']`. That is, the condition from both sentences. How can this be fixed?

The problem is that the pattern `.*` tries to match as many characters as possible. This is called *greedy matching*. One way of solving this problem is to notice that the two sentences are separated by a full-stop (.). So, instead of matching all the characters, we need to match everything but the dot character. This can be achieved by using the complement character class: `[^.]`. The hat character (`^`) in the beginning of a character class means the complement character class

After the modification the function call looks like this: `re.findall(r'[Ii]f ([^.]*), then', s)`. Another way of solving this problem is to use a non-greedy matching. The repetition specifiers `+`, `*`, `?`, and `{m,n}` have corresponding non-greedy versions: `+?`, `*?`, `??`, and `{m,n}?`. These expressions use as few characters as possible to make the whole pattern match some substring. By using non-greedy version, the function call looks like this: `re.findall(r'[Ii]f (.*?), then', s)`.

Functions in the `re` module

Below is a list of the most common functions in the `re` module

- `re.match(pattern, str)`

- `re.search(pattern, str)`
- `re.findall(pattern, str)`
- `re.finditer(pattern, str)`
- `re.sub(pattern, replacement, str, count=0)`

Functions `match` and `search` return a *match object*. A match object describes the found occurence. The function `findall` returns a list of all the occurences of the pattern. The elements in the list are strings. The function `finditer` works like `findall` function except that instead of returning a list, it returns an iterator whose items are match objects. The function `sub` replaces all the occurences of the pattern in `str` with the string replacement and returns the new string.

An example: The following program will replace all “she” words with “he”

```
import re
str = "She goes where she wants to, she's a sheriff."
newstr = re.sub(r'\b[Ss]he\b', 'he', str)
print newstr
```

This will print `he goes where he wants to, he's a sheriff.`

The `sub` function can also use backreferences to refer to the matched string. The backreferences \1, \2, and so on, refer to the groups of the pattern, in order. An example:

```
import re
str = """He is the president of Russia.
He’s a powerful man."""
newstr = re.sub(r'(\b[Hh]e\b)', r'\1 (Putin)', str, 1)
print newstr
```

This will print

```
He (Putin) is the president of Russia.
He’s a powerful man.
```

Match object

Functions `match`, `search`, and `finditer` use `match` objects to describe the found occurence. The method `groups()` of the match object returns the tuple of all the substrings matched by the groups of the pattern. Each pair of parentheses in the pattern creates a new group. These groups

are are referred to by indices 1, 2, ... The group 0 is a special one: it refers to the match created by the whole pattern.

Let's look at the match object returned by the call

```
mo = re.search(r'\d+ (\d+) \d+ (\d+)',
'first 123 45 67 890 last')
```

The call `mo.groups()` returns a tuple `('45', '890')`. We can access just some individual groups by using the method `group(gid, ...)`. For example, the call `mo.group(1)` will return `'45'`. The zeroth group will represent the whole match: `'123 45 67 890'`

In addition to accessing the strings matched by the pattern and its groups, the corresponding indices of the original string can be accessed:

- The `start(gid=0)` and `end(gid=0)` methods return the start and end indices of the matched group gid, correspondingly
- The method `span(gid)` just returns the pair of these start and end indices

The match object mo can also be used like a boolean value:

```
mo = re.search(...)
if mo:
    # do something
```

will do something if a match was found. Alternatively, the match object can be converted to a boolean value by the call `found = bool(mo)`.

Miscellaneous stuff

If the same pattern is used in many function calls, it may be wise to precompile the pattern, mainly for efficiency reasons. This can be done using the `compile(pattern, flags=0)` function in the `re` module. The function returns a so-called RE object. The RE object has method versions of the functions found in module `re`. The only difference is that the first parameter is not the pattern since the precompiled pattern is stored in the RE object.

The details of matching operation can be specified using optional flags. These flags can be given either inside the pattern or as a parameter to the compile function. Some of the more common flags are given in the following table

x	Flag
`(?i)`	re.IGNORECASE
`(?m)`	re.MULTILINE
`(?s)`	re.DOTALL

The elements on the left can appear anywhere in the pattern but preferably in the beginning. On the right there are attributes of the re module that can be given to the compile function as the second parameter

The `IGNORECASE` flag makes lower- and uppercase characters appear as equal. The `MULTILINE` flag makes the special characters `^` and `$` match the beginning and end of each line in addition to the beginning and end of the whole string. These flags make `\A` differ from `^`, and `\Z` differ from `$`. The `DOTALL` flag makes the character class `.` (dot) also accept the newline character, in addition to all the other letters.

When giving multiple flags to the compile function, the flags can be separated with the `|` sign. For example, `re.compile(pattern, re.MULTILINE | re.DOTALL)`. This is equal to `re.compile('(?m)(?s)' + pattern)`.

