Applied Python

by

Bernd Klein

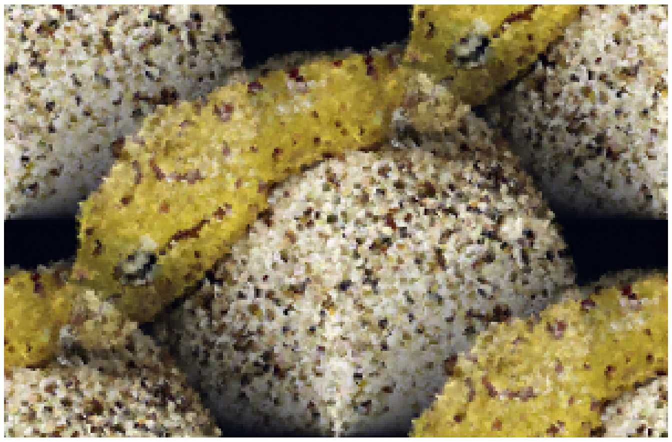
bodenseo

Python Course Applied Python Programming by Bernd Klein

PYTHON AND THE SHELL

SHELL

Shell is a term, which is often used and often misunderstood. Like the shell of an egg, either hen or Python snake, or a mussel, the shell in computer science is generally seen as a piece of software that provides an interface for a user to some other software or the operating system. So the shell can be an interface between the operating system and the services of the kernel of this operating system. But a web browser or a program functioning as an email client can be seen as shell as well.



Understanding this, it's obvious that a shell can be either

- a command-line interface (CLI) or
- a graphical user interface (GUI)

But in most cases the term shell is used as a synonym for a command line interface (CLI). The best known and most often used shells under Linux and Unix are the Bourne-Shell, C-Shell or Bash shell. The Bourne shell (sh) was modelled after the Multics shell, and is the first Unix shell. Most operating system shells can be used in both interactive and batch mode.

SYSTEM PROGRAMMING AND PYTHON

System programming (also known as systems programming) stands for the activity of programming system components or system software. System programming provides software or services to the computer hardware, while application programming produces software which provides tools or services for the user.

"System focused programming" serves as an abstraction layer between the application, i.e. the Python script or program, and the operating system, e.g. Linux or Microsoft Windows. By means of such an abstraction layer it is possible to implement platform independent applications in Python, even if they access operating specific functionalities. Python provides various modules to interact with the operating system, such as

- os
- platform
- subprocess
- shutils
- glob
- sys

Therefore Python is well suited for system programming, or even platform independent system programming. This is one of the reasons by the way why Python is wrongly considered by many as a scripting language. The proper way to say it: Python is a full-fledged programming language which can be easily used as a scripting language. The general advantages of Python are valid in system focused programming as well:

- simple and clear
- wel structured
- highly flexible

OS MODULE

The os module is the most important module for interacting with the operating system. The os module allows platform independent programming by providing abstract methods. Nevertheless it is also possible by using the system() and the exec*() function families to include system independent program parts. (Remark: The exec*()-Functions are introduced in detail in our chapter "Forks and Forking in Python" The os module provides various methods, e.g. the access to the file system.

Platform independant application often need to know on which platform the program or script is running. The `os` module provides with `os.name` the perfect command for this purpose:

```
import os

print(os.name)
```

```
posix
```

The output is of course dependant on the the operating system you are running. As of Python version 3.8 the following names are registered: `posix`, `nt`, `java`.

These names define the following operating systems:

posix
Unix-like operating systems like Unix, Linux, BSD, Minix and others.

nt
Windows systems like "Windows 10", "Windows 8.1", "Windows 8", "Windows 7" and so on.

java
Java operating system.

Most Python scripts dealing with the operating system need to know the postion in the file system. The function os.getcwd() returns a unicode string representing the current working directory.

```
print(os.getcwd())
```

```
/home/bernd/Dropbox (Bodenseo)/notebooks/advanced_topics
```

You also need the capability to move around in your filesystem. For this purpose the module `os` provides the function `chdir`. It has a parameter path, which is specified as a string. If called it will change the current working directory to the specified path.

```
os.chdir("/home/bernd/dropbox/websites/python-course.eu/cities")
print(os.getcwd())
```

```
/home/bernd/Dropbox (Bodenseo)/websites/python-course.eu/cities
```

After you have reached the desired directory, you may want to get its content. The function `listdir` returns a list containing the names of the files of this directory.

```
print(os.listdir())
```

```
['Freiburg.png', 'Stuttgart.png', 'Hamburg.png', 'Berlin.png', 'Ba
sel.png', 'Nürnberg.png', 'Erlangen.png', 'index.html', 'Ulm.pn
g', 'Singen.png', 'Bremen.png', 'Kassel.png', 'Frankfurt.png', 'Sa
arbrücken.png', 'Zürich.png', 'Konstanz.png', 'Hannover.png']
```

EXECUTING SHELL SCRIPTS WITH OS.SYSTEM()

It's not possible in Python to read a character without having to type the return key as well. On the other hand this is very easy on the Bash shell. The Bash command `read -n 1` waits for a key (any key) to be typed. If you import os, it's easy to write a script providing `getch()` by using `os.system()` and the Bash shell. `getch()` waits just for one character to be typed without a return:

```
import os
def getch():
     os.system("bash -c \"read -n 1\"")

getch()
```

The script above works only under Linux. Under Windows you will have to import the module msvcrt. Pricipially we only have to import getch() from this module. So this is the Windows solution of the problem:

```
from msvcrt import getch
```

The following script implements a platform independent solution to the problem:

```
import os, platform
if platform.system() == "Windows":
    import msvcrt
def getch():
    if platform.system() == "Linux":
        os.system('bash -c "read -n 1"')
    else:
        msvcrt.getch()

print("Type a key!")
getch()
```

```
print("Okay")
```

```
Type a key!
Okay
```

The previous script harbours a problem. You can't use the getch() function, if you are interested in the key which has been typed, because os.system() doesn't return the result of the called shell commands.

We show in the following script, how we can execute shell scripts and return the output of these scripts into python by using os.popen():

```
import os
dir = os.popen("ls").readlines()

print(dir)
```

```
['Basel.png\n', 'Berlin.png\n', 'Bremen.png\n', 'Erlangen.png\n',
'Frankfurt.png\n', 'Freiburg.png\n', 'Hamburg.png\n', 'Hannover.pn
g\n', 'index.html\n', 'Kassel.png\n', 'Konstanz.png\n', 'Nürnber
g.png\n', 'Saarbrücken.png\n', 'Singen.png\n', 'Stuttgart.png\n',
'Ulm.png\n', 'Zürich.png\n']
```

The output of the shell script can be read line by line, as can be seen in the following example:

```
import os

command = " "
while (command != "exit"):
    command = input("Command: ")
    handle = os.popen(command)
    line = " "
    while line:
        line = handle.read()
        print(line)
    handle.close()

print("Ciao that's it!")
```

```
Basel.png
Berlin.png
Bremen.png
Erlangen.png
Frankfurt.png
Freiburg.png
Hamburg.png
Hannover.png
index.html
Kassel.png
Konstanz.png
Nürnberg.png
Saarbrücken.png
Singen.png
Stuttgart.png
Ulm.png
Zürich.png



Ciao that's it!
```

SUBPROCESS MODULE

The subprocess module is available since Python 2.4. It's possible to create spawn processes with the module subprocess, connect to their input, output, and error pipes, and obtain their return codes. The module subprocess was created to replace various other modules:

- os.system
- os.spawn*
- os.popen*
- popen2.*
- commands.*

WORKING WITH THE SUBPROCESS MODULE

Instead of using the system method of the os-Module

```
os.system('Konstanz.png')
```

Output:
```
32256
```

we can use the Popen() command of the subprocess Module. By using Popen() we are capable to get the output of the script:

```
import subprocess
```

```
x = subprocess.Popen(['touch', 'xyz'])
print(x)
print(x.poll())
print(x.returncode)
```

```
<subprocess.Popen object at 0x7f92c5319290>
None
None
```

The shell command `cp -r xyz abc` can be send to the shell from Python by using the Popen() method of the subprocess-Module in the following way:

```
p = subprocess.Popen(['cp','-r', "Zürich.png", "Zurich.png"])
```

There is no need to escape the Shell metacharacters like `$, >, ...` and so on. If you want to emulate the behaviour of `` `os.system ``, the optional parameter shell has to be set to `True` and we have to use a string instead of a list:

```
p = subprocess.Popen("cp -r xyz abc", shell=True)
```

As we have mentioned above, it is also possible to catch the output from the shell command or shell script into Python. To do this, we have to set the optional parameter `stdout` of `Popen()` to `subprocess.PIPE`:

```
process = subprocess.Popen(['ls','-l'], stdout=subprocess.PIPE)
directory_content = process.stdout.readlines()
print(directory_content)
```

```
[b'total 0\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:32 abc\n',
b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Basel.png\n', b'-rw-r--
r-- 1 bernd bernd 0 Jan 23 08:07 Berlin.png\n', b'-rw-r--r-- 1 ber
nd bernd 0 Jan 23 08:07 Bremen.png\n', b'-rw-r--r-- 1 bernd bernd
0 Jan 23 08:07 Erlangen.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 2
3 08:07 Frankfurt.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:0
7 Freiburg.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Hambu
rg.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Hannover.png\
n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:05 index.html\n', b'-r
w-r--r-- 1 bernd bernd 0 Jan 23 08:07 Kassel.png\n', b'-rw-r--r--
1 bernd bernd 0 Jan 23 08:07 Konstanz.png\n', b'-rw-r--r-- 1 bern
d bernd 0 Jan 23 08:07 N\xc3\xbcrnberg.png\n', b'-rw-r--r-- 1 bern
d bernd 0 Jan 23 08:07 Saarbr\xc3\xbccken.png\n', b'-rw-r--r-- 1 b
ernd bernd 0 Jan 23 08:07 Singen.png\n', b'-rw-r--r-- 1 bernd bern
d 0 Jan 23 08:07 Stuttgart.png\n', b'-rw-r--r-- 1 bernd bernd 0 Ja
n 23 08:07 Ulm.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:28 x
yz\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:29 Zurich.png\n',
b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Z\xc3\xbcrich.png\n']
```

The `b` indicates that what you have is bytes, which is a binary sequence of bytes rather than a string of Unicode characters. Subprocesses output bytes, not characters. We can turn it to unicode string with the following list comprehension:

```
directory_content = [el.decode() for el in directory_content]
print(directory_content)
```

```
['total 0\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:32 abc\n', '-r
w-r--r-- 1 bernd bernd 0 Jan 23 08:07 Basel.png\n', '-rw-r--r-- 1
bernd bernd 0 Jan 23 08:07 Berlin.png\n', '-rw-r--r-- 1 bernd bern
d 0 Jan 23 08:07 Bremen.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 2
3 08:07 Erlangen.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07
Frankfurt.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Freibur
g.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Hamburg.png\
n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Hannover.png\n', '-r
w-r--r-- 1 bernd bernd 0 Jan 23 08:05 index.html\n', '-rw-r--r--
1 bernd bernd 0 Jan 23 08:07 Kassel.png\n', '-rw-r--r-- 1 bernd be
rnd 0 Jan 23 08:07 Konstanz.png\n', '-rw-r--r-- 1 bernd bernd 0 Ja
n 23 08:07 Nürnberg.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 0
8:07 Saarbrücken.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07
Singen.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Stuttgar
t.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Ulm.png\n', '-r
w-r--r-- 1 bernd bernd 0 Jan 23 08:28 xyz\n', '-rw-r--r-- 1 bernd
bernd 0 Jan 23 08:29 Zurich.png\n', '-rw-r--r-- 1 bernd bernd 0 Ja
n 23 08:07 Zürich.png\n']
```

FUNCTIONS TO MANIPULATE PATHS, FILES AND DIRECTORIES

Function	Description
getcwd()	returns a string with the path of the current working directory

chdir(path)	Change the current working directory to path.
	Example under Windows:
	`>>> os.chdir("c:\Windows")`
	`>>> os.getcwd()`
	`'c:\\Windows'`
	A similiar example under Linux:
	`>>> import os`
	`>>> os.getcwd()`
	`'/home/homer'`
	`>>> os.chdir("/home/lisa")`
	`>>> os.getcwd()`
	`'/home/lisa'`
	`>>>`

getcwdu()	like getcwd() but unicode as output

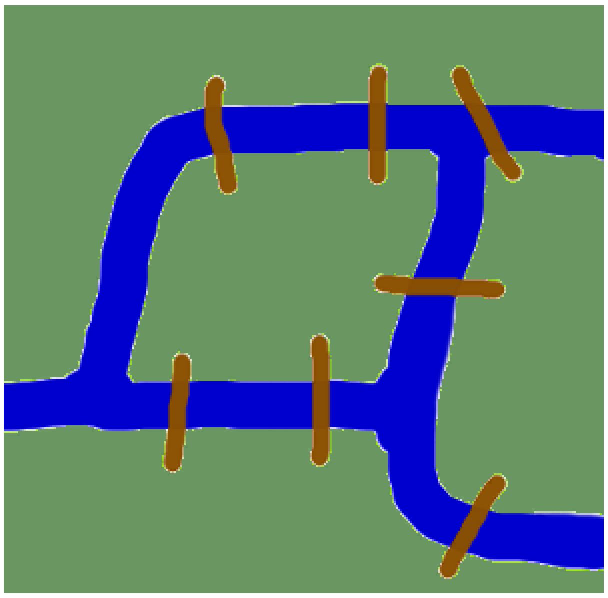
listdir(path)	A list with the content of the directory defined by "path", i.e. subdirectories and file names.
	`>>> os.listdir("/home/homer")`
	`['.gnome2', '.pulse', '.gconf', '.gconfd', '.beagl`
	`e', '.gnome2_private', '.gksu.lock', 'Public', '.IC`
	`Eauthority', '.bash_history', '.compiz', '.gvfs',`
	`'.update-notifier', '.cache', 'Desktop', 'Videos',`
	`'.profile', '.config', '.esd_auth', '.viminfo', '.s`
	`udo_as_admin_successful', 'mbox', '.xsession-error`
	`s', '.bashrc', 'Music', '.dbus', '.local', '.gstrea`
	`mer-0.10', 'Documents', '.gtk-bookmarks', 'Download`
	`s', 'Pictures', '.pulse-cookie', '.nautilus', 'exam`
	`ples.desktop', 'Templates', '.bash_logout']`

	`>>>`
mkdir(path[, mode=0755])	Create a directory named path with numeric mode "mode", if it doesn't already exist. The default mode is 0777 (octal). On some systems, mode is ignored. If it is used, the current umask value is first masked out. If the directory already exists, OSError is raised. Parent directories will not be created, if they don't exist.
makedirs(name[, mode=511])	Recursive directory creation function. Like mkdir(), but makes all intermediate-level directories needed to contain the leaf directory. Raises an error exception if the leaf directory already exists or cannot be created.
rename(old, new)	The file or directory "old" is renamed to "new" If "new" is a directory, an error will be raised. On Unix and Linux, if "new" exists and is a file, it will be replaced silently if the user has permission to do so.
renames(old, new)	Works like rename(), except that it creates recursively any intermediate directories needed to make the "new" pathname.
rmdir(path)	remove (delete) the directory "path". rmdir() works only, if the direcotry "path" is empty, otherwise an error is raised. To remove whole directory trees, shutil.rmdtree() can be used.

Further function and methods working on files and directories can be found in the module shutil. Amongst other possibilities it provides the possibility to copy files and directories with shutil.copyfile(src,dst).

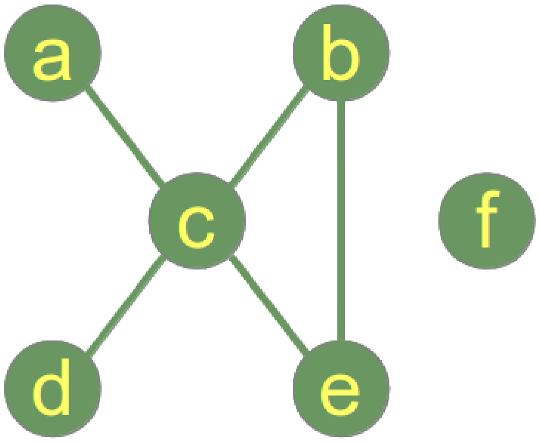
GRAPHS IN PYTHON

ORIGINS OF GRAPH THEORY



Before we start with the actual implementations of graphs in Python and before we start with the introduction of Python modules dealing with graphs, we want to devote ourselves to the origins of graph theory.
The origins take us back in time to the Künigsberg of the 18th century. Königsberg was a city in Prussia that time. The river Pregel flowed through the town, creating two islands. The city and the islands were connected by seven bridges as shown. The inhabitants of the city were moved by the question, if it was possible to take a walk through the town by visiting each area of the town and crossing each bridge only once? Every bridge must have been crossed completely, i.e. it is not allowed to walk halfway onto a bridge and then turn around and later cross the other half from the other side. The walk need not start and end at the same spot. Leonhard Euler solved the problem in 1735 by proving that it is not possible.
He found out that the choice of a route inside each land area is irrelevant and that the only thing which mattered is the order (or the sequence) in which the bridges are crossed. He had formulated an abstraction of the problem, eliminating unnecessary facts and focussing on the land areas and the bridges connecting them. This way, he created the foundations of graph theory. If we see a "land area" as a vertex and each bridge as an edge, we have "reduced" the problem to a graph.

INTRODUCTION INTO GRAPH THEORY USING PYTHON



Before we start our treatize on possible Python representations of graphs, we want to present some general definitions of graphs and its components.
A "graph"[1] in mathematics and computer science consists of "nodes", also known as "vertices". Nodes may or may not be connected with one another. In our illustration, - which is a pictorial representation of a graph,

- the node "a" is connected with the node "c", but "a" is not connected with "b". The connecting line between two nodes is called an edge. If the edges between the nodes are undirected, the graph is called an undirected graph. If an edge is directed from one vertex (node) to another, a graph is

called a directed graph. An directed edge is called an arc.
Though graphs may look very theoretical, many practical problems can be represented by graphs. They are often used to model problems or situations in physics, biology, psychology and above all in computer science. In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation,
In the latter case, the are used to represent the data organisation, like the file system of an operating system, or communication networks. The link structure of websites can be seen as a graph as well, i.e. a directed graph, because a link is a directed edge or an arc.
Python has no built-in data type or class for graphs, but it is easy to implement them in Python. One data type is ideal for representing graphs in Python, i.e. dictionaries. The graph in our illustration can be implemented in the following way:

```
graph = { "a" : {"c"},
          "b" : {"c", "e"},
          "c" : {"a", "b", "d", "e"},
          "d" : {"c"},
          "e" : {"c", "b"},
          "f" : {}
        }
```

The keys of the dictionary above are the nodes of our graph. The corresponding values are sets with the nodes, which are connectrd by an edge. A set is better than a list or a tuple, because this way, we can have only one edge between two nodes. There is no simpler and more elegant way to represent a graph.

An edge can also be ideally implemented as a set with two elements, i.e. the end nodes. This is ideal for undirected graphs. For directed graphs we would prefer lists or tuples to implement edges.