Exercise 1 (integers in brackets)

Write function `integers_in_brackets` that finds from a given string all integers that are enclosed in brackets.

Example run: `integers_in_brackets("  afd [asd] [12 ] [a34]  [ -43 ]tt [+12]xxx")` returns `[12, -43, 12]`. So there can be whitespace between the number and the brackets, but no other character besides those that make up the integer.

Test your function from the `main` function.

Basic file processing

A file can be opened with the `open` function. The call `open(filename, mode="r")` will return a *file object*, whose type is `file`. This file object can be used to refer to a file on disk. For example, when we want to read from or write to a file, we can use the methods `read` and `write` of the file object. After the file object is no longer needed, a call to the `close` method should be made.

We can control what kind of operations we can perform on a file with the *mode* parameter of the `open` function. Different options include opening a file for reading or writing, whether the file exists already or needs be created with the call to the open method, etc. Here's a list of all the opening modes:

Mode	Description
`r`	read-only mode, file must exist
`w`	write-only mode, creates, or overwrites an existing file
`a`	write-only mode, write always appends to the end
`r+`	read/write mode, file must already exist
`w+`	read/write mode, creates, or overwrites an existing file
`a+`	read/write mode, write will append to end

In the end of the mode string either the letter `t` or `b` can be appended. These stand for text mode and binary mode. If this letter is not given, the file type is text mode by default.

For binary mode the contents of the file are not interpreted in any way, and the read and write methods handle bytes. (A byte consists of 8 bits and can be used to represent a number in the range 0 to 255.)

In the text mode two interpretations happen

- On Windows operating system the end of line in files is encoded by two characters. When the file is read these two charactes are converted to `'\n'` character. During writes to a file this conversion happens in the opposite direction.
- One character is encoded in the file as one or more bytes. This conversion happens automatically during read and write operations. One common encoding between bytes and characters is utf-8. In this encoding, the Finnish character `'ä'`, for example, is encoded as the following sequence of bytes:

```
[4]: "ä".encode("utf-8")
```

```
[4]: b'\xc3\xa4'
```

Above the two bytes were expressed as hexadecimals. In decimal notation they would be 195 and 164. (Both in the range from 0 to 255.)

```
[5]: list("ä".encode("utf-8"))            # Show as a list of integers
```

```
[5]: [195, 164]
```

What is the utf-8 encoding of the letter `'a'`?

During this course we will only consider files containing text, so the default text mode is fine for us. But we might sometimes have to specify the encoding of a file, if it is not the usual utf-8.

Some common file object methods

- `read(size)` will read size characters/bytes as a string
- `write(string)` will write string/bytes to a file
- `readline()` will read a string until and including the next newline character is met
- `readlines()` will return a list of all lines of a file
- `writelines()` will write a list of lines to a file
- `flush()` will try to make sure that the changes made to a file are written to disk immediately

[6]:
```
f = open("basics.ipynb", "r") # Let's open this notebook file,
                              # which is essentially a text file.
                              # So you can open it in a texteditor as well.

for i in range(5):            # And read the first five lines
    line = f.readline()
    print(f"Line {i}: {line}", end="")
f.close()
```

```
Line 0: {
Line 1:  "cells": [
Line 2:   {
Line 3:    "cell_type": "markdown",
Line 4:    "metadata": {},
```

It is easy to forget to close the file. One can use a *context manager* to solve this problem. A context manager is created with the `with` statement. After the indented block of the `with` statement exits, the file will be automatically closed.

[7]:
```
with open("basics.ipynb", "r") as f:          # the file will be automatically closed,
                                              # when the with block exits
    for i in range(5):
        line = f.readline()
        print(f"Line {i}: {line}", end="")
```

```
Line 0: {
Line 1:  "cells": [
Line 2:   {
Line 3:    "cell_type": "markdown",
Line 4:    "metadata": {},
```

The `file` object is iterable. This means that we can iterate through the lines in the file using a for loop, like in the below example:

[8]:
```
max_len = 0
with open("basics.ipynb", "r") as f:
    for line in f:    # iterates through all the lines in the file
        if len(line) > max_len:
            max_len = len(line)
print(f"The longest line in this file has length {max_len}")
```

```
The longest line in this file has length 1046
```

Standard file objects

Python has automatically three file objects open:

- `sys.stdin` for *standard input*
- `sys.stdout` for *standard output*
- `sys.stderr` for *standard error* To read a line from a user (keyboard), you can call `sys.stdin.readline()`. To write a line to a user (screen), call `sys.stdout.write(line)`. The standard error is meant for error messages only, even though its output often goes to the same destination as standard output.

The print function uses the file `sys.stdout` and input function uses the file `sys.stdin`. An example of usage:

[9]:
```
import sys
import random
i=random.randint(-10,10)
if i >= 0:
    sys.stdout.write("Got a positive integer.\n")
else:
    sys.stderr.write("Got a negative integer.\n")
```

```
Got a negative integer.
```

These standard file objects are meant to be a basic input/output mechanism in textual form. The destinations of the file objects can be changed to point somewhere else than the usual keyboard and screen. Very often these are redirected to some files. For example, it is usual to point the stderr to a file where all error messages are logged.

sys module

We saw above that the `sys` module contains the three file objects `sys.stdin`, `sys.stdout`, and `sys.stderr`. It has also few other useful attributes. The attribute `sys.path` is the list of folders that Python uses to look for imported modules. The list `sys.argv` contains the so called *command line parameters*. For example in Linux if you are using the terminal, then you can run your program with the command `python3 programname.py param1 param2 ...`. After Python has started your program, the command line parameters are visible as follows. The name of the

program is in `sys.argv[0]`. The rest of the command line parameters are after the program name in this list: `sys.argv[1]=="param1"`, `sys.argv[2]=="param2"`, and so on. The command line parameters can be useful in adjusting the behaviour of your program. A few examples of these will be in the following exercises. (The terminal window is a textual interface to your computer instead of the usual graphical interface.)

The function `sys.exit` can be used to exit immediately your program. The integer parameter given to this function is the return value of the program. Usually the return value 0 means that the program ran successfully, and non-zero integer means that an error occurred. This return value is accessible from the terminal window from where you started the program.

Exercise 2 (file listing)

The file `src/listing.txt` contains a list of files with one line per file. Each line contains seven fields: access rights, number of references, owner's name, name of owning group, file size, date, filename. These fields are separated with one or more spaces. Note that there may be spaces also within these seven fields.

Write function `file_listing` that loads the file `src/listing.txt`. It should return a list of tuples (size, month, day, hour, minute, filename). Use regular expressions to do this (either `match`, `search`, `findall`, or `finditer` method).

An example: for line

```
-rw-r--r-- 1 jttoivon hyad-all   25399 Nov  2 21:25 exception_hierarchy.pdf
```

the function should create the tuple `(25399, "Nov", 2, 21, 25, "exception_hierarchy.pdf")`.

Exercise 3 (red green blue)

The file `src/rgb.txt` contains names of colors and their numerical representations in RGB format. The RBG format allows a color to be represented as a mixture of red, green, and blue components. Each component can have an integer value in the range [0,255]. Each line in the file contains four fields: red, green, blue, and colorname. Each field is separated by some amount of

whitespace (tab or space in this case). The text file is formatted to make it print nicely, but that makes it harder to process by a computer. Note that some color names can also contain a space character.

Write function `red_green_blue` that reads the file `rgb.txt` from the folder `src`. Remove the irrelevant first line of the file. The function should return a list of strings. Clean-up the file so that the strings in the returned list have four fields separated by a single tab character (`\t`). Use regular expressions to do this.

The first string in the returned list should be:

```
'255\t250\t250\tsnow'
```

Exercise 4 (word frequencies)

Create function `word_frequencies` that gets a filename as a parameter and returns a dict with the word frequencies. In the dictionary the keys are the words and the corresponding values are the number of times that word occurred in the file specified by the function parameter. Read all the lines from the file and split the lines into words using the `split()` method. Further, remove punctuation from the ends of words using the `strip("""!"#$%&'()*,-./:;?@[]_""")` method call.

Test this function in the main function using the file `alice.txt`. In the output, there should be a word and its count per line separated by a tab:

```
The     64
Project 83
Gutenberg   26
EBook   3
of      303
```

Exercise 5 (summary)

This exercise can give two points at maximum!

Part 1.

Create a function called `summary` that gets a filename as a parameter. The input file should contain a floating point number on each line of the file. Make your function read these numbers and then return a triple containing the sum, average, and standard deviation of these numbers for the file. As a reminder, the formula for corrected sample standard deviation is $\sqrt{\frac{\sum_{i=1}^{n}(x_i-\overline{x})^2}{n-1}}$, where $\overline{x}$ is the average.

The `main` function should call the function summary for each filename in the list `sys.argv[1:]` of command line parameters. (Skip `sys.argv[0]` since it contains the name of the current program.)

Example of usage from the command line: `python3 src/summary.py src/example.txt src/example2.txt`

Print floating point numbers using six decimals precision. The output should look like this:

```
File: src/example.txt Sum: 51.400000 Average: 10.280000 Stddev: 8.904606
File: src/example2.txt Sum: 5446.200000 Average: 1815.400000 Stddev: 3124.294045
```

Part 2.

If some line doesn't represent a number, you can just ignore that line. You can achieve this with the *try-except* block. An example of recovering from an exceptional situation:

```
try:
    x = float(line)           # The float constructor raises ValueError exception if conversion is
no possible
except ValueError:
    # Statements in here are executed when the above conversion fails
```

We will cover more about exceptions later in the course.