Function to generate the list of all edges:

```
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append({node, neighbour})

    return edges

print(generate_edges(graph))
```

```
[{'c', 'a'}, {'c', 'b'}, {'b', 'e'}, {'c', 'd'}, {'c', 'b'},
{'c', 'e'}, {'c', 'a'}, {'c', 'd'}, {'c', 'e'}, {'b', 'e'}]
```

As we can see, there is no edge containing the node "f". "f" is an isolated node of our graph. The following

Python function calculates the isolated nodes of a given graph:

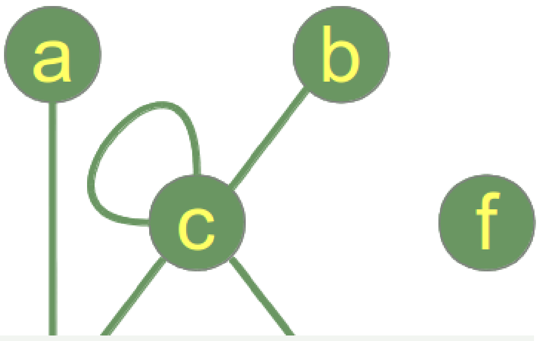
```
def find_isolated_nodes(graph):
    """ returns a set of isolated nodes. """
    isolated = set()
    for node in graph:
        if not graph[node]:
            isolated.add(node)
    return isolated
```

If we call this function with our graph, a list containing "f" will be returned: `["f"]`

GRAPHS AS A PYTHON CLASS

Before we go on with writing functions for graphs, we have a first go at a Python graph class implementation.

If you look at the following listing of our class, you can see in the **init**-method that we use a dictionary "self._graph_dict" for storing the vertices and their corresponding adjacent vertices.



```
""" A Python Class
A simple Python graph class, demonstra
ting the essential
facts and functionalities of graphs.
"""


class Graph(object):

    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self._graph_dict = graph_dict
```

```
def edges(self, vertice):
    """ returns a list of all the edges of a vertice"""
    return self._graph_dict[vertice]

def all_vertices(self):
    """ returns the vertices of a graph as a set """
    return set(self._graph_dict.keys())

def all_edges(self):
    """ returns the edges of a graph """
    return self.__generate_edges()

def add_vertex(self, vertex):
    """ If the vertex "vertex" is not in
        self._graph_dict, a key "vertex" with an empty
        list as a value is added to the dictionary.
        Otherwise nothing has to be done.
    """
    if vertex not in self._graph_dict:
        self._graph_dict[vertex] = []

def add_edge(self, edge):
    """ assumes that edge is of type set, tuple or list;
        between two vertices can be multiple edges!
    """
    edge = set(edge)
    vertex1, vertex2 = tuple(edge)
    for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:
        if x in self._graph_dict:
            self._graph_dict[x].add(y)
        else:
            self._graph_dict[x] = [y]

def __generate_edges(self):
    """ A static method generating the edges of the
        graph "graph". Edges are represented as sets
        with one (a loop back to the vertex) or two
        vertices
    """
    edges = []
    for vertex in self._graph_dict:
        for neighbour in self._graph_dict[vertex]:
            if {neighbour, vertex} not in edges:
                edges.append({vertex, neighbour})
    return edges

def __iter__(self):
    self._iter_obj = iter(self._graph_dict)
    return self._iter_obj

def __next__(self):
    """ allows us to iterate over the vertices """
    return next(self._iter_obj)

def __str__(self):
    res = "vertices: "
    for k in self._graph_dict:
        res += str(k) + " "
    res += "\nedges: "
    for edge in self.__generate_edges():
        res += str(edge) + " "
    return res
```

We want to play a little bit with our graph. We start with iterating over the graph. Iterating means iterating over the vertices.

```
g = { "a" : {"d"},
      "b" : {"c"},
      "c" : {"b", "c", "d", "e"},
      "d" : {"a", "c"},
      "e" : {"c"},
      "f" : {}
    }



graph = Graph(g)

for vertice in graph:
    print(f"Edges of vertice {vertice}: ", graph.edges(vertice))
```

```
Edges of vertice a:  {'d'}
Edges of vertice b:  {'c'}
Edges of vertice c:  {'c', 'd', 'b', 'e'}
Edges of vertice d:  {'c', 'a'}
Edges of vertice e:  {'c'}
Edges of vertice f:  {}
```

```
graph.add_edge({"ab", "fg"})
graph.add_edge({"xyz", "bla"})
```

```
print("")
print("Vertices of graph:")
print(graph.all_vertices())

print("Edges of graph:")
print(graph.all_edges())
```

```
Vertices of graph:
{'d', 'b', 'e', 'f', 'fg', 'c', 'bla', 'xyz', 'ab', 'a'}
Edges of graph:
[{'d', 'a'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'ab', 'f
g'}, {'bla', 'xyz'}]
```

Let's calculate the list of all the vertices and the list of all the edges of our graph:

```
print("")
print("Vertices of graph:")
print(graph.all_vertices())

print("Edges of graph:")
print(graph.all_edges())
```

```
Vertices of graph:
{'d', 'b', 'e', 'f', 'fg', 'c', 'bla', 'xyz', 'ab', 'a'}
Edges of graph:
[{'d', 'a'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'ab', 'f
g'}, {'bla', 'xyz'}]
```

We add a vertex and and edge to the graph:

```
print("Add vertex:")
graph.add_vertex("z")

print("Add an edge:")
graph.add_edge({"a", "d"})

print("Vertices of graph:")
print(graph.all_vertices())

print("Edges of graph:")
print(graph.all_edges())
```

```
Add vertex:
Add an edge:
Vertices of graph:
{'d', 'b', 'e', 'f', 'fg', 'z', 'c', 'bla', 'xyz', 'ab', 'a'}
Edges of graph:
[{'d', 'a'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'ab', 'f
g'}, {'bla', 'xyz'}]
```

```
print('Adding an edge {"x","y"} with new vertices:')
graph.add_edge({"x","y"})
print("Vertices of graph:")
print(graph.all_vertices())
print("Edges of graph:")
print(graph.all_edges())
```

```
Adding an edge {"x","y"} with new vertices:
Vertices of graph:
{'d', 'b', 'e', 'f', 'fg', 'z', 'x', 'c', 'bla', 'xyz', 'ab',
'y', 'a'}
Edges of graph:
[{'d', 'a'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'ab', 'f
g'}, {'bla', 'xyz'}, {'x', 'y'}]
```

PATHS IN GRAPHS

We want to find now the shortest path from one node to another node. Before we come to the Python code for this problem, we will have to present some formal definitions.

Adjacent vertices:
Two vertices are adjacent when they are both incident to a common edge.

Path in an undirected Graph:
A path in an undirected graph is a sequence of vertices $P = (v_1, v_2, ..., v_n) \in V \times V \times ... \times V$ such that v_i is adjacent to $v_{\{i+1\}}$ for $1 \leq i < n$. Such a path P is called a path of length n from v_1 to v_n.
Simple Path:
A path with no repeated vertices is called a simple path.

Example:

`(a, c, e)` is a simple path in our graph, as well as `(a,c,e,b)`. `(a,c,e,b,c,d)` is a path but not a simple path, because the node c appears twice.

We add a method `find_path` to our class `Graph`. It tries to find a path from a start vertex to an end vertex. We also add a method `find_all_paths`, which finds all the paths from a start vertex to an end vertex:

```
""" A Python Class
A simple Python graph class, demonstrating the essential
facts and functionalities of graphs.
"""


class Graph(object):

    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self._graph_dict = graph_dict

    def edges(self, vertice):
        """ returns a list of all the edges of a vertice"""
        return self._graph_dict[vertice]

    def all_vertices(self):
        """ returns the vertices of a graph as a set """
        return set(self._graph_dict.keys())

    def all_edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_vertex(self, vertex):
        """ If the vertex "vertex" is not in
            self._graph_dict, a key "vertex" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        if vertex not in self._graph_dict:
            self._graph_dict[vertex] = []

    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
```

```
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        vertex1, vertex2 = tuple(edge)
        for x, y in [(vertex1, vertex2), (vertex2, vertex1)]:
            if x in self._graph_dict:
                self._graph_dict[x].append(y)
            else:
                self._graph_dict[x] = [y]

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
            with one (a loop back to the vertex) or two
            vertices
        """
        edges = []
        for vertex in self._graph_dict:
            for neighbour in self._graph_dict[vertex]:
                if {neighbour, vertex} not in edges:
                    edges.append({vertex, neighbour})
        return edges

    def __iter__(self):
        self._iter_obj = iter(self._graph_dict)
        return self._iter_obj

    def __next__(self):
        """ allows us to iterate over the vertices """
        return next(self._iter_obj)

    def __str__(self):
        res = "vertices: "
        for k in self._graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res

    def find_path(self, start_vertex, end_vertex, path=None):
        """ find a path from start_vertex to end_vertex
            in graph """
        if path == None:
            path = []
        graph = self._graph_dict
        path = path + [start_vertex]
        if start_vertex == end_vertex:
            return path
        if start_vertex not in graph:
            return None
        for vertex in graph[start_vertex]:
            if vertex not in path:
                extended_path = self.find_path(vertex, 
                                               end_vertex, 
                                               path)
                if extended_path: 
                    return extended_path
        return None
    

    def find_all_paths(self, start_vertex, end_vertex, path=[]):
        """ find all paths from start_vertex to 
            end_vertex in graph """
        graph = self._graph_dict 
        path = path + [start_vertex]
        if start_vertex == end_vertex:
            return [path]
        if start_vertex not in graph:
            return []
        paths = []
        for vertex in graph[start_vertex]:
            if vertex not in path:
                extended_paths = self.find_all_paths(vertex, 
                                                     end_vertex, 
                                                     path)
                for p in extended_paths: 
                    paths.append(p)
        return paths
```

We check in the following the way of working of our `find_path` and `find_all_paths` methods.

```
g = { "a" : {"d"},
      "b" : {"c"},
      "c" : {"b", "c", "d", "e"},
      "d" : {"a", "c"},
      "e" : {"c"},
      "f" : {}
    }
```

```python
graph = Graph(g)

print("Vertices of graph:")
print(graph.all_vertices())

print("Edges of graph:")
print(graph.all_edges())


print('The path from vertex "a" to vertex "b":')
path = graph.find_path("a", "b")
print(path)

print('The path from vertex "a" to vertex "f":')
path = graph.find_path("a", "f")
print(path)

print('The path from vertex "c" to vertex "c":')
path = graph.find_path("c", "c")
print(path)
```

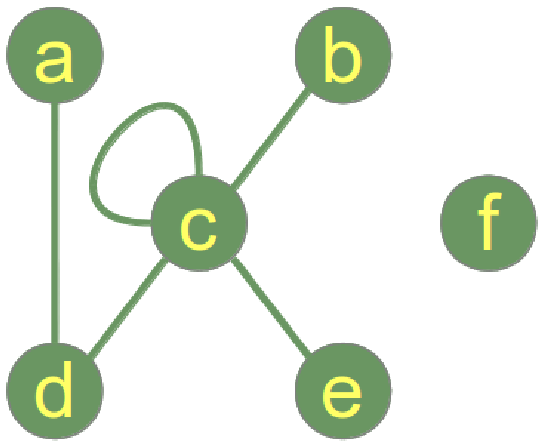
```
Vertices of graph:
{'d', 'b', 'e', 'f', 'c', 'a'}
Edges of graph:
[{'d', 'a'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}]
The path from vertex "a" to vertex "b":
['a', 'd', 'c', 'b']
The path from vertex "a" to vertex "f":
None
The path from vertex "c" to vertex "c":
['c']
```

We slightly changed our example graph by adding edges from "a" to "f" and from "f" to "d" to test the `find_all_paths` method:

```
g = { "a" : {"d", "f"},
      "b" : {"c"},
      "c" : {"b", "c", "d", "e"},
      "d" : {"a", "c", "f"},
      "e" : {"c"},
      "f" : {"a", "d"}
    }


graph = Graph(g)
print("Vertices of graph:")
print(graph.all_vertices())

print("Edges of graph:")
print(graph.all_edges())


print('All paths from vertex "a" to ve
rtex "b":')
path = graph.find_all_paths("a", "b")
print(path)

print('All paths from vertex "a" to vertex "f":')
path = graph.find_all_paths("a", "f")
print(path)

print('All paths from vertex "c" to vertex "c":')
path = graph.find_all_paths("c", "c")
print(path)
```

```
Vertices of graph:
{'d', 'b', 'e', 'f', 'c', 'a'}
Edges of graph:
[{'d', 'a'}, {'f', 'a'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c',
'e'}, {'d', 'f'}]
All paths from vertex "a" to vertex "b":
[['a', 'd', 'c', 'b'], ['a', 'f', 'd', 'c', 'b']]
All paths from vertex "a" to vertex "f":
[['a', 'd', 'f'], ['a', 'f']]
All paths from vertex "c" to vertex "c":
[['c']]
```

DEGREE AND DEGREE SEQUENCE



The **degree** of a vertex v in a graph is the number of edges connecting it, with loops counted twice. The degree of a vertex v is denoted deg(v). The maximum degree of a graph G, denoted by Δ(G), and the minimum degree of a graph, denoted by δ(G), are the maximum and minimum degree of its vertices.

In the graph on the right side, the maximum degree is 5 at vertex c and the minimum degree is 0, i.e the isolated vertex f.

If all the degrees in a graph are the same, the graph is a regular graph.

In a regular graph, all degrees are the same, and so we can speak of the degree of the graph.

The degree sum formula (Handshaking lemma):

$\sum_{v \in V} deg(v) = 2\ |E|$

This means that the sum of degrees of all the vertices is equal to the number of edges multiplied by 2. We can conclude that the number of vertices with odd degree has to be even. This statement is known as the handshaking lemma. The name "handshaking lemma" stems from a popular mathematical problem: In any group of people the number of people who have shaken hands with an odd number of other people from the group is even.

The **degree sequence** of an undirected graph is defined as the sequence of its vertex degrees in a non-increasing order. The following method returns a tuple with the degree sequence of the instance graph:

We will design a new class `Graph2` now, which inherits from our previously defined graph `Graph` and we add the following methods to it:

- `vertex_degree`
- `find_isolated_vertices`
- `delta`
- `degree_sequence`

```
class Graph2(Graph):
    def vertex_degree(self, vertex):
        """ The degree of a vertex is the number of edges connecti
ng
        it, i.e. the number of adjacent vertices. Loops are counte
d
        double, i.e. every occurence of vertex in the list
        of adjacent vertices. """
        degree =  len(self._graph_dict[vertex])
        if vertex in self._graph_dict[vertex]:
            degree += 1
        return degree

    def find_isolated_vertices(self):
        """ returns a list of isolated vertices. """
        graph = self._graph_dict
        isolated = []
        for vertex in graph:
            print(isolated, vertex)
            if not graph[vertex]:
                isolated += [vertex]
        return isolated

    def Delta(self):
        """ the maximum degree of the vertices """
        max = 0
        for vertex in self._graph_dict:
            vertex_degree = self.vertex_degree(vertex)
            if vertex_degree > max:
                max = vertex_degree
        return max

    def degree_sequence(self):
        """ calculates the degree sequence """
        seq = []
        for vertex in self._graph_dict:
            seq.append(self.vertex_degree(vertex))
        seq.sort(reverse=True)
        return tuple(seq)
```

```
g = { "a" : {"d", "f"},
      "b" : {"c"},
      "c" : {"b", "c", "d", "e"},
      "d" : {"a", "c"},
      "e" : {"c"},
      "f" : {"d"}
```

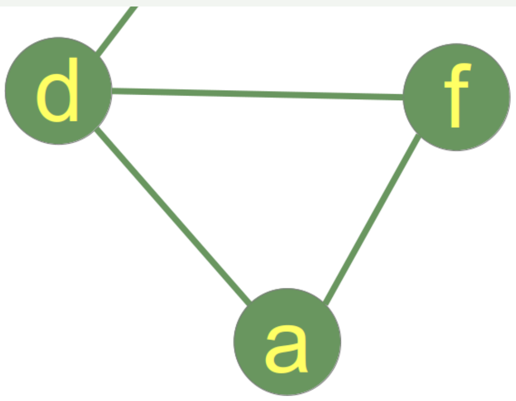
```
    }


graph = Graph2(g)
graph.degree_sequence()
```

Output:
```
(5, 2, 2, 1, 1, 1)
```

Let's have a look at the other graph:

```
g = { "a" : {"d", "f"},
      "b" : {"c"},
      "c" : {"b", "c", "d", "e"},
      "d" : {"a", "c", "f"},
      "e" : {"c"},
      "f" : {"a", "d"}
    }

graph = Graph2(g)
graph.degree_sequence()
```

Output:
```
(5, 3, 2, 2, 1, 1)
```



CURRYING

GENERAL IDEA

In mathematics and computer science, currying is the technique of breaking down the evaluation of a function that takes multiple arguments into evaluating a sequence of single-argument functions. Currying is also used in theoretical computer science, because it is often easier to transform multiple argument models into single argument models.

COMPOSITION OF FUNCTIONS

We define the composition h of two functions f and g

$h(x) = g(f(x))$

in the following Python example.

The composition of two functions is a chaining process in which the output of the inner function becomes the input of the outer function.

```
def compose(g, f):
    def h(x):
        return g(f(x))
    return h
```

We will use our compose function in the next example. Let's assume, we have a thermometer, which is not working accurately. The correct temperature can be calculated by applying the function *readjust* to the temperature values. Let us further assume that we have to convert our temperature values from Celsius to Fahrenheit. We can do this by applying *compose* to both functions:

```
def celsius2fahrenheit(t):
    return 1.8 * t + 32

def readjust(t):
    return 0.9 * t - 0.5

convert = compose(readjust, celsius2fahrenheit)

print(convert(10), celsius2fahrenheit(10))
```

```
44.5 50.0
```

The composition of two functions is generally not commutative, i.e. compose(celsius2fahrenheit, readjust) is different from compose(readjust, celsius2fahrenheit)

```
convert2 = compose(celsius2fahrenheit, readjust)

print(convert2(10), celsius2fahrenheit(10))
```

```
47.3 50.0
```

`convert2` is not a solution to our problem, because it is not readjusting the original temperatures of our thermometer but the transformed Fahrenheit values!

EXAMPLE CURRENCY CONVERSION

In our chapter on Magic Functions we had an exercise on currency conversion.

"COMPOSE" WITH ARBITRARY ARGUMENTS

The function `compose` which we have just defined can only copy with single-argument functions. We can generalize our function *compose* so that it can cope with all possible functions, along with an example using a function with two parameters.

```
def compose(g, f):
    def h(*args, **kwargs):
        return g(f(*args, **kwargs))
    return h
```

```
def BMI(weight, height):
    return weight / height**2

def evaluate_BMI(bmi):
    if bmi < 15:
        return "Very severely underweight"
    elif bmi < 16:
        return "Severely underweight"
    elif bmi < 18.5:
        return "Underweight"
    elif bmi < 25:
        return "Normal (healthy weight)"
    elif bmi < 30:
        return "Overweight"
    elif bmi < 35:
        return "Obese Class I (Moderately obese)"
    elif bmi < 40:
        return "Obese Class II (Severely obese)"
    else:
        return "Obese Class III (Very severely obese)"


f = compose(evaluate_BMI, BMI)

again = "y"
while again == "y":
    weight = float(input("weight (kg) "))
    height = float(input("height (m) "))
    print(f(weight, height))
    again = input("Another run? (y/n)")
```

```
Normal (healthy weight)
```

CURRYING FUNCTION WITH AN ARBITRARY NUMBER OF ARGUMENTS

One interesting question remains: How to curry a function across an arbitrary and unknown number of parameters?

We can use a nested function to make it possible to "curry" (accumulate) the arguments. We will need a way to tell the function calculate and return the value. If the funtions is called with arguments, these will be curried, as we have said. What if we call the function without any arguments? Right, this is a fantastic way to tell the function that we finally want to the the result. We can also clean the lists with the accumulated values:

```
def arimean(*args):
    return sum(args) / len(args)

def curry(func):
    # to keep the name of the curried function:
    curry.__curried_func_name__ = func.__name__
    f_args, f_kwargs = [], {}
    def f(*args, **kwargs):
        nonlocal f_args, f_kwargs
        if args or kwargs:
            f_args += args
            f_kwargs.update(kwargs)
            return f
        else:
            result = func(*f_args, *f_kwargs)
            f_args, f_kwargs = [], {}
            return result
    return f
```

```
curried_arimean = curry(arimean)
curried_arimean(2)(5)(9)(4, 5)
# it will keep on currying:
curried_arimean(5, 9)
print(curried_arimean())

# calculating the arithmetic mean of 3, 4, and 7
print(curried_arimean(3)(4)(7)())

# calculating the arithmetic mean of 4, 3, and 7
print(curried_arimean(4)(3, 7)())
```

```
5.571428571428571
4.666666666666667
4.666666666666667
```

Let's compare it with the result of the original arimean function:

```
print(arimean(2, 5, 9, 4, 5, 5, 9))
print(arimean(3, 4, 7))
print(arimean(4, 3, 7))
```

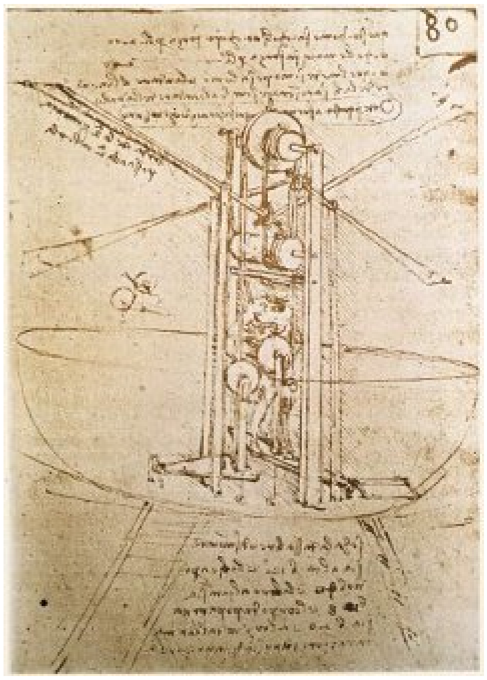
```
5.571428571428571
4.666666666666667
4.666666666666667
```

Including some prints might help to understand what's going on:

```
def arimean(*args):
    return sum(args) / len(args)

def curry(func):
    # to keep the name of the curried function:
    curry.__curried_func_name__ = func.__name__
    f_args, f_kwargs = [], {}
    def f(*args, **kwargs):
        nonlocal f_args, f_kwargs
        if args or kwargs:
            print("Calling curried function with:")
            print("args: ", args, "kwargs: ", kwargs)
            f_args += args
            f_kwargs.update(kwargs)
            print("Currying the values:")
            print("f_args: ", f_args)
            print("f_kwargs:", f_kwargs)
            return f
        else:
            print("Calling " + curry.__curried_func_name__ + " wit
h:")
            print(f_args, f_kwargs)

            result = func(*f_args, *f_kwargs)
            f_args, f_kwargs = [], {}
            return result
    return f

curried_arimean = curry(arimean)
curried_arimean(2)(5)(9)(4, 5)
# it will keep on currying:
curried_arimean(5, 9)
print(curried_arimean())
```

```
Calling curried function with:
args:  (2,) kwargs:  {}
Currying the values:
f_args:  [2]
f_kwargs: {}
Calling curried function with:
args:  (5,) kwargs:  {}
Currying the values:
f_args:  [2, 5]
f_kwargs: {}
Calling curried function with:
args:  (9,) kwargs:  {}
Currying the values:
f_args:  [2, 5, 9]
f_kwargs: {}
Calling curried function with:
args:  (4, 5) kwargs:  {}
Currying the values:
f_args:  [2, 5, 9, 4, 5]
f_kwargs: {}
Calling curried function with:
args:  (5, 9) kwargs:  {}
Currying the values:
f_args:  [2, 5, 9, 4, 5, 5, 9]
f_kwargs: {}
Calling arimean with:
[2, 5, 9, 4, 5, 5, 9] {}
5.571428571428571
```

FINITE STATE MACHINE (FSM)



A "Finite State Machine" (abbreviated FSM), also called "State Machine" or "Finite State Automaton" is an abstract machine which consists of a set of states (including the initial state and one or more end states), a set of input events, a set of output events, and a state transition function. A transition function takes the current state and an input event as an input and returns the new set of output events and the next (new) state. Some of the states are used as "terminal states".

The operation of an FSM begins with a special state, called the start state, proceeds through transitions depending on input to different states and normally ends in terminal or end states. A state which marks a successful flow of operation is known as an accept state.

Mathematical Model:
A deterministic finite state machine or acceptor deterministic finite state machine is a quintuple
$(\Sigma, S, s_0, \delta, F)$,
where:
Σ is the input alphabet (a finite, non-empty set of symbols).
S is a finite, non-empty set of states.
s_0 is an initial state, an element of S.
δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$
(in a nondeterministic finite state machine it would be $\delta : S \times \Sigma \rightarrow \wp(S)$, i.e., δ would return a set of states).
($\wp(S)$ is the Power set of S)
F is the set of final states, a (possibly empty) subset of S.

A SIMPLE EXAMPLE

We want to recognize the meaning of very small sentences with an extremely limited vocabulary and syntax.

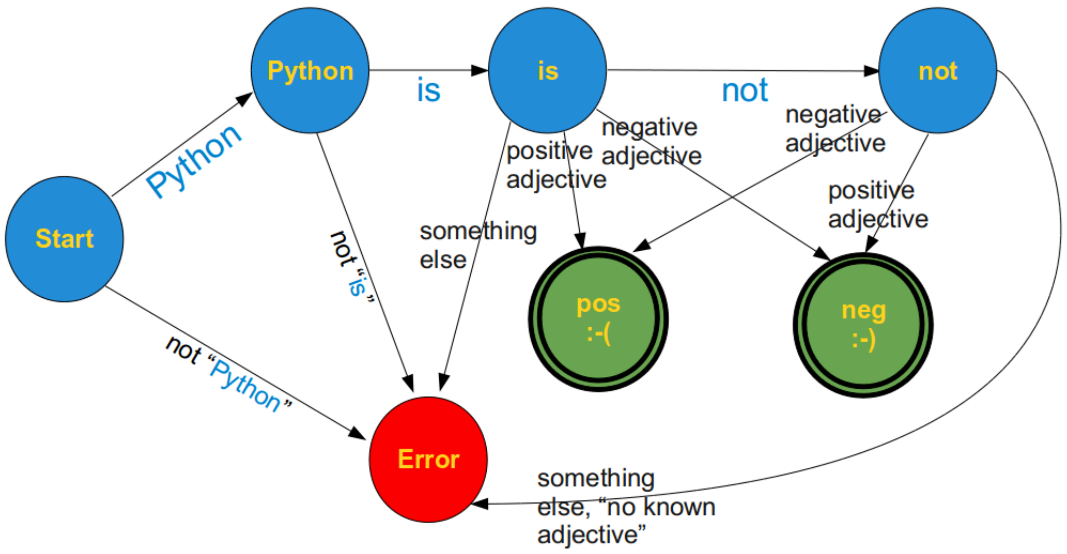
These sentences should start with "Python is" followed by

- an adjective or
- the word "not" followed by an adjective. e.g.

  "Python is great" → positive meaning

"Python is stupid" → negative meaning

"Python is not ugly" → positive meaning



A FINITE STATE MACHINE IN PYTHON

To implement the previous example, we program first a general Finite State Machine in Python.

```
class StateMachine:

    def __init__(self):
        self.handlers = {}
        self.startState = None
        self.endStates = []

    def add_state(self, name, handler, end_state=0):
        name = name.upper()
        self.handlers[name] = handler
        if end_state:
            self.endStates.append(name)
    def set_start(self, name):
        self.startState = name.upper()

    def run(self, cargo):
        try:
            handler = self.handlers[self.startState]
        except:
            raise InitializationError("must call .set_start() befo
re .run()")
        if not self.endStates:
            raise  InitializationError("at least one state must b
e an end_state")

        while True:
            (newState, cargo) = handler(cargo)
            if newState.upper() in self.endStates:
                print("reached ", newState)
                break
            else:
                handler = self.handlers[newState.upper()]
```

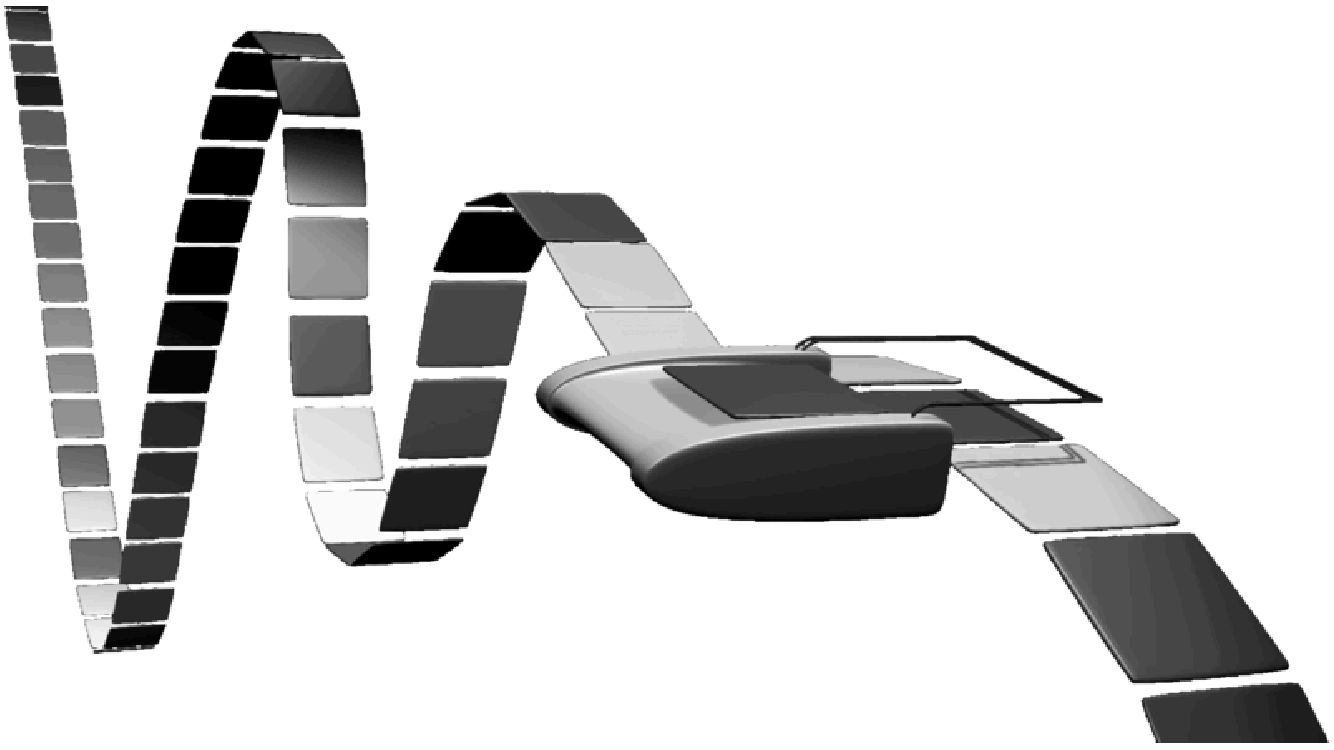
This general FSM is used in the next program:

```
positive_adjectives = ["great","super", "fun", "entertaining", "ea
sy"]
negative_adjectives = ["boring", "difficult", "ugly", "bad"]

def start_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (tx
t,"")
    if word == "Python":
        newState = "Python_state"
    else:
        newState = "error_state"
    return (newState, txt)

def python_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (tx
t,"")
    if word == "is":
        newState = "is_state"
    else:
        newState = "error_state"
    return (newState, txt)

def is_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (tx
t,"")
    if word == "not":
        newState = "not_state"
    elif word in positive_adjectives:
        newState = "pos_state"
    elif word in negative_adjectives:
        newState = "neg_state"
    else:
        newState = "error_state"
    return (newState, txt)

def not_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (tx
t,"")
    if word in positive_adjectives:
        newState = "neg_state"
    elif word in negative_adjectives:
        newState = "pos_state"
    else:
        newState = "error_state"
    return (newState, txt)

def neg_state(txt):
    print("Hallo")
    return ("neg_state", "")


m = StateMachine()
m.add_state("Start", start_transitions)
m.add_state("Python_state", python_state_transitions)
m.add_state("is_state", is_state_transitions)
m.add_state("not_state", not_state_transitions)
m.add_state("neg_state", None, end_state=1)
m.add_state("pos_state", None, end_state=1)
m.add_state("error_state", None, end_state=1)
m.set_start("Start")
m.run("Python is great")
```

```python
m.run("Python is difficult")
m.run("Perl is ugly")
```

```
reached  pos_state
reached  neg_state
reached  error_state
```

TURING MACHINE



Just to let you know straight-away: The Turing machine is not a machine. It is a mathematical model, which was formulated by the English mathematician Alan Turing in 1936. It's a very simple model of a computer, yet it has the complete computing capability of a general purpose computer. The Turing machine (TM) serves two needs in theoretical computer science:

1. The class of languages defined by a TM, i.e. structured or recursively enumerable languages
2. The class of functions a TM is capable to compute, i.e. the partial recursive functions

A Turing machine consists only of a few components: A tape on which data can be sequentially stored. The tape consists of fields, which are sequentially arranged. Each field can contain a character of a finite alphabet. This tape has no limits, it goes on infinitely in both directions. In a real machine, the tape would have to be large enough to contain all the data for the algorithm. A TM also contains a head moving in both directions over the tape. This head can read and write one character from the field over which the head resides. The Turing machine is at every moment in a certain state, one of a finite number of states. A Turing program is a list of transitions, which determine for a given state and character ("under" the head) a new state, a character which has to be written into the field under the head and a movement direction for the head, i.e. either left, right or static (motionless).

FORMAL DEFINITION OF A TURING MACHINE

A deterministic Turing machine can be defined as a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, b, q_0, q_f)$$

with

- Q is a finite, non-empty set of states
- Γ is a finite, non-empty set of the tape alphabet
- Σ is the set of input symbols with $\Sigma \subset \Gamma$
- δ is a partially defined function, the transition function:
  $\delta : (Q \setminus \{q_f\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L,N,R\}$
- b ∈ &Gamma \ Σ is the blank symbol
- $q_0 \in Q$ is the initial state
- $q_f \in Q$ is the set of accepting or final states

EXAMPLE: BINARY COMPLEMENT FUNCTION

Let's define a Turing machine, which complements a binary input on the tape, i.e. an input "1100111" e.g. will be turned into "0011000".
$\Sigma = \{0, 1\}$
$Q = \{\text{init}, \text{final}\}$
$q_0 = \text{init}$
$q_f = \text{final}$

Function Definition	Description
δ(init,0) = (init, 1, R)	If the machine is in state "init" and a 0 is read by the head, a 1 will be written, the state will change to "init" (so actually, it will not change) and the head will be moved one field to the right.
δ(init,1) = (init, 0, R)	If the machine is in state "init" and a 1 is read by the head, a 0 will be written, the state will change to "init" (so actually, it will not change) and the head will be moved one field to the right.
δ(init,b) = (final, b, N)	If a blank ("b"), defining the end of the input string, is read, the TM reaches the final state "final" and halts.

IMPLEMENTATION OF A TURING MACHINE IN PYTHON

We implement a Turing Machine in Python as a class. We define another class for the read/write tape of the Turing Machine. The core of the tape inside the class Tape is a dictionary, which contains the entries of the tape. This way, we can have negative indices. A Python list is not a convenient data structure, because Python lists are bounded on one side, i.e. bounded by 0.

We define the method **str**(self) for the class Tape. **str**(self) is called by the built-in str() function. The print function uses also the str function to calculate the "informal" string representation of an object, in our case the tape of the TM. The method get_tape() of our class TuringMachine makes use of the str representation returned by **str**.

With the aid of the method **getitem**(), we provide a reading access to the tape via indices. The definition of the method **setitem**() allows a writing access as well, as we can see e.g. in the statement `self.__tape[self.__head_position] = y[1]` of our class TuringMachine implementation.

```
class Tape(object):

    blank_symbol = " "

    def __init__(self,
                 tape_string = ""):
        self.__tape = dict((enumerate(tape_string)))
        # last line is equivalent to the following three lines:
        #self.__tape = {}
        #for i in range(len(tape_string)):
        #    self.__tape[i] = input[i]

    def __str__(self):
        s = ""
        min_used_index = min(self.__tape.keys())
        max_used_index = max(self.__tape.keys())
        for i in range(min_used_index, max_used_index):
            s += self.__tape[i]
        return s

    def __getitem__(self,index):
        if index in self.__tape:
            return self.__tape[index]
        else:
            return Tape.blank_symbol

    def __setitem__(self, pos, char):
        self.__tape[pos] = char


class TuringMachine(object):

    def __init__(self,
                 tape = "",
                 blank_symbol = " ",
                 initial_state = "",
                 final_states = None,
                 transition_function = None):
        self.__tape = Tape(tape)
        self.__head_position = 0
        self.__blank_symbol = blank_symbol
        self.__current_state = initial_state
        if transition_function == None:
            self.__transition_function = {}
        else:
            self.__transition_function = transition_function
        if final_states == None:
            self.__final_states = set()
        else:
            self.__final_states = set(final_states)

    def get_tape(self):
        return str(self.__tape)

    def step(self):
        char_under_head = self.__tape[self.__head_position]
        x = (self.__current_state, char_under_head)
        if x in self.__transition_function:
            y = self.__transition_function[x]
            self.__tape[self.__head_position] = y[1]
            if y[2] == "R":
                self.__head_position += 1
            elif y[2] == "L":
                self.__head_position -= 1
            self.__current_state = y[0]

    def final(self):
        if self.__current_state in self.__final_states:
            return True
        else:
            return False
```

Using the TuringMachine class:

```
initial_state = "init",
accepting_states = ["final"],
transition_function = {("init","0"):("init", "1", "R"),
                       ("init","1"):("init", "0", "R"),
                       ("init"," "):("final"," ", "N"),
                       }
final_states = {"final"}

t = TuringMachine("010011001 ",
                  initial_state = "init",
                  final_states = final_states,
                  transition_function=transition_function)

print("Input on Tape:\n" + t.get_tape())

while not t.final():
    t.step()

print("Result of the Turing machine calculation:")
print(t.get_tape())
```

```
Input on Tape:
010011001
Result of the Turing machine calculation:
101100110
```

TURKISH TIME AND CLOCK

INTRODUCTION

This chapter of our Python tutorial deals with turkish times. You will learn how to tell the time in turkish, but what is more him important: We will teach Python to tell the time in Turkish given the time numerically. Turkish grammar and sentence structures as well as vocabulary differ extremely from Indo-European languages like English and Persian, and also Semitic languages like Arabic and Hebrew and many other languages. So, telling the time in Turkish is also extremely difficult for speakers of English, German, French, Italian and many other languages. Fortunately, there are strict rules available. Rules which are so regular that it is easy to write a Python program to the job.