Exercise 6 (file count)

This exercise can give two points at maximum!

Part 1.

Create a function `file_count` that gets a filename as parameter and returns a triple of numbers. The function should read the file, count the number of lines, words, and characters in the file, and return a triple with these count in this order. You get division into words by splitting at whitespace. You don't have to remove punctuation.

Part 2.

Create a main function that in a loop calls `file_count` using each filename in the list of command line parameters `sys.argv[1:]` as a parameter, in turn. For call `python3 src/file_count file1 file2 ...` the output should be

```
?       ?       ?       file1
?       ?       ?       file2
...
```

The fields are separated by tabs (`\t`). The fields are in order: linecount, wordcount, charactercount, filename.

Exercise 7 (file extensions)

This exercise can give two points at maximum!

Part 1.

Write function `file_extensions` that gets as a parameter a filename. It should read through the lines from this file. Each line contains a filename. Find the extension for each filename. The function should return a pair, where the first element is a list containing all filenames with no extension (with the preceding period (`.`) removed). The second element of the pair is a dictionary with extensions as keys and corresponding values are lists with filenames having that extension.

Sounds a bit complicated, but hopefully the next example will clarify this. If the file contains the following lines

```
file1.txt
mydocument.pdf
file2.txt
archive.tar.gz
test
```

then the return value should be the pair:

```
(["test"], { "txt" : ["file1.txt", "file2.txt"], "pdf" : ["mydocument.pdf"], "gz" : ["archive.tar.gz"] } )
```

Part 2.

Write a `main` method that calls the `file_extensions` function with "src/filenames.txt" as the argument. Then print the results so that for each extension there is a line consisting of the extension and the number of files with that extension. The first line of the output should give the number of files without extensions.

With the example in part 1, the output should be

```
1 files with no extension
gz 1
pdf 1
txt 2
```

Had there been no filenames without extension then the first line would have been `0 files with no extension`. In the printout list the extensions in alphabetical order.

Objects and classes

Python is an object-oriented programming language like Java and C++. But unlike Java, Python doesn't force you to use classes, inheritance and methods. If you like, you can also choose the structural programming paradigm with functions and modules.

Every value in Python is an object. Objects are a way to combine data and the functions that handle that data. This combination is called *encapsulation*. The data items and functions of objects are called *attributes*, and in particular the function attributes are called *methods*. For example, the operator `+` on integers calls a method of integers, and the operator `+` on strings calls a method of strings.

Functions, modules, methods, classes, etc are all first class objects. This means that these objects can be

- stored in a container
- passed to a function as a parameter
- returned by a function
- bound to a variable

One can access an attribute of an object using the *dot operator*: `object.attribute`. For example: if `L` is a list, we can refer to the method `append` with `L.append`. The method call can look, for instance, like this: `L.append(4)`. Because also modules are objects in Python, we can interpret the expression `math.pi` as accessing the data attribute `pi` of module object `math`.

Numbers like 2 and 100 are instances of type `int`. Similarly, `"hello"` is an instance of type `str`. When we write `s=set()`, we are actually creating a new instance of type `set`, and bind the resulting instance object to `s`.

A user can define his own data types. These are called *classes*. A user can call these classes like they were functions, and they return a new instance object of that type. Classes can be thought as recipes for creating objects.

An example of class definition:

```
class MyClass(object):
    """Documentation string of the class"""

    def __init__(self, param1, param2):
        "This initialises an instance of type ClassName"
        self.b = param1 # creates an instance attribute
        c = param2      # creates a local variable of the function
        # statements ...

    def f(self, param1):
        """This is a method of the class"""
        # some statements

    a=1 # This creates a class attribute
```

The class definition starts with the `class` statement. With this statement you give a name for your new type, and also in parentheses list the base classes of your class. The next indented block is the *class body*. After the whole class body is read, a new type is created. Note that no instances are created yet. All the attributes and methods of the class are defined in the class body.

The example class has two methods: `__init__` and `f`. Note that their first parameter is special: `self`. It corresponds to `this` variable of C++ or Java. `__init__` does the initialisation when an instance is created. At instantiation with `i=MyClass(2,3)` the parameters `param1` and `param2` are

bound to values 2 and 3, respectively. Now that we have an instance `i`, we can call its method `f` with the dot operator: `i.f(1)`. The parameters of `f` are bound in the following way: `self=i` and `param1=1`.

There are differences in how an assignment inside a class body creates variables. The attribute `a` is at class level and is common for all instances of the class `MyClass`. The variable `c` is a local variable of the function `__init__`, and cannot therefore be used outside the function. The attribute `b` is specific to each instance of `MyClass`. Note that `self` refers to the current instance. An example: for objects `x=MyClass(1,0)` and `y=MyClass(2,0)` we have `x.b != y.b`, but `x.a == y.a`.