"Saat kaç?" means "What is the time?" in Turkish.

The task of hour program is to answer this question, given the time as string in the form "03:00" or "10:30".

We will need the words for the digits 0, 1, ... 9. We use a Pyhton list for this:

```
digits_list = ["bir", "iki", "üç",
               "dört", "beş", "altı",
               "yedi", "sekiz", "dokuz"]
```

The digits are not enough. We will need the words for the tens as well. The following list corresponds to the numbers "ten", "twenty", "thirty", "forty", and "fifty". We do not need more:

```
tens_list = ["on", "yirmi", "otuz",  "kırk",  "elli"]
```

VERBAL REPRESENTATION OF NUMBERS

Now, we need a function to transform any number between 1 and 59 to a turkish verbal representation. We can easily define this function, because it is more regular than in English. You will hopefully understand it by just looking at the function:

```
def num2txt(min):
    digits = min % 10
    tens = min // 10
    if digits > 0:
        if tens:
            return tens_list[tens-1] + " " + digits_list[digits-1]
        else:
            return digits_list[digits-1]
    else:
        return tens_list[tens-1]

for number in range(60):
    print(number, num2txt(number), end=", ")
```

```
0 elli, 1 bir, 2 iki, 3 üç, 4 dört, 5 beş, 6 altı, 7 yedi, 8 seki
z, 9 dokuz, 10 on, 11 on bir, 12 on iki, 13 on üç, 14 on dört, 15
on beş, 16 on altı, 17 on yedi, 18 on sekiz, 19 on dokuz, 20 yirm
i, 21 yirmi bir, 22 yirmi iki, 23 yirmi üç, 24 yirmi dört, 25 yirm
i beş, 26 yirmi altı, 27 yirmi yedi, 28 yirmi sekiz, 29 yirmi doku
z, 30 otuz, 31 otuz bir, 32 otuz iki, 33 otuz üç, 34 otuz dört, 3
5 otuz beş, 36 otuz altı, 37 otuz yedi, 38 otuz sekiz, 39 otuz dok
uz, 40 kırk, 41 kırk bir, 42 kırk iki, 43 kırk üç, 44 kırk dört, 4
5 kırk beş, 46 kırk altı, 47 kırk yedi, 48 kırk sekiz, 49 kırk dok
uz, 50 elli, 51 elli bir, 52 elli iki, 53 elli üç, 54 elli dört, 5
5 elli beş, 56 elli altı, 57 elli yedi, 58 elli sekiz, 59 elli dok
uz,
```

We will start now implementing the function `translate_time`. This function will produce a reply to the question "Saat kaç?". It takes `hours` and `minutes` as parameters. We will only implement the funciton for full hours, because this is the easiest case.

```
def translate_time(hours, minutes=0):
    if minutes == 0:
        return "Saat " + num2txt(hours) + "."
    else:
        return "I still do not know how to say it!"

for hour in range(1, 13):
    print(hour, translate_time(hour))
```

```
1 Saat bir.
2 Saat iki.
3 Saat üç.
4 Saat dört.
5 Saat beş.
6 Saat altı.
7 Saat yedi.
8 Saat sekiz.
9 Saat dokuz.
10 Saat on.
11 Saat on bir.
12 Saat on iki.
```

Easy, isn't it? What about "half past"? This is similar to English, if you consider that "buçuk" means "half". We extend our function:

```
def translate_time(hours, minutes=0):
    if minutes == 0:
        return "Saat " + num2txt(hours) + "."
    elif minutes == 30:
        return "Saat " + num2txt(hours) + " buçuk."
    else:
        return "I still do not know how to say it!"

for hour in range(1, 13):
    print(hour, translate_time(hour, 30))
```

```
1 Saat bir buçuk.
2 Saat iki buçuk.
3 Saat üç buçuk.
4 Saat dört buçuk.
5 Saat beş buçuk.
6 Saat altı buçuk.
7 Saat yedi buçuk.
8 Saat sekiz buçuk.
9 Saat dokuz buçuk.
10 Saat on buçuk.
11 Saat on bir buçuk.
12 Saat on iki buçuk.
```

We want to go on with telling the time in Turkish, and teach our Python program to say "a quarter past" and a "quarter to". It is getter more complicated or interesting now. Our Python implementation needs to learn some grammar rules now. The Python program has to learn now how to produce the Dative and the Accusative in Turkish.

TURKISH DATIVE

The dative case is formed by adding `e` or `a` as a suffix to the end of the word (not necessarily a noun). If the last syllable of the word has one of the vowels 'a', 'ı', 'o' or 'u', an 'a' has to be added to the end of the word. If the last syllable of the word has one of the vowels 'e', 'i', 'ö', or 'ü', an 'e' has to be appended. One more important thing: Turkish doesn't like to vowels beside of each other. Therefore, we will add a "y" in front ot the 'e' or 'a', if the last letter of the word is a vowel.

Now we are ready to implement the `dative` function. It will suffix the dative ending to a given word. The `dative` function needs a function `last_vowel` which comes back with two values:

- the vowel in the last syllable
- `True` or `False` to indicate if the word ends with a vowel

```
def last_vowel(word):
    vowels = {'ı', 'a', 'e', 'i', 'o', 'u', 'ö', 'ü'}
    ends_with_vowel = True
    for char in word[::-1]:
        if char in vowels:
            return char, ends_with_vowel
        ends_with_vowel = False

def dative(word):
    lv, ends_with_vowel = last_vowel(word)
    if lv in {'i', 'e', 'ö', 'ü'}:
        ret_vowel = 'e'
    elif lv in {'a', 'ı', 'o', 'u'}:
        ret_vowel = 'a'
    if ends_with_vowel:
        return word + "y" + ret_vowel
    else:
        return word + ret_vowel
```

We can check the `dative` function by calling it with the numbers in `digits_list`:

```
for word in digits_list:
    print(word, dative(word))
```

```
bir bire
iki ikiye
üç üçe
dört dörte
beş beşe
altı altıya
yedi yediye
sekiz sekize
dokuz dokuza
```

TURKISH ACCUSATIVE

'-i', '-ı', '-u' or '-ü' will be added to a word according to the last vowel of the word.

Last vowel	Vowel to be Added
ı or a	ı
e or i	i
o or u	u
ö or ü	ü

This can be easily implemented as a Python function:

```python
def accusative(word):
    lv, ends_with_vowel = last_vowel(word)
    if lv in {'ı', 'a'}:
        ret_vowel = 'ı'
    elif lv in {'e', 'i'}:
        ret_vowel = 'i'
    elif lv in {'o', 'u'}:
        ret_vowel = 'u'
    elif lv in {'ö', 'ü'}:
        ret_vowel = 'ü'
    if ends_with_vowel:
        return word + "y" + ret_vowel
    else:
        return word + ret_vowel
```

Let us have a look at the dative and accusative together:

```
print(f"{'word':<20s} {'dative':<20s} {'accusative':<20s}")
for word in digits_list:
    print(f"{word:<20s} {dative(word):<20s} {accusative(word):<20
s}")
```

```
word                 dative               accusative
bir                  bire                 biri
iki                  ikiye                ikiyi
üç                   üçe                  üçü
dört                 dörte                dörtü
beş                  beşe                 beşi
altı                 altıya               altıyı
yedi                 yediye               yediyi
sekiz                sekize               sekizi
dokuz                dokuza               dokuzu
```

Now, we have programmed what we need to further extend the `translate_time` function. We will add now the branches for "a quarter to" and "a quarter past". A "quarter" means "çeyrek" in Turkish. "to" and "past" are translated by adding "var" and "geçiyor" to the end. The verbal representation of the hour number has to be put into the dative in case of "var" and into the accusative form in case of "geçiyor".

```
def translate_time(hours, minutes=0):
    if minutes == 0:
        return "Saat " + num2txt(hours) + "."
    elif minutes == 30:
        return "Saat " + num2txt(hours) + " buçuk."
    elif minutes == 15:
        return "Saat " + accusative(num2txt(hours)) + " çeyrek geç
iyor."
    elif minutes == 45:
        if hours == 12:
            hours = 0
        return "Saat " + dative(num2txt(hours+1)) + " çeyrek var."
    else:
        return "I still do not know how to say it!"


for hour in range(1, 13):
    print(f"{hour:02d}:{15:02d} {translate_time(hour, 15):25s}")
    print(f"{hour:02d}:{45:02d} {translate_time(hour, 45):25s}")
```

```
01:15 Saat biri çeyrek geçiyor.
01:45 Saat ikiye çeyrek var.
02:15 Saat ikiyi çeyrek geçiyor.
02:45 Saat üçe çeyrek var.
03:15 Saat üçü çeyrek geçiyor.
03:45 Saat dörte çeyrek var.
04:15 Saat dörtü çeyrek geçiyor.
04:45 Saat beşe çeyrek var.
05:15 Saat beşi çeyrek geçiyor.
05:45 Saat altıya çeyrek var.
06:15 Saat altıyı çeyrek geçiyor.
06:45 Saat yediye çeyrek var.
07:15 Saat yediyi çeyrek geçiyor.
07:45 Saat sekize çeyrek var.
08:15 Saat sekizi çeyrek geçiyor.
08:45 Saat dokuza çeyrek var.
09:15 Saat dokuzu çeyrek geçiyor.
09:45 Saat ona çeyrek var.
10:15 Saat onu çeyrek geçiyor.
10:45 Saat on bire çeyrek var.
11:15 Saat on biri çeyrek geçiyor.
11:45 Saat on ikiye çeyrek var.
12:15 Saat on ikiyi çeyrek geçiyor.
12:45 Saat bire çeyrek var.
```

We can finish the `translate_time` function now. We have to branches for the minutes.

```python
def translate_time(hours, minutes=0):
    if minutes == 0:
        return "Saat " + num2txt(hours) + "."
    elif minutes == 30:
        return "Saat " + num2txt(hours) + " buçuk."
    elif minutes == 15:
        return "Saat " + accusative(num2txt(hours)) + " çeyrek geç
iyor."
    elif minutes == 45:
        if hours == 12:
            hours = 0
        return "Saat " + dative(num2txt(hours+1)) + " çeyrek var."
    elif minutes < 30:
        return "Saat " + accusative(num2txt(hours)) + " " + num2tx
t(minutes) + " geçiyor."
    elif minutes > 30:
        minutes = 60 - minutes
        if hours == 12:
            hours = 0
        hours = dative(num2txt(hours + 1))
        mins = num2txt(minutes)
        return  "Saat " + hours + " " + mins + " var."

for hour in range(1, 13, 1):
    for min in range(1, 60, 9):
        print(f"{hour:02d}:{min:02d} {translate_time(hour, min):25
s}")
```

```
01:01 Saat biri bir geçiyor.
01:10 Saat biri on geçiyor.
01:19 Saat biri on dokuz geçiyor.
01:28 Saat biri yirmi sekiz geçiyor.
01:37 Saat ikiye yirmi üç var.
01:46 Saat ikiye on dört var.
01:55 Saat ikiye beş var.
02:01 Saat ikiyi bir geçiyor.
02:10 Saat ikiyi on geçiyor.
02:19 Saat ikiyi on dokuz geçiyor.
02:28 Saat ikiyi yirmi sekiz geçiyor.
02:37 Saat üçe yirmi üç var.
02:46 Saat üçe on dört var.
02:55 Saat üçe beş var.
03:01 Saat üçü bir geçiyor.
03:10 Saat üçü on geçiyor.
03:19 Saat üçü on dokuz geçiyor.
03:28 Saat üçü yirmi sekiz geçiyor.
03:37 Saat dörte yirmi üç var.
03:46 Saat dörte on dört var.
03:55 Saat dörte beş var.
04:01 Saat dörtü bir geçiyor.
04:10 Saat dörtü on geçiyor.
04:19 Saat dörtü on dokuz geçiyor.
04:28 Saat dörtü yirmi sekiz geçiyor.
04:37 Saat beşe yirmi üç var.
04:46 Saat beşe on dört var.
04:55 Saat beşe beş var.
05:01 Saat beşi bir geçiyor.
05:10 Saat beşi on geçiyor.
05:19 Saat beşi on dokuz geçiyor.
05:28 Saat beşi yirmi sekiz geçiyor.
05:37 Saat altıya yirmi üç var.
05:46 Saat altıya on dört var.
05:55 Saat altıya beş var.
06:01 Saat altıyı bir geçiyor.
06:10 Saat altıyı on geçiyor.
06:19 Saat altıyı on dokuz geçiyor.
06:28 Saat altıyı yirmi sekiz geçiyor.
06:37 Saat yediye yirmi üç var.
06:46 Saat yediye on dört var.
06:55 Saat yediye beş var.
07:01 Saat yediyi bir geçiyor.
07:10 Saat yediyi on geçiyor.
07:19 Saat yediyi on dokuz geçiyor.
07:28 Saat yediyi yirmi sekiz geçiyor.
```

```
07:37 Saat sekize yirmi üç var.
07:46 Saat sekize on dört var.
07:55 Saat sekize beş var.
08:01 Saat sekizi bir geçiyor.
08:10 Saat sekizi on geçiyor.
08:19 Saat sekizi on dokuz geçiyor.
08:28 Saat sekizi yirmi sekiz geçiyor.
08:37 Saat dokuza yirmi üç var.
08:46 Saat dokuza on dört var.
08:55 Saat dokuza beş var.
09:01 Saat dokuzu bir geçiyor.
09:10 Saat dokuzu on geçiyor.
09:19 Saat dokuzu on dokuz geçiyor.
09:28 Saat dokuzu yirmi sekiz geçiyor.
09:37 Saat ona yirmi üç var.
09:46 Saat ona on dört var.
09:55 Saat ona beş var.
10:01 Saat onu bir geçiyor.
10:10 Saat onu on geçiyor.
10:19 Saat onu on dokuz geçiyor.
10:28 Saat onu yirmi sekiz geçiyor.
10:37 Saat on bire yirmi üç var.
10:46 Saat on bire on dört var.
10:55 Saat on bire beş var.
11:01 Saat on biri bir geçiyor.
11:10 Saat on biri on geçiyor.
11:19 Saat on biri on dokuz geçiyor.
11:28 Saat on biri yirmi sekiz geçiyor.
11:37 Saat on ikiye yirmi üç var.
11:46 Saat on ikiye on dört var.
11:55 Saat on ikiye beş var.
12:01 Saat on ikiyi bir geçiyor.
12:10 Saat on ikiyi on geçiyor.
12:19 Saat on ikiyi on dokuz geçiyor.
12:28 Saat on ikiyi yirmi sekiz geçiyor.
12:37 Saat bire yirmi üç var.
12:46 Saat bire on dört var.
12:55 Saat bire beş var.
```

```python
digits_list = ["bir", "iki", "üç",
               "dört", "beş", "altı",
               "yedi", "sekiz", "dokuz"]

tens_list = ["on", "yirmi", "otuz",  "kırk",  "elli"]

def num2txt(min):
    """ numbers are mapped into verbal representation """
    digits = min % 10
    tens = min // 10
    if digits > 0:
        if tens:
            return tens_list[tens-1] + " " + digits_list[digits-1]
        else:
            return digits_list[digits-1]
    else:
        return tens_list[tens-1]


def last_vowel(word):
    """ returns a tuple 'vowel', 'ends_with_vowel'
        'vowel' is the last vowel of the word 'word'
        'ends_with_vowel' is True if the word ends with a vowel
        False otherwise
    """
    vowels = {'ı', 'a', 'e', 'i', 'o', 'u', 'ö', 'ü'}
    ends_with_vowel = True
    for char in word[::-1]:
        if char in vowels:
            return char, ends_with_vowel
        ends_with_vowel = False


def dative(word):
    """  returns 'word' in dative form """
    lv, ends_with_vowel = last_vowel(word)
    if lv in {'i', 'e', 'ö', 'ü'}:
        ret_vowel = 'e'
    elif lv in {'a', 'ı', 'o', 'u'}:
        ret_vowel = 'a'
    if ends_with_vowel:
        return word + "y" + ret_vowel
    else:
        return word + ret_vowel


def accusative(word):
    """ return 'word' in accusative form """
    lv, ends_with_vowel = last_vowel(word)
    if lv in {'ı', 'a'}:
        ret_vowel = 'ı'
    elif lv in {'e', 'i'}:
        ret_vowel = 'i'
    elif lv in {'o', 'u'}:
        ret_vowel = 'u'
    elif lv in {'ö', 'ü'}:
        ret_vowel = 'ü'
    if ends_with_vowel:
        return word + "y" + ret_vowel
    else:
        return word + ret_vowel


def translate_time(hours, minutes=0):
    """ construes the verbal represention
        of the time 'hours':'minutes
    '"""
    if minutes == 0:
        return "Saat " + num2txt(hours) + "."
    elif minutes == 30:
        return "Saat " + num2txt(hours) + " buçuk."
    elif minutes == 15:
        return "Saat " + accusative(num2txt(hours)) + " çeyrek geç
iyor."
    elif minutes == 45:
        if hours == 12:
            hours = 0
        return "Saat " + dative(num2txt(hours+1)) + " çeyrek var."
    elif minutes < 30:
        return "Saat " + accusative(num2txt(hours)) + " " + num2tx
t(minutes) + " geçiyor."
    elif minutes > 30:
        minutes = 60 - minutes
        if hours == 12:
            hours = 0
        hours = dative(num2txt(hours + 1))
        mins = num2txt(minutes)
        return  "Saat " + hours + " " + mins + " var."
```

```
Overwriting turkish_time.py
```

Let us test the previous Python code with some times:

```
from turkish_time import translate_time
for hour, min in [(12, 47), (1, 23), (16, 9)]:
        print(f"{hour:02d}:{min:02d} {translate_time(hour, min):25
s}")
```

```
12:47 Saat bire on üç var.
01:23 Saat biri yirmi üç geçiyor.
16:09 Saat on altıyı dokuz geçiyor.
```

We implement in the following Python program a clock, which uses a graphical user interface. This clock shows the current time in a verbal representation. Of course, in Turkish:

In [ ]:

```
import tkinter as tk
import datetime
from turkish_time import translate_time

def get_time():
    now = datetime.datetime.now()
    return now.hour, now.minute

def time_text_label(label):
    def new_time():
        now = datetime.datetime.now()
        res = translate_time(now.hour, now.minute)
        label.config(text=res)
        label.after(1000, new_time)
    new_time()

flag_img = "images/turkish_flag_clock_background.png"
root = tk.Tk()
root.option_add( "*font", "lucida 36 bold" )
root.title("Turkish Time")
flag_img = tk.PhotoImage(file=flag_img)
flag_img = flag_img.subsample(4, 4) # shrink image, divide by 4
flag_label = tk.Label(root, image=flag_img)
flag_label.pack(side=tk.LEFT)
label = tk.Label(root,
                 font = "Latin 18 bold italic",
                 fg="dark green")
label.pack(side=tk.RIGHT)
time_text_label(label)
button = tk.Button(root,
                   text='Stop',
                   font = "Latin 12 bold italic",
                   command=root.destroy)
button.pack(side=tk.RIGHT)
root.mainloop()
```

TOWERS OF HANOI

INTRODUCTION

Why do we present a Python implementation of the "Towers of Hanoi"? The hello-world of recursion is the Factorial. This means, you will hardly find any book or tutorial about programming languages which doesn't deal with the first and introductory example about recursive functions. Another one is the calculation of the n-th Fibonacci number. Both are well suited for a tutorial because of their simplicity but they can be easily written in an iterative way as well.