All methods of a class have a mandatory first parameter which refers to the instance on which you called the method. This parameter is usually named `self`. If you want to access the class attribute `a` from a method of the class, use the fully qualified form `MyClass.a`. The methods whose names both begin and end with two underscores are called *special methods*. For example, `__init__` is a special method. These methods will be discussed in detail later.

Instances

We can create instances by calling a class like it were a function: `i = ClassName(...)`. Then parameters given in the call will be passed to the `__init__` function. In the `__init__` method you can create the instance specific attributes. If `__init__` is missing, we can create an instance without giving any parameters. As a consequence, the instance has no attributes. Later you can (re)bind attributes with the assignment `instance.attribute = new value`.

If that attribute did not exist before, it will be added to the instance with the assigned value. In Python we really can add or delete attributes to/from an existing instance. This is possible because the attribute names and the corresponding values are actually stored in a dictionary. This dictionary is also an attribute of the instance and is called `dict`. Another standard attribute in addition to dict is called `__class__`. This attribute stores the class of the instance. That is, the type of the object

Attribute lookup

Suppose `x` is an instance of class `X`, and we want to read an attribute `x.a`. The lookup has three phases:

- First it is checked whether the attribute `a` is an attribute of the instance `x`
- If not, then it is checked whether `a` is a class attribute of `x`'s class `X`
- If not, then the base classes of `X` are checked

If instead we want to bind the attribute `a`, things are much simpler. `x.a = value` will set the instance attribute. And `X.a = value` will set the class attribute. Note that if a base of `X`, the class `X`, and the instance `x` each have an attribute called `a`, then `x.a` hides `X.a`, and `X.a` hides the attribute of the base class.

Exercise 8 (prepend)

Create a class called `Prepend`. We create an instance of the class by giving a string as a parameter to the initializer. The initializer stores the parameter in an instance attribute `start`. The class also has a method `write(s)` which prints the string `s` prepended with the `start` string. An example of usage:

```
p = Prepend("+++ ")
p.write("Hello");
```

Will print

```
+++ Hello
```

Try out using the class from the `main` function.

Inheritance

Inheritance allows us to reuse the code of an existing class `B` in creating a new class `C`. Let's recap how the attribute lookup worked for classes. When looking for an attribute, the lookup procedure starts with the instance dictionary, and continues with the class attributes. If both fail, then the attribute is searched from the base classes and, recursively, from their base classes.

So, it may look like we access an attribute of a class `C`, when in reality we are accessing the attribute of its base class `B`. In this case we say that the class `C` *inherits* the attribute from its base class `B`. If we have attributes with the same name in both the class and its base class, the attribute of the base class is hidden. We say that the class `C` overrides the attribute of the base class `B`. Terminology: `B` is a base class and `C` is a derived class.

Example:

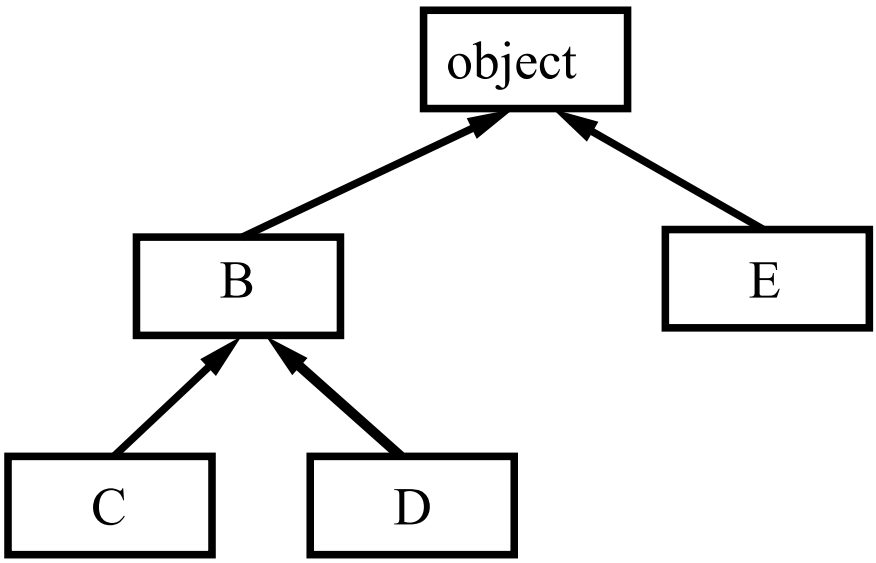
[10]:
```
class B(object):
    def f(self):
        print("Executing B.f")
    def g(self):
        print("Executing B.g")

class C(B):
    def g(self):
        print("Executing C.g")

x=C()
x.f() # inherited from B
x.g() # overridden by C
```

```
Executing B.f
Executing C.g
```

A derived class is sometimes also called a *subclass* and the base class is called *super class*. The inheritance relation of two classes `B` and `C` can be tested with function `issubclass`: `issubclass(C,B)==True` but `issubclass(B,C)==False` Function `isinstance(obj, cls)` allows us to test whether an instance has type `cls` or has an ancestor class of type `cls`. Let's create instances `x=C()` and `y=B()`. Now we have `isinstance(x,B)== isinstance(x,C)==isinstance(y,B)==True`. But `isinstance(y,C)==False`.