If you have problems in understanding recursion, we recommend that you go through the chapter "Recursive Functions" of our tutorial.

That's different with the "Towers of Hanoi". A recursive solution almost forces itself on the programmer, while the iterative solution of the game is hard to find and to grasp. So, with the Towers of Hanoi we present a recursive Python program, which is hard to program in an iterative way.

ORIGIN

There is an old legend about a temple or monastery, which contains three poles. One of them filled with 64 gold disks. The disks are of different sizes, and they are put on top of each other, according to their size, i.e. each disk on the pole a little smaller than the one beneath it. The priests, if the legend is about a temple, or the monks, if it is about a monastery, have to move this stack from one of the three poles to another one. But one rule has to be applied: a large disk can never be placed on top of a smaller one. When they would have finished their work, the legend tells, the temple would crumble into dust, and the world would end.

But don't be afraid, it's not very likely that they will finish their work soon, because $2^{64} - 1$ moves are necessary, i.e. 18,446,744,073,709,551,615 to move the tower according to the rules.

But there is - most probably - no ancient legend. The legend and the game "towers of Hanoi" had been conceived by the French mathematician Edouard Lucas in 1883.

RULES OF THE GAME

The rules of the game are very simple, but the solution is not so obvious. The game "Towers of Hanoi" uses three rods. A number of disks is stacked in decreasing order from the bottom to the top of one rod, i.e. the largest disk at the bottom and the smallest one on top. The disks build a conical tower.

The aim of the game is to move the tower of disks from one rod to another rod.

The following rules have to be obeyed:

- Only one disk may be moved at a time.
- Only the most upper disk from one of the rods can be moved in a move
- It can be put on another rod, if this rod is empty or if the most upper disk of this rod is larger than the one which is moved.

NUMBER OF MOVES

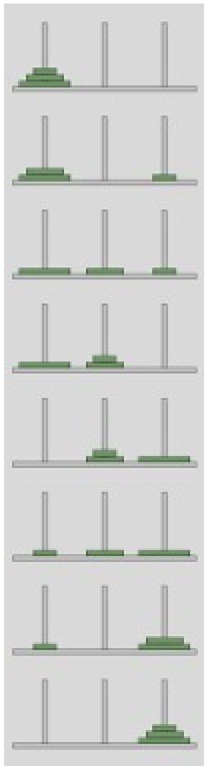
The number of moves necessary to move a tower with n disks can be calculated as: $2^n - 1$

PLAYING AROUND TO FIND A SOLUTION

From the formula above, we know that we need 7 moves to move a tower of size 3 from the most left rod (let's call it SOURCE to the most right tower (TARGET).

The pole in the middle (we will call it AUX) is needed as an auxiliary stack to deposit disks temporarily.

Before we examine the case with 3 disks, as it is depicted in the image on the right side, we will have a look at towers of size 1 (i.e. just one disk) and size 2. The solution for a tower with just one disk is straightforward: We will move the one disk on the SOURCE tower to the TARGET tower and we are finished.

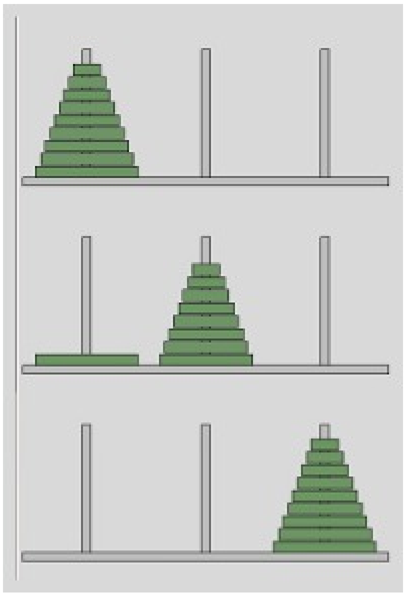


Let's look now at a tower with size 2, i.e. two disks. There are two possibilities to move the first disk, the disk on top of the stack of SOURCE: We can move this disk either to TARGET or to AUX.

- So let's start by moving the smallest disk from SOURCE to TARGET. Now there are two choices: We can move this disk again, either back to the SOURCE peg, which obviously doesn't make sense, or we could move it to AUX, which doesn't make sense either, because we could have moved there as our first step. So the only move which makes sense is moving the other disk, i.e. the largest disk, to peg AUX. Now, we have to move the smallest disk again, because we don't want to move the largest disk back to SOURCE again. We can move the smallest disk to AUX. Now we can see that we have moved the tower of size 2 to the peg AUX, but the target had been peg TARGET. We have already used the maximal number of moves, i.e. $2^2 - 1 = 3$
- Moving the smallest disk from peg SOURCE to TARGET as our first step has not shown to be successful. So, we will move this disk to peg AUX in our first step. After this we move the second disk to TARGET. After this we move the smallest disk from AUX to TARGET and we have finished our task!

  </ul> We have seen in the cases n=1 and n=2 that it depends on the first move, if we will be able to successfully and with the minimal number of moves solve the riddle. We know from our formula that the minimal number of moves necessary to move a tower of size 3 from the SOURCE peg to the target peg is 7 ($2^3 - 1$)
  You can see in the solution, which we present in our image that the first disk has to be moved from the peg SOURCE to the peg TARGET. If your first step consists of moving the smallest disk to AUX, you will not be capable of finishing the task with less than 9 moves.

Let's number the disks as D_1 (smallest), D_2 and D_3 (largest) and name the pegs as S (SOURCE peg), A (AUX), T (TARGET). We can see that we move in three moves the tower of size 2 (the disks D_1 and D_2) to A. Now we can move D_3 to T, where it is finally positioned. The last three moves move the tower consisting of D_2D_1 from peg A to T to place them on top of D_3.



There is a general rule for moving a tower of size n (n > 1) from the peg S to the peg T:

- move a tower of n - 1 discs D_{n-1} ... D_1 from S to A. Disk D_n is left alone on peg S
- Move disk D_n to T
- move the tower of n - 1 discs D_{n-1} ... D_1 on A to T, i.e. this tower will be put on top of Disk D_n

</ul> The algorithm, which we have just defined, is a recursive algorithm to move a tower of size n. It actually is the one, which we will use in our Python implementation to solve the Towers of Hanoi. Step 2 is a simple move of a disk. But to accomplish the steps 1 and 3, we apply the same algorithm again on a tower of n-1. The calculation will finish with a finite number of steps, because very time the recursion will be started with a tower which is 1 smaller than the one in the calling function. So finally we will end up with a tower of size n = 1, i.e. a simple move.

RECURSIVE PYTHON PROGRAM

The following Python script contains a recursive function "hanoi", which implements a recursive solution for Towers of Hanoi:

```
def hanoi(n, source, helper, target):
    if n > 0:
        # move tower of size n - 1 to helper:
        hanoi(n - 1, source, target, helper)
        # move disk from source peg to target peg
        if source:
            target.append(source.pop())
        # move tower of size n-1 from helper to target
        hanoi(n - 1, helper, source, target)

source = [4,3,2,1]
target = []
```

```python
helper = []
hanoi(len(source),source,helper,target)

print(source, helper, target)
```

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

This function is an implementation of what we have explained in the previous subchapter. First we move a tower of size n-1 from the peg `source` to the `helper` peg. We do this by calling

`hanoi(n - 1, source, target, helper)`

After this, there will be the largest disk left on the peg source. We move it to the empty peg target by the statement

```
            if source:
                target.append(source.pop())
```

After this, we have to move the tower from "helper" to "target", i.e. on top of the largest disk:

```
            hanoi(n - 1, helper, source, target)
```

If you want to check, what's going on, while the recursion is running, we suggest the following Python programm. We have slightly changed the data structure. Instead of passing just the stacks of disks to the function, we pass tuples to the function. Each tuple consists of the stack and the function of the stack:

```python
def hanoi(n, source, helper, target):
    print("hanoi( ", n, source, helper, target, " called")
    if n > 0:
        # move tower of size n - 1 to helper:
        hanoi(n - 1, source, target, helper)
        # move disk from source peg to target peg
        if source[0]:
            disk = source[0].pop()
            print("moving " + str(disk) + " from " + source[1] +
" to " + target[1])
            target[0].append(disk)
        # move tower of size n-1 from helper to target
        hanoi(n - 1, helper, source, target)

source = ([4,3,2,1], "source")
target = ([], "target")
helper = ([], "helper")
hanoi(len(source[0]),source,helper,target)

print(source, helper, target)
```

```
hanoi(  4 ([4, 3, 2, 1], 'source') ([], 'helper') ([], 'target')
called
hanoi(  3 ([4, 3, 2, 1], 'source') ([], 'target') ([], 'helper')
called
hanoi(  2 ([4, 3, 2, 1], 'source') ([], 'helper') ([], 'target')
called
hanoi(  1 ([4, 3, 2, 1], 'source') ([], 'target') ([], 'helper')
called
hanoi(  0 ([4, 3, 2, 1], 'source') ([], 'helper') ([], 'target')
called
moving 1 from source to helper
hanoi(  0 ([], 'target') ([4, 3, 2], 'source') ([1], 'helper')  ca
lled
moving 2 from source to target
hanoi(  1 ([1], 'helper') ([4, 3], 'source') ([2], 'target')  call
ed
hanoi(  0 ([1], 'helper') ([2], 'target') ([4, 3], 'source')  call
ed
moving 1 from helper to target
hanoi(  0 ([4, 3], 'source') ([], 'helper') ([2, 1], 'target')  ca
lled
moving 3 from source to helper
hanoi(  2 ([2, 1], 'target') ([4], 'source') ([3], 'helper')  call
ed
hanoi(  1 ([2, 1], 'target') ([3], 'helper') ([4], 'source')  call
ed
hanoi(  0 ([2, 1], 'target') ([4], 'source') ([3], 'helper')  call
ed
moving 1 from target to source
hanoi(  0 ([3], 'helper') ([2], 'target') ([4, 1], 'source')  call
ed
moving 2 from target to helper
hanoi(  1 ([4, 1], 'source') ([], 'target') ([3, 2], 'helper')  ca
lled
hanoi(  0 ([4, 1], 'source') ([3, 2], 'helper') ([], 'target')  ca
lled
moving 1 from source to helper
hanoi(  0 ([], 'target') ([4], 'source') ([3, 2, 1], 'helper')  ca
lled
moving 4 from source to target
hanoi(  3 ([3, 2, 1], 'helper') ([], 'source') ([4], 'target')  ca
lled
hanoi(  2 ([3, 2, 1], 'helper') ([4], 'target') ([], 'source')  ca
lled
hanoi(  1 ([3, 2, 1], 'helper') ([], 'source') ([4], 'target')  ca
lled
```

```
hanoi(  0 ([3, 2, 1], 'helper') ([4], 'target') ([], 'source')  ca
lled
moving 1 from helper to target
hanoi(  0 ([], 'source') ([3, 2], 'helper') ([4, 1], 'target')  ca
lled
moving 2 from helper to source
hanoi(  1 ([4, 1], 'target') ([3], 'helper') ([2], 'source')  call
ed
hanoi(  0 ([4, 1], 'target') ([2], 'source') ([3], 'helper')  call
ed
moving 1 from target to source
hanoi(  0 ([3], 'helper') ([4], 'target') ([2, 1], 'source')  call
ed
moving 3 from helper to target
hanoi(  2 ([2, 1], 'source') ([], 'helper') ([4, 3], 'target')  ca
lled
hanoi(  1 ([2, 1], 'source') ([4, 3], 'target') ([], 'helper')  ca
lled
hanoi(  0 ([2, 1], 'source') ([], 'helper') ([4, 3], 'target')  ca
lled
moving 1 from source to helper
hanoi(  0 ([4, 3], 'target') ([2], 'source') ([1], 'helper')  call
ed
moving 2 from source to target
hanoi(  1 ([1], 'helper') ([], 'source') ([4, 3, 2], 'target')  ca
lled
hanoi(  0 ([1], 'helper') ([4, 3, 2], 'target') ([], 'source')  ca
lled
moving 1 from helper to target
hanoi(  0 ([], 'source') ([], 'helper') ([4, 3, 2, 1], 'target')
called
([], 'source') ([], 'helper') ([4, 3, 2, 1], 'target')
```

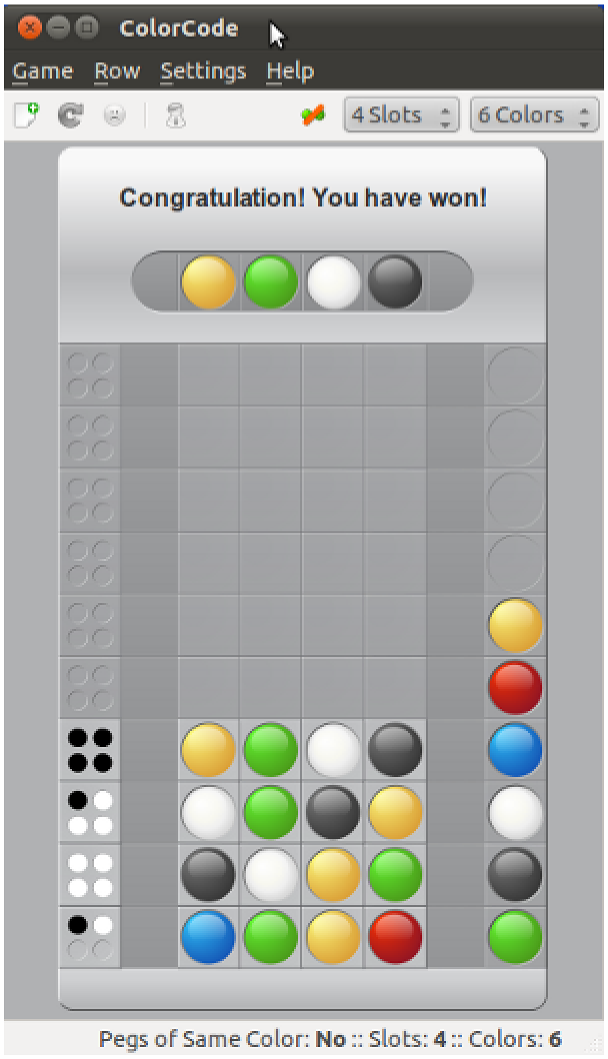
MASTERMIND / BULLS AND COWS

INTRODUCTION

In this chapter of our advanced Python topics we present an implementation of the game Bulls and Cows. This game, which is also known as "Cows and Bulls" or "Pigs and Bulls", is an old code-breaking game played by two players. The game goes back to the 19th century and can be played with paper and pencil. Bulls and Cows -- also known as Cows and Bulls or Pigs and Bulls or Bulls and Cleots -- was the inspirational source of Mastermind, a game invented in 1970 by Mordecai Meirowitz. The game is played by two players. Mastermind and "Bulls and Cows" are very similar and the underlying idea is essentially the same, but Mastermind is sold in a box with a decoding board and pegs for the coding and the feedback pegs. Mastermind uses colours as the underlying code information, while Bulls and Cows uses digits.

RULES OF THE GAME

Bulls and Cows



Congratulation! You have won!

Pegs of Same Color: **No** :: Slots: **4** :: Colors: **6**

We start with the rules of "Bulls and Cows":
It's a pencil and paper game played by two players. The two players write a 4-digit number on a sheet of paper. The digits must be all different, but there is a version, where digits can be used more than once. Each player has to find out the opponent's secret code. To this purpose, the players - in turn - try to guess the opponent's number. The opponent has to score the guess: A "bull" is a digit which is located at the right position. If, for example, the hidden code is "4 3 2 5" and the guess is "4 3 1 2", then we have the two bulls "4" and "3" in the guess "4 3 1 2". A "Cow" on the other hand is a correct number, which is on the wrong position. The "2" of the previous example is a cow.

The first player to reveal the other's secret number is the winner of the game.

The secret numbers for bulls and cows are usually 4-digit-numbers, but the game can be played with 3 to 6 digit numbers.

There are lots of computer implementations of "Bulls and Cows". The first one was a program called moo, written in PL/I.

Mastermind

Mastermind obeys essentially the same rules as "Bulls and Cows", but colours are used instead of digits. Mastermind is a commercial board game, which is played on a decoding board with a shield on one side hiding a row of four large holes to put in coloured pegs, i.e. the secret code, which has to be found out by the opponent. There are 8 to 12 rows of four holes for the guesses, which are open to the view. Beside of each row, there is a set of four small holes for the black and white score pegs.

THE IMPLEMENTATION

The following program is a Python implementation (Python3), which is capable of breaking a colour code of four positions and six colours, but these numbers are scalable. The colour code is not allowed to contain multiple occurrences of colours. We need the function all_colours from the following combinatorics module, which can be saved as combinatorics.py:

In [ ]:

```
import random

def fac(n):
    if n == 0:
        return 1
    else:
        return (fac(n-1) * n)

def permutations(items):
    n = len(items)
    if n==0: yield []
    else:
        for i in range(len(items)):
            for cc in permutations(items[:i]+items[i+1:]):
                yield [items[i]]+cc

def k_permutations(items, n):
    if n==0: yield []
    else:
        for i in range(len(items)):
            for ss in k_permutations(items, n-1):
                if (not items[i] in ss):
                    yield [items[i]]+ss

def random_permutation(list):
    length = len(list);
    max = fac(length);
    index = random.randrange(0, max)
    i = 0
    for p in permutations(list):
        if i == index:
            return p
        i += 1

def all_colours(colours, positions):
    colours = random_permutation(colours)
    for s in k_permutations(colours, positions):
        yield(s)
```

If you want to know more about permutations, you may confer to our chapter about "generators and iterators". The function all_colours is essentially like k_permutations, but it starts the sequence of permutations with a random permutations, because we want to make sure that the computer starts each new game with a completely new guess.

The colours, which are used for the guesses, are defined in a list assigned to the variable "colours". It's possible to put in different elements into this list, e.g.

In [ ]:

```
colours = ["a", "b", "c", "d", "e", "f"]
```

or something like the following assignment, which doesn't make a lot of sense in terms of colours:

In [ ]:

```
colours = ["Paris blue", "London green", "Berlin black", "Vienna y
ellow", "Frankfurt red", "Hamburg brown"]
```

The list "guesses" has to be empty. All the guesses with the scores from the human player will be saved in this list. It might look like this in a game:

```
[(['pink', 'green', 'blue', 'orange'], (1, 1)),
 (['pink', 'blue', 'red', 'yellow'], (1, 2)),
 (['pink', 'orange', 'yellow', 'red'], (0, 2))]
```

We can see that guesses is a list of tuples. Each tuple contains a list of colours[1] and a 2-tuple with the scores from the human player, e.g. (1, 2) with 1 the number of "bulls" or "blacks" in the Mastermind lingo and the 2 is the number of "cows" (or "whites" in the mastermind jargon) in ['pink', 'blue', 'red', 'yellow']

We use the generator all_colours() to create the first guess:

```
number_of_positions = 4
permutation_iterator = all_colours(colours, number_of_positions)
current_colour_choices = next(permutation_iterator)
current_colour_choices
```

The following while loop presents the guesses to the human player, evaluates the answers and produces a new guess. The while loop ends, if either the number of "black" ("cows") (new_guess[1][0]) is equal to number_of_positions or if new_guess[1][0] == -1, which means that the answers were inconsistent.

We have a closer look at the function new_evaluation. At first, it calls the function get_evaluation() which returns the "bulls" and "cows" or terms of our program the values for rightly_positioned and permutated.

```
def get_evaluation():
   """ asks the human player for an evaluation """
   show_current_guess(new_guess[0])
   rightly_positioned = int(input("Blacks: "))
```

```
        permutated = int(input("Whites: "))
        return (rightly_positioned, permutated)
```

The game is over, if the number of rightly positioned colours, as returned by the function get_evaluation(), is equal to the number_of_positions, i.e. 4:

```
        if rightly_positioned == number_of_positions:
           return(current_colour_choices, (rightly_positioned, permutate
    d))
```

In the next if statement, we check, if the human answer makes sense. There are combinations of whites and blacks, which don't make sense, for example three blacks and one white don't make sense, as you can easily understand. The function answer_correct() is used to check, if the input makes sense:

```
    def answer_ok(a):
       (rightly_positioned, permutated) = a
       if (rightly_positioned + permutated > number_of_positions) \
           or (rightly_positioned + permutated < len(colours) - numbe
    r_of_positions):
          return False
       if rightly_positioned == 3 and permutated == 1:
          return False
       return True
```

The function answer_ok() should be self-explanatory. If answer_ok returns False, new_evaluation will be left with the return value -1 for the blacks, which in turn will end the while loop in the main program:

```
        if not answer_ok((rightly_positioned, permutated)):
           print("Input Error: Sorry, the input makes no sense")
           return(current_colour_choices, (-1, permutated))
```

If the check was True, the guess will be appended to the previous guesses and will be shown to the user:

```
        guesses.append((current_colour_choices, (rightly_positioned, per
    mutated)))
        view_guesses()
```

After this steps, a new guess will be created. If a new guess could be created, it will be shown, together with the black and white values from the previous guess, which are checked in the while loop. If a new guess can not be created, a -1 will be returned for the blacks:

```
        current_colour_choices = create_new_guess()
        if not current_colour_choices:
           return(current_colour_choices, (-1, permutated))
        return(current_colour_choices, (rightly_positioned, permutated))
```

The following code is the complete program, which you can save and start, but don't forget to use Python3:

```
import random
from combinatorics import all_colours

def inconsistent(p, guesses):
    """ the function checks, if a permutation p, i.e. a list of
colours like p = ['pink', 'yellow', 'green', 'red'] is consistent
with the previous colours. Each previous colour permuation gues
s[0]
compared (check()) with p has to return the same amount of blacks
 (rightly positioned colours) and whites (right colour at wrong
position) as the corresponding evaluation (guess[1] in the
list guesses) """
    for guess in guesses:
        res = check(guess[0], p)
        rightly_positioned, permutated = guess[1]
        if res != [rightly_positioned, permutated]:
           return True # inconsistent
    return False # i.e. consistent

def answer_ok(a):
    """ checking of an evaluation given by the human player makes
    sense. 3 blacks and 1 white make no sense for example. """
    rightly_positioned, permutated = a
    if (rightly_positioned + permutated > number_of_positions) \
       or (rightly_positioned + permutated < len(colours) - numbe
r_of_positions):
        return False
    if rightly_positioned == 3 and permutated == 1:
        return False
    return True

def get_evaluation():
    """ asks the human player for an evaluation """
    show_current_guess(new_guess[0])
    rightly_positioned = int(input("Blacks: "))
    permutated = int(input("Whites: "))
    return (rightly_positioned, permutated)

def new_evaluation(current_colour_choices):
    """ This funtion gets an evaluation of the current guess, chec
ks
    the consistency of this evaluation, adds the guess together wi
th
    the evaluation to the list of guesses, shows the previous gues
ses
```

```
    and creates a new guess """
    rightly_positioned, permutated = get_evaluation()
    if rightly_positioned == number_of_positions:
        return current_colour_choices, (rightly_positioned, permut
ated)

    if not answer_ok((rightly_positioned, permutated)):
        print("Input Error: Sorry, the input makes no sense")
        return current_colour_choices, (-1, permutated)
    guesses.append((current_colour_choices, (rightly_positioned, p
ermutated)))
    view_guesses()

    current_colour_choices = create_new_guess()
    if not current_colour_choices:
        return current_colour_choices, (-1, permutated)
    return(current_colour_choices, (rightly_positioned, permutate
d))


def check(p1, p2):
    """ check() calculates the number of bulls (blacks) and cows
(whites)
    of two permutations """
    blacks = 0
    whites = 0
    for i in range(len(p1)):
        if p1[i] == p2[i]:
            blacks += 1
        else:
            if p1[i] in p2:
                whites += 1
    return [blacks, whites]

def create_new_guess():
    """ a new guess is created, which is consistent to the
    previous guesses """
    next_choice = next(permutation_iterator)
    while inconsistent(next_choice, guesses):
        try:
            next_choice = next(permutation_iterator)
        except StopIteration:
            print("Error: Your answers were inconsistent!")
            return ()
    return next_choice
```

```python
def show_current_guess(new_guess):
    """ The current guess is printed to stdout """
    print("New Guess: ", end=" ")

    for c in new_guess:
        print(c, end=" ")
    print()

def view_guesses():
    """ The list of all guesses with the corresponding evaluation
s
        is printed """
    print("Previous Guesses:")
    for guess in guesses:
        guessed_colours = guess[0]
        for c in guessed_colours:
            print(c, end=" ")
        for i in guess[1]:
            print(" %i " % i, end=" ")
        print()


colours = ["red","green","blue","yellow","orange","pink"]
guesses = []
number_of_positions = 4

permutation_iterator = all_colours(colours, number_of_positions)
current_colour_choices = next(permutation_iterator)

new_guess = (current_colour_choices, (0,0) )
while (new_guess[1][0] == -1) or (new_guess[1][0] != number_of_pos
itions):
    new_guess = new_evaluation(new_guess[0])
```

```
New Guess:  red green pink yellow

Previous Guesses:
red green pink yellow  2   0
New Guess:  red green blue orange

Previous Guesses:
red green pink yellow  2   0
red green blue orange  2   2
New Guess:  red green orange blue
```

PYTHON AND JSON

INTRODUCTION

JSON stands for JavaScript Object Notation. JSON is an open standard file and data interchange format. The content of a JSON file or JSON data is human-readable. JSON is used for storing and exchanging data. The JSON data objects consist of attribute–value pairs. The data format of JSON looke very similar to a Python dictionary, but JSON is a language-independent data format. The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. JSON filenames use the extension .json.

DUMPS AND LOAD

It is possible to serialize a Python dict object to a JSON formatted string by using `dumps` from the `json` module:

```
import json

d = {"a": 3, "b": 3, "c": 12}

json.dumps(d)
```

Output:
```
'{"a": 3, "b": 3, "c": 12}'
```

The JSON formatted string looks exactly like a Python dict in a string format. In the followoing example, we can see a difference: "True" and "False" are turned in "true" and "false":

```
d = {"a": True, "b": False, "c": True}

d_json = json.dumps(d)
d_json
```

Output:
```
'{"a": true, "b": false, "c": true}'
```

We can transform the json string back in a Python dictionary:

```
json.loads(d_json)
```

Output:
```
{'a': True, 'b': False, 'c': True}
```

DIFFERENCES BETWEEN JSON AND PYTHON DICTIONARIES

If you got the idea that turning dictionaries in json strings is always structure-preserving, you are wrong:

```
persons = {"Isabella": {"surname": "Jones",
                        "address": ("Bright Av.",
                                    34,
                                    "Village of Sun")},
           "Noah": {"surname": "Horton",
                    "address": (None,
                                None,
                                "Whoville")}
          }


persons_json = json.dumps(persons)
print(persons_json)
```

```
{"Isabella": {"surname": "Jones", "address": ["Bright Av.", 34, "V
illage of Sun"]}, "Noah": {"surname": "Horton", "address": [null,
null, "Whoville"]}}
```

We can see that the address tuple is turned into a list!

```
json.loads(persons_json)
```

Output:
```
{'Isabella': {'surname': 'Jones',
  'address': ['Bright Av.', 34, 'Village of Sun']},
 'Noah': {'surname': 'Horton', 'address': [None, None, 'Whovi
lle']}}
```

You can prettyprint JSON by using the optinional `indent` parameter:

```
persons_json = json.dumps(persons, inden
t=4)
print(persons_json)
```

```
{
    "Isabella": {
        "surname": "Jones",
        "address": [
            "Bright Av.",
            34,
            "Village of Sun"
        ]
    },
    "Noah": {
        "surname": "Horton",
        "address": [
            null,
            null,
            "Whoville"
        ]
    }
}
```

RELATIONSHIP BETWEEN PYTHON DICTS AND JSON OBJECTS

PYTHON OBJECT	JSON OBJECT
dict	object
list, tuple	array
str	string
int, long, float	numbers
True	true
False	false
None	null

```python
import json

d = {"d": 45, "t": 123}
x = json.dumps(d)
print(x)
```

```
lst = [34, 345, 234]
x = json.dumps(lst)
print(x)

int_obj = 199
x = json.dumps(int_obj)
print(x)
```

```
{"d": 45, "t": 123}
[34, 345, 234]
199
```

There is another crucial difference, because JSON accepts onls keys `str`, `int`, `float`, `bool` or `None` as keys, as we can see in the following example:

```
board = {(1, "a"): ("white", "rook"),
         (1, "b"): ("white", "knight"),
         (1, "c"): ("white", "bishop"),
         (1, "d"): ("white", "queen"),
         (1, "e"): ("white", "king"),
         # further data skipped
        }
```

Calling `json.dumps` with `board` as an argument would result in the exeption `TypeError: keys must be str, int, float, bool or None, not tuple`.

To avoid this, we could use the optional key `skipkeys`:

```
board_json = json.dumps(board,
                        skipkeys=True)

board_json
```

Output: `'{}'`

We avoided the exception, but the result is not satisfying, because the data is missing!

A better solution is to turn the tuples into string, as we do in the following:

```
board2 = dict((str(k), val) for k, val in board.items())
board2
```

Output:
```
{"(1, 'a')": ('white', 'rook'),
 "(1, 'b')": ('white', 'knight'),
 "(1, 'c')": ('white', 'bishop'),
 "(1, 'd')": ('white', 'queen'),
 "(1, 'e')": ('white', 'king')}
```

```
board_json = json.dumps(board2)
board_json
```

Output:
```
'{"(1, \'a\')": ["white", "rook"], "(1, \'b\')": ["white", "k
night"], "(1, \'c\')": ["white", "bishop"], "(1, \'d\')": ["w
hite", "queen"], "(1, \'e\')": ["white", "king"]}'
```

```


board2 = dict((str(k[0])+k[1], val) for k, val in board.items())
board2
```

Output:
```
{'1a': ('white', 'rook'),
 '1b': ('white', 'knight'),
 '1c': ('white', 'bishop'),
 '1d': ('white', 'queen'),
 '1e': ('white', 'king')}
```

board_json = json.dumps(board2) board_json

```
board2 = dict((str(key[0])+key[1], value) for key, value in boar
d.items())
board2
```

Output:
```
{'1a': ('white', 'rook'),
 '1b': ('white', 'knight'),
 '1c': ('white', 'bishop'),
 '1d': ('white', 'queen'),
 '1e': ('white', 'king')}
```

```
board_json = json.dumps(board2)
board_json
```

Output:
```
'{"1a": ["white", "rook"], "1b": ["white", "knight"], "1c":
["white", "bishop"], "1d": ["white", "queen"], "1e": ["whit
e", "king"]}'
```

In [ ]:

```

```

READING A JSON FILE

We will read in now a JSON example file `json_example.jsonwhich can be found in our` data `directory. We use an example from` json.org```.

```
json_ex = json.load(open("data/json_example.json"))
print(type(json_ex), json_ex)
```

```
<class 'dict'> {'glossary': {'title': 'example glossary', 'GlossDi
v': {'title': 'S', 'GlossList': {'GlossEntry': {'ID': 'SGML', 'Sor
tAs': 'SGML', 'GlossTerm': 'Standard Generalized Markup Languag
e', 'Acronym': 'SGML', 'Abbrev': 'ISO 8879:1986', 'GlossDef': {'pa
ra': 'A meta-markup language, used to create markup languages suc
h as DocBook.', 'GlossSeeAlso': ['GML', 'XML']}, 'GlossSee': 'mark
up'}}}}}
```

if you work with jupyter-lab or jupyter-notebook, you might have wondered about the data format used by it. You may guess already: It is JSON. Let's read in a notebook file with the extension ".ipynb":

```
nb = json.load(open("data/example_notebook.ipynb"))
print(nb)
```

```
{'cells': [{'cell_type': 'markdown', 'metadata': {}, 'source':
['# Titel\n', '\n', '## Introduction\n', '\n', 'This is some text\
n', '\n', '- apples\n', '- pears\n', '- bananas']}, {'cell_type':
'markdown', 'metadata': {}, 'source': ['# some code\n', '\n', 'x
= 3\n', 'y = 4\n', 'z = x + y\n']}], 'metadata': {'kernelspec':
{'display_name': 'Python 3', 'language': 'python', 'name': 'python
3'}, 'language_info': {'codemirror_mode': {'name': 'ipython', 'ver
sion': 3}, 'file_extension': '.py', 'mimetype': 'text/x-python',
'name': 'python', 'nbconvert_exporter': 'python', 'pygments_lexe
r': 'ipython3', 'version': '3.7.6'}}, 'nbformat': 4, 'nbformat_min
or': 4}
```

```
for key, value in nb.items():
    print(f"{key}:\n    {value}")
```

```
cells:
    [{'cell_type': 'markdown', 'metadata': {}, 'source': ['# Tite
l\n', '\n', '## Introduction\n', '\n', 'This is some text\n',
'\n', '- apples\n', '- pears\n', '- bananas']}, {'cell_type': 'mar
kdown', 'metadata': {}, 'source': ['# some code\n', '\n', 'x = 3\
n', 'y = 4\n', 'z = x + y\n']}]
metadata:
    {'kernelspec': {'display_name': 'Python 3', 'language': 'pytho
n', 'name': 'python3'}, 'language_info': {'codemirror_mode': {'nam
e': 'ipython', 'version': 3}, 'file_extension': '.py', 'mimetyp
e': 'text/x-python', 'name': 'python', 'nbconvert_exporter': 'pyth
on', 'pygments_lexer': 'ipython3', 'version': '3.7.6'}}
nbformat:
    4
nbformat_minor:
    4
```

```python
fh = open("data/disaster_mission.json")
data = json.load(fh)
print(list(data.keys()))
```

```
['Reference number', 'Country', 'Name', 'Function']
```

READ JSON WITH PANDAS

We can read a JSON file with the modue Pandas as well.

```python
import pandas

data = pandas.read_json("data/disaster_mission.json")
data
```

Output:

	Reference number	Country	Name	Function
0	1	France	Thomas Durand	Medical Aid
1	2	France	Maxime Petit	Social Worker
2	3	France	Alexandre Petit	Medical Aid
3	4	Switzerland	Mélissa Meier	Social Worker
4	5	Switzerland	Daniel Schneider	Medical Aid
...	...	...	...	...
195	196	Germany	Manfred Weber	Security Aid
196	197	France	Marie Dubois	Security Aid
197	198	France	Alexandre Dubois	Medical Aid
198	199	France	Nicolas Fontaine	Medical Aid
199	200	Switzerland	Laura Schmid	Medical Aid

200 rows × 4 columns

WRITE JSON FILES WITH PANDAS

We can also write data to Pandas files:

```
import pandas as pd
data.to_json("data/disaster_mission2.txt")
```

CREATING MUSICAL SCORES WITH PYTHON

INTRODUCTION

In this chapter of our Python course, we provide a tutorial on music engravings. We use Python to create an input file for the music engraving program Lilypond. Our Python program will translate an arbitrary text into a musical score. Each character of the alphabet is translated by our Python program into a one or more notes of a music piece.

LILYPOND

Lilypond Logo Before we can start with the code for our Python implementation, we have to give some information about Lilypond. What is Lilypond? The makers of Lilypond define it like this on lilypond.org:

"LilyPond is a music engraving program, devoted to producing the highest-quality sheet music possible. It brings the aesthetics of traditionally engraved music to computer printouts."

They describe their goal as following: "LilyPond came about when two musicians wanted to go beyond the soulless look of computer-printed sheet music. Musicians prefer reading beautiful music, so why couldn't programmers write software to produce elegant printed parts?

The result is a system which frees musicians from the details of layout, allowing them to focus on making music. LilyPond works with them to create publication-quality parts, crafted in the best traditions of classical music engraving."

A simple example to get you started with Lilypond:

```
%%writefile simple.ly
\version "2.12.3"
{
  c' e' g' a'
}
```

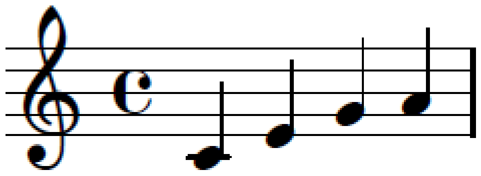
```
Overwriting simple.ly
```

```
!lilypond simple.ly
```

```
GNU LilyPond 2.20.0
Processing `simple.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `/tmp/lilypond-4T11WW'...
Converting to `simple.pdf'...
Deleting `/tmp/lilypond-4T11WW'...
Success: compilation successfully completed
```

```
!evince simple.pdf
```

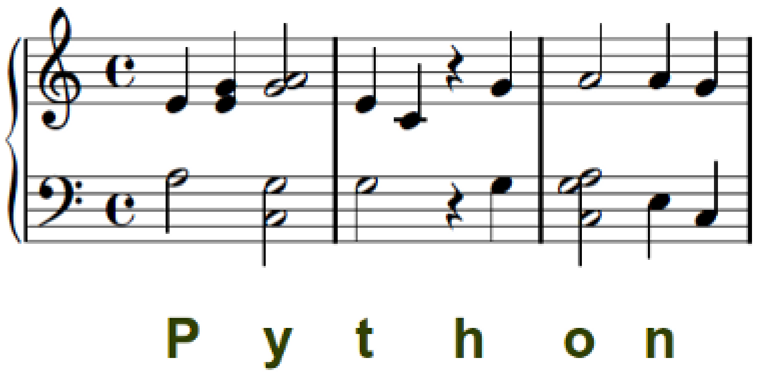
The result is saved in a pdf file "simple.pdf", which looks like this:



We do not want to give a complete tutorial. We recomment using the Learning Manual

USING PYTHON WITH LILYPOND

We write a program, which translates an arbitrary text string into a piece of music, which can be played on the piano. To this purpose every character of the Latin alphabet is mapped to two quarters of a four-four time, both for the left and right hand of a piano score.