`object` should be a base class or an ancestor class of every other class. This means that `isinstance(x, object)==True` for all instances `x`.

By deriving from an existing class we can modify and/or extend its behaviour, without touching the original class. For example, if we want to add one method to a list class, we can use inheritance. Therefore we have to only code the part that has changed and reuse the rest of the code of type list. Another use of inheritance is to create conceptual hierarchies. For instance, later we will learn about the exception hierarchy of Python. Third use would be to use classes to

create interfaces. There can be several classes that have same interface (that is, they offer the same attributes), but their behaviour or implementation can be very different. This allows changing a part of your program with minimal changes required elsewhere in the code.

If in the definition of the method `C.f` we need to call the corresponding method of class `A`, we can use the fully qualified call `A.f(...)`. This is called delegation. It is useful, for instance, when you want to call the init method of the base class from the init of the derived class to initialise the base class attributes.

Special methods

We have already encountered one special method, namely the `__init__` method. This method sets the instance attributes to some initial value. Its first parameter is `self`, and the subsequent parameters are the ones that were passed to the call of the class. The `__init__` method should return no value. Next the main general purpose special methods are introduced. They are executed when certain operations on objects are performed.

In the following, `C` is a class and `x` and `y` are its instances. `__hash__` returns an int value, with the following requirement: `x==y` implies `x.__hash__() == y.__hash__()`. The value is used in storing objects in dictionaries and sets. The instances `x` and `y` must be immutable A class with `__call__` method makes its instances callable. I.e. the call `x(a,b, ...)` will result in calling this special method with the given parameters. The method `__del__` gets called when the corresponding instance gets deleted. Method `__new__` is used to control the creation of new instances. It can be used, for example, to create classes that have only one instance.

The method `__str__` is called when the print statement needs to print the value of an instance. It returns a string. The print-format expression calls this for conversion `%s`. The method `__repr__` is called when the interactive interpreter prints the value of an evaluated expression, and when the conversion `%r` for print-format expression is used. Returns a canonical representation string that (at least in theory) can be used to recreate the original object. Special methods `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, and `__ne__` get called when the corresponding operators `x==y`, `x>=y`, `x>y`, `x<=y`, `x<y`, and `x!=y` are used.

If you want the instances of your class to support the numeric operations (like +, -, *, /, etc), you must define a set of special methods in you class. For example, the expression `x+y` will result in a call `x.__add__(y)` which should return the result of the operation. Here are a few of the most common numerical special methods:

Method	Description
`__add__`	addition (+)

Method	Description
`__sub__`	subtraction (-)
`__mul__`	multiplication (*) | | `__truediv__` | division (/) | | `__floordiv__` | division (//) | ————————

The corresponding augmented assignments += -= *= /= have special methods iadd , isub , imul , idiv. The conversion functions complex(), float(), int() and long() call the following special methods:

Method	Description
`__complex__`	convert to a complex number
`__float__`	convert to a float
`__int__`	convert to an integer

In addition to the normal methods of containers, like the `append` method of the list, there are several operations that are handled by calls to special methods of the container class. The test whether `x` is a member of container `c` is done by the operation `x in c`. The corresponding special method call is `x.__contains__(y)`. Deletion of an element of container `c` can be done with the operation `del c[key]`. This will result in the method call `x.__delitem__`.

Reading an item of a container `c` is done with the operation `c[key]`. The corresponding method call is `c.__getitem__(key)`. Similarly, setting an item with `c[key]=value` results in the call `c.__setitem__(key,value)`. The number of elements in a container `c` can be queried with the function call `len(c)`. This function call actually calls the special method `c.__len__`. The call `iter(c)` will call the special method `__iter__`.

Exercise 9 (rational)

Create a class `Rational` whose instances are rational numbers. A new rational number can be created with the call to the class. For example, the call `r=Rational(1,4)` creates a rational number “one quarter”. Make the instances support the following operations: `+` `-` `*` `/` `<` `>` `==` with their natural behaviour. Make the rationals also printable so that from the printout we can clearly see that they are rational numbers.

Exceptions

When an error occurs, what can we do?

- Print an error message
- Stop the execution of a program
- Indicate the error by returning a special value, like -1 or None
- Ignore the error
- ...