The image on the right side illustrates this with the String "Python". We use a pentatonic scale to ensure that the result will not sound too bad.

We implement the mapping with a Python dictionary:

```
char2notes = {
  ' ':("a4 a4 ", "r2 "),
  'a':("<c a>2 ", "<e' a'>2 "),
  'b':("e2 ", "e'4 <e' g'> "),
  'c':("g2 ", "d'4 e' "),
  'd':("e2 ", "e'4 a' "),
  'e':("<c g>2 ", "a'4 <a' c'> "),
  'f':("a2 ", "<g' a'>4 c'' "),
  'g':("a2 ", "<g' a'>4 a' "),
  'h':("r4 g ", " r4 g' "),
  'i':("<c e>2 ", "d'4 g' "),
  'j':("a4 a ", "g'4 g' "),
  'k':("a2 ", "<g' a'>4 g' "),
  'l':("e4 g ", "a'4 a' "),
  'm':("c4 e ", "a'4 g' "),
  'n':("e4 c ", "a'4 g' "),
  'o':("<c a g>2  ", "a'2 "),
  'p':("a2 ", "e'4 <e' g'> "),
  'q':("a2 ", "a'4 a' "),
  'r':("g4 e ", "a'4 a' "),
  's':("a2 ", "g'4 a' "),
  't':("g2 ", "e'4 c' "),
  'u':("<c e g>2  ", "<a' g'>2"),
  'v':("e4 e ", "a'4 c' "),
  'w':("e4 a ", "a'4 c' "),
  'x':("r4 <c d> ", "g' a' "),
  'y':("<c g>2  ", "<a' g'>2"),
  'z':("<e a>2 ", "g'4 a' "),
  '\n':("r1 r1 ", "r1 r1 "),
  ',':("r2 ", "r2"),
  '.':("<c e a>2 ", "<a c' e'>2")
}
```

The mapping of a string to the notes can be realized by a simple for loop in Python:

```
txt = "Love one another and you will be happy. It is as simple as
that."

upper_staff = ""
lower_staff = ""
for i in txt.lower():
    (l, u) = char2notes[i]
    upper_staff += u
    lower_staff += l
```

The following code sequence embeds the strings upper_staff and lower_staff into a Lilypond format, which can be processed by Lilypond:

```
staff = "{\n\\new PianoStaff << \n"
staff += "  \\new Staff {" + upper_staff + "}\n"
staff += "  \\new Staff { \clef bass " + lower_staff + "}\n"
staff += ">> \midi {}\n}\n"

title = """\header {
  title = "Love One Another"
  composer = "Bernd Klein using Python"
  tagline = "Copyright: Bernd Klein"
}"""

open("piano_score.ly", "w").write(title + staff)
```

Output:
```
1023
```

Putting the code together and saving it under text_to_music.py, we can create our piano score on the command with the following command:

```
!lilypond piano_score.ly
```

```
GNU LilyPond 2.20.0
Processing `piano_score.ly'
Parsing...
piano_score.ly:9:4: error: syntax error, unexpected \midi
>>
   \midi {}
piano_score.ly:5:2: error: errors found, ignoring music expression
}
 {
piano_score.ly:1: warning: no \version statement found, please add

\version "2.20.0"

for future compatibility
fatal error: failed files: "piano_score.ly"
```

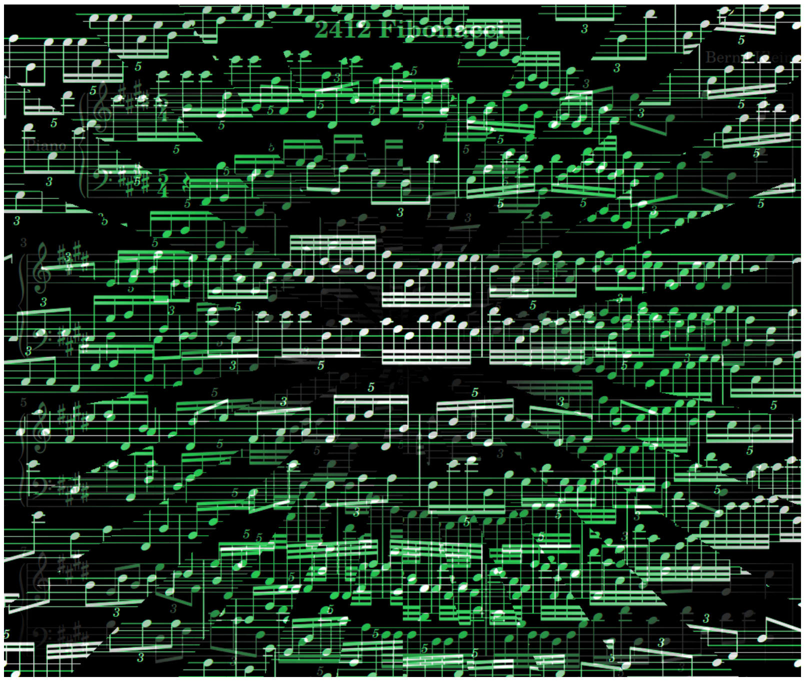
```
!evince piano_score.pdf
```

In [ ]:

```

```

MUSIC AND FIBONACCI

FIBONACCI SEQUENCE



The Fibonacci sequence or numbers - named after Leonardo of Pisa, also known as Fibonacci - have fascinated not only mathematicians for centuries. Fibonacci formulated an exercise about the rabbits and their reproduction: In the beginning there is one pair of rabbits (male and female), born out of the blue sky. It takes one month until they can mate. At the end of the second month the female gives birth to a new pair of rabbits. It works the same way with every newly born pair of rabbits, i.e. that it takes one month until the female one can mate and after another month she gives birth to a new pair of rabbits. Now let's suppose that every female rabbit will bring forth another pair of rabbits every month after the end of the first month. Be aware of the fact that these "mathematical" rabbits are immortal. So the population for the the generations look like this:

1, 1, 2, 3, 5, 8, 13, 21, ...

We can easily see that each new number is the sum of the previous two.

We get to music very soon, and feel free to skip the mathematics, but one thing is also worth mentioning. The Fibonacci numbers are strongly related to the Golden Ratio φ:

$$\varphi = \frac{1+\sqrt{5}}{2}$$

because the quotient of last and the previous to last number in this seqence is getting closer and closer to φ:

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \varphi$$

($F_n stands for the n-th Fibonacci number)

The Fibonacci numbers, often presented in conjunction with the golden ratio, are a popular theme in culture.

The Fibonacci numbers have been used in the visual art and architecture. The have been used in music very often. My favorite example is the song Lateralus by Tool. The text is rhythmically grouped in Fibonacci numbers. If you look at the following lines , you can count the syllables and you will get 1, 1, 2, 3, 5, 8, 5, 3, 13, 8, 5, 3:

```
Black
then
white are
all I see
in my infancy
Red and yellow then came to be,
reaching out to me
Lets me see

As below, so above and beyond, I imagine
Drawn beyond the lines of reason
Push the envelope,
watch it bend
```

We will create a piano score for Fibonacci numbers in this chapter. There is no unique way to do this. We will create both a PDF score our "composition" and a midi file, so that you can listen to the result. We will use LilyPond to create the score:

LILYPOND

What is LilyPond? On the websitelypond.org they write the following:

> LilyPond is a music engraving program, devoted to producing the highest-quality sheet music possible. It brings the aesthetics of traditionally engraved music to computer printouts. LilyPond is free software and part of the GNU Project.

You can learn more about Lilypond in the chapter python_scores.php of our Python tutorial. The following page give you also a great impression how LilyPond works: 'Compiling' Music

```
%%writefile simple_example.ly
\version "2.14.1"
\include "english.ly"

\score {
  \new Staff {
    \key d \major
    \numericTimeSignature
    \time 2/4
    <cs' d'' b''>16 <cs' d'' b''>8.
    %% Here: the tie on the D's looks funny
    %% Too tall? Left-hand endpoint is not aligned with the B ti
e?
    ~
    <cs' d'' b''>8 [ <b d'' a''> ]
  }
}
```

```
Overwriting simple_example.ly
```

```
!lilypond  simple_example.ly
```

```
GNU LilyPond 2.20.0
Processing `simple_example.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `/tmp/lilypond-R5QDyE'...
Converting to `simple_example.pdf'...
Deleting `/tmp/lilypond-R5QDyE'...
Success: compilation successfully completed
```

You can see the result by looking at the pdf file simple_example.pdf. The original file from which the PDF was created is simple_example.ly

FIBONACCI SCORE

THE FIBONACCI FUNCTION

To create our Fibonacci score we wil use the following Fibonacci function. You can find further explanation - most probably not necessary for the understanding of this chapter - concerning the Fibonacci function in our chapter Recursive Functions and additionally in our chapters Memoization with Decorators and python3_generators.php.

```
class FibonacciLike:

    def __init__(self, i1=0, i2=1):
        self.memo = {0:i1, 1:i2}

    def __call__(self, n):
        if n not in self.memo:
            self.memo[n] = self.__call__(n-1) + self.__cal
l__(n-2)
```

```
        return self.memo[n]

fib = FibonacciLike()
```

Furthmore, we will use the function gcd, which calculates the greatest common divisor of two positive numbers:

```
def gcd(a, b):
    """ returns the greatest common divisor of the
    numbers a and b """
    while b != 0:
        a, b = b, a%b
    return a
```

NUMBERS TO NOTES

We map the numbers 0 to 10 to the notes of the E major scale:

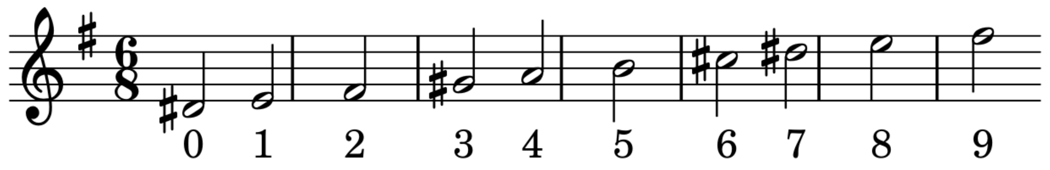
```
%%writefile digits_to_notes.ly
<<
  \relative c' {
    \key g \major
    \time 6/8
    dis2 e2 fis2 gis2 a2 b2 cis2 dis2 e2 fis2
  }
  \addlyrics {
    "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
  }
>>
```

```
Overwriting digits_to_notes.ly
```

```
!lilypond  digits_to_notes.ly
```

```
GNU LilyPond 2.20.0
Processing `digits_to_notes.ly'
Parsing...
digits_to_notes.ly:1: warning: no \version statement found, pleas
e add

\version "2.20.0"

for future compatibility
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `/tmp/lilypond-KrL4cS'...
Converting to `digits_to_notes.pdf'...
Deleting `/tmp/lilypond-KrL4cS'...
Success: compilation successfully completed
```

We need a mapping of digits to notes. We will use an e major scale and give each note the following numbers:



In Python we do this mapping by using a list and the indices correspond to the numbers abo

```
digits2notes = ["dis", "e", "fis",
                "gis", "a", "b",
                "cis'", "dis'", "e'",
                "fis'"]
#digits2notes = ["b", "c", "d", "e", "f", "g", "a'", "b'", "c'",
"d'"]
```

```
notes = str(fib(24)) + str(fib(12))
notes = str(fib(24))
notes
```

Output: `'46368'`

```
def notesgenerator(func, *numbers,k=1):
    """
    notesgenerator takes a function 'func', which will be applied
to
```

```
    the tuple of 'numbers'. The results are transformed into strin
gs
    and concatenated into a string 'notes'
    The length of the string 'notes' is the first value which wil
l be yielded
    by a generator object.
    After this it will cycle forever over the notes and will each
    time return the next k digits. The value of k can be changed b
e
    calling the iterator with send and a new k-value.


    Example:
    notesgenerator(fib, 1, 1, 2, 3, 5, 8, k=2)
    The results of applying fib to the numbers are
    1, 1, 1, 2, 5, 21
    The string concatenations gives us
    1112521
    When called the first time, it will yield 7, i.e. the length o
f
    the string notes.
    After this it will yield 11, 12, 52, 11, 11, 25, 21, 11, 12,
...
    """
    func_values_as_strings = [str(func(x)) for x in numbers]
    notes = "".join( func_values_as_strings)
    pos = 0
    n = len(notes)
    rval = yield n
    while True:
        new_pos = (pos + k) % n
        if pos + k < n:
            rval = yield notes[pos:new_pos]
        else:
            rval = yield notes[pos:] + notes[:new_pos]
        if rval:
            k = rval
        pos = new_pos
```

```
from itertools import cycle
note = notesgenerator(fib, 5, 8, 13, k=1)

print("length of the created 'notes' string", next(note))
for i in range(5):
    print(next(note), end=", ")
print("We set k to 2:")
print(note.send(2))
for i in range(5):
    print(next(note), end=", ")
```

```
length of the created 'notes' string 6
5, 2, 1, 2, 3, We set k to 2:
35
21, 23, 35, 21, 23,
```

```
def create_variable_score(notes_gen,
                          upper,
                          lower,
                          beat_numbers,
                          number_of_notes=80):
    """ create_variable_score creates the variable part of our sco
re,
    i.e. without the header. It consists of a melody with a chord
s accompaniment.
    """

    counter = 0
    old_res = ""
    for beat in cycle(beat_numbers):
        res = notes_gen.send(beat)
        #print(res)
        seq = ""
        if beat == 1:
            upper += digits2notes[int(res[0])] + "4 "
        elif beat == 2:
            for i in range(beat):
                upper += digits2notes[int(res[i])] + "8 "
        elif beat == 3:
            upper += " \\times 2/3 { "
            note = digits2notes[int(res[0])]
            upper += note + "8 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                upper += note + " "
            upper += "}"
        #elif beat == 4:
        #    for i in range(beat):
        #        upper += digits2notes[int(res[i])] + "16 "
        elif beat == 5:
            upper += " \\times 4/5 { "
            note = digits2notes[int(res[0])]
            upper += note + "16 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                upper += note + " "
            upper += "}"
        elif beat == 8:
            upper += " \\times 8/8 { "
            note = digits2notes[int(res[0])]
            upper += note + "32 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                upper += note + " "
            upper += "}"
        if counter > number_of_notes:
            break

        if old_res:
            accord = set()
            for i in range(len(old_res)):
                note = digits2notes[int(old_res[i])]
                accord.add(note)
            accord = " ".join(list(accord))
            lower += "<" +  accord + ">"

        else:
            lower += "r4 "

        counter += 1

        old_res = res

    return upper + "}", lower +  "}"
```

```
def create_score_header(key="e \major", time="5/4"):

    """ This creates the score header """

    upper = r"""
\version "2.20.0"
upper = \fixed c' {
  \clef treble
  \key """ + key  +       r"""
  \time """ + time + r"""
```

```
"""

    lower = r"""
lower = \fixed c {
  \clef bass
  \key """ + key  +       r"""
  \time """ + time + r"""

"""
    return upper, lower
```

```
def create_score(upper,
                 lower,
                 number_seq):
    """ create score combines everything to the final score """

    title = ",".join([str(x) for x in number_seq])
    score = upper + "\n" + lower + "\n" + r"""
    \header {
      title = " """

    score += title
    score += r""" Fibonacci"
      composer = "Bernd Klein"
    }

    \score {
      \new PianoStaff \with { instrumentName = "Piano" }
      <<
        \new Staff = "upper" \upper
        \new Staff = "lower" \lower
      >>
      \layout { }
      \midi {\tempo 4 = 45 }
    }
    """
    return score
```

```
from itertools import cycle
number_seq = (2021,)
x = notesgenerator(fib, *number_seq)

print(next(x))

upper, lower = create_score_header()
```

```
t = (1, 1, 2, 3, 5, 5, 3, 2, 1, 1, 2, 3, 5, 8, 8, 5, 3, 2, 1, 1)
# = (1, 1, 2, 3, 5, 8)
ng = notesgenerator(fib, *number_seq)
number_of_notes = next(ng)
print(f"n of notes: {number_of_notes}")
upper, lower = create_variable_score(ng,
                                     upper,
                                     lower,
                                     t,
                                    number_of_notes)

score = create_score(upper, lower, number_seq)

open("score.ly", "w").write(score)
```

```
423
n of notes: 423
```

Output:
```
13555
```

With the previous code we created the file score.ly which is the lilypond reüresentation of our score. To create a PDF version and a midi file, we have to call lilypond on a command line with

```
lilypond score.ly
```

The following call with `!lilypond score.ly` is the equivalent in a jupter notebook cell:

```
!lilypond  score.ly
```

```
GNU LilyPond 2.20.0
Processing `score.ly'
Parsing...
Interpreting music...[8][16][24][32][40][48][56][64][72][80]
Preprocessing graphical objects...
Interpreting music...
MIDI output to `score.midi'...
Finding the ideal number of pages...
Fitting music on 6 or 7 pages...
Drawing systems...
Layout output to `/tmp/lilypond-TcDnC5'...
Converting to `score.pdf'...
Deleting `/tmp/lilypond-TcDnC5'...
Success: compilation successfully completed
```

In case you cannot create create it yourself, you can downoad it here:

- score.pdf
- score.midi
- score.mp3

We created the mp3 file with the following Linux commands:

```
timidity score.midi -Ow -o - | ffmpeg -i - -acodec libmp3lame -ab 64k
score.mp3
```

The program `timidity` can also be used to listen to the midi file by calling it on the command line:

```
timidity  score.midi
```

POLYPHONIC SCORE FROM FIBONACCI NUMBERS

While the first score consists of a melody underlaid with chords, we now want to create a score that has a melody both in the left and right hand.

```python
def create_polyphone_score(notes_gen,
                           upper,
                           lower,
                           beat_numbers,
                           number_of_notes=80):
    """ This function is similar to create_variable score
    but it creates melodies in the left and right hand."""

    counter = 0
    old_res = ""
    group = ""
    print(beat_numbers)
    for beat in cycle(beat_numbers):
        res = notes_gen.send(beat)
        #print(beat, res)
        seq = ""
        old_group = group
        if beat == 1:
            group = digits2notes[int(res[0])] + "4 "
            upper += group
        elif beat == 2:
            group = ""
            for i in range(beat):
                group += digits2notes[int(res[i])] + "8 "
            upper += group
        elif beat == 3:
            group = ""
            group += "\\times 2/3 { "
            note = digits2notes[int(res[0])]
            group += note + "8 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                group += note + " "
            group += "}"
            upper += group
        #elif beat == 4:
        #    for i in range(beat):
        #        upper += digits2notes[int(res[i])] + "16 "
        elif beat == 5:
            group = ""
            group += "\\times 4/5 { "
            note = digits2notes[int(res[0])]
            group += note + "16 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                group += note + " "
            group += "}"
            upper += group
        elif beat == 8:
            group = ""
            group += "{ "
            note = digits2notes[int(res[0])]
            group += note + "32 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                group += note + " "
            group += "}"
            upper += group
        elif beat == 13:
            group = ""
            group += "\\times 4/13 { "
            note = digits2notes[int(res[0])]
            group += note + "16 "
            for i in range(1, beat):
                note = digits2notes[int(res[i])]
                group += note + " "
            group += "}"
            upper += group
        if old_group:
            lower += old_group
        else:
            lower += "r4 "

        counter += beat
        if counter > number_of_notes:
            break

        old_res = res
    return upper +  "}", lower +  "}"
```

```
from itertools import cycle

upper, lower = create_score_header()

t = (1, 1, 2, 3, 5, 5, 3, 2, 1, 1, 2, 3, 5, 8, 8, 5, 3, 2, 1, 1)
t = (1, 1, 2, 3, 5, 8, 13, 5, 1, 1)
ng = notesgenerator(fib, *number_seq)
number_of_notes = next(ng)
print(f"n of notes: {number_of_notes}")
upper, lower = create_polyphone_score(ng,
                                      upper,
                                      lower,
                                      t,
                                      number_of_notes*3)


score = create_score(upper, lower, number_seq)

open("score2.ly", "w").write(score)
```

```
n of notes: 423
(1, 1, 2, 3, 5, 8, 13, 5, 1, 1)
```

Output: `14497`

```
!lilypond  score2.ly
```

```
GNU LilyPond 2.20.0
Processing `score2.ly'
Parsing...
Interpreting music...[8][16][24][32][40][48][56][64][64]
Preprocessing graphical objects...
Interpreting music...
MIDI output to `score2.midi'...
Finding the ideal number of pages...
Fitting music on 4 or 5 pages...
Drawing systems...
Layout output to `/tmp/lilypond-jakQyo'...
Converting to `score2.pdf'...
Deleting `/tmp/lilypond-jakQyo'...
Success: compilation successfully completed
```

You can download the previously created files:

- score2.pdf
- score2.midi
- score2.mp3

PENTATONIC SCORE FROM FIBONACCI NUMBERS

In the following example, we will use a Pentatonic scale:

```
digits2notes = ["a,", "c", "d",
                "e", "g", "a",
                "c'", "d'", "e'",
                "g'"]
```

```
from itertools import cycle



upper, lower = create_score_header(key="a \minor", time="4/4")


t = (1, 1, 2, 3, 5, 5, 3, 2, 1, 1, 2, 3, 5, 8, 8, 5, 3, 2, 1, 1)
t = (1, 1, 2, 3, 5, 8, 13, 5, 1, 1)
ng = notesgenerator(fib, *number_seq)
number_of_notes = next(ng)

print(f"n of notes: {number_of_notes}")
upper, lower = create_polyphone_score(ng,
                                      upper,
                                                                            lower,
                                                                            t,
                                                                            number_of_notes*3)


score = create_score(upper, lower, number_seq)

open("score3.ly", "w").write(score)
```

```
n of notes: 423
(1, 1, 2, 3, 5, 8, 13, 5, 1, 1)
```

Output:
```
11591
```

```
!lilypond  score3.ly
```

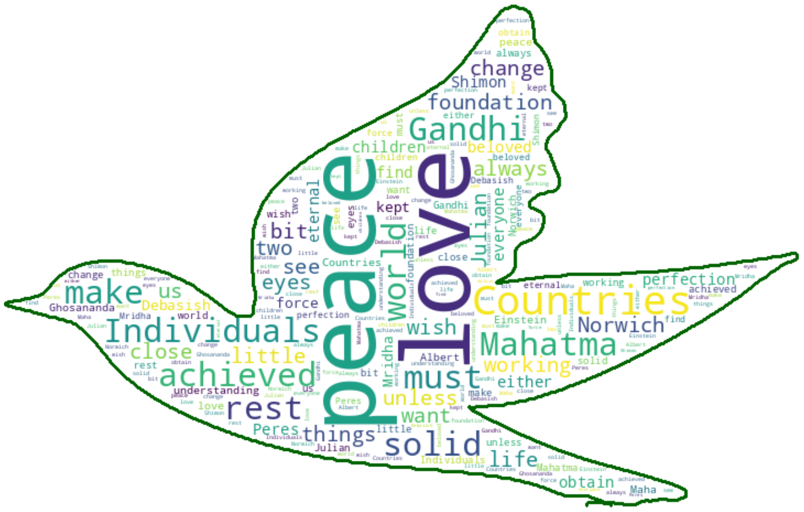
```
GNU LilyPond 2.20.0
Processing `score3.ly'
Parsing...
Interpreting music...[8][16][24][32][40][48][56][64][72][80][80]
Preprocessing graphical objects...
Interpreting music...
MIDI output to `score3.midi'...
Finding the ideal number of pages...
Fitting music on 7 or 8 pages...
Drawing systems...
Layout output to `/tmp/lilypond-Lj1dnx'...
Converting to `score3.pdf'...
Deleting `/tmp/lilypond-Lj1dnx'...
Success: compilation successfully completed
```

You can download the previously created files:

- score3.pdf
- score3.midi
- score3.mp3

WORD CLOUDS

INTRODUCTION



Word Clouds (WordClouds) are quite often called Tag clouds, but I prefer the term word cloud. It think this term is more general and easier to be understood by most people. The term tag is used for annotating texts and especially websites. This means finding out the most important words or terms characterizing or classifying a text. In the early days of web development people had to tag their websites so that search engines could easier classify them. Spemmer used this to manipulate the search engines by giving incorrect or even misleading tags so that their websites ranked higher. Google changed this by automatically finding out the importance of the text components. Google more or less disregarding the tags which the owners of the websites assigned to their pages. "Word clouds" as we use them also find out automatically what are the most important words. Of course, we do it naively by just counting the number of occurrances and using stop words. This is not the correct way to find out about the "real" importance of words, but leads to very interesting results, as we will see in the following.

We still haven't defined what a "word cloud" is. It is a visual representation of text data. Size and colors are used to show the relative importance of words or terms in a text. The bigger a term is the greater is its weight. So the size reflects the frequency of a words, which may correspond to its importance.

We will demonstrate in this tutorial how to create you own WordCloud with Python. We will use the Python modules Numpy, Matplotlib, Pillow, Pandas, and wordcloud in this tutorial.

The module wordcloud is not part of most of the Python distribution. If you use Anaconda, you can easily install it with the shell command

```
conda install -c conda-forge wordcloud
```

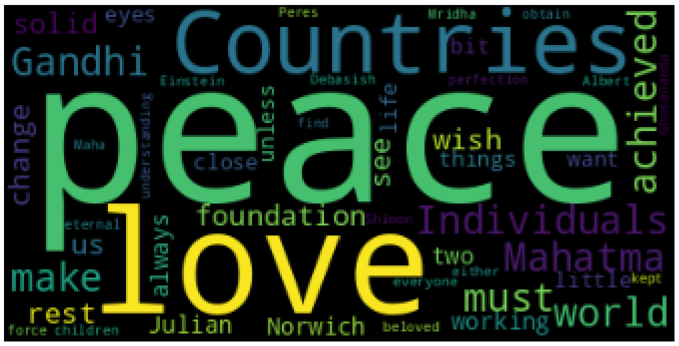
Unfortunately, this is not enough for all the things we are doing in this tutorial. So you will have to install the latest version from github:

```
    git clone https://github.com/amueller/word_cloud.git
    cd word_cloud
    pip install .
```

```
# importing the necessary modules:
```

```
import numpy as np
import pandas as pd
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator

import matplotlib.pyplot as plt
from PIL import Image
```

```
text = open("data/peace_and_love.txt").read()

wordcloud = WordCloud().generate(text)

# Display the generated image:
plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```



We will play around with the numerous parameters of WordCloud. We create a square picture with a transparant background. We also increase the likelihood of vertically oriented words by setting `prefer_horizontal` to 0.5 instead of 0.9 which is the default:

```
wordcloud = WordCloud(width=500,
                      height=500,
                      prefer_horizontal=0.5,
                      background_color="rgba(255, 255, 255, 0)",
                      mode="RGBA").generate(text)

plt.imshow(wordcloud)
plt.axis("off")
plt.show()
wordcloud.to_file("img_dir/peace_and_love.png")
```

Mahatma
love
achieved
must
world
solid
change
Gandhi
wish
peace
rest
Norwich
foundation
make
Julian working
Countries
Individuals

Output:
```
<wordcloud.wordcloud.WordCloud at 0x7f59ebbc0670>
```

```
wordcloud = WordCloud().generate(text)

# Display the generated image:
plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```

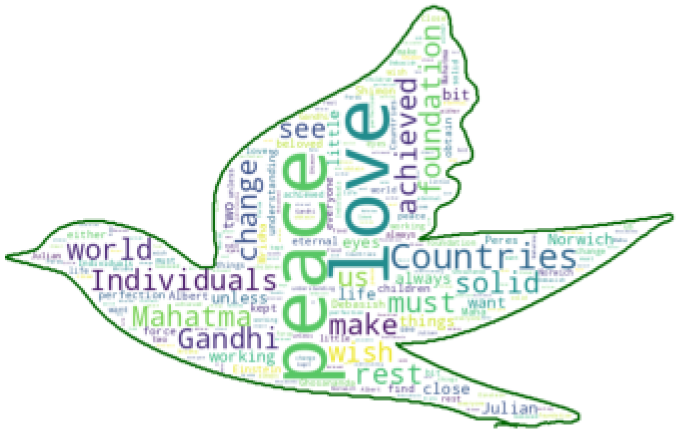
peace
world
Individuals
love
must
Gandhi
Countries
Mahatma
achieved
make
Peres
obtain
unless
change
eternal
perfection

We will show in the following how we can create word clouds with special shapes. We will use the shape of the dove from the following picture:

```
dove_mask = np.array(Image.open("img_dir/dove.png"))
#dove_mask[230:250, 240:250]
plt.imshow(dove_mask)
plt.axis("off")
plt.show()
```

We will create in the following example a wordclous in the shape of the previously loaded "peace dove". If the parameter `repeat` is set to `True` the words and phrases will be repeated until `max_words` (default 200) or `min_font_size` (default 4) is reached.

```
wordcloud = WordCloud(background_color="white",
                      mask=dove_mask,
                      contour_width=3,
                      repeat=True,
                      min_font_size=3,
                      contour_color='darkgreen')

# Generate a wordcloud
wordcloud.generate(text)

# store to file
wordcloud.to_file("img_dir/dove_wordcloud.png")

# show

plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```

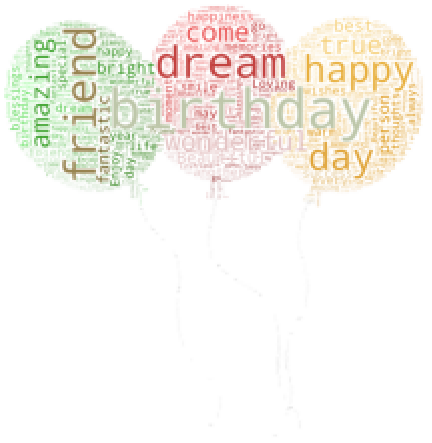


We will use now a colored mask with christmas bubles to create a word cloud with differenctly colored areas:

balloons.png

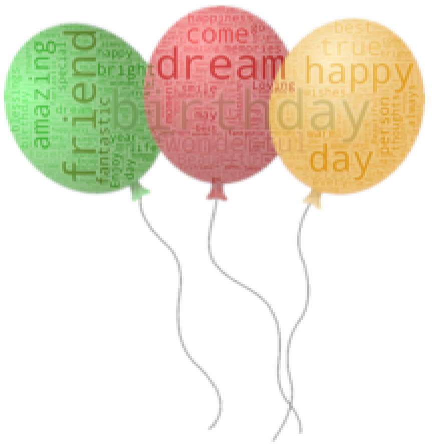
The following Python code can be used to create the colored wordcloud. We visualize the result with Matplotlib:

```
balloon_mask = np.array(Image.open("img_dir/balloons.png"))

image_colors = ImageColorGenerator(balloon_mask)

wc_balloons = WordCloud(stopwords=STOPWORDS,
                        background_color="white",
                        mode="RGBA",
                        max_words=1000,
                        #contour_width=3,
                        repeat=True,
                        mask=balloon_mask)

text = open("data/birthday_text.txt").read()
wc_balloons.generate(text)
wc_balloons.recolor(color_func = image_colors)

plt.imshow(wc_balloons)
plt.axis("off")
plt.show()
```



So that it looks better, we overlay this picture with the original picture of the balloons!

```
balloons_img = Image.fromarray(wc_balloons.to_array())
balloon_mask_img = Image.fromarray(balloon_mask)

new_img = Image.blend(balloons_img,
                      balloon_mask_img,
                      0.5)
new_img.save("balloons_with_text.png","PNG")
plt.imshow(new_img)
plt.axis("off")
plt.show()
```



EXERCISES

When I created the wordcloud tutorial it was the 23rd of December. This explains why the exercises are dealing with Christmas. Actually, I used the pictures as Christmas cards. So, you wil lbe able to create your customized Christmas and birthday card with Python!

Create a wordcloud in the shape of a christmas tree with Python. You can use the following black-and-white christmas tree for this purpose:

xmas_tree.png

We also provided a text filled with words related to Xmas:

Christmas Phrases

This exercise is Xmas related as well. This time, you may use the pictures

The first one can be used to create the wordcloud:

christmas_tree_bulbs.jpg

The second one can be overlayed with the wordcloud:

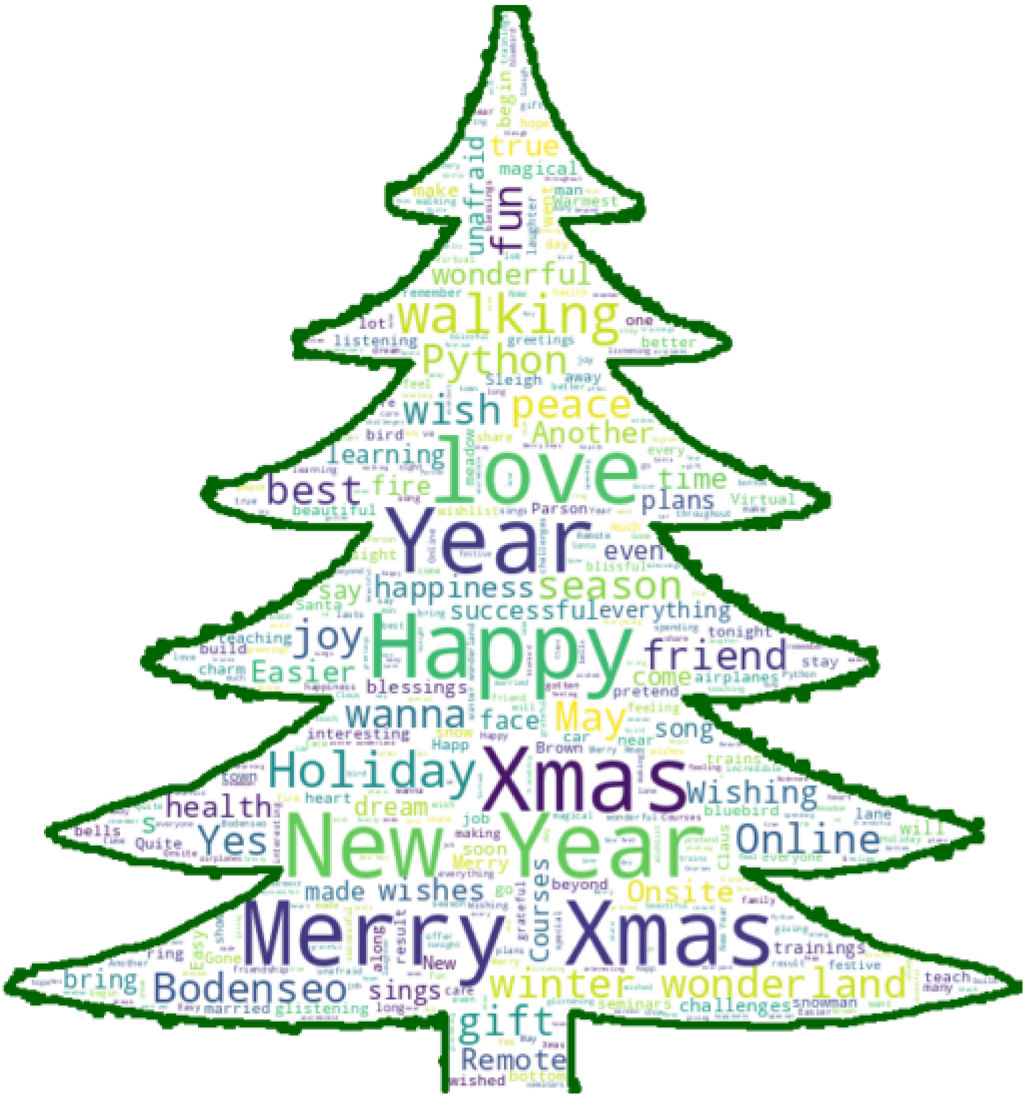
christmas_tree_bulbs.jpg

SOLUTIONS

SOLUTION TO EXERCISE 1

```
# load Chrismas tree mask
xmas_tree_mask = np.array(Image.open("img_dir/xmas_tree.png"))


text = open("data/xmas.txt").read()

wc = WordCloud(background_color="white",
               max_words=1000,
               mask=xmas_tree_mask,
               repeat=True,
               stopwords=STOPWORDS,
               contour_width=7,
               contour_color='darkgreen')

# Generate a wordcloud
```

```
wc.generate(text)

# store to file
wc.to_file("images/xmas_tree_wordcloud.png")

# show
plt.figure(figsize=[20,10])
plt.imshow(wc, interpolation='bilinear')
plt.axis("off")
plt.show()
```



```
# read text
```

```
text = open("data/xmas_jackie.txt").read()


tree_bulbs_img = np.array(Image.open("img_dir/christmas_tree_bulb
s.jpg"))
wordcloud_bulbs = WordCloud(background_color="white",
                            mode="RGB",
                            repeat=True,
                            max_words=1000,
                            mask=tree_bulbs_img).generate(text)

# create coloring from image
image_colors = ImageColorGenerator(tree_bulbs_img)
```

We will overlay the wordcloud image now with the picture including leaves:

```
tree_bulbs_img = Image.fromarray(wordcloud_bulbs.to_array())
tree_bulbs_leaves_img = np.array(Image.open("img_dir/christmas_tre
e_bulbs_leaves.jpg"))
tree_bulbs_leaves_img = Image.fromarray(tree_bulbs_leaves_img)


new_img = Image.blend(tree_bulbs_img,
                      tree_bulbs_leaves_img,
                      0.5)

# to save the newly created image uncomment the following line
new_img.save("images/christmas_tree_bulbs_wordcloud_jackie.png","P
NG")

plt.figure(figsize=[15, 16])
plt.imshow(new_img)

plt.axis("off")
plt.show()
```

Merry
Happy New
walking
Year
Wishing
friend
health
learning
love
winter wonderland
Jackie Trixie
Xmas
singing
wish
season
gift
New Year
Xmas
Holiday
Happy
peace
Another
wanna
best wishes
wonderful