These solutions tend to combine the indication of a problem and the reaction to the problem indication. The behaviour of the program in error situations cannot the changed, they are fixed in the implementation of the function. When an erroneous situation is noticed, it may not be clear how to handle the situation. Usually the user or an instance that called a function knows what to do.

Most modern computer languages have a system called *exception handling*. This system separates the recognition of errors and the handling of these situations. We can signal an error or anomalous situation by *raising* an exception. Exceptions can be raised in Python with the `raise` statement:

- `raise` instance
- `raise` exception class [, expression]

In the second form, if the expression exists, it is a tuple of parameters given to exception class.

The functions of the Python standard library raise exceptions in error situations. Sometimes exceptions aren't really errors. For example, when an iterator runs out of elements, it will signal this by raising the `StopIteration` exception. Another less erroneus exception is the `Warning` exception.

The general form of exception catching statement is the following:

```
try:
    # here are the statements that can cause exceptions
except (Exceptionname1, Exceptionname2, ...):
    # here we handle the exceptions
else:
    # this gets executed if try-block caused no exceptions
finally:
    # this is always executed, clean-up code
```

Usually, just the try and except parts are needed.

[11]:
```
L=[1,2,3]
```

```
try:
    print(L[3])
except IndexError:
    print("Index does not exist")
```

```
Index does not exist
```

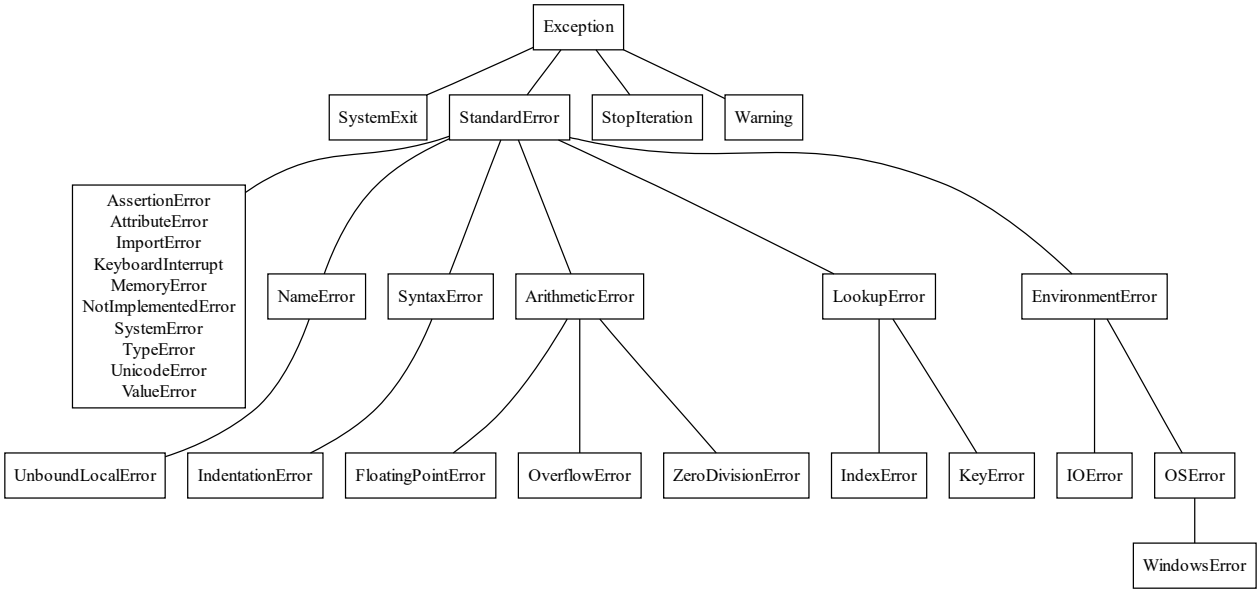
[12]:
```
def compute_average(L):
    n=len(L)
    s=sum(L)
    return float(s)/n # error is noticed here !!!
mylist=[]
while True:
    try:
        x=float(input("Give a number (non-number quits): "))
        mylist.append(x)
    except ValueError:
        break
try:
    average=compute_average(mylist)
    print("Average is", average)
except ZeroDivisionError:
    # and the error is handled here
    if len(mylist) == 0:
        print("Tried to compute the average of empty list of numbers")
    else:
        print("Something strange happened")
```

```
Give a number (non-number quits): 1
Give a number (non-number quits):
Average is 1.0
```

Exception hierarchy

In Python exceptions are objects, like all values in Python. These objects are instantiated from exception classes. Exception classes form naturally hierarchies:

- New exception classes can be made by inheriting from existing exception classes and extending them
- The root of this hierarchy is the class `Exception`
- Python defines several base classes to derive from, and several ready-to-use exception classes



Too general exception specifications

The exception hierarchy allows to catch multiple similar exceptions by catching their common base class. This feature has to be used carefully. Over-general exception specification, like `except Exception:`, can hide the real reason for an error. Example of this:

[13]:
```
import sys
s=input("Give a number: ")
s=s[:-1] # strip the \n character from the end
try:
    x=int(s)
    sys.stdout.wr1te(f"You entered {x}\n")
except Exception:
    print("You didn't enter a number")
```

```
Give a number: 1
You didn't enter a number
```

In the previous example, if the user doesn't enter a string that represents an integer, a `ValueError` is raised by the `int` function. Instead of catching the `ValueError`, we catch the root of the exception hierarchy, namely `Exception`. This results in catching all possible exceptions. But this will cause one typing error in the program to go undetected. Change the exception specification from `Exception` to `ValueError` to see what this error is.

What is the error handling policy in Python

Python uses a different approach to error checking than many other common languages. Instead of trying to beforehand check that all the inputs are of correct type and then contents of input variables are sensible for some operations, Python first tries the operations and then checks

whether they caused any exceptions. This is partly what duck typing is about: a function works for a set of inputs if all the operations in the function body make sense for those inputs. So, that's why the parameters of functions aren't specified to be of any certain type.

Exercise 10 (extract numbers)

Write a function `extract_numbers` that gets a string as a parameter. It should return a list of numbers that can be both ints and floats. Split the string to words at whitespace using the `split()` method. Then iterate through each word, and initially try to convert to an int. If unsuccesful, then try to convert to a float. If not a number then skip the word.

Example run: `print(extract_numbers("abd 123 1.2 test 13.2 -1"))` will return `[123, 1.2, 13.2, -1]`

Sequences, iterables, generators: revisited

In simple terms, a container is *iterable*, if we can go through all its elements using a `for` loop. All the sequences are iterable, but there are other iterable objects as well. We can even create iterable types ourselves. In our class there needs to be a special method `__iter__` that returns an *iterator* for the container. An iterator is an object that has method `__next__`, which returns the next element from the container. Let's have a look at a simple example where the container and its iterator is the same class.

[14]:
```
class WeekdayIterator(object):
    """Iterator over the weekdays."""
    def __init__(self):
        self.i=0           # Start from Monday
        self.weekdays =
("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday")
    def __iter__(self):    # If this object were a container, then this method would return
the iterator over the
                           # elements of the container.
        return self        # However, this object is already an iterator, hence we return
self.
    def __next__(self):    # Returns the next weekday
        if self.i == 7:
            raise StopIteration # Signal that all weekdays were already iterated over
        else:
            weekday = self.weekdays[self.i]
            self.i += 1
            return weekday

for w in WeekdayIterator():
    print(w)
```

```
Monday
Tuesday
Wednesday
```

```
Thursday
Friday
Saturday
Sunday
```

We can now check whether the WeekdayIterator is a Sequence type:

[15]:
```
from collections import abc  # Get the abstract base classes
containers = ["efg", [1,2,3], (4,5), WeekdayIterator()]
for c in containers:
    if isinstance(c, abc.Sequence):
        print(c, "is a sequence")
    else:
        print(c, "is not a sequence")
```

```
efg is a sequence
[1, 2, 3] is a sequence
(4, 5) is a sequence
<__main__.WeekdayIterator object at 0x7f343c3d6898> is not a sequence
```

Weekday is not a sequence because, for instance, you cannot index it with the brackets `[]`, but it is an iterable:

[16]:
```
isinstance(WeekdayIterator(), abc.Iterable)
```

[16]:
```
True
```

So it is possible to create iterators ourselves, but the syntax was quite complicated. There is an easier option using *generators*. A generator is a function that contains a `yield` statement. Note the difference between generators and generator expressions we saw in the first week. Both however produce iterables. Here's an example of a generator:

[17]:
```
def mydate(day=1, month=1):   # Generates dates starting from the given date
    lengths=(31,28,31,30,31,30,31,31,30,31,30,31)   # How many days in a month
    first_day=day
    for m in range(month, 13):
        for d in range(first_day, lengths[m-1] + 1):
            yield (d, m)
        first_day=1
# Create the generator by calling the function:
gen = mydate(26, 2)   # Start from 26th of February
for i, (day, month) in enumerate(gen):
    if i == 5: break                # Print only the first five dates from the generator
    print(f"Index {i}, day {day}, month {month}")
```

```
Index 0, day 26, month 2
Index 1, day 27, month 2
Index 2, day 28, month 2
Index 3, day 1, month 3
Index 4, day 2, month 3
```

Tenga en cuenta que no sería posible escribir el iterable anterior usando una expresión generadora, y hubiera sido muy torpe escribirlo explícitamente como un iterador como hicimos con el `WeekdayIterator`. La siguiente figura muestra las relaciones entre los diferentes iterables que hemos visto: