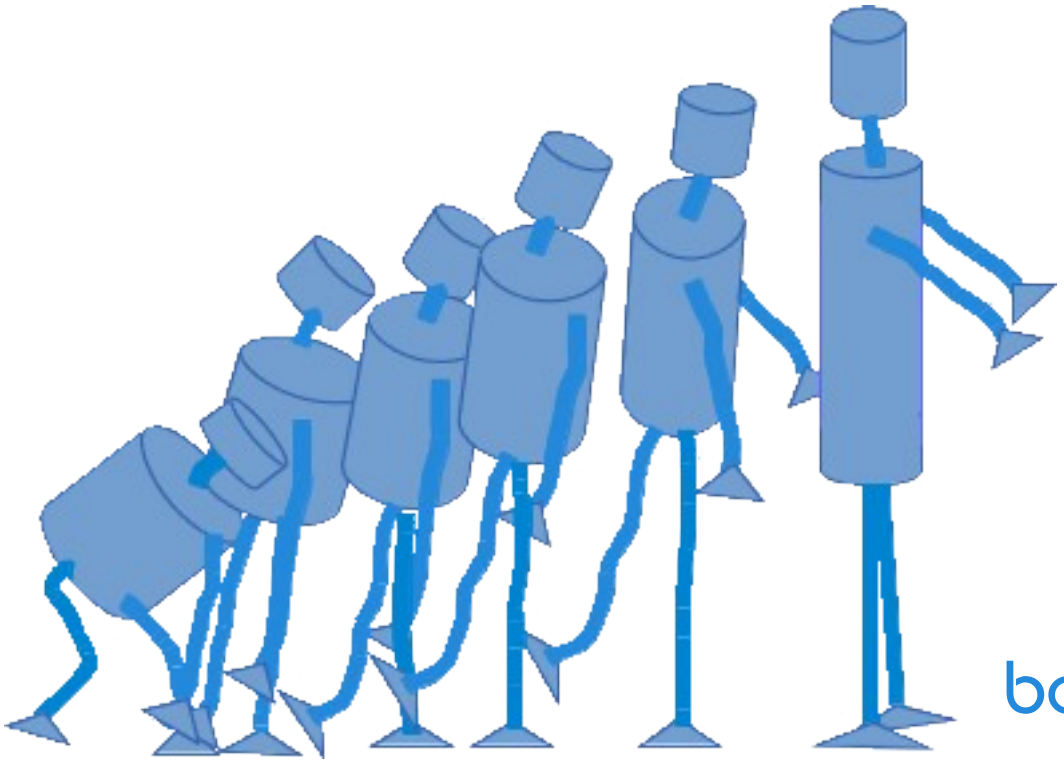
Applied Python

Bernd Klein



bodenseo

SYS-MODUL

INFORMATION ON THE PYTHON INTERPRETER

Like all the other modules, the sys module has to be imported with the import statement, i.e.

```
import sys
```

If there are questions about the import statement, we recommend the introductory chapter of our basic course concerning this topic Modular Programming and Modules

The sys module provides information about constants, functions and methods of the Python interpreter. dir(system) gives a summary of the available constants, functions and methods. Another possibility is the help() function. Using help(sys) provides valuable detail information.

The module sys informs e.g. about the maximal recursion depth (`sys.getrecursionlimit()`) and provides the possibility to change (sys.setrecursionlimit())
The current version number of Python can be accessed as well:

```
>>> import sys
>>> sys.version
'2.6.5 (r265:79063, Apr 16 2010, 13:57:41) \n[GCC 4.4.3]'
>>> sys.version_info
(2, 6, 5, 'final', 0)
>>>
```

COMMAND-LINE ARGUMENTS

Lots of scripts need access to the arguments passed to the script, when the script was started.
`argv` argv (or to be precise `sys.argv`) is a list, which contains the command-line arguments passed to the script. The first item of this list contains the name of the script itself. The arguments follow the script name.
The following script iterates over the `sys.argv` list :

```
#!/usr/bin/python

import sys

# it's easy to print this list of course:
print sys.argv

# or it can be iterated via a for loop:

for i in range(len(sys.argv)):
    if i == 0:
        print "Function name: %s" % sys.argv[0]
    else:
        print "%d. argument: %s" % (i,sys.argv[i])
```

We save this script as `arguments.py` . If we call it, we get the following output::

```
$ python arguments.py arg1 arg2
['arguments.py', 'arg1', 'arg2']
Function name: arguments.py
1. argument: arg1
2. argument: arg2
$
```

CHANGING THE OUTPUT BEHAVIOUR OF THE INTERACTIVE PYTHON SHELL

Python's interactive mode is one of the things which make Python special among other programming languages like Perl or Java. As we have seen in the chapter Interactive mode of our introductory tutorial, it's enough to write an expression on the command line and get back a meaningful output. However some users might prefer a different output behaviour. To change the way the interpreter prints interactively entered expressions, you will have to rebind `sys.displayhook` to a callable object.

We will demonstrate this in the following interactive example session:

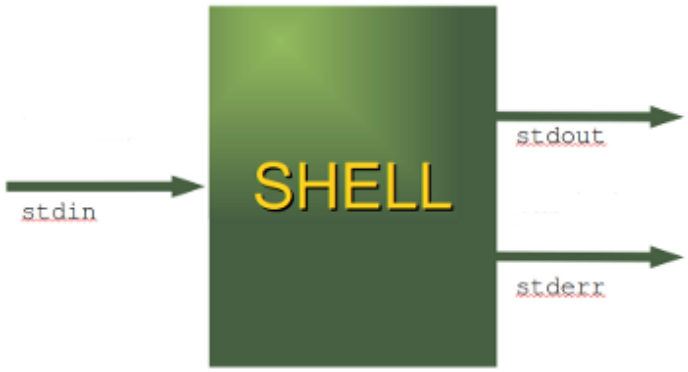
```
>>> import sys
>>> x = 42
>>> x
42
```

```
>>> import sys
>>> def my_display(x):
...     print "out: ",
...     print x
...
>>> sys.displayhook = my_display
>>> x
out:  42
>>>
>>> print x
42
```

We can see from this example that the standard behaviour of the print() function will not be changed.

STANDARD DATA STREAMS

Every serious user of a UNIX or Linux operating system knows standard streams, i.e. input, standard output and standard error. They are known as pipes. They are commonly abbreviated as stdin, stdout, stderr.
The standard input (stdin) is normally connected to the keyboard, while the standard error and standard output go to the terminal (or window) in which you are working.



These data streams can be accessed from Python via the objects of the sys module with the same names, i.e. sys.stdin, sys.stdout and sys.stderr.

```
>>> import sys
>>> for i in (sys.stdin, sys.stdout, sys.stderr):
...     print(i)
...

', mode 'w' at 0x7f3397a2c150>
```

```
', mode 'w' at 0x7f3397a2c1e0>
>>>
```

The following example illustrates the usage of the standard data streams:

```
>>> import sys
>>> print "Going via stdout"
Going via stdout
>>> sys.stdout.write("Another way to do it!\n")
Another way to do it!
>>> x = raw_input("read value via stdin: ")
read value via stdin: 42
>>> print x
42
>>> print "type in value: ", ; sys.stdin.readline()[:-1]
type in value: 42

'42'
>>>
```

The following example combines input and output:

```
import sys

while True:
  # output to stdout:
  print "Yet another iteration ..."
  try:
    # reading from sys.stdin (stop with Ctrl-D):
    number = raw_input("Enter a number: ")
  except EOFError:
    print "\nciao"
    break
  else:
    number = int(number)
    if number == 0:
      print >> sys.stderr, "0 has no inverse"
    else:
      print "inverse of %d is %f" % (number, 1.0/number)
```

If we save the previous example under "streams.py" and use a file called "number.txt" with numbers (one number per line) for the input, we can call the script in the following way from the

Bash shell:

```
$ python streams.py < numbers.txt
```

It's also possible to redirect the output into a file:

```
$ python streams.py < numbers.txt > output.txt
```

Now the only output left in the shell will be:

```
0 has no inverse
```

because this comes via the sterr stream.

REDIRECTIONS

There is hardly a user of a Linux or a Unix Shell, e.g. the Bourne or the Bash Shell, who hasn't used input or output redirections. It's not exaggerated to say that a useful work is not possible without redirections.

The standard output (stdout) can be redirected e.g. into a file, so that we can process this file later with another program. The same is possible with the standard error stream, we can redirect it into a file as well. We can redirect both stderr and stdout into the same file or into separate files.

The following script should be self-explanatory. But nevertheless here are some explanations: The first statement uses the regular standard output (stdout), i.e. the text "Coming through stdout" will be printed into the terminal from which the script has been called. In the next line we are storing the standard output channel in the variable save_stdout, so that we will be capable of restoring the original state at a later point in the script. After this we open a file "test.txt" for writing. After the statement `sys.stdout = fh` all print statements will directly print into this file. The original condition is restored with `sys.stdout = save_stdout` .

```
import sys

print("Coming through stdout")
```

```
# stdout is saved
save_stdout = sys.stdout

fh = open("test.txt","w")

sys.stdout = fh
print("This line goes to test.txt")

# return to normal:
sys.stdout = save_stdout

fh.close()
```

The following example shows how to redirect the standard error stream into a file:

```
import sys

save_stderr = sys.stderr
fh = open("errors.txt","w")
sys.stderr = fh

x = 10 / 0

# return to normal:
sys.stderr = save_stderr

fh.close()
```

It's possible to write into the error stream directly, i.e. without changing the general output behaviour. This can be achieved by appending `>> sys.stderr` to a print statement.

```
import sys

save_stderr = sys.stderr
fh = open("errors.txt","w")
sys.stderr = fh

print >> sys.stderr, "printing to error.txt"

# return to normal:
```

```
sys.stderr = save_stderr

fh.close()
```

OTHER INTERESTING VARIABLES AND CONSTANTS IN THE SYS MODULE

The largest positive integer supported by the platform's Py_ssize_t type, and thus the maximum size lists, strings, dicts, and many other containers can have.

Name	Description

byteorder	An indicator of the native byte order.
	The following output was created on a Linux machine and Python 2.6.5:
	`>>> sys.byteorder`
	`'little'`
	`>>>`
	The value will be 'big' on big-endian (most-significant byte first) platforms, and 'little' on little-endian (least-significant byte first) platforms.

executable	A string containing the name of the executable binary (path and executable file name) for the Python interpreter. E.g. "c:\\Python31\\python.exe on Windows 7 or "/usr/bin/python" on Linux.
	`>>> sys.executable`
	`'/usr/bin/python'`

maxint	This attribute contains the largest positive integer supported by Python's regular integer type. The minimum value for the maximal integer value is at least 2^{31-1} . The largest negative integer corresponds to the value of `- maxint - 1` . This asymmetry results from the use of 2's complement binary arithmetic. The value of maxint depends on the operating system.

	Remark concerning Python 3.X: The sys.maxint constant was removed with Python 3.0, since there is no longer a limit to the values of integers. However, sys.maxsize can be used as an integer larger than any practical list or string index. It conforms to the implementation's "natural" integer size and is typically the same as sys.maxint in previous releases on the same platform (assuming the same build options).
maxsize	
maxunicode	An integer giving the largest supported code point for a Unicode character. The value of this depends on the configuration option that specifies whether Unicode characters are stored as UCS-2 or UCS-4.
modules	The value of sys.modules is a dictionary mapping the names of modules to modules which have already been loaded. This can be manipulated e.g. to enforce the reloading of modules. Note that removing a module from this dictionary is not the same as calling reload() on the corresponding module object.

path	Contains the search pyth, where Python is looking for modules.
	`>>> sys.path`
	`['', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2', '/usr/lib/python2.6/lib-tk',`
	`'/usr/lib/python2.6/lib-old', '/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/dist-packages',`
	`'/usr/lib/python2.6/dist-packages/PIL',`
	`'/usr/lib/python2.6/dist-packages/gst-0.10',`
	`'/usr/lib/pymodules/python2.6', '/usr/lib/python2.6/dist-packages/gtk-2.0',`
	`'/usr/lib/pymodules/python2.6/gtk-2.0',`
	`'/usr/lib/python2.6/dist-packages/wx-2.8-gtk2-unicode',`
	`'/usr/local/lib/python2.6/dist-packages']`
	`>>>`

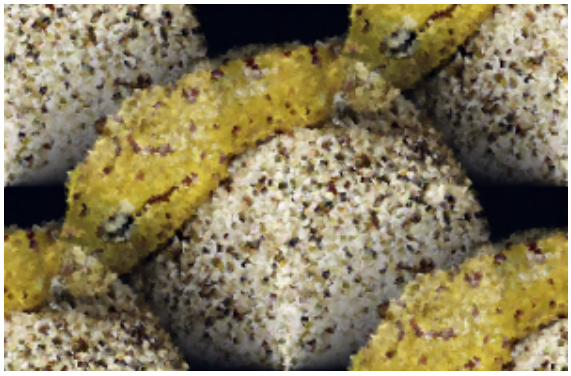
platform	Name of the platform on which Python is running, e.g. "linux2" for Linux and "win32" for Windows

	>>> sys.platform 'linux2' >>>
version	Version number of the Python interpreter >>> sys.version '2.6.5 (r265:79063, Apr 16 2010, 13:57:41) \n[GCC 4.4.3]' >>>
version_info	Similar information than sys.version, but the output is a tuple containing the five components of the version number: major, minor, micro, release-level, and serial. The values of this tuple are integers except the value for the release level, which is one of the following: 'alpha', 'beta', 'candidate', or 'final'. >>> sys.version_info (2, 6, 5, 'final', 0) >>>
__stdin__ __stdout__ __stderr__	These attributes contain the original values of stdin, stderr and stdout at the start of the program. They can be useful to print to the actual standard stream no matter if the sys.std* object has been redirected. They can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. But instead of using these values, the original stream should always be explicitly saved in a variable (as shown in our previous examples) before replacing it, and the original state should be restored by using the saved object.

getrecursionlimit()
setrecursionlimit(limit)
	getrecursionlimit() returns the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.
	>>> sys.getrecursionlimit()
	>>> 1000
	
	setrecursionlimit(limit) sets the maximum depth of the Python interpreter stack to the value of "limit".

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', 'Basel.png', 'Nürnberg.png', 'Erlangen.png', 'index.html', 'Ulm.png', 'Singen.png', 'Bremen.png', 'Kassel.png', 'Frankfurt.png', 'Saarbrücken.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', 'E
rlangen.png\n', 'Frankfurt.png\n', 'Freiburg.png
\n', 'Hamburg.png\n', 'Hannover.png\n', 'index.htm
l\n', 'Kassel.png\n', 'Konstanz.png\n', 'Nürnberg.
png\n', 'Saarbrücken.png\n', 'Singen.png\n', 'Stut
tgart.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.pn
g"])
```

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 J
an 23 08:07 Berlin.png\n', b'-rw-r--r-- 1 bernd be
rnd 0 Jan 23 08:07 Bremen.png\n', b'-rw-r--r-- 1 b
ernd bernd 0 Jan 23 08:07 Erlangen.png\n', b'-rw-r
--r-- 1 bernd bernd 0 Jan 23 08:07 Frankfurt.png
\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Fre
iburg.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23
08:07 Hamburg.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'-rw-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 bernd bernd 0 Jan 23 08:07 N\xc3\xb
crnberg.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 2
3 08:07 Saarbr\xc3\xbccken.png\n', b'-rw-r--r-- 1
bernd bernd 0 Jan 23 08:07 Singen.png\n', b'-rw-r-
-r-- 1 bernd bernd 0 Jan 23 08:07 Stuttgart.png
\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Ul
m.png\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:2
8 xyz\n', b'-rw-r--r-- 1 bernd bernd 0 Jan 23 08:2
9 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_conte
nt]
print(directory_content)
```

```
['total 0\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 0
8:32 abc\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 0
8:07 Basel.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan
23 08:07 Berlin.png\n', '-rw-r--r-- 1 bernd bernd
0 Jan 23 08:07 Bremen.png\n', '-rw-r--r-- 1 bernd
bernd 0 Jan 23 08:07 Erlangen.png\n', '-rw-r--r--
1 bernd bernd 0 Jan 23 08:07 Frankfurt.png\n', '-r
w-r--r-- 1 bernd bernd 0 Jan 23 08:07 Freiburg.png
\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Hamb
urg.png\n', '-rw-r--r-- 1 bernd bernd 0 Jan 23 08:
07 Hannover.png\n', '-rw-r--r-- 1 bernd bernd 0 Ja
n 23 08:05 index.html\n', '-rw-r--r-- 1 bernd bern
d 0 Jan 23 08:07 Kassel.png\n', '-rw-r--r-- 1 bern
d bernd 0 Jan 23 08:07 Konstanz.png\n', '-rw-r--r-
- 1 bernd bernd 0 Jan 23 08:07 Nürnberg.png\n', '-
rw-r--r-- 1 bernd bernd 0 Jan 23 08:07 Saarbrücke
n.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 Stuttgart.png\n', '-rw-r--r-- 1 bernd bernd
0 Jan 23 08:07 Ulm.png\n', '-rw-r--r-- 1 bernd ber
nd 0 Jan 23 08:28 xyz\n', '-rw-r--r-- 1 bernd bern
d 0 Jan 23 08:29 Zurich.png\n', '-rw-r--r-- 1 bern
d bernd 0 Jan 23 08:07 Zürich.png\n']
```

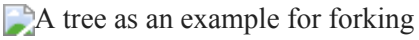
FUNCTIONS TO MANIPULATE PATHS, FILES AND DIRECTORIES

</table>
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).

In [ ]:

FORK UND PROZESSE

FORK


A tree as an example for forking

Long before biologists started their research of cloning, computer scientists had a successful history of cloning. They cloned processes, though they didn't call it cloning but forking.
Forking is one of the most important aspects of Unix and Linux. When a process forks, it creates a copy of itself. More generally, a fork in a multithreading environment means that a thread of execution is duplicated, creating a child thread from the parent thread. they are identical but can be told apart.

The fork operation creates a separate address space for the child. The child process has an exact copy of all the memory of the parent process. The execution of the parent and child process is independent of each other.

In computer science the term fork stands for at least two different aspects:

- The cloning of a process, as roughly described above.
- In software engineering, a project fork happens when developers take a legal copy of source code from one software package and start independent development on it. This way starting a distinct piece of software.

FORK IN PYTHON

The system function call fork() creates a copy of the process, which has called it. This copy runs as a child process of the calling process. The child process gets the data and the code of the parent process. The child process receives a process number (PID, Process IDentifier) of its own from the operating system. The child process runs as an independent instance, this means independent of a parent process. With the return value of fork() we can decide in which process we are: 0 means that we are in the child process while a positive return value means that we are in the parent process. A negative return value means that an error occurred while trying to fork.

To be able to fork processes we need to import the os module in Python.

The following Python3 example shows a parent process, which forks every time the user types in a "c", when prompted. Both the child process and the parent process continue after the "if

newpid == 0:" statement. The value of newpid is greater than 0 in the parent process and 0 in the child process. The exit statement `os.exit(0)` of the child function is necessary, because otherwise the child process would return into the parent process, i.e. to the input statement.

```
import os

def child():
   print('\nA new child ',  os.getpid())
   os._exit(0)

def parent():
   while True:
      newpid = os.fork()
      if newpid == 0:
         child()
      else:
         pids = (os.getpid(), newpid)
         print("parent: %d, child: %d\n" % pids)
      reply = input("q for quit / c for new fork")
      if reply == 'c':
          continue
      else:
          break

parent()
```

STARTING INDEPENDENT PROCESSES VIA FORK()

So far we have called functions in our examples which are defined in the same script file.

Forks are often used to start independent programs. To do this we need the exec*() functions.

They execute a new program by replacing the current process by this program. They do not return to the program which has called them. They even receive the same process ID as the calling program.

THE EXEC*()-FUNCTIONS

The exec*()-Funktionen are available in various formats:

- os.execl(path, arg0, arg1, ...)

- os.execle(path, arg0, arg1, ..., env)
- os.execlp(file, arg0, arg1, ...)
- os.execlpe(file, arg0, arg1, ..., env)
- os.execv(path, args)
- os.execve(path, args, env)
- os.execvp(file, args)
- os.execvpe(file, args, env)

We will explain these functions with examples, because they are hardly explained in literature.

We will use a bash shell script, which we save under test.sh in the directory /home/monty/bin2 To understand the following examples it's only necessary that the script test.sh is not included in a directory which is included in the PATH (the environment variable $PATH of bash). test.sh has to be executable:

```
chmod 755 test.sh
```

```
#!/bin/bash

script_name=$0
arg1=$1
current=`pwd`
echo $script_name, $arg1
echo "XYZ: "$XYZ
echo "PATH: "$PATH
echo "current directory: $current"
```

The Python script execvp.py, which calls our script test.sh, is saved in a different directory, e.g. /home/monty/python:

```
#!/usr/bin/python
import os
args = ("test","abc")
os.execvp("test.sh", args)
```

As test.sh can't be found in any of the $PATH locations, we get an error message, if we call execvp in a command line:

```
$ ./execvp.py
Traceback (most recent call last):
  File "./execvp.py", line 6, in <module>
    os.execvp("test.sh", args)
  File "/usr/lib/python2.6/os.py", line 344, in execvp
    _execvpe(file, args)
  File "/usr/lib/python2.6/os.py", line 380, in _execvpe
    func(fullname, *argrest)
OSError: [Errno 2] No such file or directory
```

To prevent this error message, and to achieve that our shell script is called, we extend the PATH environment variable with the directory which contains test.sh, in our case /home/monty/bin2:

```
$ PATH=$PATH:/home/monty/bin2
$ ./execvp.py
/home/monty/bin2/test.sh, abc
XYZ:
PATH: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/bin:/home/monty/bin:/home/monty/bin
2
current directory: /home/monty/python
```

Another elegant possibility to extend the search path PATH offers the execvpe()-function. This function has a third parameter which is a dictionary with environment variables. If environment variables already exist, they will be replaced by the corresponding values of this dictionary. If an environment variable doesn't exist, e.g. "XYZ" in the next example, it will be created.

```
import os
env =  {"PATH":"/home/monty/bin2", "XYZ":"BlaBla"}
args = ("test","abc")
os.execvpe("test.sh", args, env)
```

If we save this script under the file name execvpe.py, we receive the following output, if we call it:

```
$ ./execvpe.py
/home/monty/bin2/test.sh, abc
XYZ: BlaBla
PATH: /home/monty/bin2/
current directory: /home/monty/python/
$
```

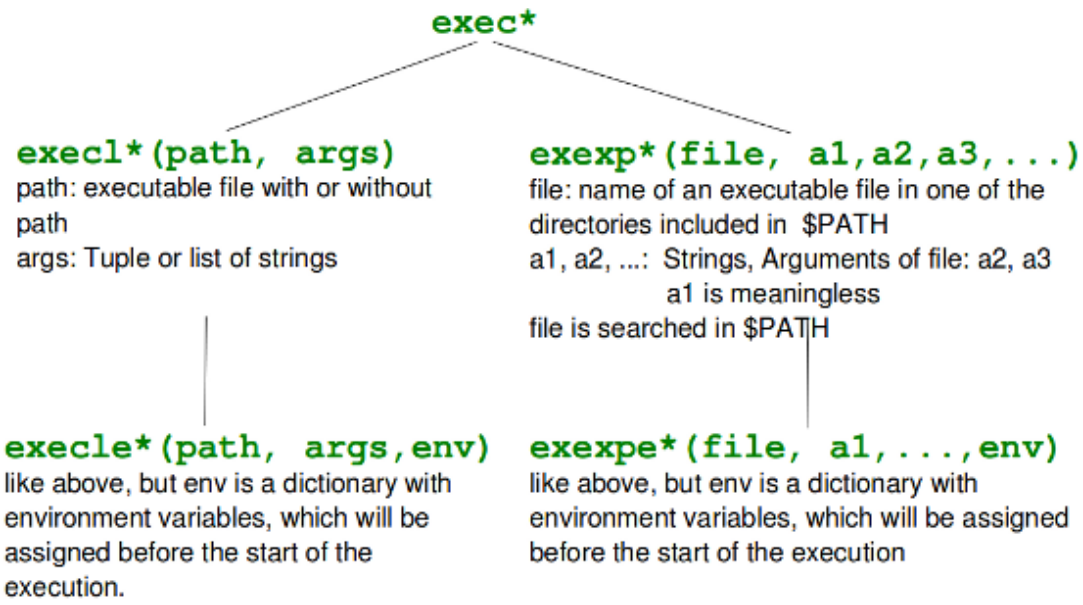
The value of the shell environment variable $PATH is replaced by the new value of our dictionary. If you want to avoid this, i.e. if you want to new directory to be appended to the existing path "PATH", you have to change the code like this:

```
import os

path = os.environ["PATH"] + ":/home/monty/bin2/"
env =  {"PATH":path, "XYZ":"BlaBla"}
os.execlpe("test.sh", "test","abc", env)
```

It's possible to use execlpe() instead of execvpe(), but the code of our Python script has to be changed in the following way:

```
import os
env =  {"PATH":"/home/monty/bin2/", "XYZ":"BlaBla"}
os.execlpe("test.sh", "test","abc", env)
```

OVERVIEW OF THE EXEC FUNCTIONS



THREADS IN PYTHON

DEFINITION OF A THREAD

A Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system. Threads are normally created by a fork of a computer script or program in two or more parallel (which is implemented on a single processor by multitasking) tasks. Threads are usually contained in processes. More than one thread can exist within the same process. These threads share the memory and the state of the process. In other words: They share the code or instructions and the values of its variables.

There are two different kind of threads:

- Kernel threads
- User-space Threads or user threads

Kernel Threads are part of the operating system, while User-space threads are not implemented in the kernel.

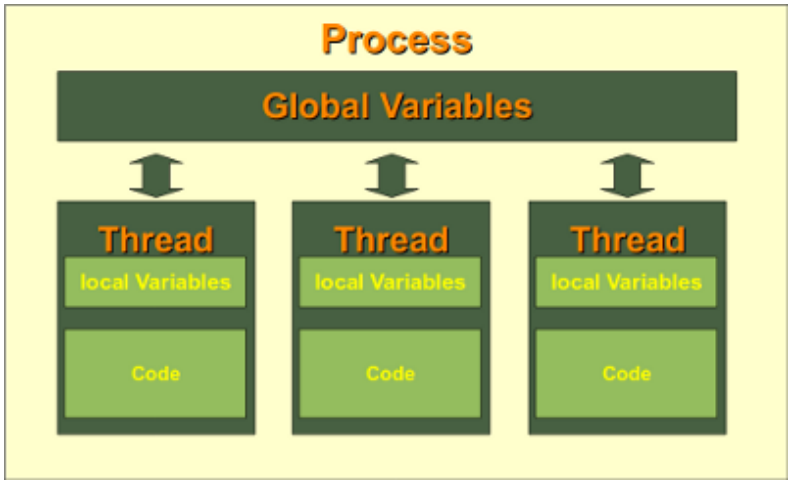
In a certain way, user-space threads can be seen as an extension of the function concept of a programming language. So a thread user-space thread is similar to a function or procedure call. But there are differences to regular functions, especially the return behaviour.

Every process has at least one thread, i.e. the process itself. A process can start multiple threads. The operating system executes these threads like parallel "processes". On a single processor machine, this parallelism is achieved by thread scheduling or timeslicing.

Advantages of Threading:

- Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be executed truly concurrent.
- A program can remain responsive to input. This is true both on single and on multiple CPU

- Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.



The handling of threads is simpler than the handling of processes for an operating system. That's why they are sometimes called light-weight process (LWP)

THREADS IN PYTHON

There are two modules which support the usage of threads in Python:

- thread
  and
- threading

Please note: The thread module has been considered as "deprecated" for quite a long time. Users have been encouraged to use the threading module instead. So,in Python 3 the module "thread" is not available anymore. But that's not really true: It has been renamed to "_thread" for backwards incompatibilities in Python3.

The module "thread" treats a thread as a function, while the module "threading" is implemented in an object oriented way, i.e. every thread corresponds to an object.

THE THREAD MODULE

It's possible to execute functions in a separate thread with the module Thread. To do this, we can use the function thread.start_new_thread:

```
thread.start_new_thread(function, args[, kwargs])
```

This method starts a new thread and return its identifier. The thread executes the function

"function" (function is a reference to a function) with the argument list args (which must be a list or a tuple). The optional kwargs argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

Example for a Thread in Python:

```
from thread import start_new_thread

def heron(a):
    """Calculates the square root of a"""
    eps = 0.0000001
    old = 1
    new = 1
    while True:
        old,new = new, (new + a/new) / 2.0
        print old, new
        if abs(new - old) < eps:
            break
    return new

start_new_thread(heron,(99,))
start_new_thread(heron,(999,))
start_new_thread(heron,(1733,))

c = raw_input("Type something to quit.")
```

The raw_input() in the previous example is necessary, because otherwise all the threads would be exited, if the main program finishes. raw_input() waits until something has been typed in.

We expand the previous example with counters for the threads.

```
from thread import start_new_thread

num_threads = 0
def heron(a):
    global num_threads
    num_threads += 1

    # code has been left out, see above
    num_threads -= 1
    return new

start_new_thread(heron,(99,))
start_new_thread(heron,(999,))
start_new_thread(heron,(1733,))
```

```
start_new_thread(heron,(17334,))

while num_threads > 0:
    pass
```

The script above doesn't work the way we might expect it to work. What is wrong?
The problem is that the final while loop will be reached even before one of the threads could have incremented the counter num_threads.

But there is another serious problem:
The problem arises by the assignments to num_thread
`num_threads += 1`
and
`num_threads -= 1`
These assignment statements are not atomic. Such an assignment consists of three actions:

- Reading the value of num_thread
- A new int instance will be incremented or decremented by 1
- the new value has to be assigned to num_threads

Errors like this happen in the case of increment assignments:
The first thread reads the variable num_threads, which still has the value 0. After having read this value, the thread is put to sleep by the operating system. Now it is the second thread's turn: It also reads the value of the variable num_threads, which is still 0, because the first thread has been put to sleep too early, i.e. before it had been able to increment its value by 1. Now the second thread is put to sleep. Now it is the third thread's turn, which again reads a 0, but the counter should have been 2 by now. Each of these threads assigns now the value 1 to the counter. Similiar problems occur with the decrement operation.

SOLUTION

Problems of this kind can be solved by defining critical sections with lock objects. These sections will be treated atomically, i.e. during the execution of such a section a thread will not be interrupted or put to sleep.
The methode thread.allocate_lock is used to create a new lock object:

```
lock_object = thread.allocate_lock()
```

The beginning of a critical section is tagged with `lock_object.acquire()` and the end with `lock_object.release()` .
The solution with locks looks like this:

```
from thread import start_new_thread, allocate_lock
num_threads = 0
thread_started = False
lock = allocate_lock()
def heron(a):
    global num_threads, thread_started
    lock.acquire()
    num_threads += 1
    thread_started = True
    lock.release()

    ...

    lock.acquire()
    num_threads -= 1
    lock.release()
    return new

start_new_thread(heron,(99,))
start_new_thread(heron,(999,))
start_new_thread(heron,(1733,))

while not thread_started:
    pass
while num_threads > 0:
    pass
```

THREADING MODULE

We want to introduce the threading module with an example. The Thread of the example doesn't do a lot, essentially it just sleeps for 5 seconds and then prints out a message:

```
import time
from threading import Thread

def sleeper(i):
    print "thread %d sleeps for 5 seconds" % i
    time.sleep(5)
    print "thread %d woke up" % i

for i in range(10):
    t = Thread(target=sleeper, args=(i,))
    t.start()
```

Method of operation of the threading.Thread class: The class threading.Thread has a method start(), which can start a Thread. It triggers off the method run(), which has to be overloaded.

The join() method makes sure that the main program waits until all threads have terminated.

The previous script returns the following output:

```
thread 0 sleeps for 5 seconds
thread 1 sleeps for 5 seconds
thread 2 sleeps for 5 seconds
thread 3 sleeps for 5 seconds
thread 4 sleeps for 5 seconds
thread 5 sleeps for 5 seconds
thread 6 sleeps for 5 seconds
thread 7 sleeps for 5 seconds
thread 8 sleeps for 5 seconds
thread 9 sleeps for 5 seconds
thread 1 woke up
thread 0 woke up
thread 3 woke up
thread 2 woke up
thread 5 woke up
thread 9 woke up
thread 8 woke up
thread 7 woke up
thread 6 woke up
thread 4 woke up
```

The next example shows a thread, which determines, if a number is prime or not. The Thread is defined with the threading module:

```
import threading

class PrimeNumber(threading.Thread):
  def __init__(self, number):
    threading.Thread.__init__(self)
    self.Number = number

  def run(self):
    counter = 2
    while counter*counter < self.Number:
      if self.Number % counter == 0:
        print "%d is no prime number, because %d = %d * %
d" % ( self.Number, self.Number, counter, self.Number / c
ounter)
                return
            counter += 1
        print "%d is a prime number" % self.Number
threads = []
while True:
```

```
    input = long(raw_input("number: "))
    if input < 1:
        break

    thread = PrimeNumber(input)
    threads += [thread]
    thread.start()

for x in threads:
    x.join()
```

With locks it should look like this:

```
class PrimeNumber(threading.Thread):
    prime_numbers = {}
    lock = threading.Lock()

    def __init__(self, number):
        threading.Thread.__init__(self)
        self.Number = number
        PrimeNumber.lock.acquire()
        PrimeNumber.prime_numbers[number] = "None"
        PrimeNumber.lock.release()

    def run(self):
        counter = 2
        res = True
        while counter*counter < self.Number and res:
            if self.Number % counter == 0:
               res = False
            counter += 1
        PrimeNumber.lock.acquire()
        PrimeNumber.prime_numbers[self.Number] = res
        PrimeNumber.lock.release()
threads = []
while True:
    input = long(raw_input("number: "))
    if input < 1:
        break

    thread = PrimeNumber(input)
    threads += [thread]
    thread.start()

for x in threads:
    x.join()
```

PINGING WITH THREADS

The previous examples of this chapter are of purely didactical interest, and have no practical applicability. The following example shows an interesting application, which can be easily used. If you want to determine in a local network which addresses are active or which computers are active, this script can be used. But you have to be careful with the range, because it can jam the network, if too many pings are started at once. Manually we would do the following for a network 192.168.178.x: We would ping the addresses 192.168.178.0, 192.168.178.1, 192.168.178.3 until 192.168.178.255 in turn. Every time we would have to wait a few seconds for the return values. This can be programmed in Python with a for loop over the address range of the IP addresses and a os.popen("ping -q -c2 "+ip,"r").

A solution without threads is highly inefficient, because the script will have to wait for every ping.

Solution with threads:

```
import os, re

received_packages = re.compile(r"(\d) received")
status = ("no response","alive but losses","alive")

for suffix in range(20,30):
   ip = "192.168.178."+str(suffix)
   ping_out = os.popen("ping -q -c2 "+ip,"r")
   print "... pinging ",ip
   while True:
      line = ping_out.readline()
      if not line: break
      n_received = received_packages.findall(line)
      if n_received:
         print ip + ": " + status[int(n_received[0])]
```

To understand this script, we have to look at the results of a ping on a shell command line:

```
$ ping -q -c2 192.168.178.26
PING 192.168.178.26 (192.168.178.26) 56(84) bytes of dat
a.

--- 192.168.178.26 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 9
99ms
rtt min/avg/max/mdev = 0.022/0.032/0.042/0.010 ms
```

If a ping doesn't lead to success, we get the following output:

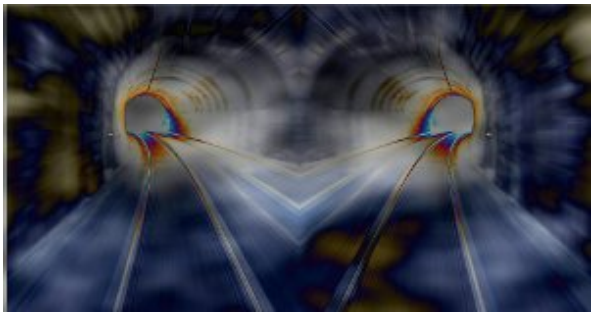
```
$ ping -q -c2 192.168.178.23
PING 192.168.178.23 (192.168.178.23) 56(84) bytes of dat
a.

--- 192.168.178.23 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet
loss, time 1006ms
```

This is the fast solution with threads:

```
import os, re, threading

class ip_check(threading.Thread):
   def __init__ (self,ip):
      threading.Thread.__init__(self)
      self.ip = ip
      self.__successful_pings = -1
   def run(self):
      ping_out = os.popen("ping -q -c2 "+self.ip,"r")
      while True:
        line = ping_out.readline()
        if not line: break
        n_received = re.findall(received_packages,line)
        if n_received:
           self.__successful_pings = int(n_received[0])
   def status(self):
      if self.__successful_pings == 0:
         return "no response"
      elif self.__successful_pings == 1:
         return "alive, but 50 % package loss"
      elif self.__successful_pings == 2:
         return "alive"
      else:
         return "shouldn't occur"
received_packages = re.compile(r"(\d) received")

check_results = []
for suffix in range(20,70):
   ip = "192.168.178."+str(suffix)
   current = ip_check(ip)
   check_results.append(current)
   current.start()

for el in check_results:
   el.join()
   print "Status from ", el.ip,"is",el.status()
```

PIPES IN PYTHON

PIPE



Unix or Linux without pipes is unthinkable, or at least, pipelines are a very important part of Unix and Linux applications. Small elements are put together by using pipes. Processes are chained together by their standard streams, i.e. the output of one process is used as the input of another process. To chain processes like this, so-called anonomymous pipes are used.
The concept of pipes and pipelines was introduced by Douglas McIlroy, one of the authors of the early command shells, after he noticed that much of the time they were processing the output of one program as the input to another. Ken Thompson added the concept of pipes to the UNIX operating system in 1973. Pipelines have later been ported to other operating systems like DOS, OS/2 and Microsoft Windows as well.

Generally there are two kinds of pipes:

- anonymous pipes
  and
- named pipes

Anonymous pipes exist solely within processes and are usually used in combination with forks.

BEER PIPE IN PYTHON

"99 Bottles of Beer" is a traditional song in the United States and Canada. The song is derived from the English "Ten Green Bottles". The song consists of 100 verses, which are very similar. Just the number of bottles varies. Only one, i.e. the hundredth verse is slightly different. This song is often sung on long trips, because it is easy to memorize, especially when drunken, and it can take a long time to sing.

Here are the lyrics of this song:

Ninety-nine bottles of beer on the wall, Ninety-nine bottles of beer. Take one down, pass it around, Ninety-eight bottles of beer on the wall.

The next verse is the same starting with 98 bottles of beer. So the general rule is, each verse one bottle less, until there in none left. The song normally ends here. But we want to implement the Aleph-Null (i.e. the infinite) version of this song with an additional verse:

No more bottles of beer on the wall, no more bottles of beer. Go to the store and buy some more, Ninety-nine bottles of beer on the wall.

This song has been implemented in all conceivable computer languages like "Whitespace" or "Brainfuck". You find the collection at http://99-bottles-of-beer.net
We program the Aleph-Null variant of the song with a fork and a pipe:

```
import os

def child(pipeout):
  bottles = 99
  while True:
    bob = "bottles of beer"
    otw = "on the wall"
    take1 = "Take one down and pass it around"
    store = "Go to the store and buy some more"

    if bottles > 0:
      values =  (bottles, bob, otw, bottles, bob, take1,
bottles - 1,bob,otw)
      verse = "%2d %s %s,\n%2d %s.\n%s,\n%2d %s %s." % va
lues
      os.write(pipeout, verse)
      bottles -= 1
    else:
      bottles = 99
      values =  (bob, otw, bob, store, bottles, bob,otw)
      verse = "No more %s %s,\nno more %s.\n%s,\n%2d %s %
s." % values
      os.write(pipeout, verse)

def parent():
    pipein, pipeout = os.pipe()
```

```
    if os.fork() == 0:
        child(pipeout)
    else:
        counter = 1
        while True:
            if counter % 100:
                verse = os.read(pipein, 117)
            else:
                verse = os.read(pipein, 128)
            print 'verse %d\n%s\n' % (counter, verse)
            counter += 1

parent()
```

The problem in the code above is that we or better the parent process have to know exactly how many bytes the child will send each time. For the first 99 verses it will be 117 Bytes (`verse = os.read(pipein, 117)`) and for the Aleph-Null verse it will be 128 bytes (`verse = os.read(pipein, 128)`

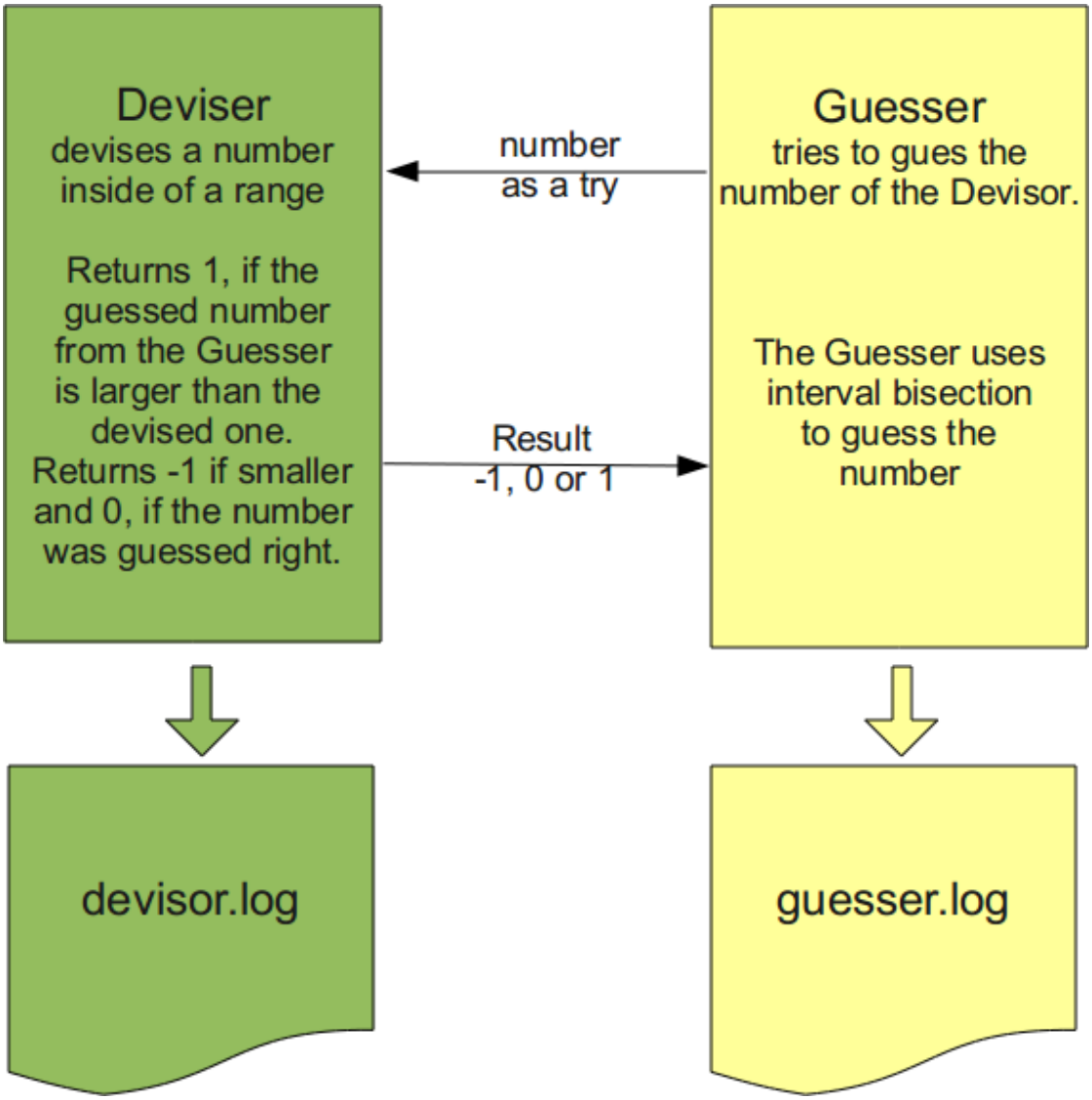
We fixed this in the following implementation, in which we read complete lines:

```
import os

def child(pipeout):
  bottles = 99
  while True:
    bob = "bottles of beer"
    otw = "on the wall"
    take1 = "Take one down and pass it around"
    store = "Go to the store and buy some more"

    if bottles > 0:
      values =  (bottles, bob, otw, bottles, bob, take1,
bottles - 1,bob,otw)
      verse = "%2d %s %s,\n%2d %s.\n%s,\n%2d %s %s.\n" %
values
      os.write(pipeout, verse)
      bottles -= 1
    else:
      bottles = 99
      values =  (bob, otw, bob, store, bottles, bob,otw)
      verse = "No more %s %s,\nno more %s.\n%s,\n%2d %s %
s.\n" % values
      os.write(pipeout, verse)
def parent():
    pipein, pipeout = os.pipe()
    if os.fork() == 0:
```

```
            os.close(pipein)
            child(pipeout)
        else:
            os.close(pipeout)
            counter = 1
            pipein = os.fdopen(pipein)
            while True:
                print 'verse %d' % (counter)
                for i in range(4):
                    verse = pipein.readline()[:-1]
                    print '%s' % (verse)
                counter += 1
                print

parent()
```

BIDIRECTIONAL PIPES

Now we come to something completely non-alcoholic. It's a simple guessing game, which small children often play. We want to implement this game with bidirectional Pipes. There is an explanation of this game in our tutorial in the chapter about loops. The following diagram explains both the rules of the game and the way we implemented it:



The deviser, the one who devises the number, has to imagine a number between a range of 1 to n. The Guesser inputs his guess. The deviser informs the player, if this number is larger, smaller or equal to the secret number, i.e. the number which the deviser has randomly created. Both the deviser and the guesser write their results into log files, i.e. deviser.log and guesser.log respectively.

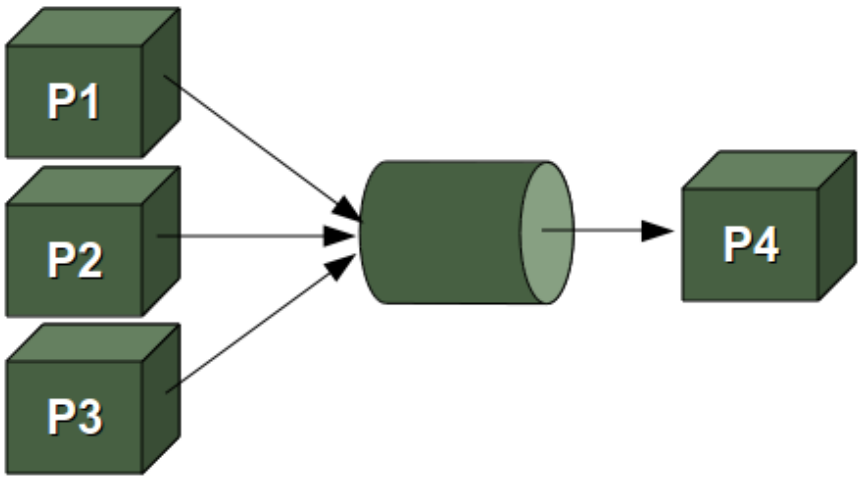
This is the complete implementation:

```
import os, sys, random


def deviser(max):
    fh = open("deviser.log","w")
    to_be_guessed = int(max * random.random()) + 1

    guess = 0
    while guess != to_be_guessed:
        guess = int(raw_input())
        fh.write(str(guess) + " ")
        if guess > 0:
            if guess > to_be_guessed:
                print 1
            elif guess < to_be_guessed:
                print -1
            else:
                print 0
            sys.stdout.flush()
        else:
            break
    fh.close()

def guesser(max):
    fh = open("guesser.log","w")
    bottom = 0
    top = max
    fuzzy = 10
    res = 1
    while res != 0:
        guess = (bottom + top) / 2
        print guess
        sys.stdout.flush()
        fh.write(str(guess) + " ")
        res = int(raw_input())
        if res == -1: # number is higher
            bottom = guess
        elif res == 1:
            top = guess
        elif res == 0:
            message = "Wanted number is %d" % guess
            fh.write(message)
        else: # this case shouldn't occur
            print "input not correct"
            fh.write("Something's wrong")
```

```
n = 100
stdin  = sys.stdin.fileno() # usually 0
stdout = sys.stdout.fileno() # usually 1

parentStdin, childStdout  = os.pipe()
childStdin,  parentStdout = os.pipe()
pid = os.fork()
if pid:
    # parent process
    os.close(childStdout)
    os.close(childStdin)
    os.dup2(parentStdin,  stdin)
    os.dup2(parentStdout, stdout)
    deviser(n)
else:
    # child process
    os.close(parentStdin)
    os.close(parentStdout)
    os.dup2(childStdin,  stdin)
    os.dup2(childStdout, stdout)
    guesser(n)
```

NAMED PIPES, FIFOS

Under Unix as well as under Linux it's possible to create Pipes, which are implemented as files.



These Pipes are called "named pipes" or sometimes Fifos (First In First Out).

A process reads from and writes to such a pipe as if it were a regular file. Sometimes more than one process write to such a pipe but only one process reads from it.

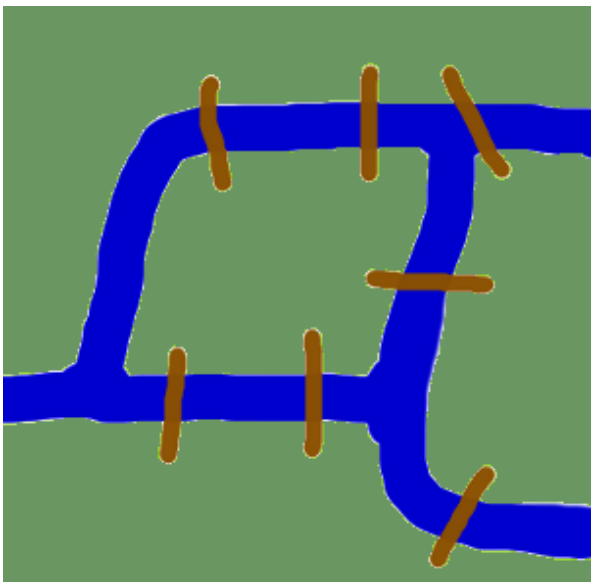
The following example illustrates the case, in which one process (child process) writes to the

pipe and another process (the parent process) reads from this pipe.

```
import os, time, sys
pipe_name = 'pipe_test'

def child( ):
    pipeout = os.open(pipe_name, os.O_WRONLY)
    counter = 0
    while True:
        time.sleep(1)
        os.write(pipeout, 'Number %03d\n' % counter)
        counter = (counter+1) % 5

def parent( ):
    pipein = open(pipe_name, 'r')
    while True:
        line = pipein.readline()[:-1]
        print 'Parent %d got "%s" at %s' % (os.getpid(),
line, time.time( ))

if not os.path.exists(pipe_name):
    os.mkfifo(pipe_name)
pid = os.fork()
if pid != 0:
    parent()
else:
    child()
```

GRAPHS IN PYTHON

ORIGINS OF GRAPH THEORY

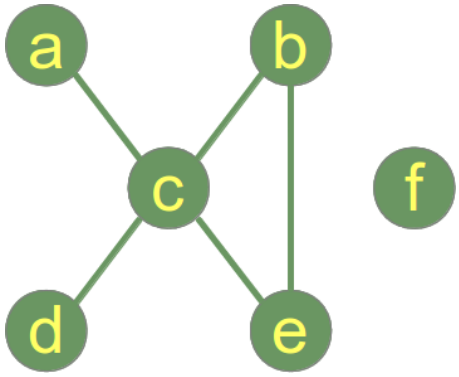


Before we start with the actual implementations of graphs in Python and before we start with the introduction of Python modules dealing with graphs, we want to devote ourselves to the origins of graph theory.
The origins take us back in time to the Künigsberg of the 18th century. Königsberg was a city 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 lists with the nodes, which are connecting by an edge. There is no simpler and more elegant way to represent a graph.

An edge can be seen as a 2-tuple with nodes as elements, i.e. ("a","b")

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))
```

This code generates the following output, if combined with the previously defined graph dictionary:

```
$ python3 graph_simple.py
[('a', 'c'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('c',
'e'), ('b', 'c'), ('b', 'e'), ('e', 'c'), ('e', 'b'),
('d', 'c')]
```

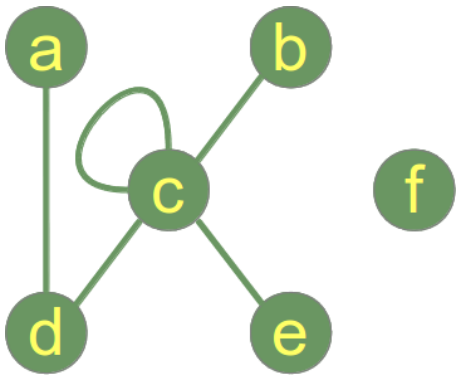
As we can see, there is no edge containing the node "f". "f" is an isolated node of our graph. The following Python function calculates the isolated nodes of a given graph:

```
def find_isolated_nodes(graph):
    """ returns a list of isolated nodes. """
    isolated = []
    for node in graph:
        if not graph[node]:
            isolated += 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, demo
nstrating the essential
facts and functionalities of grap
hs.
"""


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 vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def 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 emp
ty
            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 li
st;
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        (vertex1, vertex2) = tuple(edge)
        if vertex1 in self.__graph_dict:
            self.__graph_dict[vertex1].append(vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2]

    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 __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


    if __name__ == "__main__":

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


        graph = Graph(g)

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

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

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

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

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

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

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

        print('Adding an edge {"x","y"} with new vertices:')
```

```
graph.add_edge({"x","y"})
print("Vertices of graph:")
print(graph.vertices())
print("Edges of graph:")
print(graph.edges())
```

If you start this module standalone, you will get the following result:

```
$ python3 graph.py
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f']
Edges of graph:
[{'a', 'd'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}]
Add vertex:
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f', 'z']
Add an edge:
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f', 'z']
Edges of graph:
[{'a', 'd'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'},
{'a', 'z'}]
Adding an edge {"x","y"} with new vertices:
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f', 'y', 'z']
Edges of graph:
[{'a', 'd'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'},
{'a', 'z'}, {'y', 'x'}]
```

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.

The following method finds a path from a start vertex to an end vertex:

```
    def find_path(self, start_vertex, end_vertex, path=No
ne):
        """ 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_verte
x,
                                               path)
                if extended_path:
                    return extended_path
        return None
```

If we save our graph class including the find_path method as "graphs.py", we can check the way of working of our find_path function:

```
from graphs import Graph

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


graph = Graph(g)
```

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

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

The result of the previous program looks like this:

```
Vertices of graph:
['e', 'a', 'd', 'f', 'c', 'b']
Edges of graph:
[{'e', 'c'}, {'a', 'd'}, {'d', 'c'}, {'b', 'c'}, {'c'}]
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']
```

The method find_all_paths finds all the paths between a start vertex to an end vertex:

```
    def find_all_paths(self, start_vertex, end_vertex, pa
th=[]):
        """ 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(vert
ex,
                                                     end_
vertex,
                                                     pat
h)
                for p in extended_paths:
                    paths.append(p)
        return paths
```

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

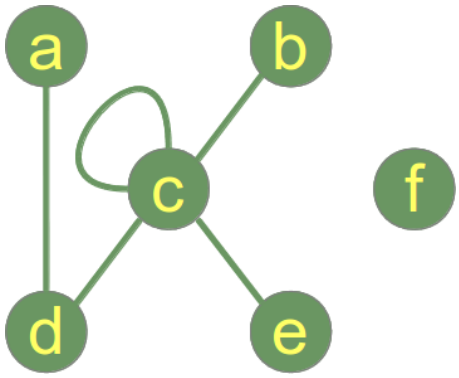
```
from graphs import Graph

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


graph = Graph(g)

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

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


print('All paths from vertex "a" to vertex "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)
```

The result looks like this:

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

DEGREE

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 following method calculates the degree of a vertex:

```
    def vertex_degree(self, vertex):
        """ The degree of a vertex is the number of edges
connecting
            it, i.e. the number of adjacent vertices. Loo
ps are counted
            double, i.e. every occurence of vertex in the
list
            of adjacent vertices. """
        adj_vertices =  self.__graph_dict[vertex]
        degree = len(adj_vertices) + adj_vertices.count(v
ertex)
        return degree
```

The following method calculates a list containing the isolated vertices of a graph:

```
    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
```

The methods delta and Delta can be used to calculate the minimum and maximum degree of a vertex respectively:

```
    def delta(self):
        """ the minimum degree of the vertices """
        min = 100000000
        for vertex in self.__graph_dict:
            vertex_degree = self.vertex_degree(vertex)
            if vertex_degree < min:
                min = vertex_degree
        return min

    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
```

DEGREE SEQUENCE

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:

```
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)
```

The degree sequence of our example graph is the following sequence of integers: (5,2,1,1,1,0). Isomorphic graphs have the same degree sequence. However, two graphs with the same degree sequence are not necessarily isomorphic.

There is the question whether a given degree sequence can be realized by a simple graph. The Erdös-Gallai theorem states that a non-increasing sequence of n numbers d_i (for i = 1, ..., n) is the degree sequence of a simple graph if and only if the sum of the sequence is even and the following condition is fulfilled:

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k) \quad \text{for } k \in \{1, \ldots, n\}$$

IMPLEMENTATION OF THE ERDÖS-GALLAI THEOREM

Our graph class - see further down for a complete listing - contains a method "erdoes_gallai" which decides, if a sequence fulfills the Erdös-Gallai theorem. First, we check, if the sum of the elements of the sequence is odd. If so the function returns False, because the Erdös-Gallai theorem can't be fulfilled anymore. After this we check with the static method is_degree_sequence whether the input sequence is a degree sequence, i.e. that the elements of

the sequence are non-increasing. This is kind of superfluous, as the input is supposed to be a degree sequence, so you may drop this check for efficiency. Now, we check in the body of the second if statement, if the inequation of the theorem is fulfilled:

```
    @staticmethod
    def erdoes_gallai(dsequence):
        """ Checks if the condition of the Erdoes-Gallai
inequality
            is fullfilled
        """
        if sum(dsequence) % 2:
            # sum of sequence is odd
            return False
        if Graph.is_degree_sequence(dsequence):
            for k in range(1,len(dsequence) + 1):
                left = sum(dsequence[:k])
                right =  k * (k-1) + sum([min(x,k) for x
in dsequence[k:]])
                if left > right:
                    return False
        else:
            # sequence is increasing
            return False
        return True
```

Version without the superfluous degree sequence test:

```
    @staticmethod
    def erdoes_gallai(dsequence):
        """ Checks if the condition of the Erdoes-Gallai
inequality
            is fullfilled
            dsequence has to be a valid degree sequence
        """
        if sum(dsequence) % 2:
            # sum of sequence is odd
            return False
        for k in range(1,len(dsequence) + 1):
            left = sum(dsequence[:k])
            right =  k * (k-1) + sum([min(x,k) for x in d
sequence[k:]])
            if left > right:
                return False
        return True
```

GRAPH DENSITY

The graph density is defined as the ratio of the number of edges of a given graph, and the total number of edges, the graph could have. In other words: It measures how close a given graph is to a complete graph.
The maximal density is 1, if a graph is complete. This is clear, because the maximum number of edges in a graph depends on the vertices and can be calculated as:
max. number of edges = ½ * |V| * (|V| - 1).

On the other hand the minimal density is 0, if the graph has no edges, i.e. it is an isolated graph.
For undirected simple graphs, the graph density is defined as:

$$D = \frac{2|E|}{|V|\,(|V| - 1)}$$

A dense graph is a graph in which the number of edges is close to the maximal number of edges. A graph with only a few edges, is called a sparse graph. The definition for those two terms is not very sharp, i.e. there is no least upper bound (supremum) for a sparse density and no greatest lower bound (infimum) for defining a dense graph.

The precisest mathematical notation uses the big O notation:

Sparse Graph: Dense Graph:
A dense graph is a graph G = (V, E) in which $|E| = \Theta(|V|^2)$.

"density" is a method of our class to calculate the density of a graph:

```
    def density(self):
        """ method to calculate the density of a graph
"""
        g = self.__graph_dict
        V = len(g.keys())
        E = len(self.edges())
        return 2.0 * E / (V *(V - 1))
```

We can test this method with the following script.

```
from graph2 import Graph

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

```
        "e" : ["c"],
        "f" : ["a"]
}

complete_graph = {
    "a" : ["b","c"],
    "b" : ["a","c"],
    "c" : ["a","b"]
}

isolated_graph = {
    "a" : [],
    "b" : [],
    "c" : []
}


graph = Graph(g)
print(graph.density())

graph = Graph(complete_graph)
print(graph.density())

graph = Graph(isolated_graph)
print(graph.density())
```

A complete graph has a density of 1 and isolated graph has a density of 0, as we can see from the results of the previous test script:

```
$ python test_density.py
0.466666666667
1.0
0.0
```

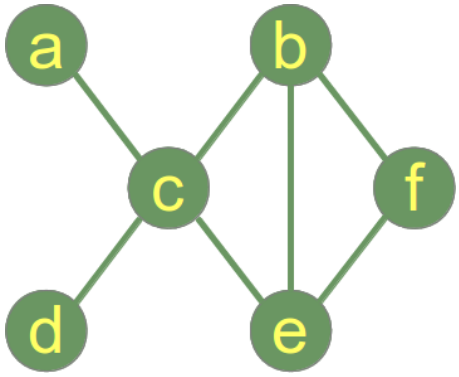
CONNECTED GRAPHS

A graph is said to be connected if every pair of vertices in the graph is connected. The example graph on the right side is a connected graph.

It possible to determine with a simple algorithm whether a graph is connected:

1. Choose an arbitrary node x of the graph G as the starting point
2. Determine the set A of all the nodes which can be reached from x.

3. If A is equal to the set of nodes of G, the graph is connected; otherwise it is disconnected.



We implement a method is_connected to check if a graph is a connected graph. We don't put emphasis on efficiency but on readability.

```
    def is_connected(self,
                     vertices_enc
ountered = None,
                     start_vertex
=None):
        """ determines if the gra
ph is connected """
        if vertices_encountered is None:
            vertices_encountered = set()
        gdict = self.__graph_dict
        vertices = list(gdict.keys()) # "list" necessary
in Python 3
        if not start_vertex:
            # chosse a vertex from graph as a starting po
int
            start_vertex = vertices[0]
        vertices_encountered.add(start_vertex)
        if len(vertices_encountered) != len(vertices):
            for vertex in gdict[start_vertex]:
                if vertex not in vertices_encountered:
                    if self.is_connected(vertices_encount
ered, vertex):
                        return True
        else:
            return True
        return False
```

If you add this method to our graph class, we can test it with the following script. Assuming that you save the graph class as graph2.py:

```
from graph2 import Graph

g = { "a" : ["d"],
      "b" : ["c"],
      "c" : ["b", "c", "d", "e"],
      "d" : ["a", "c"],
      "e" : ["c"],
```

```
      "f" : []
}

g2 = { "a" : ["d","f"],
       "b" : ["c"],
       "c" : ["b", "c", "d", "e"],
       "d" : ["a", "c"],
       "e" : ["c"],
       "f" : ["a"]
}

g3 = { "a" : ["d","f"],
       "b" : ["c","b"],
       "c" : ["b", "c", "d", "e"],
       "d" : ["a", "c"],
       "e" : ["c"],
       "f" : ["a"]
}


graph = Graph(g)
print(graph)
print(graph.is_connected())

graph = Graph(g2)
print(graph)
print(graph.is_connected())

graph = Graph(g3)
print(graph)
print(graph.is_connected())
```

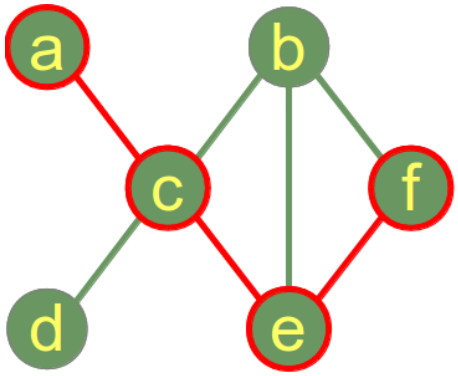
A connected component is a maximal connected subgraph of G. Each vertex belongs to exactly one connected component, as does each edge.

DISTANCE AND DIAMETER OF A GRAPH

The distance "dist" between two vertices in a graph is the length of the shortest path between these vertices. No backtracks, detours, or loops are allowed for the calculation of a distance.

In our example graph on the right, the distance between the vertex a and the vertex f is 3, i.e. dist(a,f) = 3, because the shortest way is via the vertices c and e (or c and b alternatively).

The eccentricity of a vertex s of a graph g is the maximal distance to every other vertex of the graph:
e(s) = max({ dist(s,v) | v ∈ V })
(V is the set of all vertices of g)



The diameter d of a graph is defined as the maximum eccentricity of any vertex in the graph. This means that the diameter is the length of the shortest path between the most distanced nodes. To determine the diameter of a graph, first find the shortest path between each pair of vertices. The greatest length of any of these paths is the diameter of the graph.

We can directly see in our example graph that the diameter is 3, because the minimal length between a and f is 3 and there is no other pair of vertices with a longer path.

The following method implements an algorithm to calculate the diameter.

```
    def diameter(self):
        """ calculates the diameter of the graph """

        v = self.vertices()
        pairs = [ (v[i],v[j]) for i in range(len(v)-1) fo
r j in range(i+1, len(v))]
        smallest_paths = []
        for (s,e) in pairs:
            paths = self.find_all_paths(s,e)
            smallest = sorted(paths, key=len)[0]
            smallest_paths.append(smallest)

        smallest_paths.sort(key=len)

        # longest path is at the end of list,
        # i.e. diameter corresponds to the length of this
path
        diameter = len(smallest_paths[-1]) - 1
        return diameter
```

We can calculate the diameter of our example graph with the following script, assuming again, that our complete graph class is saved as graph2.py:

```
from graph2 import Graph

g = { "a" : ["c"],
      "b" : ["c","e","f"],
```

```
        "c" : ["a","b","d","e"],
        "d" : ["c"],
        "e" : ["b","c","f"],
        "f" : ["b","e"]
}


graph = Graph(g)

diameter = graph.diameter()

print(diameter)
```

It will print out the value 3.

THE COMPLETE PYTHON GRAPH CLASS

In the following Python code, you find the complete Python Class Module with all the discussed methodes: graph2.py

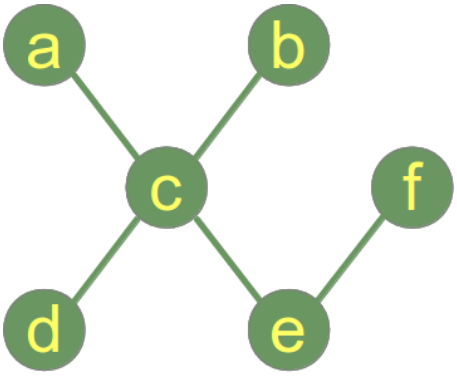
TREE / FOREST

A tree is an undirected graph which contains no cycles. This means that any two vertices of the graph are connected by exactly one simple path.

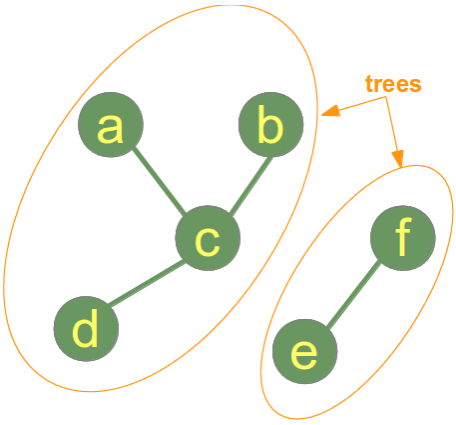
A forest is a disjoint union of trees. Contrary to forests in nature, a forest in graph theory can consist of a single tree!

A graph with one vertex and no edge is a tree (and a forest).

An example of a tree:



While the previous example depicts a graph which is a tree and forest, the following picture shows a graph which consists of two trees, i.e. the graph is a forest but not a tree:

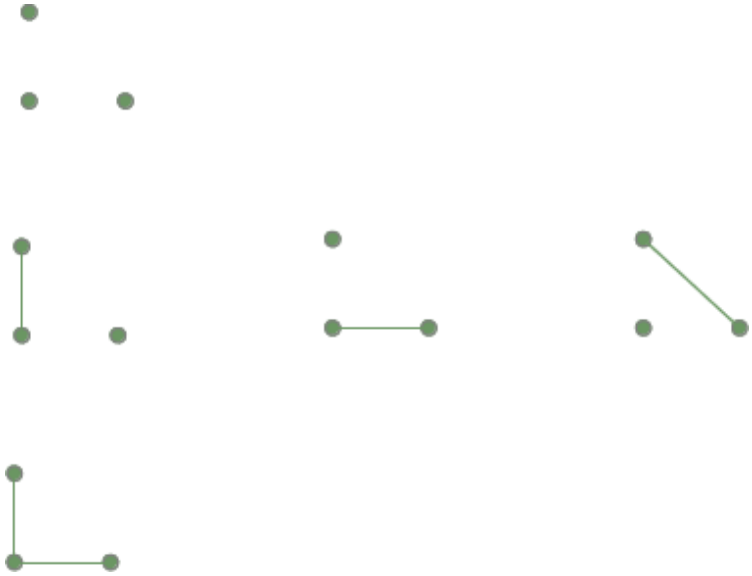


OVERVIEW OF FORESTS:

With one vertex:

Forest graphs with two vertices:

Forest graphs with three vertices:

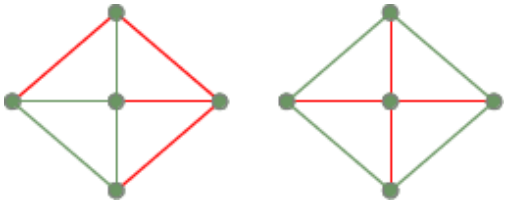


SPANNING TREE

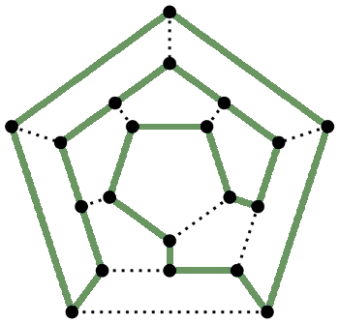
A spanning tree T of a connected, undirected graph G is a subgraph G' of G, which is a tree, and G' contains all the vertices and a subset of the edges of G. G' contains all the edges of G, if G is a tree graph. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex. That is, every vertex lies in the tree, but no cycles (or loops) are contained.

Example:
A fully connected graph:

Two spanning trees from the previous fully connected graph:



HAMILTONIAN GAME



An Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once. A Hamiltonian cycle (or circuit) is a Hamiltonian path that is a cycle.
Note for computer scientists: Generally, it is not not possible to determine, whether such paths or cycles exist in arbitrary graphs, because the Hamiltonian path problem has been proven to be NP-complete.

It is named after William Rowan Hamilton who invented the so-called "icosian game", or Hamilton's puzzle, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Hamilton solved this problem using the icosian calculus, an algebraic structure based on roots of unity with many similarities to the quaternions, which he also invented.

COMPLETE LISTING OF THE GRAPH CLASS

```
""" 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 d
ictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self.__graph_dict = graph_dict


    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def 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 emp
ty
            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 li
st;
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        vertex1 = edge.pop()
        if edge:
            # not a loop
```

```
            vertex2 = edge.pop()
        else:
            # a loop
            vertex2 = vertex1
        if vertex1 in self.__graph_dict:
            self.__graph_dict[vertex1].append(vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2]

    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 __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_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 find_path(self, start_vertex, end_vertex, path=
[]):
        """ find a path 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 None
        for vertex in graph[start_vertex]:
            if vertex not in path:
                extended_path = self.find_path(vertex,
                                               end_verte
x,
                                               path)
                if extended_path:
                    return extended_path
        return None


    def find_all_paths(self, start_vertex, end_vertex, pa
th=[]):
        """ 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(vert
ex,
                                                     end_
vertex,
                                                     pat
h)
                for p in extended_paths:
                    paths.append(p)
        return paths

    def is_connected(self,
                     vertices_encountered = None,
                     start_vertex=None):
        """ determines if the graph is connected """
        if vertices_encountered is None:
            vertices_encountered = set()
        gdict = self.__graph_dict
        vertices = list(gdict.keys()) # "list" necessary
in Python 3
```

```
        if not start_vertex:
            # chosse a vertex from graph as a starting po
int
            start_vertex = vertices[0]
        vertices_encountered.add(start_vertex)
        if len(vertices_encountered) != len(vertices):
            for vertex in gdifrom graph2 import Graph

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

graph = Graph(g)
print(graph)

for node in graph.vertices():
    print(graph.vertex_degree(node))

print("List of isolated vertices:")
print(graph.find_isolated_vertices())

print("""A path from "a" to "e":""")
print(graph.find_path("a", "e"))

print("""All pathes from "a" to "e":""")
print(graph.find_all_paths("a", "e"))

print("The maximum degree of the graph is:")
print(graph.Delta())

print("The minimum degree of the graph is:")
print(graph.delta())

print("Edges:")
print(graph.edges())

print("Degree Sequence: ")
ds = graph.degree_sequence()
print(ds)

fullfilling = [ [2, 2, 2, 2, 1, 1],
                      [3, 3, 3, 3, 3, 3],
                      [3, 3, 2, 1, 1]
```

```
                     ]
non_fullfilling = [ [4, 3, 2, 2, 2, 1, 1],
                    [6, 6, 5, 4, 4, 2, 1],
                    [3, 3, 3, 1] ]

for sequence in fullfilling + non_fullfilling :
    print(sequence, Graph.erdoes_gallai(sequence))

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

print("Add edge ('x','y'): ")
graph.add_edge(('x', 'y'))
print(graph)

print("Add edge ('a','d'): ")
graph.add_edge(('a', 'd'))
print(graph)
ct[start_vertex]:
                if vertex not in vertices_encountered:
                    if self.is_connected(vertices_encount
ered, vertex):
                        return True
        else:
            return True
        return False

    def vertex_degree(self, vertex):
        """ The degree of a vertex is the number of edges
connecting
            it, i.e. the number of adjacent vertices. Loo
ps are counted
            double, i.e. every occurence of vertex in the
list
            of adjacent vertices. """
        adj_vertices =  self.__graph_dict[vertex]
        degree = len(adj_vertices) + adj_vertices.count(v
ertex)
        return degree

    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)

    @staticmethod
    def is_degree_sequence(sequence):
        """ Method returns True, if the sequence "sequenc
e" is a
            degree sequence, i.e. a non-increasing sequen
ce.
            Otherwise False is returned.
        """
        # check if the sequence sequence is non-increasin
g:
        return all( x>=y for x, y in zip(sequence, sequen
ce[1:]))


    def delta(self):
        """ the minimum degree of the vertices """
        min = 100000000
        for vertex in self.__graph_dict:
            vertex_degree = self.vertex_degree(vertex)
            if vertex_degree < min:
                min = vertex_degree
        return min

    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 density(self):
        """ method to calculate the density of a graph
"""
        g = self.__graph_dict
        V = len(g.keys())
        E = len(self.edges())
        return 2.0 * E / (V *(V - 1))

    def diameter(self):
        """ calculates the diameter of the graph """

        v = self.vertices()
        pairs = [ (v[i],v[j]) for i in range(len(v)) for
```

```
j in range(i+1, len(v)-1)]
        smallest_paths = []
        for (s,e) in pairs:
            paths = self.find_all_paths(s,e)
            smallest = sorted(paths, key=len)[0]
            smallest_paths.append(smallest)

        smallest_paths.sort(key=len)

        # longest path is at the end of list,
        # i.e. diameter corresponds to the length of this
path
        diameter = len(smallest_paths[-1]) - 1
        return diameter

    @staticmethod
    def erdoes_gallai(dsequence):
        """ Checks if the condition of the Erdoes-Gallai
inequality
            is fullfilled
        """
        if sum(dsequence) % 2:
            # sum of sequence is odd
            return False
        if Graph.is_degree_sequence(dsequence):
            for k in range(1,len(dsequence) + 1):
                left = sum(dsequence[:k])
                right =  k * (k-1) + sum([min(x,k) for x
in dsequence[k:]])
                if left > right:
                    return False
        else:
            # sequence is increasing
            return False
        return True
```

We can test this Graph class with the following program:

```
from graphs import Graph

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

```
graph = Graph(g)
print(graph)

for node in graph.vertices():
    print(graph.vertex_degree(node))

print("List of isolated vertices:")
print(graph.find_isolated_vertices())

print("""A path from "a" to "e":""")
print(graph.find_path("a", "e"))

print("""All pathes from "a" to "e":""")
print(graph.find_all_paths("a", "e"))

print("The maximum degree of the graph is:")
print(graph.Delta())

print("The minimum degree of the graph is:")
print(graph.delta())

print("Edges:")
print(graph.edges())

print("Degree Sequence: ")
ds = graph.degree_sequence()
print(ds)

fullfilling = [ [2, 2, 2, 2, 1, 1],
                      [3, 3, 3, 3, 3, 3],
                      [3, 3, 2, 1, 1]
                    ]
non_fullfilling = [ [4, 3, 2, 2, 2, 1, 1],
                    [6, 6, 5, 4, 4, 2, 1],
                    [3, 3, 3, 1] ]

for sequence in fullfilling + non_fullfilling :
    print(sequence, Graph.erdoes_gallai(sequence))

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

print("Add edge ('x','y'): ")
graph.add_edge(('x', 'y'))
print(graph)
```

```
print("Add edge ('a','d'): ")
graph.add_edge(('a', 'd'))
print(graph)
```

If we start this program, we get the following output:

```
vertices: c e f a d b
edges: {'c', 'b'} {'c'} {'c', 'd'} {'c', 'e'} {'d', 'a'}
5
1
0
1
2
1
List of isolated vertices:
[] c
[] e
[] f
['f'] a
['f'] d
['f'] b
['f']
A path from "a" to "e":
['a', 'd', 'c', 'e']
All pathes from "a" to "e":
[['a', 'd', 'c', 'e']]
The maximum degree of the graph is:
5
The minimum degree of the graph is:
0
Edges:
[{'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'d', 'a'}]
Degree Sequence:
(5, 2, 1, 1, 1, 0)
[2, 2, 2, 2, 1, 1] True
[3, 3, 3, 3, 3, 3] True
[3, 3, 2, 1, 1] True
[4, 3, 2, 2, 2, 1, 1] False
[6, 6, 5, 4, 4, 2, 1] False
[3, 3, 3, 1] False
Add vertex 'z':
vertices: c e f z a d b
edges: {'c', 'b'} {'c'} {'c', 'd'} {'c', 'e'} {'d', 'a'}
Add edge ('x','y'):
vertices: c e f z a y d b
edges: {'c', 'b'} {'c'} {'c', 'd'} {'c', 'e'} {'d', 'a'}
```

```
{'y', 'x'}
Add edge ('a','d'):
vertices: c e f z a y d b
edges: {'c', 'b'} {'c'} {'c', 'd'} {'c', 'e'} {'d', 'a'}
{'y', 'x'}
```

Footnotes:

[1] The graphs studied in graph theory (and in this chapter of our Python tutorial) should not be confused with the graphs of functions

[2] A singleton is a set that contains exactly one element.

Credits:
Narayana Chikkam, nchikkam(at)gmail(dot)com, pointed out an index error in the "erdoes_gallai" method. Thank you very much, Narayana!

PYTHON-MODUL PYGRAPH

This chapter is still not finished. We are working on it. This software provides a suitable data structure for representing graphs and a whole set of important algorithms. Provided features and algorithms:

- Support for directed, undirected, weighted and non-weighted graphs
- Support for hypergraphs
- Canonical operations
- XML import and export
- DOT-Language output (for usage with Graphviz)
- Random graph generation
- Accessibility (transitive closure)
- Breadth-first search
- Critical path algorithm
- Cut-vertex and cut-edge identification
- Cycle detection
- Depth-first search
- Heuristic search (A* algorithm)
- Identification of connected components
- Minimum spanning tree (Prim's algorithm)
- Mutual-accessibility (strongly connected components)
- Shortest path search (Dijkstra's algorithm)
- Topological sorting
- Transitive edge identification

PYTHON-MODUL PYGRAPHVIZ

With Pygraphviz you can create, edit, read, write, and draw graphs using Python to access the Graphviz graph data structure and layout algorithms.

NETWORKX

OVERVIEW

This chapter is still not finished. We are working on it.

NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Pygraphviz is a Python interface to the Graphviz graph layout and visualization package.

- Python language data structures for graphs, digraphs, and multigraphs.
- Nodes can be "anything" (e.g. text, images, XML records)
- Edges can hold arbitrary data (e.g. weights, time-series)
- Generators for classic graphs, random graphs, and synthetic networks
- Standard graph algorithms
- Network structure and analysis measures
- Basic graph drawing
- Open source BSD license
- Well tested: more than 1500 unit tests
- Additional benefits from Python: fast prototyping, easy to teach, multi-platform

CREATING A GRAPH

CREATE AN EMPTY GRAPH

Our first example of a graph will be an empty graph. To see the proper mathematical definition of a graph, you can have a look at our previous chapter Graphs in Python. The following little Python script uses NetworkX to create an empty graph:

```
import networkx as nx

G=nx.Graph()

print(G.nodes())
print(G.edges())

print(type(G.nodes()))
print(type(G.edges()))
```

If we save this script as "empty.py" and start it, we get the following output:

```
$ python3 empyty.py
[]
[]
<class 'list'>
<class 'list'>
```

We can see that the result from the graph methods nodes() and edges() are lists.

ADDING NODES TO OUR GRAPH

Now we will add some nodes to our graph. We can add one node with the method add_node() and a list of nodes with the method add_nodes_from():

```
import networkx as nx

G=nx.Graph()

# adding just one node:
G.add_node("a")
# a list of nodes:
G.add_nodes_from(["b","c"])

print("Nodes of graph: ")
print(G.nodes())
print("Edges of graph: ")
print(G.edges())
```

ADDING EDGES TO OUR GRAPH

G can also be created or increased by adding one edge at a time by the method add_edge(), which has the two nodes of the edge as the two parameters. If we have a tuple or a list as the edge, we can use the asterisk operator to unpack the tuple or the list:

```
import networkx as nx

G=nx.Graph()
G.add_node("a")
G.add_nodes_from(["b","c"])

G.add_edge(1,2)
edge = ("d", "e")
G.add_edge(*edge)
edge = ("a", "b")
G.add_edge(*edge)
```

```
print("Nodes of graph: ")
print(G.nodes())
print("Edges of graph: ")
print(G.edges())
```

In our previous example, the first edge consists of the nodes 1 and 2, which had not been included in our graph so far. The same is true for the second edge with the tuple `("d", "e")`. We can see that the nodes will be automatically included as well into the graph, as we can see from the output:

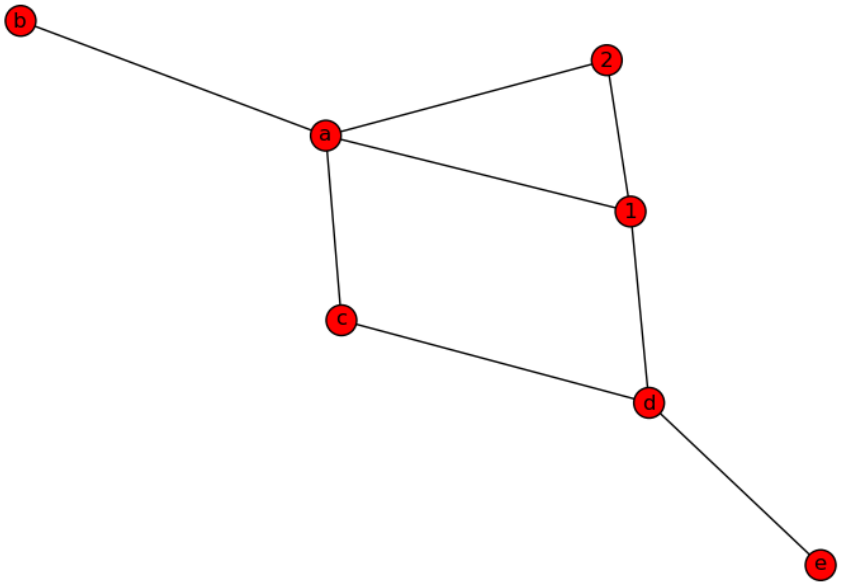
```
Nodes of graph:
['a', 1, 'c', 'b', 'e', 'd', 2]
Edges of graph:
[('a', 'b'), (1, 2), ('e', 'd')]
```

We can add a bunch of edges as a list of edges in the form of 2 tuples.

```
# adding a list of edges:
G.add_edges_from([("a","c"),("c","d"), ("a",1), (1,"d"),
("a",2)])
```

We can also print the resulting graph by using matplotlib:

```
nx.draw(G)
plt.savefig("simple_path.png") # save as png
plt.show() # display
```

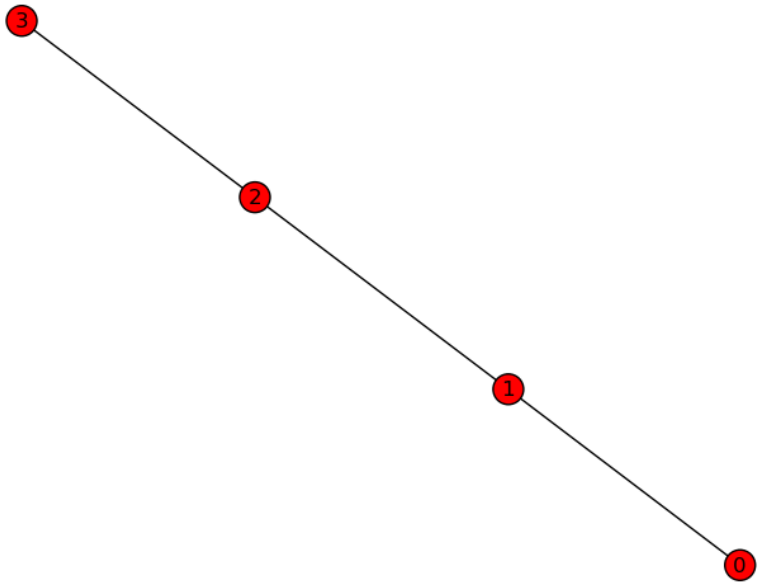


GENERATE PATH GRAPH

We can create a Path Graph with linearly connected nodes with the method path_graph(). The Python code code uses matplotlib. pyplot to plot the graph. We will give detailed information on matplotlib at a later stage of the tutorial:

```
import networkx as nx
import matplotlib.pyplot as plt

G=nx.path_graph(4)

print("Nodes of graph: ")
print(G.nodes())
print("Edges of graph: ")
print(G.edges())
nx.draw(G)
plt.savefig("path_graph1.png")
plt.show()
```

The created graph is an undirected linearly connected graph, connecting the integer numbers 0 to 3 in their natural order:



RENAMING NODES

Sometimes it is necessary to rename or relabel the nodes of an existing graph. For this purpose the function relabel_nodes is the ideal tool.

```
networkx.relabel.relabel_nodes(G, mapping, copy=True)
```

The parameter G is a Graph, the mapping has to be a dictionary and the last parameter is optional. If copy is set to True, - which is the default - a copy will be returned, otherwise, i.e. if it is set to False, the nodes of the graph will be relabelled in place.
In the following example we create again the Path graph with the node labels from 0 to 3. After this we define a dictionary, in which we map each node label into a new value, i.e. city names:

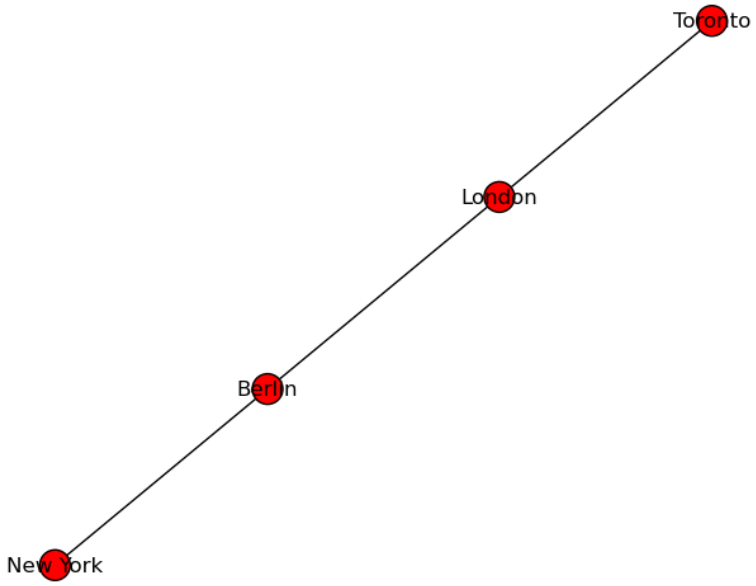
```
import networkx as nx
import matplotlib.pyplot as plt

G=nx.path_graph(4)
cities = {0:"Toronto",1:"London",2:"Berlin",3:"New York"}
```

```
H=nx.relabel_nodes(G,cities)

print("Nodes of graph: ")
print(H.nodes())
print("Edges of graph: ")
print(H.edges())
nx.draw(H)
plt.savefig("path_graph_cities.png")
plt.show()
```

The Python program returns the following output:

```
Nodes of graph:
['Toronto', 'Berlin', 'New York', 'London']
Edges of graph:
[('Toronto', 'London'), ('Berlin', 'New York'), ('Berli
n', 'London')]
```

The visualized graph looks liks this:



When we relabelled the graph G in our previous Python exampls, we create a new graph H,

while the original graph G was not changed. By setting the copy parameter flag to False, we can relabel the nodes in place without copying the graph. In this case the line

`H=nx.relabel_nodes(G,cities)`

will be changed to

`nx.relabel_nodes(G,cities, copy=False)`

This approach might lead to problems, if the mapping is circular, while copying is always safe. The mapping from the nodes of the original node labels to the new node labels doesn't have to be complete. An example of a partial in-place mapping:

```
import networkx as nx

G=nx.path_graph(10)
mapping=dict(zip(G.nodes(),"abcde"))
nx.relabel_nodes(G, mapping, copy=False)

print("Nodes of graph: ")
print(G.nodes())
```

Only the nodes 0 to 4 are nenamed, while the other nodes keep the numerical value, as we can see in the output from the program:

```
$ python3 partial_relabelling.py
Nodes of graph:
[5, 6, 7, 8, 9, 'c', 'b', 'a', 'e', 'd']
```

The mapping for the nodes can be a function as well:

```
import networkx as nx

G=nx.path_graph(10)

def mapping(x):
    return x + 100

nx.relabel_nodes(G, mapping, copy=False)

print("Nodes of graph: ")
print(G.nodes())
```

The result:

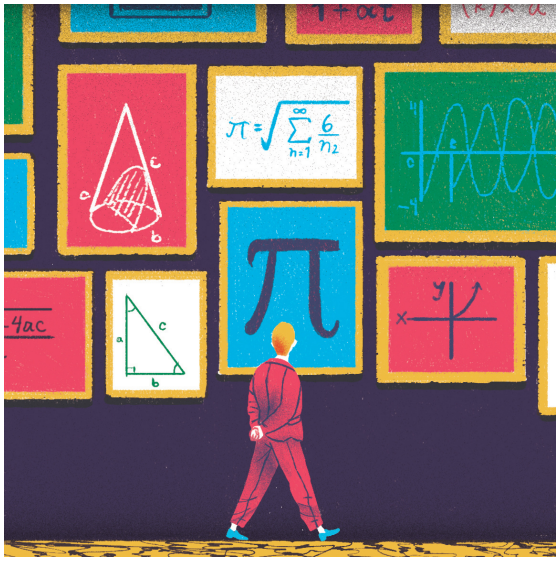
```
$ python3 relabelling_with_function.py
Nodes of graph:
[107, 106, 103, 108, 109, 104, 105, 100, 102, 101]
```

POLYNOMIALS

INTRODUCTION

If you have been to highschool, you will have encountered the terms **polynomial** and **polynomial function**. This chapter of our Python tutorial is completely on polynomials, i.e. we will define a class to define polynomials. The following is an example of a polynomial with the degree 4:

$$p(x) = x^4 - 4 \cdot x^2 + 3 \cdot x$$



You will find out that there are lots of similarities to integers. We will define various arithmetic operations for polynomials in our class, like addition, subtraction, multiplication and division. Our polynomial class will also provide means to calculate the derivation and the integral of polynomials. We will not miss out on plotting polynomials.

There is a lot of beaty in polynomials and above all in how they can be implemented as a Python class. We like to say thanks to Drew Shanon who granted us permission to use his great picture, treating math as art!

SHORT MATHEMATICAL INTRODUCTION

We will only deal with polynomial in a single indeterminate (also called variable) x. A general form of a polynomial in a single indeterminate looks like this:

$$a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

where $a_0, a_1, \ldots a_n$ are the constants - non-negative integers - and x is the indeterminate or variable. The term "indeterminate" means that x represents no particular value, but any value may be substituted for it.

This expression is usually written with the summation operator:

$$\sum_{k=0}^{n} a_k \cdot x^k = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

A polynomial function is a function that can be defined by evaluating a polynomial. A function f of one argument can be defined as:

$$f(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

POLYNOMIAL FUNCTIONS WITH PYTHON

It's easy to implement polynomial functions in Python. As an example we define the polynomial function given in the introduction of this chapter, i.e. $p(x) = x^4 - 4 \cdot x^2 + 3 \cdot x$

The Python code for this polynomial function looks like this:

We can call this function like any other function:

```
-1 -6
0 0
2 6
3.4 97.59359999999998
```

```
<Figure size 640x480 with 1 Axes>
```

POLYNOMIAL CLASS

We will define now a class for polynomial functions. We will build on an idea which we have developed in the chapter on decorators of our Python tutorial. We introduced polynomial factories.

A polynomial is uniquely determined by its coefficients. This means, an instance of our polynomial class needs a list or tuple to define the coefficients.

We can the instantiate the polynomial of our previous example polynomial function like this:

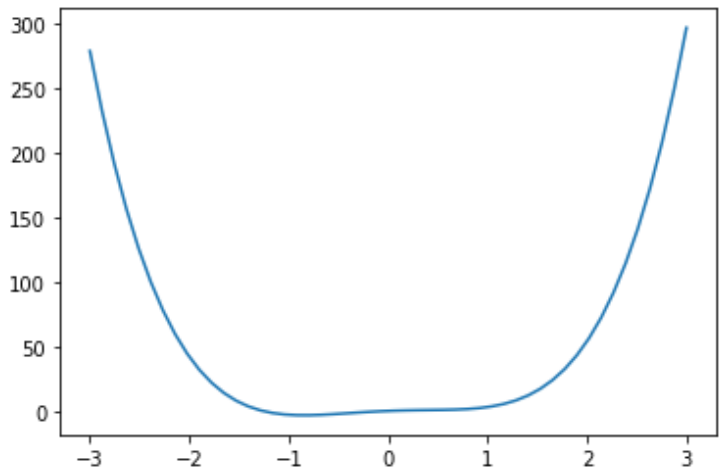
```
Polynomial(4, 0, -4, 3, 0)
Polynomial(4, 0, -4, 3, 0)
```

So far, we have defined **polynomials**, but what we actually need are **polynomial functions**. For this purpose, we turn instances of the Polynomial class into **callables** by defining the **call** method:

It is possible now to call an instance of our class like a function. We call it with an argument and the instannce, - which is a callable, - behaves like a polynomial function:

```
-3 279
-2 42
-1 -3
0 0
1 3
2 54
```

Just for fun, let us plot the previously defined function:



Before we further refine our class, let us give a numerically more efficient variant of our `__call__` method. We introduced this variant in our chapter on decorators.

Every polynomial

$f(x) = \sum_{k=0}^{n} a_k \cdot x^k$

can be also written in the form

$f(x) = (a_n x + x_{n-1})x + \cdot + a_1)x + a_0$

To emperically see that they are equivalent, we rewrite our class definition, but call it `Polynomial2` so that we can use both versions:

We will test both classes now:

```
True
```

It is possible to define addition and subtractions for polynomials. All we have to do is to add or subtract the coefficients with the same exponents from both polynomials.

If we have polynomial functions $f(x) = \sum_{k=0}^{n} a_k \cdot x^k$ and $g(x) = \sum_{k=0}^{n} b_k \cdot x^k$, the addition is defined as

$$(f+g)(x) = \sum_{k=0}^{n} (a_k + b_k) \cdot x^k$$

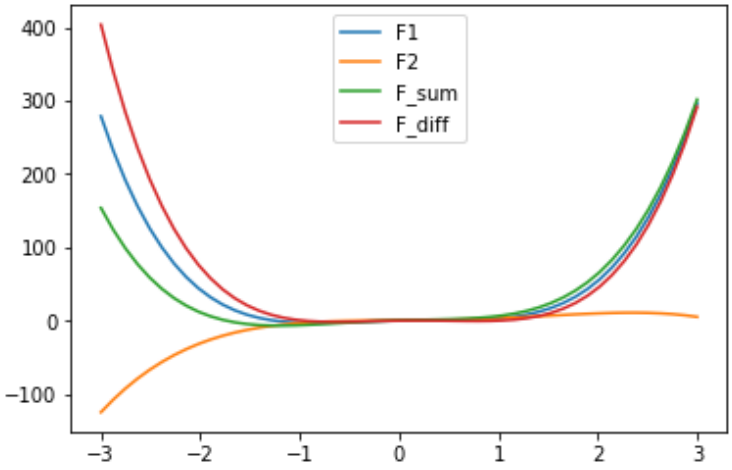
and correspondingly subtraction is defined as

$$(f-g)(x) = \sum_{k=0}^{n} (a_k - b_k) \cdot x^k$$

Before we can add the methods __add__ and __sub__, which are necessary for addition and subtraction, we add a generatur zip_longest(). It works similar to zip for two parameters, but it does not stop if one of the iterators is exhausted but uses "fillvalue" instead.

```
(2, -1)
(0, 4)
(0, 5)
```

We will add this generator to our class as a static method. We are now capable of adding the __add__ and __sub__ methods as well. We will need the generator `zip_longest` from itertools. `zip_longest` takes and arbitrary number of iterables and a keyword parameter `fillvalue` . It creates an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.



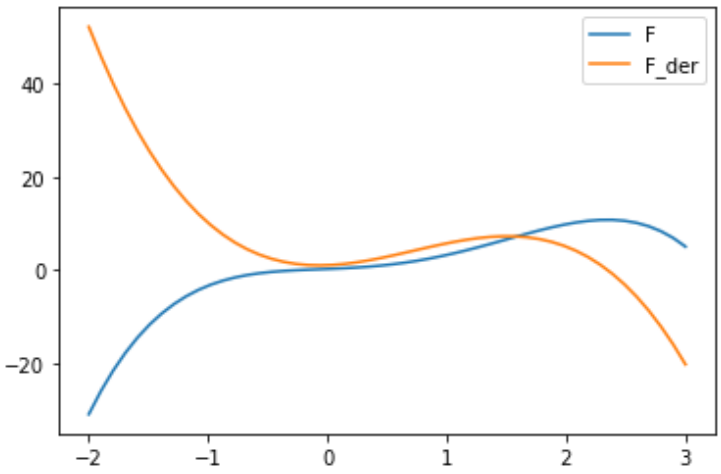
It is incredibly easy to add differentiation to our class. Mathematically, it is defined as

$$f'(x) = \sum_{k=0}^{n} k \cdot a_k \cdot x^{k-1}$$

if

$$f(x) = \sum_{k=0}^{n} a_k \cdot x^k$$

This can be easily implemented in our method 'derivative':



Playing around a little bit:

```
1x^4 + 2x^3 - 3x^2 + 4x^1 - 55
4x^3 + 6x^2 - 6x^1 + 4
1x^2 + 2x^1 + 3
-52x^4 + 6x^3 - 2x^2 + 2x^1 + 1
```

In [ ]:

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 not only used in programming but in theoretical computer science as well. The reason is that it is often easier to transform multiple argument models into single argument models.

The need for currying arises for example in the following case: Let us assume that we have a context, in which we can only use a function with one argument. We have to use a function with multiple parameters. So we need a way to transform this function into a function with just one parameter. Currying provides the solution to this problem. Currying means rearranging a multiple-parameter function into a chain of functions applied to one argument. It is always possible to transform a function with multiple arguments into a chain of single-argument functions.

Python is not equipped for this programming style. This means there are no special syntactical constructs available to support currying. On the other hand, Python is well suited to simulate this way of programming. We will introduce various ways to accomplish this in this chapter of our tutorial.

Some readers not interested in the mathematical details can skip the following two subchapters, because the a mathematically focussed.

COMPOSITION OF FUNCTIONS

We define the composition h of two function f and g

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

often written as

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

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.

CURRYING

As we have already metnioned in the introduction, currying means transforming a function with multiple parameters into a chain of functions with one parameter.

We will start with the simplest case, i.e. two parameters. Given is a function f with two parameters x and y. We can curry the function in the following way:

We have to find a function g which returns a function h when it is applied to the second parameter y of f. h is a function which can be applied to the first parameter x of f, satisfying the condition

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

Now we have a look at the general case of currying. Let us assume that we have a function f with n parameters:

$$f(x_1, x_2, \ldots x_n)$$

Currying leads to a cascade of functions:

$$\begin{aligned} f_{n-1} &= f_n(x_n) \\ &\vdots \\ f_1 &= f_2(x_2) \\ f(x_1, x_2, \ldots x_n) &= f_1(x_1) \end{aligned}$$

COMPOSITION OF FUNCTIONS IN PYTHON

TWO FUNCTIONS

The function *compose* can be used to create to compose two functions:

We will use our compose function in the next example. Let's assume, we have a thermometer, which is not working accurate. 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 into degrees fahrenheit. We can do this by applying *compose* to both functions:

Output::

```
(44.5, 50.0)
```

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

Output::

```
(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!

"COMPOSE" WITH AN ARBITRARY NUMBER OF ARGUMENTS

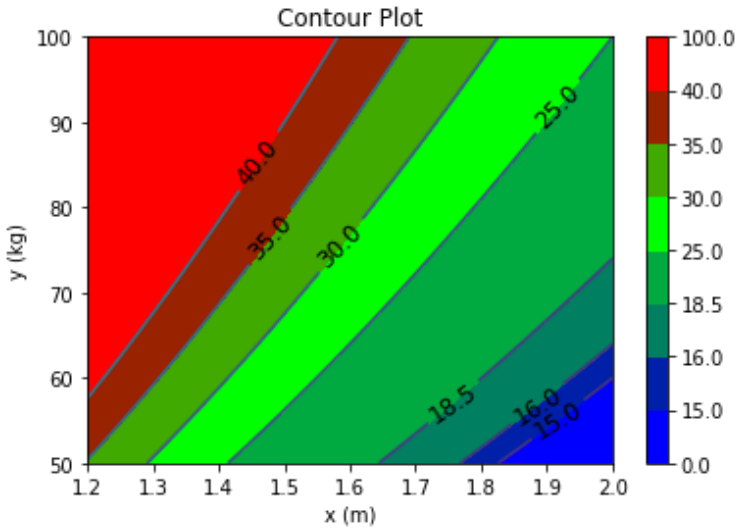
The function *compose* which we have just defined can only cope with single-argument functions. We can generalize our function *compose* so that it can cope with all possible functions. This is not currying of course but nevertheless also an interesting function.

Example using a function with two parmameters.

```
weight (kg) 70
height (m) 1.76
Normal (healthy weight)
weight (kg) 0
height (m) 1
Very severely underweight
```

BMI CHART

This is off-topic, because it is not about currying. Yet, it is nice to have a BMI chart, especially if it is created with Python means. We use contour plots from the Matplotlib module in our program. If you want to learn more about contour plots, you can go to chapter on Contour Plots of our Matplotlib tutorial.



CURRYING EXAMPLES IN PYTHON

CURRYING BMI

We used the function BMI in a composition in the previous example. We will now use it as a first currying example. The height of grown-ups is principally a constant. Okay, I know, we shrink every day from dawn to dusk about one centimetre and we get the loss returned over night. We define a function *f* which takes a height and returns a function whith one parameter (weight) to return the BMI:

```
0.00223081499107674 0.00223081499107674
0.0021258503401360546 0.0021258503401360546
0.0020281233098972417 0.0020281233098972417
0.002528256989886972 0.002528256989886972
0.002409297052154195 0.002409297052154195
0.002298539751216874 0.002298539751216874
0.002751338488994646 0.002751338488994646
0.0026218820861678006 0.0026218820861678006
0.002501352082206598 0.002501352082206598
```

EXAMPLE: CURRENCY CONVERSION

In the chapter on Magic Functions of our tutorial we had an excercise, in which we defined a class for currency conversions.

We will define now a function *exchange*, which takes three arguments:

1. The source currency
2. The target currency
3. The amount in the source currency

To function needs the actual exchange rates. We can download them from finance.yahoo.com website with the function get_currencies. Though in our example we use some old exchange rates:

Output::

```
137.56418155678784
```

We can now define *curried* functions from the function *exchange*:

```
83.17005545286507
83.17005545286507
```

We want to rewrite the function *exchange* in a curryable version:

We can redefine *exchange_from_CHF* and *CHF2EUR* in a properly curried way:

```
<function curry_exchange.<locals>.f.<locals>.h at 0x
7f7543673268>
83.17005545286507
```

You will find various calls to curry_exchange in the following examples:

```
<function curry_exchange.<locals>.f.<locals>.h at 0x
7f7543438950>
92.41117272540563
<function curry_exchange.<locals>.f.<locals>.h at 0x
7f754372a730>
92.41117272540563
92.41117272540563
92.41117272540563
<function curry_exchange.<locals>.f at 0x7f754343884
0>
92.41117272540563
<function curry_exchange.<locals>.f.<locals>.h at 0x
7f7543673268>
92.41117272540563
CHF 120.00000000000001
CAD 165.07701786814542
GBP 98.87861983980284
JPY 12684.9044978435
EUR 110.89340727048676
USD 123.22858903265559
```

So far we have written custom-made curry functions. We will define a general currying function in the following chapter of our tutorial.

GENERAL CURRYING

```
5.571428571428571
```

Output::

```
16
```

```
<function curry.<locals>.f at 0x7f7543702bf8>
9.241117272540563
```

Output::

```
102.69049086054633
```

```
<function curry.<locals>.f at 0x7f7543457048>
9.241117272540563
```

Output::

```
102.69049086054633
```

PARTIAL-FUNCTION

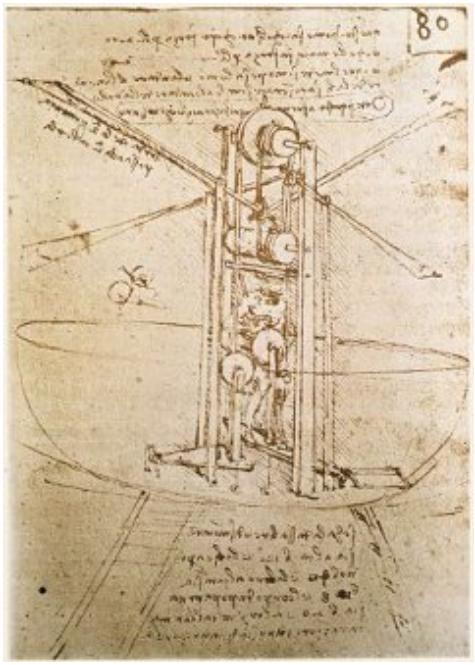
The function *partial* from the module *functools* can be used to simulate currying as well. It can also be used to "freeze" some of the function's arguments. This means it simplifies the functions signature.

We use it now to curry our function exchange once more in a different way:

Output::

```
102.69049086054633
```

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".

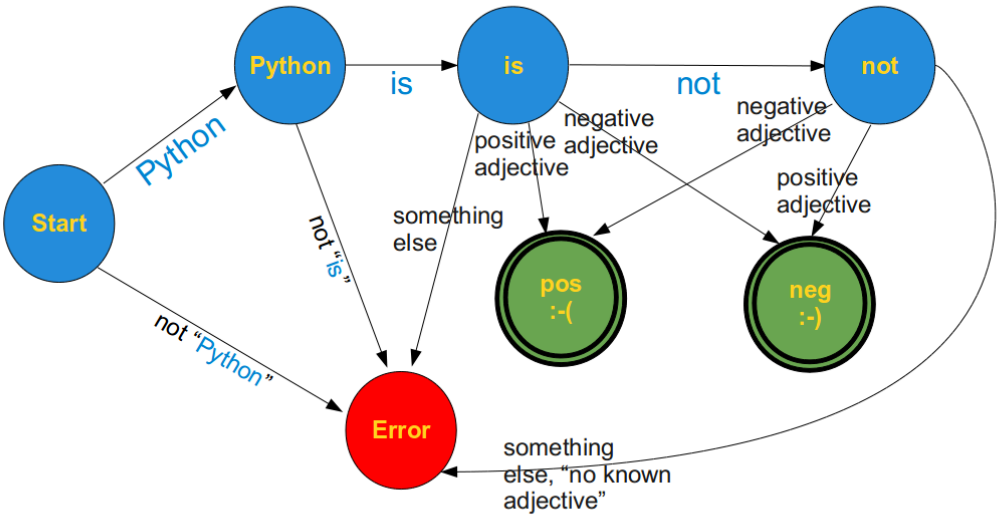
The operation of an FSM begins with a special state, called the start state, proceeds through transitions depending on input to different states and normally ends in terminal or end states. A state which marks a successful flow of operation is known as an accept state.

Mathematical Model:
A deterministic finite state machine or acceptor deterministic finite state machine is a quintuple (Σ,S,s_0,δ,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. We save this class as statemachine.py:

```
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_sta
    rt() before .run()")
            if not self.endStates:
                raise  InitializationError("at least one stat
    e must be 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:

```
from statemachine import StateMachine

positive_adjectives = ["great","super", "fun", "entertain
ing", "easy"]
negative_adjectives = ["boring", "difficult", "ugly", "ba
d"]

def start_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 els
e (txt,"")
    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 els
e (txt,"")
```

```
    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 els
e (txt,"")
    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 els
e (txt,"")
    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", "")

if __name__== "__main__":
    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")
```

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

If we save the application of our general Finite State Machine in statemachine_test.py and call it with

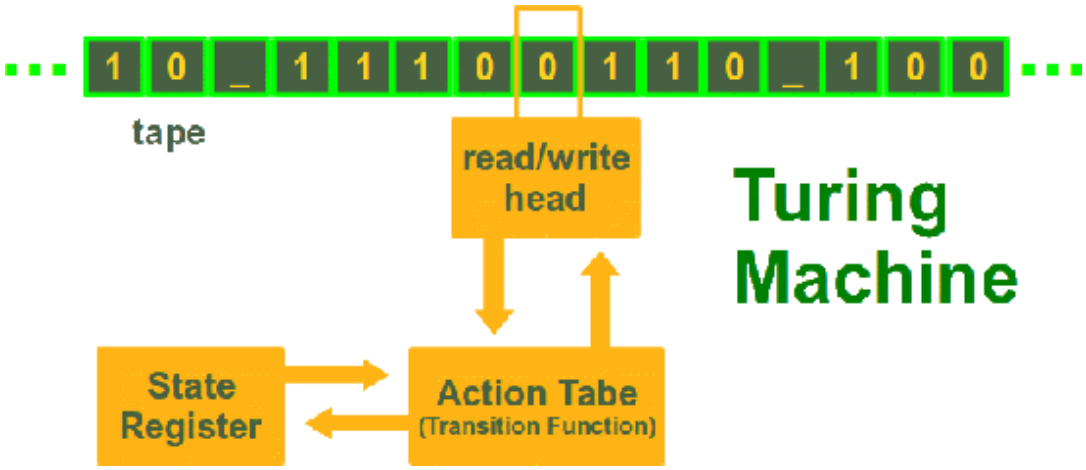
```
python statemachine_test.py
```

we get the following results:

```
$ python statemachine_test.py
reached  pos_state which is an end state
reached  neg_state which is an end state
reached  error_state which is an end state
```

The code of the finite state machine is compatible with Python3 as well!

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".
Σ = {0, 1}
Q = {init, final}
q_0 = init
q_f = final

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'`
	An 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', '.beagle', '.gnome2_private', '.gksu.lock', 'Public', '.ICEauthority', '.bash_history', '.compiz', '.gvfs', '.update-notifier', '.cache', 'Desktop', 'Videos', '.profile', '.config', '.esd_auth', '.viminfo', '.sudo_as_admin_successful', 'mbox', '.xsession-errors', '.bashrc', 'Music', '.dbus', '.local', '.gstreamer-0.10', 'Documents', '.gtk-bookmarks', 'Downloads', 'Pictures', '.pulse-cookie', '.nautilus', 'examples.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.

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.

The class TuringMachine:
We define the method __str__(self), which is called by the str() built-in function and by the print statement to compute the "informal" string representation of an object, in our case the string representation of a tape.

```
    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_funct
ion
        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_positio
n]
        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:

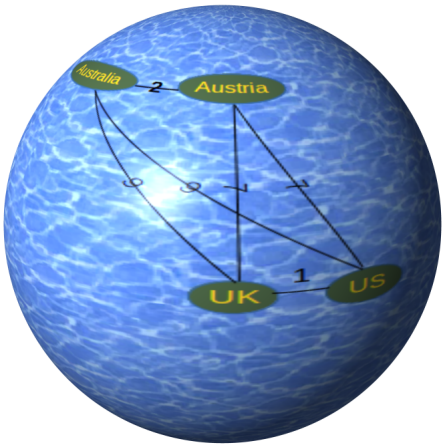
```
from turing_machine import TuringMachine

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("010011 ",
                  initial_state = "init",
                  final_states = final_states,
                  transition_function=transition_functio
n)

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())
```

The previous program prints the following results:

```
Input on Tape:
010011
Result of the Turing machine calculation:
101100
```

LEVENSHTEIN DISTANCE

INTRODUCTION



This chapter covers the Levenshtein distance and presents some Python implementations for this measure. There are lots of use cases for the Levenshtein distances. The Levenshtein Distance and the underlying ideas are widely used in areas like computer science, computer linguistics, and even bioinformatics, molecular biology, DNA analysis. You can even measure the similarity of melodies or rhythms in music[1]. The Levenshtein distance has widely permeated our everyday life. Whenever you use a program or an application using some form of spell checking and error correction, the programmers most likely will have used "edit distance" or as it is also called "Levenshtein distance".

You might already have encountered another possible use case for this concept: Imagine that you are using a Python dictionary, in which you use strings as keys.

Let us look at the following example dictionary with city names of the United States, which are often misspelled:

So, trying to get the corresponding state names via the following dictionary accesses will raise exceptions:

```
cities["Tuscon"]
cities["Pittsburg"]
cities["Cincinati"]
cities["Albequerque"]
```

If a human reader looks at these misspellings, he or she will have no problem in recognizing the city you have in mind. The Python dictionary on the other hand is pedantic and unforgivable. It only accepts a key, if it is exactly identical.

The question is to what degree are two strings similar? What we need is a string similarity metric or a measure for the "distance" of strings.

A string metric is a metric that measures the distance between two text strings. One of the best known string metrics is the so-called Levenshtein Distance, also known as Edit Distance. Levenshtein calculates the the number of substitutions and deletions needed in order to transform one string into another one.

THE MINIMUM EDIT DISTANCE OR LEVENSHTEIN DINSTANCE

The minimum edit distance between two strings is the minimum numer of editing operations needed to convert one string into another. The editing operations can consist of insertions, deletions and substitutions.

The simplest sets of edit operations can be defined as:

- Insertion of a single symbol. This means that we add a character to a string s. Example: If we have the string s = "Manhatan", we can insert the character "t" to get the correct spelling:

```
>>> s = "Manhatan"
>>> s = s[:5] + "t" + s[5:]
>>> print(s)
Manhattan
```

- Deletion of a single symbol Example:

```
>>> s = "Mannhattan"
>>> s = s[:2] + s[3:]
>>> s
'Manhattan'
```

- Substitution of a single symbol In the following example, we have to change the letter "o" into the letter "a" to get the correct spelling:

```
>>> s = "Manhatton"
>>> s = s[:7] + "a" + s[8:]
>>> s
'Manhattan'
```

The minimum edit distance between the two strings "Mannhaton" and "Manhattan" corresponds to the value 3, as we need three basic editing operation to transform the first one into the second one:

```
>>> s = "Mannhaton"
>>> s = s[:2] + s[3:]          # deletion
>>> s
'Manhaton'
```

```
>>> s = s[:5] + "t" + s[5:]   # insertion
>>> s
'Manhatton'
>>> s = s[:7] + "a" + s[8:]   # substitution
>>> s
'Manhattan'
```

We can assign assign a weight or costs to each of these edit operations, e.g. setting each of them to 1. It is also possible to argue that substitutions should be more expensive than insertations or deletions, so sometimes the costs for substitutions are set to 2.

MATHEMATICAL DEFINITION OF THE LEVENSHTEIN DISTANCE

The Levenshtein distance between two strings a and b is given by $lev_{a,b}(len(a), len(b))$ where $lev_{a,b}(i, j)$ is equal to

- max(i, j) if min(i, j)=0
- otherwise:

$$\min(lev_{a,b}(i-1, j) + 1,\ lev_{a,b}(i, j-1) + 1,\ lev_{a,b}(i-1, j-1) + 1_{a_i \neq b_j})$$

where $1_{a_i \neq b_j}$</sub> is the indicator function equal to 0 when $a_i=b_j$ and equal to 1 otherwise, and $lev_{a,b}(i, j)$ is the distance between the first i characters of a and the first j characters of b.

The Levenshtein distance has the following properties:

- It is zero if and only if the strings are equal.
- It is at least the difference of the sizes of the two strings.
- It is at most the length of the longer string.
- Triangle inequality: The Levenshtein distance between two strings is no greater than the sum of their Levenshtein distances from a third string.

RECURSIVE LEVENSHTEIN FUNCTION IN PYTHON

The following Python function implements the Levenshtein distance in a recursive way:

```
3
```

This recursive implementation is very inefficient because it recomputes the Levenshtein distance of the same substrings over and over again. We count the number of calls in the following version by using a decorator function. If you don't know them, you can learn about them in our chapter on Memoization and Decorators:

```
3
LD was called 29737 times!
[("('', 'P'){}", 5336), ("('P', ''){}", 4942), ("
('', ''){}", 3653), ("('P', 'P'){}", 3653), ("('',
'Pe'){}", 2364), ("('P', 'Pe'){}", 1683), ("('Py',
''){}", 1666), ("('Py', 'P'){}", 1289), ("('', 'Pe
i'){}", 912), ("('Py', 'Pe'){}", 681), ("('P', 'Pe
i'){}", 681), ("('Pyt', ''){}", 462), ("('Pyt', 'P')
{}", 377), ("('Py', 'Pei'){}", 321), ("('', 'Peit')
{}", 292), ("('Pyt', 'Pe'){}", 231), ("('P', 'Peit')
{}", 231), ("('Py', 'Peit'){}", 129), ("('Pyt', 'Pe
i'){}", 129), ("('Pyth', ''){}", 98), ("('Pyth',
'P'){}", 85), ("('', 'Peith'){}", 72), ("('Pyt', 'Pe
it'){}", 63), ("('Pyth', 'Pe'){}", 61), ("('P', 'Pei
th'){}", 61), ("('Py', 'Peith'){}", 41), ("('Pyth',
'Pei'){}", 41), ("('Pyth', 'Peit'){}", 25), ("('Py
t', 'Peith'){}", 25), ("('Pytho', ''){}", 14), ("('P
yth', 'Peith'){}", 13), ("('Pytho', 'P'){}", 13), ("
('', 'Peithe'){}", 12), ("('Pytho', 'Pe'){}", 11),
("('P', 'Peithe'){}", 11), ("('Py', 'Peithe'){}",
9), ("('Pytho', 'Pei'){}", 9), ("('Pyt', 'Peithe')
{}", 7), ("('Pytho', 'Peit'){}", 7), ("('Pyth', 'Pei
the'){}", 5), ("('Pytho', 'Peith'){}", 5), ("('Pyth
o', 'Peithe'){}", 3), ("('Python', 'Pei'){}", 1), ("
('Python', 'Peithe'){}", 1), ("('', 'Peithen'){}",
1), ("('P', 'Peithen'){}", 1), ("('Pytho', 'Peithe
n'){}", 1), ("('Py', 'Peithen'){}", 1), ("('Python',
'P'){}", 1), ("('Python', 'Peit'){}", 1), ("('Pyt',
'Peithen'){}", 1), ("('Pyth', 'Peithen'){}", 1), ("
('Python', 'Peith'){}", 1), ("('Python', ''){}", 1),
("('Python', 'Pe'){}", 1), ("('Python', 'Peithen')
{}", 1)]
```

We can see that this recursive function is highly inefficient. The Levenshtein distance of the string s="" and t="P" was calculated 5336 times. In the following version we add some "memory" to our recursive Levenshtein function by adding a dictionary memo:

```
3
The function was called 49 times!
```

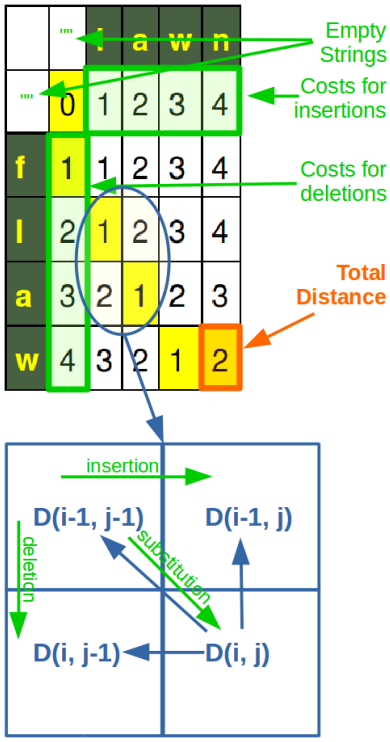
The previous recursive version is now efficient, but it has a design flaw in it. We polluted the code with the statements to update our dictionary mem. Of course, the design is a lot better, if we do not pollute our code by adding the logic for saving the values into our Levenshtein function. We can also "outsource" this code into a decorator. The following version uses a decorator "memoize" to save these values:

```
3
The function was called 127 times!
```

The additional calls come from the fact that we have three unconditional calls as arguments of the function "min".

ITERATIVE COMPUTATION OF THE LEVENSHTEIN DISTANCE

To compute the Levenshtein distance in a non-recursive way, we use a matrix containing the Levenshtein distances between all prefixes of the first string and all prefixes of the second one. We can dynamically compute the values in this matrix. The last value computed will be the distance between the two full strings. This is an algorithmic example of a bottom-up dynamic programming.



The algorithm works like this:
We set the cost for an insertion, a deletion and a substitution to 1. We want to calculate the distance between two string s and t with len(s) == m and len(t) == n. A matrix D is used, which contains in the (i,j)-cell the Levenshtein distance between s[:i+1] and t[:j+1]. The values of the matrix will be calculated starting with the upper left corner and ending with the lower right corner.

We start with filling in the base cases, i.e. the row and the column with the index 0. Calculation in this case means that we fill the row with index 0 with the lenghts of the substrings of t and respectively fill the column with the index 0 with the lengths of the substrings of s.

The values of all the other elements of the matrix only depend on the values of their left neighbour, the top neightbour and the top left one.

The calculation of the D(i,j) for both i and j greater 0 works like this: D(i,j) means that we are calculating the Levenshtein distance of the substrings s[0:i-1] and t[0:j-1]. If the last characters of these substrings are equal, the edit distance corresponds to the distance of the substrings s[0:-1] and t[0:-1], which may be empty, if s or t consists of only one character, which means that we will use the values from the 0th column or row. If the last characters of s[0:i-1] and t[0:j-1] are not equal, the edit distance D[i,j] will be set to the sum of 1 + min(D[i, j-1], D[i-1, j], D[i-1, j-1])-

We illustrate this in the following diagram:

```
[0, 1, 2, 3, 4]
[1, 1, 2, 3, 4]
[2, 1, 2, 3, 4]
[3, 2, 1, 2, 3]
[4, 3, 2, 1, 2]
2
```

The following picture of the matrix of our previous calculation contains - coloured in yellow - the optimal path through the matrix. We start with a deletion ("f"), we keep the "l" (no costs added), after this we keep the "a" and "w". The last step is an insertion, raising the costs to 2, which is the final Levenstein distance.

For the sake of another example, let us use the Levenshtein distance for our initial example of this chapter. So, we will virtually "go back" to New York City and its thrilling borough Manhattan. We compare it with a misspelling "Manahaton", which is the combination of various common misspellings.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
[2, 1, 0, 1, 2, 3, 4, 5, 6, 7]
[3, 2, 1, 0, 1, 2, 3, 4, 5, 6]
[4, 3, 2, 1, 1, 1, 2, 3, 4, 5]
[5, 4, 3, 2, 1, 2, 1, 2, 3, 4]
[6, 5, 4, 3, 2, 2, 2, 1, 2, 3]
[7, 6, 5, 4, 3, 3, 3, 2, 2, 3]
[8, 7, 6, 5, 4, 4, 3, 3, 3, 3]
[9, 8, 7, 6, 5, 5, 4, 4, 4, 3]
3
```

		M	a	n	a	h	a	t	o	n
	0	1	2	3	4	5	6	7	8	9
M	1	0	1	2	3	4	5	6	7	8
a	2	1	0	1	2	3	4	5	6	7
n	3	2	1	0	1	2	3	4	5	6
h	4	3	2	1	1	1	2	3	4	5
a	5	4	3	2	1	2	1	2	3	4
t	6	5	4	3	2	2	2	1	2	3
t	7	6	5	4	3	3	3	2	2	3
a	8	7	6	5	4	4	3	3	3	3
n	9	8	7	6	5	5	4	4	4	3

So far we have had fixed costs for insertions, deletions and substitutions, i.e. each of them was set to 1.

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

The situation in the call to iterative_levenshtein with default costs, i.e. 1 for insertions, deletions and substitutions:

		x	y	z
	0	1	2	3
a	1	1	2	3
b	2	2	2	3
c	3	3	3	3

The content of the matrix if we the substitutions are twice as expensive as the insertions and deletions, i.e. the call iterative_levenshtein("abc", "xyz", costs=(1, 1, 2)):

		x	y	z
	0	1	2	3
a	1	2	3	4
b	2	3	4	5
c	3	4	5	6

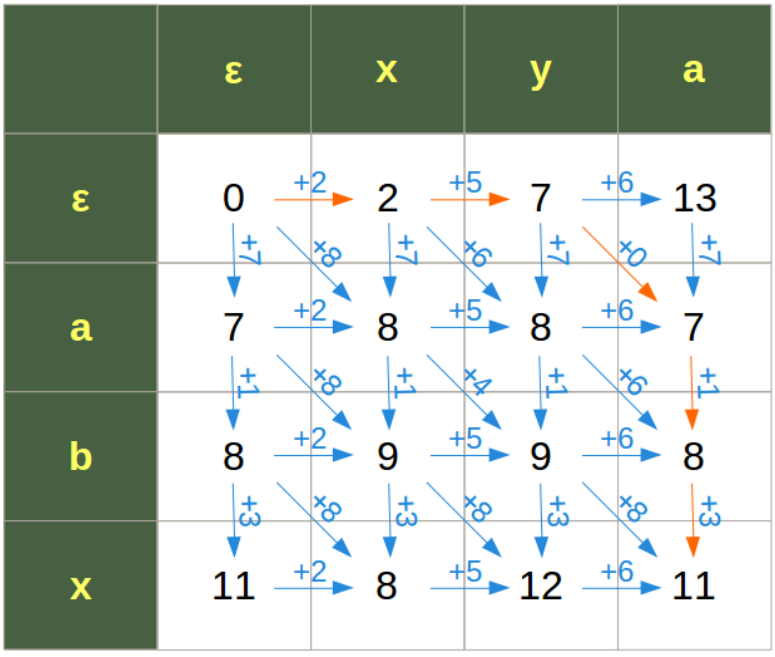
Now we call iterative_levenshtein("abc", "xyz", costs=(2, 2, 1)), which means that a substitution is half as expension as an insertion or a deletion:

		x	y	z
	0	2	4	6
a	2	1	3	5
b	4	3	2	4
c	6	5	4	3

It is also possible to have indivual weights for each character. Instead of passing a tuple with three values to the function, we will use a dictionary with values for every character.

```
subs:  8 deletes:  7 inserts:  2 a x
subs:  8 deletes:  1 inserts:  2 b x
subs:  0 deletes:  3 inserts:  2 x x
subs:  6 deletes:  7 inserts:  5 a y
subs:  4 deletes:  1 inserts:  5 b y
subs:  8 deletes:  3 inserts:  5 x y
subs:  0 deletes:  7 inserts:  6 a a
subs:  6 deletes:  1 inserts:  6 b a
subs:  8 deletes:  3 inserts:  6 x a
[0, 2, 7, 13]
[7, 8, 8, 7]
[8, 9, 9, 8]
[11, 8, 12, 11]
11
```

We demonstrate in the following diagram how the algorithm works with the weighted characters. The orange arrows show the path to the minimal edit distance 11:



FOOTNOTES

[1] Measurement of Similarity in Music: A Quantitative Approach for Non-parametric Representations

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: $2^n - 1$

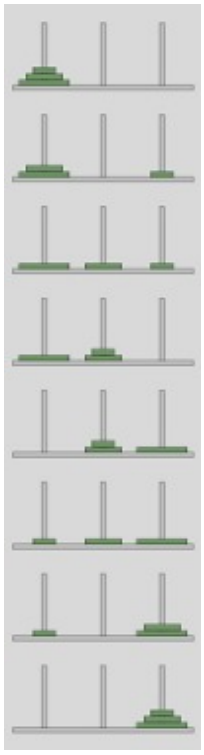
PLAYING AROUND TO FIND A SOLUTION

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!

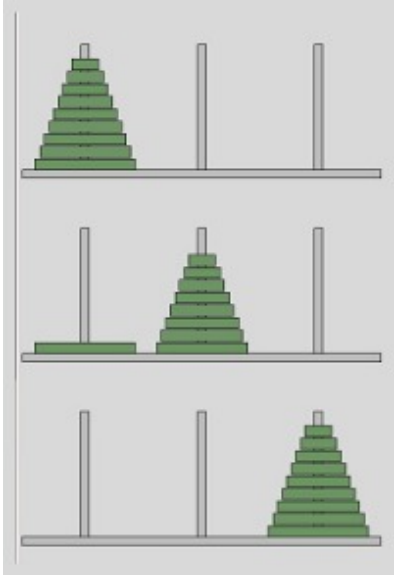
We have seen in the cases n=1 and n=2 that it depends on the first move, if we will be able to successfully and with the minimal number of moves solve the riddle. We know from our formula that the minimal number of moves necessary to move a tower of size 3 from the SOURCE peg to the target peg is 7 ($2^3 - 1$)
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

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

print source, helper, target
```

This function is implementing, 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:

```
    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 " + sour
    ce[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
```

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

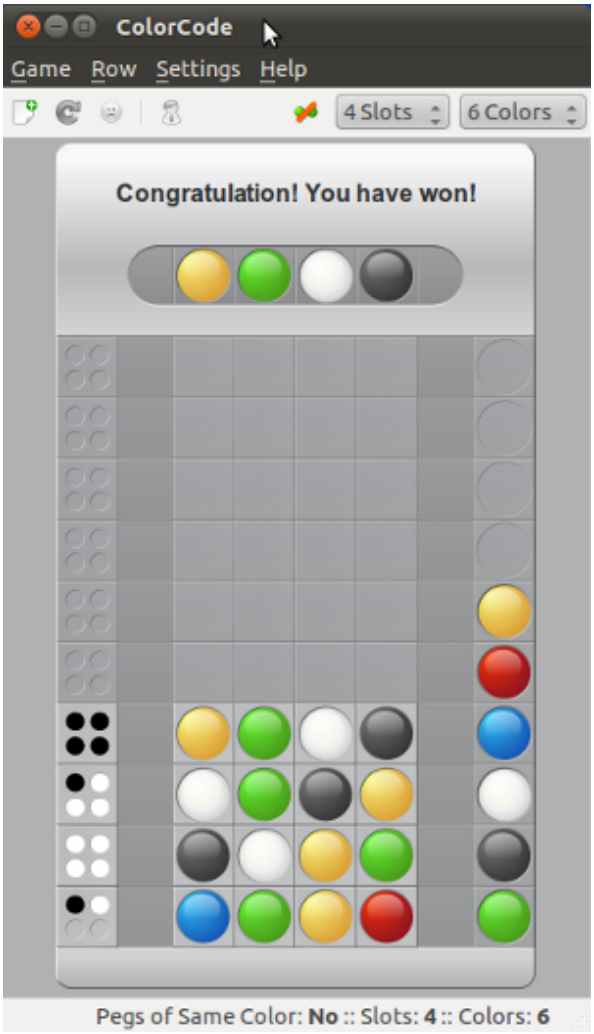
Bulls and Cows

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:

```
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.

```
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:

```
colours = ["Paris blue", "London green", "Berlin black",
"Vienna yellow", "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:

```
permutation_iterator = all_colours(colours, number_of_pos
itions)
current_colour_choices = next(permutation_iterator)
```

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,
permutated))
```

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_positi
ons) \
        or (rightly_positioned + permutated < len(colours)
- number_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 sens
e")
        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_posit
ioned, permutated)))
    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, pe
rmutated))
```

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 li
st of
colours like p = ['pink', 'yellow', 'green', 'red'] is co
nsistent
with the previous colours. Each previous colour permuatio
n guess[0]
compared (check()) with p has to return the same amount o
f blacks
(rightly positioned colours) and whites (right colour at
wrong
position) as the corresponding evaluation (guess[1] in th
e
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 playe
r makes
sense. 3 blacks and 1 white make no sense for example.
"""
   (rightly_positioned, permutated) = a
   if (rightly_positioned + permutated > number_of_positi
ons) \
       or (rightly_positioned + permutated < len(colours)
- number_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 gue
ss, checks
the consistency of this evaluation, adds the guess togeth
er with
```

```
the evaluation to the list of guesses, shows the previous
guesses
and creates a new guess """
   rightly_positioned, permutated = get_evaluation()
   if rightly_positioned == number_of_positions:
      return(current_colour_choices, (rightly_positioned,
permutated))

   if not answer_ok((rightly_positioned, permutated)):
      print("Input Error: Sorry, the input makes no sens
e")
      return(current_colour_choices, (-1, permutated))
   guesses.append((current_colour_choices, (rightly_posit
ioned, permutated)))
   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, pe
rmutated))


def check(p1, p2):
   """ check() calculates the number of bulls (blacks) an
d 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

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 eva
luations
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()

if __name__ == "__main__":
    colours = ["red","green","blue","yellow","orange","pin
k"]
    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] != n
umber_of_positions):
        new_guess = new_evaluation(new_guess[0])
```

GUI FOR MASTERMIND

You can find an implementation of the previous program with a graphical user interface using Tkinter in our Tkinter Tutorial.

Footnotes:

[1] The colour guesses don't have to be lists, we could have used tuples to represent the colours guesses as well.

CREATING DYNAMIC WEBSITES WITH PYTHON WITH MOD_PYTHON AND WSGI

INTRODUCTION

Please notice:
Work on this topic is under process. (August 2014)

mod_python is an Apache HTTP Server module. It's purpose is to integrate Python programming with the Apache web server, or in other words a Python language binding for the Apache HTTP Server. The official website of mod_python says that it possible to write "with mod_python web-based applications in Python that run many times faster than traditional CGI and will have access to advanced features such as ability to retain database connections and other data between hits and access to Apache internals." mod_python has been pronounced dead some years ago. So it didn't look to be a good idea to use it for new projects. It never died, it was only "sleeping". It came to life again in 2013!

PYTHON AND MOD_PYTHON

If we want to use Python on am Apache web server, we need the mod_python module for Apache. This module provides a Python language binding so that we can integrate Python. It's a more efficient approach than using CGI, because CGI will start a new Python process for every request.

Mod_python consists of two components: The dynamically loadable module mod_python.so for Apache and the Python package mod_python. If you are using Debian or Ubuntu Linux, it's satisfying to install the package libapache2-mod-python for this purpose, assuming apache2 is already installed:

```
sudo apt-get install libapache2-mod-python
```

If apache2 has to be installed as well, do the following installation first:

```
sudo apt-get install apache2
```

You have to add the following lines into /etc/apache2/sites-enabled/000-default:

```
AddHandler mod_python .py
     PythonHandler mod_python.publisher
     PythonDebug On
```

It may look like this: ServerAdmin webmaster@localhost DocumentRoot /var/www/html AddHandler mod_python .py PythonHandler mod_python.publisher PythonDebug On Now we have to restart the Apache server:

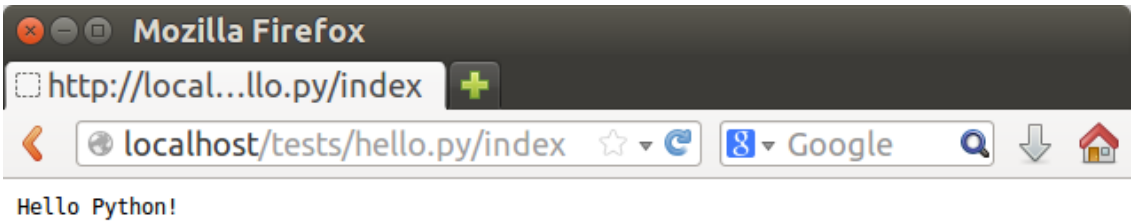
```
sudo /etc/init.d/apache2 restart
```

A SIMPLE DYNAMIC PAGE WITH MOD_PYTHON

We will create a subdirectory "tests" in the documents root of the Apache server. In case of Debian and Ubuntu, this will be /var/www/html/. We save the following Python program as "hello.py" in the previously created subdirectory:

```
def index():
    return "Hello Python!"
```

We have to start a browser and go to the location "localhost/tests/hello.py/index". It works with "localhost/tests/hello.py" as well. We get the following output in the browser window:



ANOTHER MORE "USEFUL" WEBPAGE

We save the following website as timesite.py. It will print out the current date and time, as well as the timezone:

```
import time
def index():
        html = """
<html><head>
<title>mod_python.publisher first html page</title>
</head>
<body>
```

```
<h1>This page was generated by mod_python.publisher</h1><
hr>
The local time of this server is:  %s
<br>The timezone of this server is  :  %s
</body>
</html>
""" % (time.ctime(time.time()), time.timezone/3600)
        return html
</pre>
```

We get the following output:

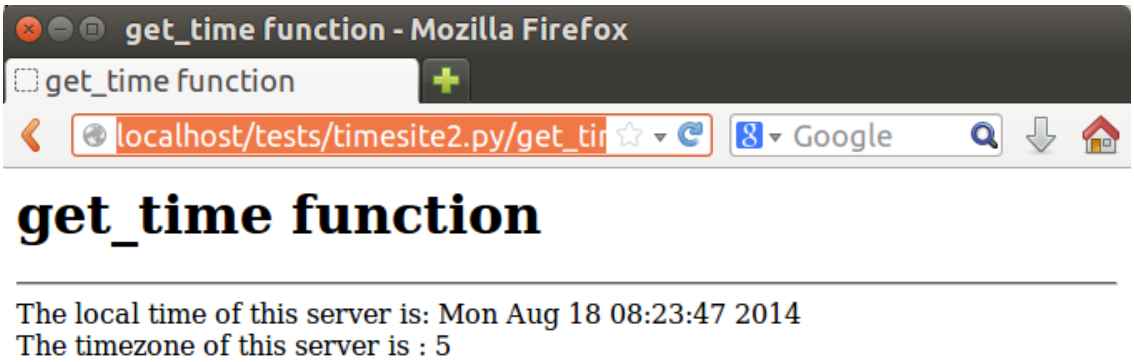
ANOTHER PAGE NAME

So far we used index as the default website name. We can also define other functions and by doing so create websites with other names. We write the get_time function in following example and modify the index function:

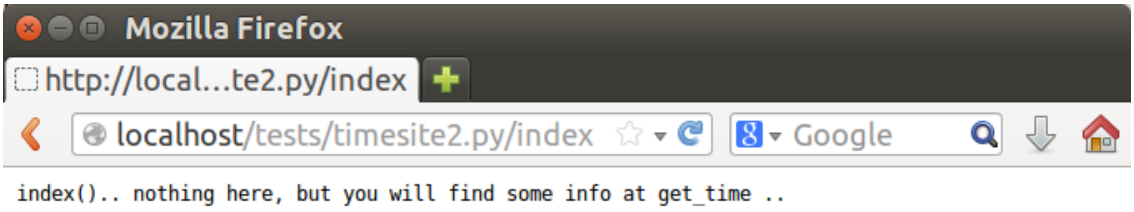
```
import time
def index():
        return "index().. nothing here, but you will find
some info at get_time .."
def get_time():
```

```
        html = """
<html><head>
<title>get_time function</title>
</head>
<body>
<h1>get_time function</h1>
<hr>
The local time of this server is:  %s <br>
The timezone of this server is  :  %s <br>
</body>
</html>""" % (time.ctime(time.time()), time.timezone/360
0)
        return html
```

Calling the location "http://localhost/tests/timesite2.py/get_time", returns the following output:



Using the address "http://localhost/tests/timesite2.py/index", supplies this:



USING FORMS

HTML forms are used to pass data to a server. We can do this with mod_python as well. The following html form inside our Python program "form.py" contains fields for the first name, last name email address and radio buttons for the gender:

```
def index():

        return """
<html><head>
<title>Formular</title>
</head>
<body>
<FORM value="form" action="get_info" method="post">
  <P>
        <LABEL for="firstname">First Name: </LABEL>
        <INPUT type="text" name="firstname"><BR>
        <LABEL for="lastname">Last Name: </LABEL>
        <INPUT type="text" name="lastname"><BR>
        <LABEL for="email">email: </LABEL>
        <INPUT type="text" name="email"><BR>
        <INPUT type="radio" name="gender" value="Male">Ma
le<BR>
        <INPUT type="radio" name="gender" value="Female">
Female<BR>
        <INPUT type="submit" value="Send"> <INPUT type="r
eset">
  </P>
</FORM>
```

```
</body>
</html>
"""

def get_info(req):
        info = req.form
        first = info['firstname']
        last = info['lastname']
        email = info['email']
        gender = info['gender']
        return """
<html><head>
<title>POST method using mod_python</title>
</head>
<body>
<h1>POST Method using mod_python</h1>
<hr>
Thanks for using our service:<br>
Your first name: %s <br>
Your last name: %s <br>
Your email address: %s <br>
Your gender: %s <br>
</body>
</html>
""" %(first, last.upper(), email, gender.lower())
```

Calling the above program with the URL "http://localhost/tests/form.py/index" gives us the following entry form:

To see the result page, i.e. the result of the function get_info, we have to push the "send" button:

WSGI

WHAT IS WSGI?

WSGI is the Web Server Gateway Interface. It is a specification that describes how web servers communicate with web applications. It is a framework for the Python. It was originally specified in 2003. WSGI has become a standard for Python web application development. WSGI has been specified in PEP 3333. The abstract says "This document specifies a proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers."

CREATING DYNAMIC WEBSITES WITH PYTHON AND WSGI

INTRODUCTION

Please notice: Work on this topic is under process. (August 2014)

WSGI is the Web Server Gateway Interface. It is a specification that describes how web servers communicate with web applications. It is a framework for the Python. It was originally specified in 2003. WSGI has become a standard for Python web application development. WSGI has been specified in PEP 3333. The abstract says "This document specifies a proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers."

SIMPLE EXAMPLE WITH WSGI

We demonstrate the way of working with a simple example, actually as simple as simple can be. All it does will be to greet a visitor of the website with "Hello my friend!", we just had enough of "Hello World".

```
from wsgiref.simple_server import make_server

def application(environ, start_response):
    start_response("200 OK", [("Content-type", "text/plai
n")])
    return ["Hello my friend!".encode("utf-8")]


server = make_server('localhost', 8080, application)
server.serve_forever()
```

You can save this program wherever you want on your computer. If you start it with "python3 greeting.py", you can visit the URL "localhost:8080" with a browser of your choice. You should

see now: "Hello my friend!" This text will appear within your browser. On the terminal, you will the an output, which looks similar to this:

```
$ python3 hello_wsgi.py
127.0.0.1 - - [19/Aug/2014 10:31:39] "GET / HTTP/1.1" 200
16
```

You may have noticed that we used Python 3 to start the server. The program runs with Python 2 as well. There are lots of similar examples out there on the web which don't work for Python 3. The reason is in many cases, that they don't return a bytes string, but a "simple" str class instance. So, if you drop the method call '.encode("utf-8")', the program will only run with Pyhton2 but not with Python3 anymore.

We used the simple reference implementation wsgiref of WSGI, which is included in Python's standard library. It's easier to for testing purposes. The make_server method takes five parameters:

- host: the host name can be 'localhost' or any other host name, like your server name "server = make_server('saturn', 8080, application)" or a domain or IP address, e.g. server = make_server('192.168.170.128', 8080, application), which makes it possible to access this web server from another computer in the network.
- port: We use 8080 as the port number
- app: This has to be a reference to a function, which is returning a list with the results. Every element of this list has to be a bytes string.
- We will not discuss the optional keyword parameters "server_class=, handler_class"

The function "app" - used as the third parameter of make_server - needs two parameters:

- environ
- start_response: start_response has to be a callable with three parameters: status, response_headers, exc_info=None
  status contains the numeric HTTP status code of the response, e.g. "200 OK", "404 NOT FOUND", or "500 SERVER ERROR". response_headers contains the HTTP message for the status code used. exc_info used for traceback information is optional.

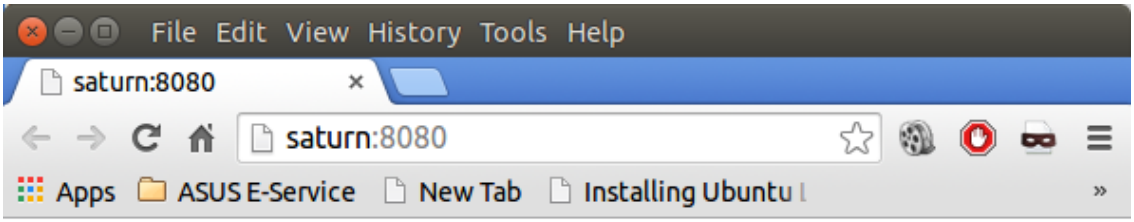
ANOTHER EXAMPLE

The following program nothing new. It's just the previous example, which has to be extended so that the function application returns the first 30 lines of text from the novel Ulysses by James

Joyce:

```
from wsgiref.simple_server import make_server

def application(environ, start_response):
    start_response("200 OK", [("Content-type", "text/plai
n")])

    fh = open("ulysses.txt")
    lines = [fh.readline().encode("utf-8") for i in range
(30)]

    return lines


server = make_server('saturn', 8080, application)
server.serve_forever()
```

The browser output looks like this:

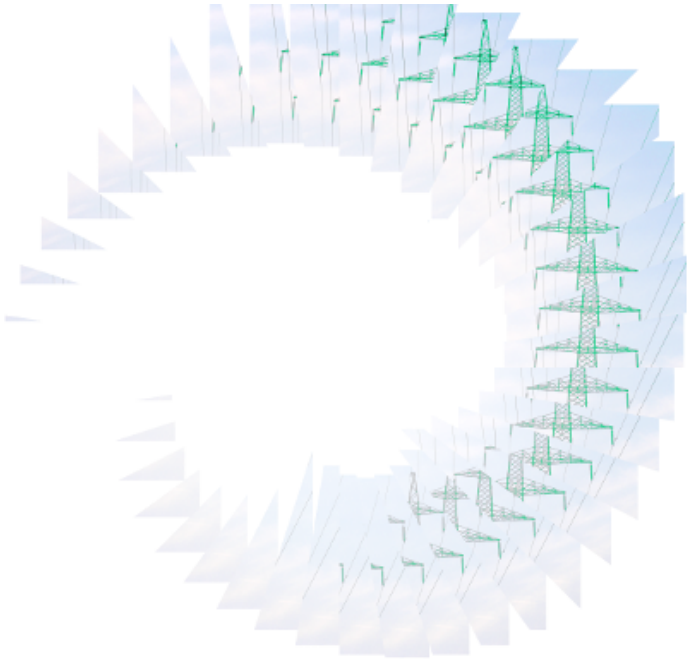



```
ULYSSES

by James Joyce



-- I --

Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of
lather on which a mirror and a razor lay crossed. A yellow dressinggown,
ungirdled, was sustained gently behind him on the mild morning air. He
held the bowl aloft and intoned:

--_Introibo ad altare Dei_.

Halted, he peered down the dark winding stairs and called out coarsely:

--Come up, Kinch! Come up, you fearful jesuit!

Solemnly he came forward and mounted the round gunrest. He faced about
and blessed gravely thrice the tower, the surrounding land and the
awaking mountains. Then, catching sight of Stephen Dedalus, he bent
towards him and made rapid crosses in the air, gurgling in his throat
and shaking his head. Stephen Dedalus, displeased and sleepy, leaned
his arms on the top of the staircase and looked coldly at the shaking
gurgling face that blessed him, equine in its length, and at the light
untonsured hair, grained and hued like pale oak.

Buck Mulligan peeped an instant under the mirror and then covered the
```

CREATING DYNAMIC WEBSITES WITH PYTHON AND PYLONS

INTRODUCTION

Please notice:
Work on this topic is under process.
(August 2014)



The picture on the right side is misleading: We are not talking about this kind of Pylons, i.e. electricity pylons, or transmission towers, used to support an overhead power line.

But their is some interesting aspect in this comparison. It is a set of web application frameworks written in Python. So it can be seen as the steel lattice tower of an electricity pylon, but it is not supporting power lines but web pages. James Gardner, a co-founder of Pylons, defined this framework as "*Pylons is a lightweight web framework emphasizing flexibility and rapid development using standard tools from the Python community.*"

We will demonstrate in simple examples - starting with the inevitable "Hello World" project - how to use it. We will also introduce the usage of Macros.

INSTALLATION

Before you can use Pylons, you have to install it. If you use Debian, Ubuntu or Mint, all you have to do is execute the following code on a command shell:

```
apt-get install python-pylons
```

You are able to create your first Pylons project now:

```
$ paster create --template=pylons MyProject
```

You will be asked two questions and you can answer by keeping the default:

```
Enter template_engine (mako/genshi/jinja2/etc: Template l
anguage) ['mako']:
Enter sqlalchemy (True/False: Include SQLAlchemy configur
ation) [False]:
```

We have created now a directory MyProject, which contains the following files and subdirectories:

- development.ini
- ez_setup.py
- MANIFEST.in
- myproject
- MyProject.egg-info
- README.txt
- setup.cfg
- setup.py
- test.ini

The content of the subdirectory myproject looks like this:

- config
  contains the files
- controllers
- __init__.py
- __init__.pyc
- lib
- model
- public
- templates
- tests
- websetup.py

```
cd MyProject
paster serve --reload development.ini
```

Now the web server is running. If you visit the URL http://localhost:5000/ with a browser of your choice, you will see the following welcome screen:

It's generated from ./myproject/public/index.html inside of your MyProject directory.

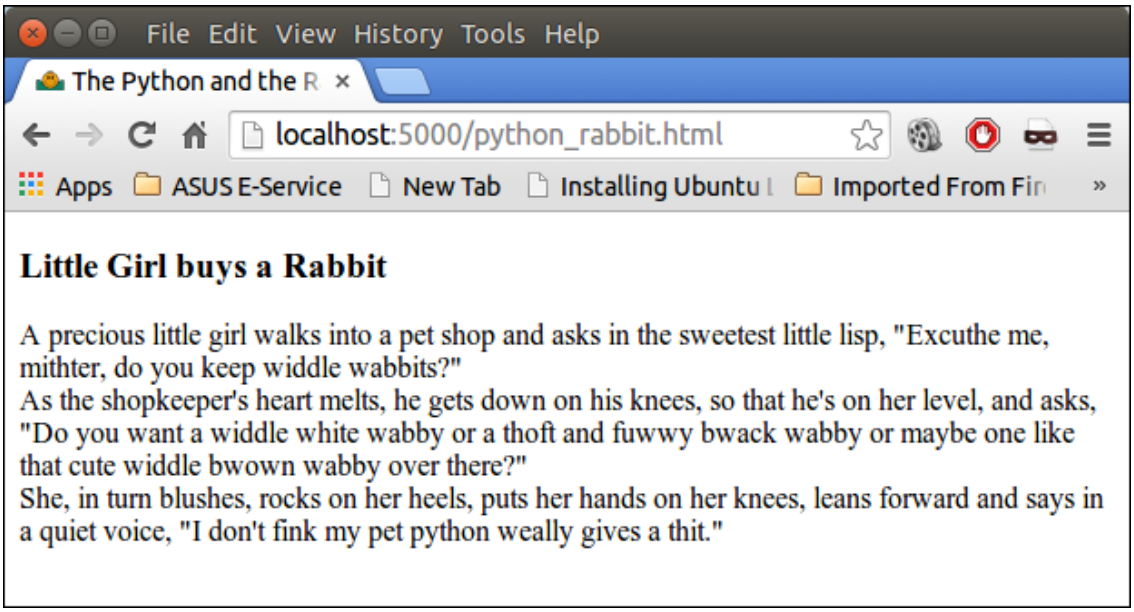
STATIC PAGES

You can put other html files into the directory ./myproject/public/. These files can be visited by the web browser as well. You can save the following html code as python_rabbit.html:

```
<html>
<head>
<title>The Python and the Rabbit</title>
</head>
<body>
<h3>Little Girl buys a Rabbit</h3>
A precious little girl walks into a pet shop and asks in
the
sweetest little lisp, "Excuthe me, mithter, do you keep w
iddle wabbits?"
<br>
```

```
As the shopkeeper's heart melts, he gets down on his knee
s, so that he's
on her level, and asks, "Do you want a widdle white wabby
or a thoft and
fuwwy bwack wabby or maybe one like that cute widdle bwow
n wabby over
there?"
<br>
She, in turn blushes, rocks on her heels, puts her hands
on her knees,
leans forward and says in a quiet voice, "I don't fink my
pet python
weally gives a thit."
</body>
</html>
```

You can visit the URL "http://localhost:5000/python_rabbit.html" and the html file above will be rendered:

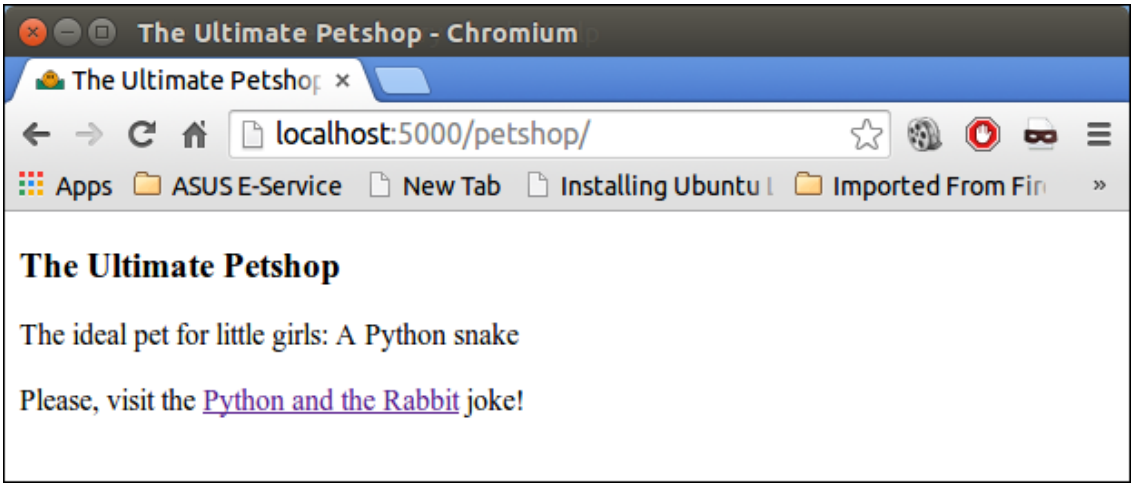


Any files in the directory "public" are treated as static files. It may contain subdirectories as well, which are part of the URL in the usual way. If we create a subdirectory "petshop" in "public", we will receive a "404 Not Found" error, unless we create an index.html file as well and place it inside of the subdirectory, in our case "petshop". So, there is no directory index default view, as we can have it with Apache. This is for security reasons.

We will create now the subdirectory "petshop" and inlcude the following index.html file:

```
<html>
<head>
<title>The Ultimate Petshop</title>
</head>
<body>
<h3>The Ultimate Petshop</h3>
The ideal pet for little girls: A Python snake
<br><br>
Please, visit the <a href="../python_rabbit.html">Python
and the Rabbit</a> joke!

</body>
</html>
```

Visiting "http://localhost:5000/petshop/" or "http://localhost:5000/petshop/index.html", we will encounter the following content:

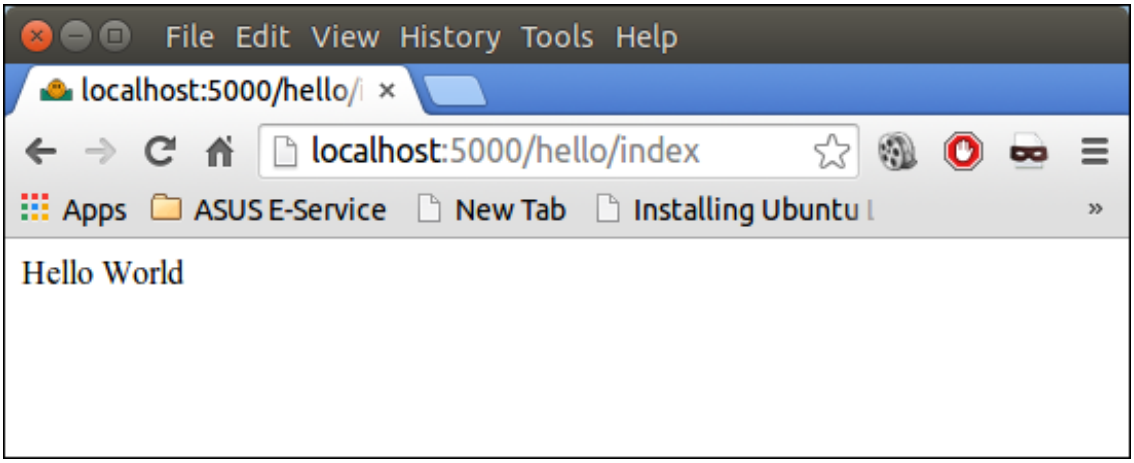


GETTING DYNAMIC WITH CONTROLLERS

We will create now a dynamic "Hello World" application by creating a controller in our project. The controller is able to handle requests. We can create the controller with the following command:

```
$ paster controller hello
```

If we visit the URL http://localhost:5000/hello/index we will get a website with the content "Hello World":



We want to change our application into a multilingual website, i.e. visitors should be greeted in their languages. To this purpose we will have to change the file hello.py, which can be found in MyProject/myproject/controllers/:

```
import logging

from pylons import request, response, session, tmpl_conte
xt as c, url
from pylons.controllers.util import abort, redirect

from myproject.lib.base import BaseController, render

log = logging.getLogger(__name__)

class HelloController(BaseController):

    def index(self):
        # Return a rendered template
        #return render('/hello.mako')
        # or, return a string
        return 'Hello World'
```

We will change the index method of the HelloController class:

```
    def index(self, id=None):
        # Return a rendered template
        #return render('/hello.mako')
        # or, return a string
        if id == "fr":
            return 'Bonjour Monde'
        if id == "de":
            return 'Hallo Welt'
        if id == "it":
            return 'Ciao mondo'
        return 'Hello World'
```

Calling the URL "http://localhost:5000/hello/index" will still return "Hello World", but adding "/de" - i.e. http://localhost:5000/hello/index/de - to the previous URL will return "Hallo Welt".

MAKO TEMPLATES

You may have noticed the commented line "#return render('/hello.mako')". If we uncomment this line, the server will use a mako template "hello.mako" or better it will try to use a template. This template is supposed to be located in the directory "myproject/templates/". So far this directory is empty. Uncommenting the line will create the following error message in your browser window:

```
WebError Traceback:
-> TopLevelLookupException: Cant locate template for uri
'/hello.mako'
```

A Mako is a template library - also called a template engine - written in Python. It is the default template language included with the Pylons and Pyramid web frameworks. Mako's syntax and API borrows from other approaches, e.g. Django, Jinja2, Myghty, and Genshi. Mako is an embedded Python language, used for Python server pages.

A Mako template contains various kind of content, for example, XML, HTML, email text, and so on.

A template can contain directives which represent variable or expression substitutions, control structures, or blocks of Python code. It may also contain tags that offer additional functionality. A Mako is compiled into real Python code.

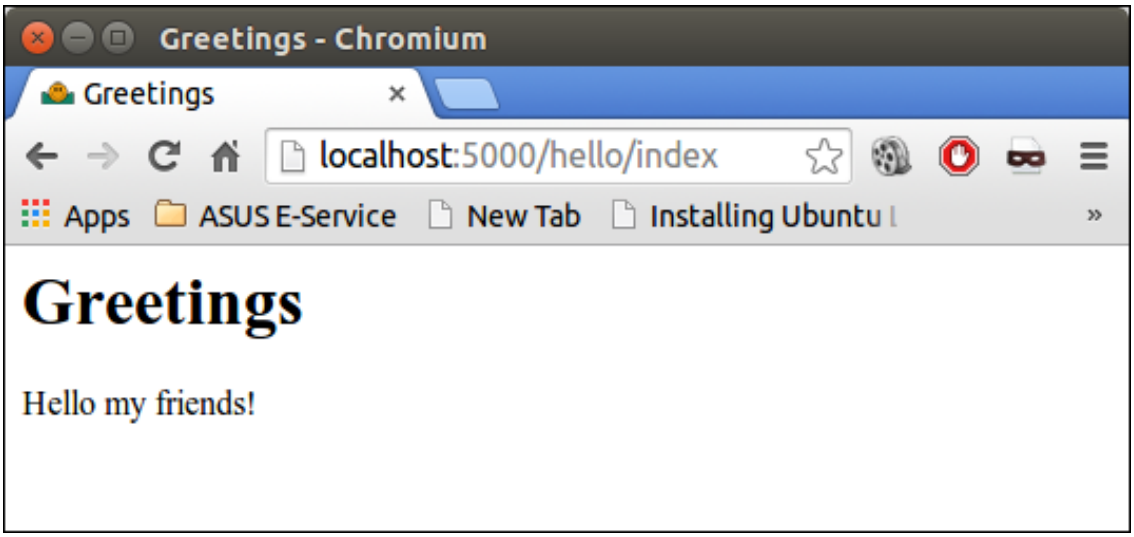
The easiest possible Mako template consists solely of html code. The following Mako template is statically greeting all friends:

```
<html>
<head>
  <title>Greetings</title>
</head>
<body>
  <h1>Greetings</h1>
  <p>Hello my friends!</p>
</body>
</html>
```

We change our hello.py in the directory "controllers" to the following code:

```
from myproject.lib.base import BaseController, render

class HelloController(BaseController):

    def index(self, id=None):
        # Return a rendered template
        return render('/hello.mako')
```

We have to save it in the templates subdirectory. The new page looks like this on a chromium browser:



This introduction is about dynamic web pages with Pylons, so we have to personalize our greeting. At first, we want to demonstrate how to use Python variables inside of a template. To this purpose, the tmpl_context is available in the pylons module, and refers to the template context. Objects attached to it are available in the template namespace as tmpl_context. In most cases tmpl_context is aliased as c in controllers and templates for convenience. Our hello.py needs to import tmpl_context and we set c.name to "Frank":

```
from pylons import tmpl_context as c
from myproject.lib.base import BaseController, render


class HelloController(BaseController):

    def index(self, id=None):
        c.name = "Frank"
        # Return a rendered template
        return render('/hello.mako')
```

The output in a broser hasn't changed dramatically. Just "Hello Frank" instead of "Hello my friends!". Admittedly, still not very dynamical!

But we can use the URL with our id parameter again:

```
from pylons import tmpl_context as c
from myproject.lib.base import BaseController, render


class HelloController(BaseController):

    def index(self, id=None):
        c.name = id if id else "my friends"
        # Return a rendered template
        return render('/hello.mako')
```

Visiting http://localhost:5000/hello/index/Frank renders the output "Hello Frank!", whereas the URL http://localhost:5000/hello/index will produce "Hello my friends!"

MAPPING THE ROOT URL

We have seen that we can put various static html files under the root directory "myproject/public/index.html" of our project. We saw that index.html of the public directory is

the own which is shown, if we point the browser to http://localhost:5000/ You may want to have a dynamic page right at the root. So, we have to map the root URL http://localhost:5000/ to a controller action. This can be done by modifying the routing.py file in the config directory. If you look at this file, you will find a comment line "`# CUSTOM ROUTES HERE`". You have to add the following line after this line:

```
# CUSTOM ROUTES HERE

map.connect('/', controller='hello', action='index')
```

There is one more thing you shouldn't forget: You have to remove the index.html file in public. Otherwise this file will be still served.

PYTHON AND SQL

INTRODUCTION

The history of SQL goes back to the early 70th. SQL is a Structured Query Language, which is based on a relational model, as it was described in Edgar F. Codds's 1970 paper "A Relational Model of Data for Large Shared Data Banks. SQL is often pronounced like "sequel". SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987. As most people coming to this website are already familiar with mSQL, PostgresSQL, MySQL or other variants of SQL, we will not enlarge on SQL itself.

A database is an organized collection of data. The data are typically organized to model aspects of reality in a way that supports processes requiring this information. The term "database" can both refer to the data themselves or to the database management system. The Database management system is a software application for the interaction between users database itself. Users don't have to be human users. They can be other programs and applications as well. We will learn how Python or better a Python program can interact as a user of an SQL database.

This is an introduction into using SQLite and MySQL from Python. The Python standard for database interfaces is the Python DB-API, which is used by Python's database interfaces. The DB-API has been defined as a common interface, which can be used to access relational databases. In other words, the code in Python for communicating with a database should be the same, regardless of the database and the database module used. Even though we use lots of SQL examples, this is not an introduction into SQL but a tutorial on the Python interface. To learn SQL you have to consult a SQL tutorial.

SQLITE

SQLite is a simple relational database system, which saves its data in regular data files or even in the internal memory of the computer, i.e. the RAM. It was developped for embedded applications, like Mozilla-Firefox (Bookmarks), Symbian OS or Android. SQLITE is "quite" fast, even though it uses a simple file. It can be used for large databases as well. If you want to use SQLite, you have to import the module sqlite3. To use a database, you have to create first a Connection object. The connection object will represent the database. The argument of connection - in the following example "companys.db" - functions both as the name of the file, where the data will be stored, and as the name of the database. If a file with this name exists, it will be opened. It has to be a SQLite database file of course! In the following example, we will open a database called company. The file does not have to exist.:

```
>>> import sqlite3
>>> connection = sqlite3.connect("company.db")
```

We have now created a database with the name "company". It's like having sent the command "CREATE DATABASE company;" to a SQL server. If you call "sqlite3.connect('company.db')" again, it will open the previously created database.

After having created an empty database, you will most probably add one or more tables to this database. The SQL syntax for creating a table "employee" in the database "company" looks like this:

```
CREATE TABLE employee (
staff_number INT NOT NULL AUTO_INCREMENT,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE,
PRIMARY KEY (staff_number) );
```

This is the way, somebody might do it on a SQL command shell. Of course, we want to do this directly from Python. To be capable to send a command to "SQL", or SQLite, we need a cursor object. Usually, a cursor in SQL and databases is a control structure to traverse over the records in a database. So it's used for the fetching of the results. In SQLite (and other Python DB interfaces)it is more generally used. It's used for performing all SQL commands.

We get the cursor object by calling the cursor() method of connection. An arbitrary number of cursors can be created. The cursor is used to traverse the records from the result set. We can define a SQL command with a triple quoted string in Python:

```
sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
```

```
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""
```

Concerning the SQL syntax: You may have noticed that the AUTOINCREMENT field is missing in the SQL code within our Python program. We have defined the staff_number field as "INTEGER PRIMARY KEY" A column which is labelled like this will be automatically auto-incremented in SQLite3. To put it in other words: If a column of a table is declared to be an INTEGER PRIMARY KEY, then whenever a NULL will be used as an input for this column, the NULL will be automatically converted into an integer which will one larger than the highest value so far used in that column. If the table is empty, the value 1 will be used. If the largest existing value in this column has the 9223372036854775807, which is the largest possible INT in SQLite, an unused key value is chosen at random.

Now we have a database with a table but no data included. To populate the table we will have to send the "INSERT" command to SQLite. We will use again the execute method. The following example is a complete working example. To run the program you will either have to remove the file company.db or uncomment the "DROP TABLE" line in the SQL command:

```
import sqlite3
connection = sqlite3.connect("company.db")

cursor = connection.cursor()

# delete
#cursor.execute("""DROP TABLE employee;""")

sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""

cursor.execute(sql_command)

sql_command = """INSERT INTO employee (staff_number, fnam
e, lname, gender, birth_date)
    VALUES (NULL, "William", "Shakespeare", "m", "1961-10
-25");"""
cursor.execute(sql_command)
```

```
sql_command = """INSERT INTO employee (staff_number, fnam
e, lname, gender, birth_date)
    VALUES (NULL, "Frank", "Schiller", "m", "1955-08-1
7");"""
cursor.execute(sql_command)

# never forget this, if you want the changes to be saved:
connection.commit()

connection.close()
```

Of course, in most cases, you will not literally insert data into a SQL table. You will rather have a lot of data inside of some Python data type e.g. a dictionary or a list, which has to be used as the input of the insert statement.

The following working example, assumes that you have already an existing database company.db and a table employee. We have a list with data of persons which will be used in the INSERT statement:

```
import sqlite3
connection = sqlite3.connect("company.db")

cursor = connection.cursor()

staff_data = [ ("William", "Shakespeare", "m", "1961-10-2
5"),
               ("Frank", "Schiller", "m", "1955-08-17"),
               ("Jane", "Wall", "f", "1989-03-14") ]

for p in staff_data:
    format_str = """INSERT INTO employee (staff_number, f
name, lname, gender, birth_date)
    VALUES (NULL, "{first}", "{last}", "{gender}", "{birt
hdate}");"""

    sql_command = format_str.format(first=p[0], last=p
[1], gender=p[2], birthdate = p[3])
    cursor.execute(sql_command)
```

The time has come now to finally query our employee table:

```
import sqlite3
connection = sqlite3.connect("company.db")
```

```
cursor = connection.cursor()

cursor.execute("SELECT * FROM employee")
print("fetchall:")
result = cursor.fetchall()
for r in result:
    print(r)
cursor.execute("SELECT * FROM employee")
print("\nfetch one:")
res = cursor.fetchone()
print(res)
```

If we run this program, saved as "sql_company_query.py", we get the following result, depending on the actual data:

```
$ python3 sql_company_query.py
fetchall:
(1, 'William', 'Shakespeare', 'm', None, '1961-10-25')
(2, 'Frank', 'Schiller', 'm', None, '1955-08-17')
(3, 'Bill', 'Windows', 'm', None, '1963-11-29')
(4, 'Esther', 'Wall', 'm', None, '1991-05-11')
(5, 'Jane', 'Thunder', 'f', None, '1989-03-14')

fetch one:
(1, 'William', 'Shakespeare', 'm', None, '1961-10-25')
```

MYSQL

If you work under a Python 2.x version, the module MySQLdb can be used. It has to be installed. This can be accomplished under Debian or Ubuntu like this:

```
sudo apt-get install python-MySQLdb
```

If you work with Python 3, you have to make sure that you write everything lowercase:

```
sudo apt-get install python3-mysqldb
```

Of course, you have also the possibility to install it via "pip install" inside a virtualenv:

```
pip install mysqlclient
```

- import the MySQLdb modul

- Open a connection to the SQL server
- Sending and receiving commands
- Closing the connection to SQL

Importing and connecting looks like this:

```
import MySQLdb

connection = MySQLdb.connect (host = "localhost",
                              user = "testuser",
                              passwd = "testpass",
                              db = "company")

cursor = connection.cursor()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone()
print("server version:", row[0])
cursor.close()
connection.close()
```

For the following examples, we assume that you have created a user "pytester". You can do this e.g. on the command line with the following commands: First we start a mysql session with:

```
mysql -u root -p
```

On the mysql shell we continue with:

```
mysql> CREATE USER 'pytester'@'localhost' IDENTIFIED BY
'monty';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON *.* TO 'pytester'@'localho
st';
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

Let's check the MySQL server version by using the previously created connection. To do this, we have to create a cursor object first:

```
import mysql.connector as mc

connection = mc.connect (host = "localhost",
                         user = "pytester",
```

```
                            passwd = "monty",
                            db = "company")
cursor = connection.cursor()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone()
print("server version:", row[0])
cursor.close()
connection.close()
```

The output may look like this:

```
server version: 5.5.52-0ubuntu0.14.04.1
```

Like in our example for sqlite3 in the beginning of this chapter, we will create a table employee and fill it with some data. The program works only under Python 3:

```
import sys
import mysql.connector as mc

try:
    connection = mc.connect (host = "localhost",
                             user = "pytester",
                             passwd = "monty",
                             db = "company")
except mc.Error as e:
    print("Error %d: %s" % (e.args[0], e.args[1]))
    sys.exit(1)

cursor = connection.cursor()

cursor.execute ("DROP TABLE IF EXISTS employee")

# delete
#cursor.execute("""DROP TABLE employee;""")

sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""

cursor.execute(sql_command)

staff_data = [ ("William", "Shakespeare", "m", "1961-10-2
```

```
5"),
                ("Frank", "Schiller", "m", "1955-08-17"),
                ("Jane", "Wall", "f", "1989-03-14"),
                ]

for staff, p in enumerate(staff_data):
    format_str = """INSERT INTO employee (staff_number, f
name, lname, gender, birth_date)
    VALUES ({staff_no}, '{first}', '{last}', '{gender}',
'{birthdate}');"""

    sql_command = format_str.format(staff_no=staff, first
=p[0], last=p[1], gender=p[2], birthdate = p[3])
    print(sql_command)
    cursor.execute(sql_command)

connection.commit()


cursor.close()
connection.close()
```

This program returns the following output which corresponds to the insertions into the table 'employee':

```
INSERT INTO employee (staff_number, fname, lname, gender,
birth_date)
    VALUES (0, 'William', 'Shakespeare', 'm', '1961-10-2
5');
INSERT INTO employee (staff_number, fname, lname, gender,
birth_date)
    VALUES (1, 'Frank', 'Schiller', 'm', '1955-08-17');
INSERT INTO employee (staff_number, fname, lname, gender,
birth_date)
    VALUES (2, 'Jane', 'Wall', 'f', '1989-03-14');
```

After this, we want to query our database again:

```
import sys
import mysql.connector as mc

try:
    connection = mc.connect (host = "localhost",
                             user = "pytester",
                             passwd = "monty",
                             db = "company")
except mc.Error as e:
```

```
        print("Error %d: %s" % (e.args[0], e.args[1]))
        sys.exit(1)

    cursor = connection.cursor()

    cursor.execute("SELECT * FROM employee")
    print('''Result of "SELECT * FROM employee":''')
    result = cursor.fetchall()
    for r in result:
        print(r)

    cursor.close()
    connection.close()
```

It generates the following output:

```
Result of "SELECT * FROM employee":
(0, 'William', 'Shakespeare', 'm', None, datetime.date(19
61, 10, 25))
(1, 'Frank', 'Schiller', 'm', None, datetime.date(1955,
8, 17))
(2, 'Jane', 'Wall', 'f', None, datetime.date(1989, 3, 1
4))
```

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

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:

```
\version "2.12.3"
{
```

```
  c' e' g' a'
}
```

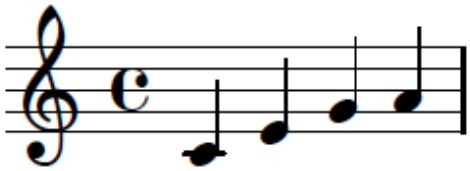
Saving the Lilypond code above in a file called simple.ly, we can start Lilypond on a command shell:

```
lilypond simple.ly
```

This call creates the following output:

```
GNU LilyPond 2.12.3
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 `simple.ps'...
Converting to `./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

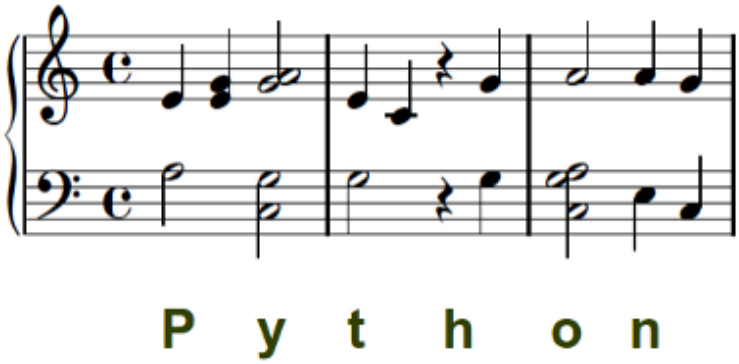
We write a program, which translates an arbitrary text string into a piece of music, which can be played on the piano. To this purpose every character of the Latin alphabet is mapped to two quarters of a four-four time, both for the left and right hand of a piano score.

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 s
imple 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 += ">>\n}\n"

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

print title + staff
```

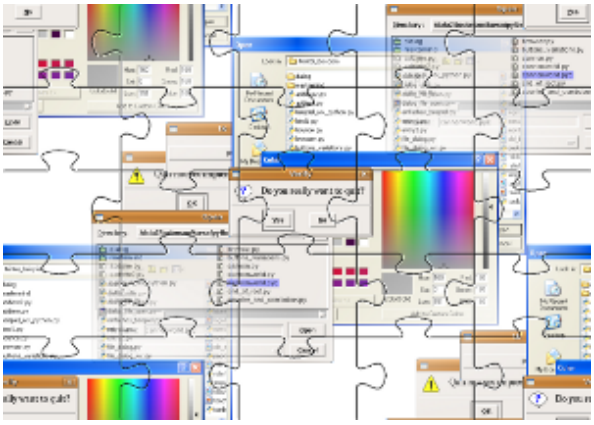
Putting the code together and saving it under text_to_music.py, we can create our piano score on the command with the following command:

```
python text_to_music.py > piano_score.ly
```

This will create a PDF file called piano_score.pdf.

PYTHON TKINTER

INTRODUCTION



We have often been asked: "Is there no Tk for Python?" or "Is Tkinter the same as Tk?" Of course, there is Tk for Python. Without Tk Python would be less attractive to many users. Tk is called Tkinter in Python, or to be precise, Tkinter is the Python interface for Tk. Tkinter is an acronym for "Tk interface".

Tk was developed as a GUI extension for the Tcl scripting language by John Ousterhout. The first release was in 1991. Tk proved as extremely successful in the 1990's, because it is easier to learn and to use than other toolkits. So it is no wonder that many programmers wanted to use Tk independently of Tcl. That's why bindings for lots of other programming languages have been developed, including Perl, Ada (called TASH), Python (called Tkinter), Ruby, and Common Lisp.

Tk provides the following widgets:

- button
- canvas
- checkbutton
- combobox
- entry
- frame
- label
- labelframe
- listbox
- menu
- menubutton
- message
- notebook
- tk_optionMenu
- panedwindow
- progressbar

- radiobutton
- scale
- scrollbar
- separator
- sizegrip
- spinbox
- text
- treeview

It provides the following top-level windows:

- tk_chooseColor - pops up a dialog box for the user to select a color.
- tk_chooseDirectory - pops up a dialog box for the user to select a directory.
- tk_dialog - creates a modal dialog and waits for a response.
- tk_getOpenFile - pops up a dialog box for the user to select a file to open.
- tk_getSaveFile - pops up a dialog box for the user to select a file to save.
- tk_messageBox - pops up a message window and waits for a user response.
- tk_popup - posts a popup menu.
- toplevel - creates and manipulates toplevel widgets.

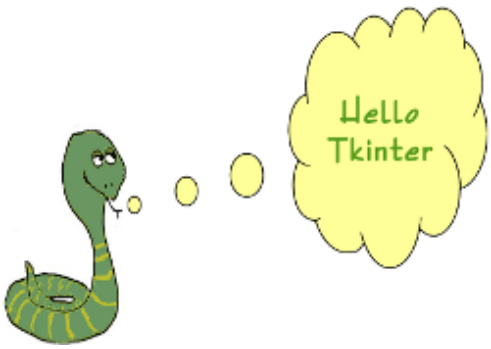
Tk also provides three geometry managers:

- place - which positions widgets at absolute locations
- grid - which arranges widgets in a grid
- pack - which packs widgets into a cavity

TKINTER

HELLO TKINTER LABEL

We will start our tutorial with one of the easiest widgets of Tk (Tkinter), i.e. a label. A Label is a Tkinter Widget class, which is used to display text or an image. The label is a widget that the user just views but not interact with.



There is hardly any book or introduction into a programming language, which doesn't start with the "Hello World" example. We will draw on tradition but will slightly modify the output to "Hello Tkinter" instead of "Hello World".

The following Python script uses Tkinter to create a window with the text "Hello Tkinter". You can use the Python interpretor to type this script line after line, or you can save it in a file, for example, "hello.py":

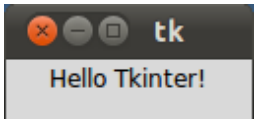
```
import tkinter as tk

# if you are still working under a Python 2 version,
# comment out the previous line and uncomment the followi
ng line
# import Tkinter as tk

root = tk.Tk()

w = tk.Label(root, text="Hello Tkinter!")
w.pack()

root.mainloop()
```

STARTING OUR EXAMPLE

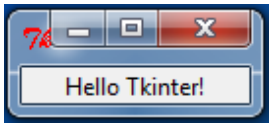
If we save the script under the name hello.py, we can start it like this using the command shell:

```
$ python3 hello.py
```

If you run the command under the Gnome and Linux, the window the window will look like this:



Under Windows it appears in the Windows look and feel:



EXPLANATION

The tkinter module, containing the Tk toolkit, has always to be imported. In our example, we imported tkinter by renaming it into tk, which is the preferred way to do it:

```
import tkinter as tk
```

To initialize tkinter, we have to create a Tk root widget, which is a window with a title bar and other decoration provided by the window manager. The root widget has to be created before any other widgets and there can only be one root widget.

```
root = tk.Tk()
```

The next line of code contains the Label widget. The first parameter of the Label call is the name of the parent window, in our case "root". So our Label widget is a child of the root widget. The keyword parameter "text" specifies the text to be shown:

```
w = tk.Label(root, text="Hello Tkinter!")
```

The pack method tells Tk to fit the size of the window to the given text.

```
w.pack()
```

The window won't appear until we enter the Tkinter event loop:

```
root.mainloop()
```

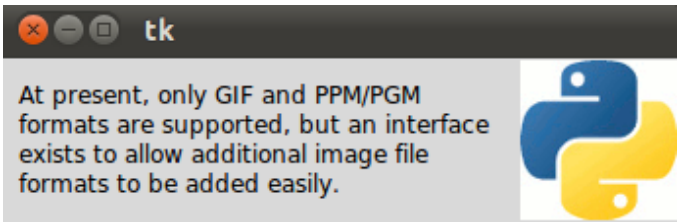
Our script will remain in the event loop until we close the window.

USING IMAGES IN LABELS

As we have already mentioned, labels can contain text and images. The following example contains two labels, one with a text and the other one with an image.

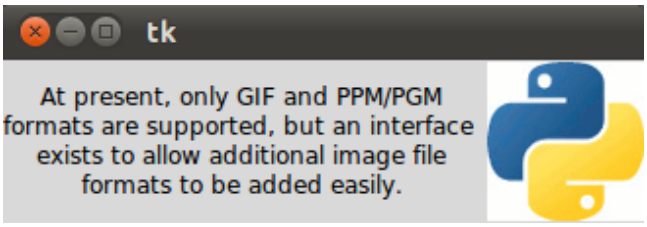
```
import tkinter as tk

root = tk.Tk()
logo = tk.PhotoImage(file="python_logo_small.gif")

w1 = tk.Label(root, image=logo).pack(side="right")

explanation = """At present, only GIF and PPM/PGM
formats are supported, but an interface
exists to allow additional image file
formats to be added easily."""

w2 = tk.Label(root,
              justify=tk.LEFT,
              padx = 10,
              text=explanation).pack(side="left")
root.mainloop()
```

If you start this script, it will look like this using Ubuntu Linux with Gnome desktop:



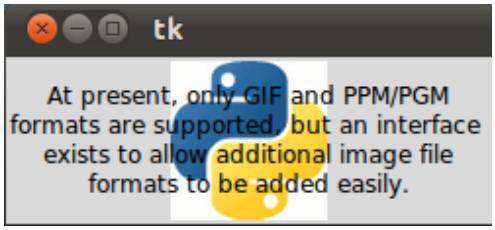
The "justify" parameter can be used to justify a text on the LEFT, RIGHT or CENTER. padx can be used to add additional horizontal padding around a text label. The default padding is 1 pixel. pady is similar for vertical padding. The previous example without justify (default is

centre) and padx looks like this:



You want the text drawn on top of the image? No problem! We need just one label and use the image and the text option at the same time. By default, if an image is given, it is drawn instead of the text. To get the text as well, you have to use the compound option. If you set the compound option to CENTER the text will be drawn on top of the image:

```
import tkinter as tk

root = tk.Tk()
logo = tk.PhotoImage(file="python_logo_small.gif")

explanation = """At present, only GIF and PPM/PGM
formats are supported, but an interface
exists to allow additional image file
formats to be added easily."""

w = tk.Label(root,
             compound = tk.CENTER,
             text=explanation,
             image=logo).pack(side="right")

root.mainloop()
```



We can have the image on the right side and the text left justified with a padding of 10 pixel on

the left and right side by changing the Label command like this:

```
w = Label(root,
          justify=LEFT,
          compound = LEFT,
          padx = 10,
          text=explanation,
          image=logo).pack(side="right")
```

If the compound option is set to BOTTOM, LEFT, RIGHT, or TOP, the image is drawn correspondingly to the bottom, left, right or top of the text.

COLORIZED LABELS IN VARIOUS FONTS

Some Tk widgets, like the label, text, and canvas widget, allow you to specify the fonts used to display text. This can be achieved by setting the attribute "font". typically via a "font" configuration option. You have to consider that fonts are one of several areas that are not platform-independent.

The attribute fg can be used to have the text in another colour and the attribute bg can be used to change the background colour of the label.

```
import tkinter as tk

root = tk.Tk()
tk.Label(root,
                 text="Red Text in Times Font",
                 fg = "red",
                 font = "Times").pack()
tk.Label(root,
                 text="Green Text in Helvetica Font",
                 fg = "light green",
                 bg = "dark green",
                 font = "Helvetica 16 bold italic").pack
()
tk.Label(root,
                 text="Blue Text in Verdana bold",
                 fg = "blue",
                 bg = "yellow",
                 font = "Verdana 10 bold").pack()
```

```
root.mainloop()
```

The result looks like this:

DYNAMICAL CONTENT IN A LABEL

The following script shows an example, where a label is dynamically incremented by 1 until the stop button is pressed:

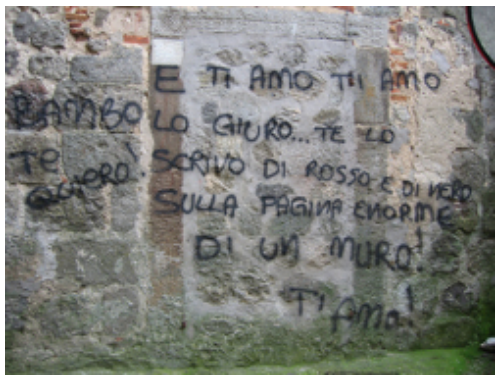
```
import tkinter as tk

counter = 0
def counter_label(label):
  def count():
    global counter
    counter += 1
    label.config(text=str(counter))
    label.after(1000, count)
  count()


root = tk.Tk()
root.title("Counting Seconds")
label = tk.Label(root, fg="green")
label.pack()
counter_label(label)
button = tk.Button(root, text='Stop', width=25, command=r
oot.destroy)
button.pack()
root.mainloop()
```

The result of the previous script looks like this:


Counting Seconds
16
Stop

TKINTER

MESSAGE WIDGET



The widget can be used to display short text messages. The message widget is similar in its functionality to the Label widget, but it is more flexible in displaying text, e.g. the font can be changed while the Label widget can only display text in a single font. It provides a multiline object, that is the text may span more than one line. The text is automatically broken into lines and justified. We were ambiguous, when we said, that the font of the message widget can be changed. This means that we can choose arbitrarily a font for one widget, but the text of this widget will be rendered solely in this font. This means that we can't change the font within a widget. So it's not possible to have a text in more than one font. If you need to display text in multiple fonts, we suggest to use a Text widget.

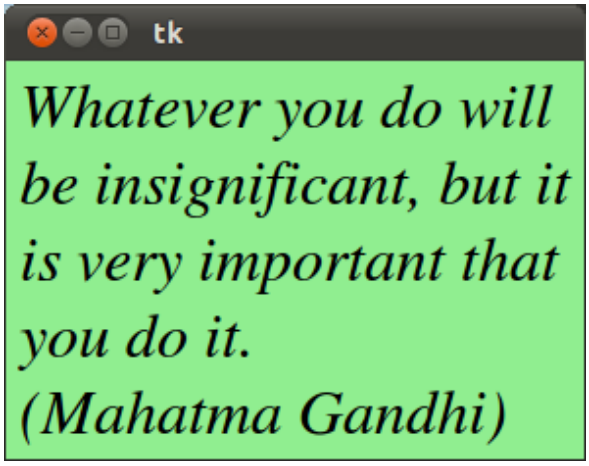
The syntax of a message widget:

w = Message (master, option, ...)

Let's have a look at a simple example. The following script creates a message with a famous saying by Mahatma Gandhi:

```
import tkinter as tk
master = tk.Tk()
whatever_you_do = "Whatever you do will be insignificant,
but it is very important that you do it.\n(Mahatma Gandh
i)"
msg = tk.Message(master, text = whatever_you_do)
msg.config(bg='lightgreen', font=('times', 24, 'italic'))
msg.pack()
tk.mainloop()
```

The widget created by the script above looks like this:



THE OPTIONS IN DETAIL

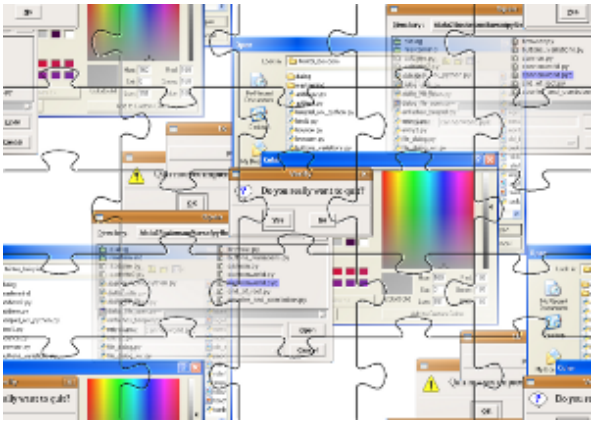
Option	Meaning
anchor	The position, where the text should be placed in the message widget: N, NE, E, SE, S, SW, W, NW, or CENTER. The Default is CENTER.
aspect	Aspect ratio, given as the width/height relation in percent. The default is 150, which means that the message will be 50% wider than it is high. Note that if the width is explicitly set, this option is ignored.
background	The background color of the message widget. The default value is system specific.
bg	Short for background.
borderwidth	Border width. Default value is 2.
bd	Short for borderwidth.
cursor	Defines the kind of cursor to show when the mouse is moved over the message widget. By default the standard cursor is used.
font	Message font. The default value is system specific.
foreground	Text color. The default value is system specific.
fg	Same as foreground.
highlightbackground	Together with highlightcolor and highlightthickness, this option controls how to draw the highlight region.
highlightcolor	See highlightbackground.

highlightthickness	See highlightbackground.
justify	Defines how to align multiple lines of text. Use LEFT, RIGHT, or CENTER. Note that to position the text inside the widget, use the anchor option. Default is LEFT.
padx	Horizontal padding. Default is -1 (no padding).
pady	Vertical padding. Default is -1 (no padding).
relief	Border decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE.
takefocus	If true, the widget accepts input focus. The default is false.
text	Message text. The widget inserts line breaks if necessary to get the requested aspect ratio. (text/Text)
textvariable	Associates a Tkinter variable with the message, which is usually a StringVar. If the variable is changed, the message text is updated.
width	Widget width given in character units. A suitable width based on the aspect setting is automatically chosen, if this option is not given.

TKINTER

TKINTER BUTTONS

The Button widget is a standard Tkinter widget, which is used for various kinds of buttons. A button is a widget which is designed for the user to interact with, i.e. if the button is pressed by mouse click some action might be started. They can also contain text and images like labels. While labels can display text in various fonts, a button can only display text in a single font. The text of a button can span more than one line.



A Python function or method can be associated with a button. This function or method will be executed, if the button is pressed in some way.

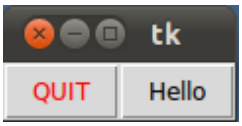
EXAMPLE FOR THE BUTTON CLASS

The following script defines two buttons: one to quit the application and another one for the action, i.e. printing the text "Tkinter is easy to use!" on the terminal.

```
import tkinter as tk


def write_slogan():
    print("Tkinter is easy to use!")

root = tk.Tk()
frame = tk.Frame(root)
frame.pack()

button = tk.Button(frame,
                   text="QUIT",
                   fg="red",
                   command=quit)
button.pack(side=tk.LEFT)
```

```
slogan = tk.Button(frame,
                   text="Hello",
                   command=write_slogan)
slogan.pack(side=tk.LEFT)

root.mainloop()
```

The result of the previous example looks like this:



DYNAMICAL CONTENT IN A LABEL

The following script shows an example, where a label is dynamically incremented by 1 until a stop button is pressed:

```
import tkinter as tk

counter = 0
def counter_label(label):
  counter = 0
  def count():
    global counter
    counter += 1
    label.config(text=str(counter))
    label.after(1000, count)
  count()


root = tk.Tk()
root.title("Counting Seconds")
label = tk.Label(root, fg="dark green")
label.pack()
counter_label(label)
button = tk.Button(root, text='Stop', width=25, command=r
oot.destroy)
```

```
button.pack()
root.mainloop()
```

The result of the previous example looks like this:

TKINTER

VARIABLE CLASSES

Some widgets (like text entry widgets, radio buttons and so on) can be connected directly to application variables by using special options: variable, textvariable, onvalue, offvalue, and value. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value. These Tkinter control variables are used like regular Python variables to keep certain values. It's not possible to hand over a regular Python variable to a widget through a variable or textvariable option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in the Tkinter module. They are declared like this:

- x = StringVar() # Holds a string; default value ""
- x = IntVar() # Holds an integer; default value 0
- x = DoubleVar() # Holds a float; default value 0.0
- x = BooleanVar() # Holds a boolean, returns 0 for False and 1 for True

To read the current value of such a variable, call the method get(). The value of such a variable can be changed with the set() method.

TKINTER

RADIO BUTTONS

A radio button, sometimes called option button, is a graphical user interface element of Tkinter, which allows the user to choose (exactly) one of a predefined set of options. Radio buttons can contain text or images. The button can only display text in a single font. A Python function or method can be associated with a radio button. This function or method will be called, if you press this radio button.

Radio buttons are named after the physical buttons used on old radios to select wave bands or preset radio stations. If such a button was pressed, other buttons would pop out, leaving the pressed button the only pushed in button.

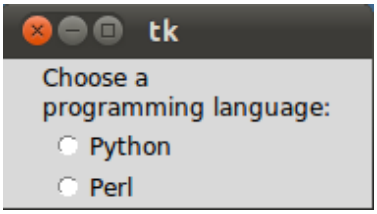
Each group of Radio button widgets has to be associated with the same variable. Pushing a button changes the value of this variable to a predefined certain value.

SIMPLE EXAMPLE WITH RADIO BUTTONS

```
import tkinter as tk

root = tk.Tk()

v = tk.IntVar()

tk.Label(root,
         text="""Choose a
programming language:""",
         justify = tk.LEFT,
         padx = 20).pack()
tk.Radiobutton(root,
               text="Python",
               padx = 20,
               variable=v,
```

```
                value=1).pack(anchor=tk.W)
tk.Radiobutton(root,
               text="Perl",
               padx = 20,
               variable=v,
               value=2).pack(anchor=tk.W)

root.mainloop()
```

The result of the previous example looks like this:



IMPROVING THE EXAMPLE

In many cases, there are more than two radio buttons. It would be cumbersome, if we have to define and write down each button. The solution is shown in the following example. We have a list "languages", which contains the button texts and the corresponding values. We can use a for loop to create all the radio buttons.

```
import tkinter as tk

root = tk.Tk()

v = tk.IntVar()
v.set(1)  # initializing the choice, i.e. Python

languages = [
    ("Python",1),
    ("Perl",2),
    ("Java",3),
    ("C++",4),
    ("C",5)
]

def ShowChoice():
    print(v.get())
```

```
tk.Label(root,
         text="""Choose your favourite
programming language:""",
         justify = tk.LEFT,
         padx = 20).pack()

for val, language in enumerate(languages):
    tk.Radiobutton(root,
                   text=language,
                   padx = 20,
                   variable=v,
                   command=ShowChoice,
                   value=val).pack(anchor=tk.W)


root.mainloop()
```

The result of the previous example looks like this:

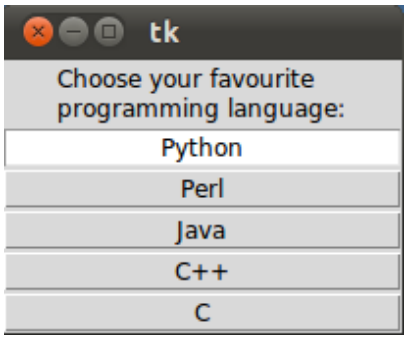
INDICATOR

Instead of having radio buttons with circular holes containing white space, we can have radio buttons with the complete text in a box. We can do this by setting the indicatoron (stands for "indicator on") option to 0, which means that there will be no separate radio button indicator. The default is 1.

We exchange the definition of the Radiobutton in the previous example with the following one:

```
tk.Radiobutton(root,
               text=language,
               indicatoron = 0,
               width = 20,
               padx = 20,
               variable=v,
               command=ShowChoice,
               value=val).pack(anchor=tk.W)
```

We have added the option indicatoron and the option width.



CHECKBOXES

INTRODUCTION

Checkboxes, also known as tickboxes or tick boxes or check boxes, are widgets that permit the user to make multiple selections from a number of different options. This is different to a radio button, where the user can make only one choice.

Usually, checkboxes are shown on the screen as square boxes that can contain white spaces (for false, i.e not checked) or a tick mark or X (for true, i.e. checked).

A caption describing the meaning of the checkbox is usually shown adjacent to the checkbox. The state of a checkbox is changed by clicking the mouse on the box. Alternatively it can be done by clicking on the caption, or by using a keyboard shortcut, for example, the space bar.

A Checkbox has two states: on or off.

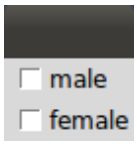
The Tkinter Checkbutton widget can contain text, but only in a single font, or images, and a button can be associated with a Python function or method. When a button is pressed, Tkinter calls the associated function or method. The text of a button can span more than one line.

SIMPLE EXAMPLE

The following example presents two checkboxes "male" and "female". Each checkbox needs a different variable name (IntVar()).

```
from tkinter import *
master = Tk()
var1 = IntVar()
Checkbutton(master, text="male", variable=var1).grid(row=
0, sticky=W)
var2 = IntVar()
Checkbutton(master, text="female", variable=var2).grid(ro
w=1, sticky=W)
mainloop()
```

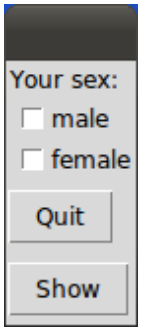
If we start this script, we get the following window:



We can improve this example a little bit. First we add a Label to it. Furthermore we add two Buttons, one to leave the application and the other one to view the values var1 and var2.

```
from tkinter import *
master = Tk()

def var_states():
   print("male: %d,\nfemale: %d" % (var1.get(), var2.get
()))

Label(master, text="Your sex:").grid(row=0, sticky=W)
var1 = IntVar()
Checkbutton(master, text="male", variable=var1).grid(row=
1, sticky=W)
var2 = IntVar()
Checkbutton(master, text="female", variable=var2).grid(ro
w=2, sticky=W)
Button(master, text='Quit', command=master.quit).grid(row
=3, sticky=W, pady=4)
Button(master, text='Show', command=var_states).grid(row=
4, sticky=W, pady=4)
mainloop()
```

The result of the previous script looks like this:



If we check "male" and click on "Show", we get the following output:

```
male: 1,
female: 0
```

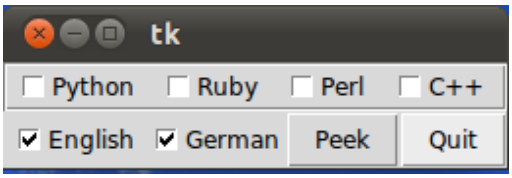
ANOTHER EXAMPLE WITH CHECKBOXES

We write an application, which depicts a list of programming languages, e.g. ['Python', 'Ruby', 'Perl', 'C++'] and a list of natural languages, e.g. ['English','German'] as checkboxes. So it's possible to choose programming languages and natural languages. Furthermore, we have two buttons: A "Quit" button for ending the application and a "Peek" button for checking the state of the checkbox variables.

```
#!/usr/bin/python3

from tkinter import *
class Checkbar(Frame):
   def __init__(self, parent=None, picks=[], side=LEFT, a
nchor=W):
      Frame.__init__(self, parent)
      self.vars = []
      for pick in picks:
         var = IntVar()
         chk = Checkbutton(self, text=pick, variable=var)
         chk.pack(side=side, anchor=anchor, expand=YES)
         self.vars.append(var)
   def state(self):
      return map((lambda var: var.get()), self.vars)
if __name__ == '__main__':
   root = Tk()
```

```
    lng = Checkbar(root, ['Python', 'Ruby', 'Perl', 'C+
+'])
    tgl = Checkbar(root, ['English','German'])
    lng.pack(side=TOP,  fill=X)
    tgl.pack(side=LEFT)
    lng.config(relief=GROOVE, bd=2)

    def allstates():
       print(list(lng.state()), list(tgl.state()))
    Button(root, text='Quit', command=root.quit).pack(side
=RIGHT)
    Button(root, text='Peek', command=allstates).pack(side
=RIGHT)
    root.mainloop()
```

The window looks like this:



ENTRY WIDGETS

INTRODUCTION

Entry widgets are the basic widgets of Tkinter used to get input, i.e. text strings, from the user of an application. This widget allows the user to enter a single line of text. If the user enters a string, which is longer than the available display space of the widget, the content will be scrolled. This means that the string cannot be seen in its entirety. The arrow keys can be used to move to the invisible parts of the string. If you want to enter multiple lines of text, you have to use the text widget. An entry widget is also limited to single font.

The syntax of an entry widget looks like this:

w = Entry(master, option, ...)

"master" represents the parent window, where the entry widget should be placed. Like other widgets, it's possible to further influence the rendering of the widget by using options. The comma separated list of options can be empty.

The following simple example creates an application with two entry fields. One for entering a last name and one for the first name. We use Entry without options.

```
import tkinter as tk

master = tk.Tk()
tk.Label(master, text="First Name").grid(row=0)
tk.Label(master, text="Last Name").grid(row=1)

e1 = tk.Entry(master)
e2 = tk.Entry(master)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
```

```
master.mainloop()
```

The window created by the previous script looks like this:

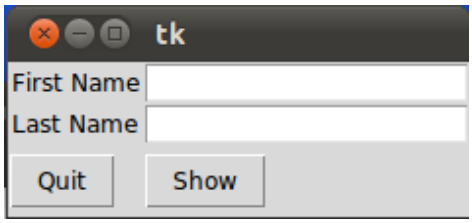
Okay, we have created Entry fields, so that the user of our program can put in some data. But how can our program access this data? How do we read the content of an Entry?

To put it in a nutshell: The get() method is what we are looking for. We extend our little script by two buttons "Quit" and "Show". We bind the function show_entry_fields(), which is using the get() method on the Entry objects, to the Show button. So, every time this button is clicked, the content of the Entry fields will be printed on the terminal from which we had called the script.

```
import tkinter as tk

def show_entry_fields():
    print("First Name: %s\nLast Name: %s" % (e1.get(), e
2.get()))

master = tk.Tk()
tk.Label(master,
         text="First Name").grid(row=0)
tk.Label(master,
         text="Last Name").grid(row=1)

e1 = tk.Entry(master)
e2 = tk.Entry(master)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

tk.Button(master,
          text='Quit',
          command=master.quit).grid(row=3,
                                    column=0,
                                    sticky=tk.W,
```

```
                                                pady=4)
tk.Button(master,
          text='Show', command=show_entry_fields).grid(ro
w=3,
                                                                co
lumn=1,
                                                                st
icky=tk.W,
                                                                pa
dy=4)

tk.mainloop()
```

The complete application looks now like this:



Let's assume now that we want to start the Entry fields with default values, e.g. we fill in "Miller" or "Baker" as a last name, and "Jack" or "Jill" as a first name. The new version of our Python program gets the following two lines, which can be appended after the Entry definitions, i.e. "e2 = tk.Entry(master)":

```
e1.insert(10, "Miller")
e2.insert(10, "Jill")
```

What about deleting the input of an Entry object, every time, we are showing the content in our function show_entry_fields()? No problem! We can use the delete method. The delete() method has the format delete(first, last=None). If only one number is given, it deletes the character at index. If two are given, the range from "first" to "last" will be deleted. Use delete(0, END) to delete all text in the widget.

```
import tkinter as tk

def show_entry_fields():
    print("First Name: %s\nLast Name: %s" % (e1.get(), e
```

```
2.get()))
    e1.delete(0, tk.END)
    e2.delete(0, tk.END)

master = tk.Tk()
tk.Label(master, text="First Name").grid(row=0)
tk.Label(master, text="Last Name").grid(row=1)

e1 = tk.Entry(master)
e2 = tk.Entry(master)
e1.insert(10, "Miller")
e2.insert(10, "Jill")

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

tk.Button(master,
          text='Quit',
          command=master.quit).grid(row=3,
                                    column=0,
                                    sticky=tk.W,
                                    pady=4)
tk.Button(master, text='Show', command=show_entry_field
s).grid(row=3,

column=1,

sticky=tk.W,

pady=4)

master.mainloop()

tk.mainloop()
```

The next example shows, how we can elegantly create lots of Entry field in a more Pythonic way. We use a Python list to hold the Entry descriptions, which we include as labels into the application.

```
import tkinter as tk

fields = 'Last Name', 'First Name', 'Job', 'Country'

def fetch(entries):
    for entry in entries:
        field = entry[0]
```

```
            text  = entry[1].get()
            print('%s: "%s"' % (field, text))

    def makeform(root, fields):
        entries = []
        for field in fields:
            row = tk.Frame(root)
            lab = tk.Label(row, width=15, text=field, anchor
    ='w')
            ent = tk.Entry(row)
            row.pack(side=tk.TOP, fill=tk.X, padx=5, pady=5)
            lab.pack(side=tk.LEFT)
            ent.pack(side=tk.RIGHT, expand=tk.YES, fill=tk.X)
            entries.append((field, ent))
        return entries

    if __name__ == '__main__':
        root = tk.Tk()
        ents = makeform(root, fields)
        root.bind('', (lambda event, e=ents: fetch(e)))
        b1 = tk.Button(root, text='Show',
                       command=(lambda e=ents: fetch(e)))
        b1.pack(side=tk.LEFT, padx=5, pady=5)
        b2 = tk.Button(root, text='Quit', command=root.quit)
        b2.pack(side=tk.LEFT, padx=5, pady=5)
        root.mainloop()
```

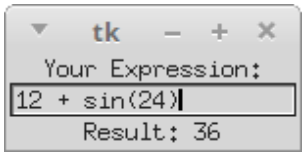
If you start this Python script, it will look like this:

CALCULATOR

We are not really writing a calculator, we rather provide a GUI which is capable of evaluating any mathematical expression and printing the result.

```
import tkinter as tk
from math import *

def evaluate(event):
    res.configure(text = "Result: " + str(eval(entry.get
()))

w = tk.Tk()
tk.Label(w, text="Your Expression:").pack()
entry = tk.Entry(w)
entry.bind("", evaluate)
entry.pack()
res = tk.Label(w)
res.pack()
w.mainloop()
```

Our widget looks like this:



INTEREST CALCULATION

The following formula can be used to calculate the balance B_k after k payments (balance index), starting with an initial balance (also known as the loan principal) and a period rate r:

$$B_k = (1+r)^k \cdot B_0 - \frac{(1+r)^k - 1}{r} \cdot p$$

where
rate = interest rate in percent, e.g. 3 %
i = rate / 100, annual rate in decimal form
r = period rate = i / 12
B_0 = initial balance, also called loan principal
B_k = balance after k payments

k = number of monthly payments
p = period (monthly) payment
If we want to find the necessary monthly payment if the loan is to be paid off in n payments one sets Bn = 0 and gets the formula:

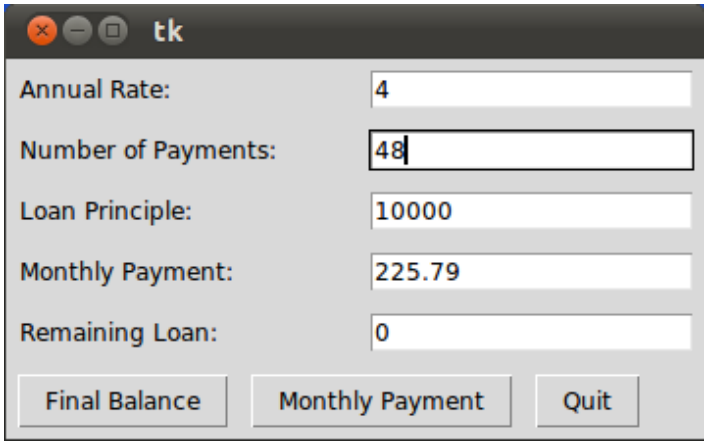
$$p = r \cdot \frac{(1+r)^n B_0 - B_n}{(1+r)^n - 1}$$

where
n = number of monthly payments to pay back the principal loan

```
import tkinter as tk

fields = ('Annual Rate', 'Number of Payments', 'Loan Prin
ciple', 'Monthly Payment', 'Remaining Loan')

def monthly_payment(entries):
    # period rate:
    r = (float(entries['Annual Rate'].get()) / 100) / 12
    print("r", r)
    # principal loan:
    loan = float(entries['Loan Principle'].get())
    n =  float(entries['Number of Payments'].get())
    remaining_loan = float(entries['Remaining Loan'].get
())
    q = (1 + r)** n
    monthly = r * ( (q * loan - remaining_loan) / ( q - 1
))
    monthly = ("%8.2f" % monthly).strip()
    entries['Monthly Payment'].delete(0, tk.END)
    entries['Monthly Payment'].insert(0, monthly )
    print("Monthly Payment: %f" % float(monthly))

def final_balance(entries):
    # period rate:
    r = (float(entries['Annual Rate'].get()) / 100) / 12
    print("r", r)
    # principal loan:
    loan = float(entries['Loan Principle'].get())
    n =  float(entries['Number of Payments'].get())
    monthly = float(entries['Monthly Payment'].get())
    q = (1 + r) ** n
    remaining = q * loan  - ( (q - 1) / r) * monthly
```

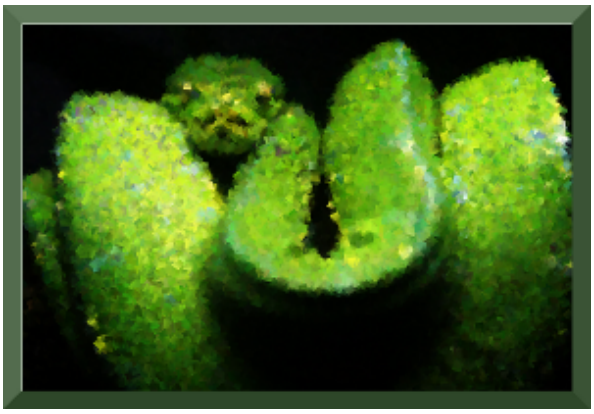
```
    remaining = ("%8.2f" % remaining).strip()
    entries['Remaining Loan'].delete(0, tk.END)
    entries['Remaining Loan'].insert(0, remaining )
    print("Remaining Loan: %f" % float(remaining))

def makeform(root, fields):
    entries = {}
    for field in fields:
        print(field)
        row = tk.Frame(root)
        lab = tk.Label(row, width=22, text=field+": ", an
chor='w')
        ent = tk.Entry(row)
        ent.insert(0, "0")
        row.pack(side=tk.TOP,
                 fill=tk.X,
                 padx=5,
                 pady=5)
        lab.pack(side=tk.LEFT)
        ent.pack(side=tk.RIGHT,
                 expand=tk.YES,
                 fill=tk.X)
        entries[field] = ent
    return entries

if __name__ == '__main__':
    root = tk.Tk()
    ents = makeform(root, fields)
    b1 = tk.Button(root, text='Final Balance',
           command=(lambda e=ents: final_balance(e)))
    b1.pack(side=tk.LEFT, padx=5, pady=5)
    b2 = tk.Button(root, text='Monthly Payment',
           command=(lambda e=ents: monthly_payment(e)))
    b2.pack(side=tk.LEFT, padx=5, pady=5)
    b3 = tk.Button(root, text='Quit', command=root.quit)
    b3.pack(side=tk.LEFT, padx=5, pady=5)
    root.mainloop()
```

Our loan calculator looks like this, if we start it with Python3:


tk
Annual Rate:
4
Number of Payments:
48
Loan Principle:
10000
Monthly Payment:
225.79
Remaining Loan:
0
Final Balance
Monthly Payment
Quit

CANVAS WIDGETS

INTRODUCTION



The Canvas widget supplies graphics facilities for Tkinter. Among these graphical objects are lines, circles, images, and even other widgets. With this widget it's possible to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets.

We demonstrate in our first example, how to draw a line.
The method create_line(coords, options) is used to draw a straight line. The coordinates "coords" are given as four integer numbers: x_1, y_1, x_2, y_2 This means that the line goes from the point (x_1, y_1) to the point (x_2, y_2). After these coordinates follows a comma separated list of additional parameters, which may be empty. We set, for example, the colour of the line to the special green of our website: fill="#476042"

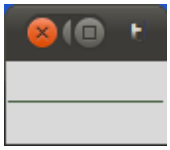
We kept the first example intentionally very simple. We create a canvas and draw a straight horizontal line into this canvas. This line vertically cuts the canvas into two areas.

The casting to an integer value in the assignment "y = int(canvas_height / 2)" is superfluous, because create_line can work with float values as well. They are automatically turned into integer values. In the following you can see the code of our first simple script:

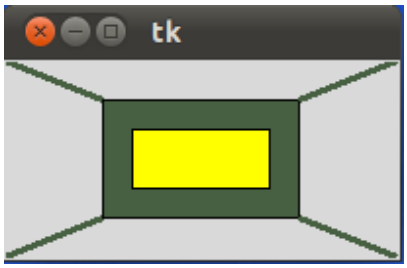
```
from tkinter import *
master = Tk()

canvas_width = 80
canvas_height = 40
w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()
```

```
y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y, fill="#476042")


mainloop()
```

If we start this program, using Python 3, we get the following window:



For creating rectangles we have the method create_rectangle(coords, options). Coords is again defined by two points, but this time the first one is the top left point and the bottom right point of the rectangle.

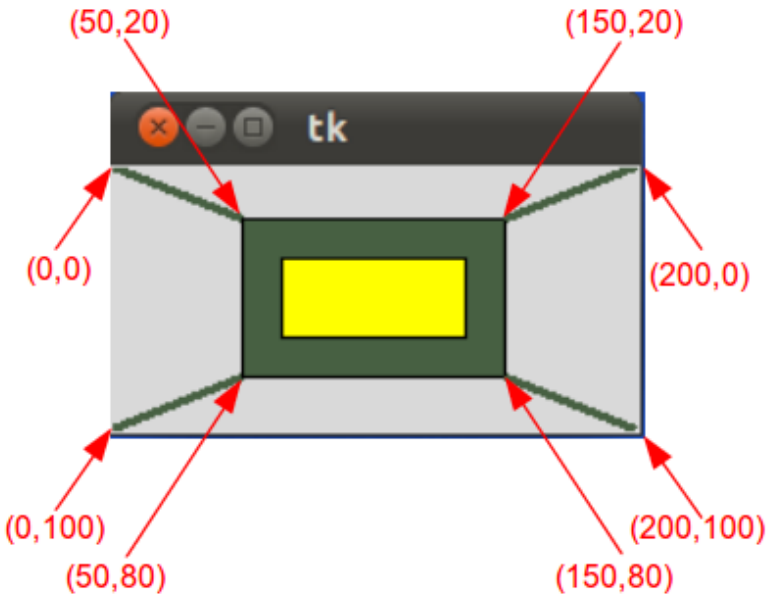


The window, you see above, is created by the following Python tkinter code:

```
from tkinter import *

master = Tk()

w = Canvas(master, width=200, height=100)
w.pack()

w.create_rectangle(50, 20, 150, 80, fill="#476042")
w.create_rectangle(65, 35, 135, 65, fill="yellow")
w.create_line(0, 0, 50, 20, fill="#476042", width=3)
w.create_line(0, 100, 50, 80, fill="#476042", width=3)
w.create_line(150,20, 200, 0, fill="#476042", width=3)
```

```
w.create_line(150, 80, 200, 100, fill="#476042", width=3)

mainloop()
```

The following image with the coordinates will simplify the understanding of application of create_lines and create_rectangle in our previous example.



TEXT ON CANVAS

We demonstrate now how to print text on a canvas. We will extend and modify the previous example for this purpose. The method create_text() can be applied to a canvas object to write text on it. The first two parameters are the x and the y positions of the text object. By default, the text is centred on this position. You can override this with the anchor option. For example, if the coordinate should be the upper left corner, set the anchor to NW. With the keyword parameter text, we can define the actual text to be displayed on the canvas.

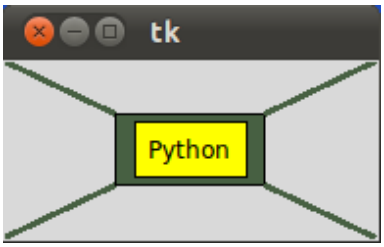
```
from tkinter import *

canvas_width = 200
canvas_height = 100

colours = ("#476042", "yellow")
box=[]
```

```
for ratio in ( 0.2, 0.35 ):
   box.append( (canvas_width * ratio,
                canvas_height * ratio,
                canvas_width * (1 - ratio),
                canvas_height * (1 - ratio) ) )

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

for i in range(2):
   w.create_rectangle(box[i][0], box[i][1],box[i][2],box
[i][3], fill=colours[i])

w.create_line(0, 0,                  # origin of canvas
              box[0][0], box[0][1], # coordinates of left
upper corner of the box[0]
              fill=colours[0],
              width=3)
w.create_line(0, canvas_height,     # lower left corner o
f canvas
              box[0][0], box[0][3], # lower left corner o
f box[0]
              fill=colours[0],
              width=3)
w.create_line(box[0][2],box[0][1],  # right upper corner
of box[0]
              canvas_width, 0,      # right upper corner
of canvas
              fill=colours[0],
              width=3)
w.create_line(box[0][2], box[0][3], # lower right corner
pf box[0]
              canvas_width, canvas_height, # lower right
corner of canvas
              fill=colours[0], width=3)

w.create_text(canvas_width / 2,
              canvas_height / 2,
              text="Python")
mainloop()
```

Though the code of our example program is changed drastically, the graphical result looks still

the same except for the text "Python":

You can understand the benefit of our code changes, if you change for example the height of the canvas to 190 and the width to 90 and modify the ratio for the first box to 0.3. Image doing this in the code of our first example. It would be a lot tougher. The result looks like this:

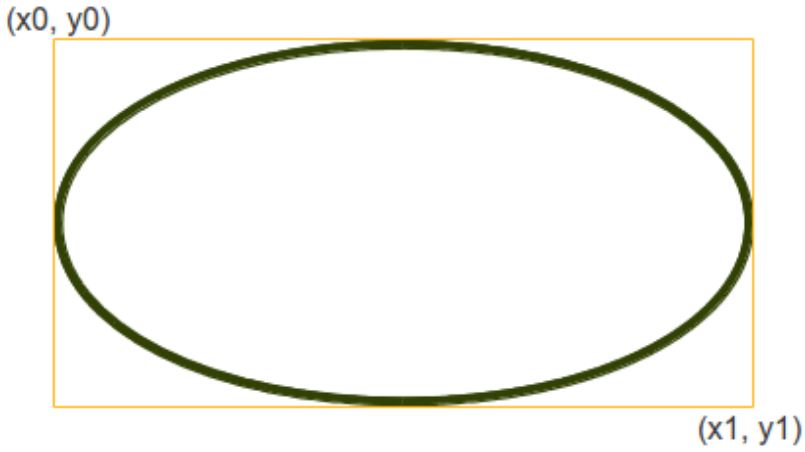


OVAL OBJECTS

An oval (or an ovoid) is any curve resembling an egg (ovum means egg in Latin). It resembles an ellipse, but it is not an ellipse. The term "oval" is not well-defined. Many different curves are called ovals, but they all have in common:

- They are differentiable, simple (not self-intersecting), convex, closed, plane curves
- They are very similar in shape to ellipses
- There is at least one axis of symmetry

The word oval stems from Latin ovum meaning "egg" and that's what it is: A figure which resembles the form of an egg. An oval is constructed from two pairs of arcs, with two different radii A circle is a special case of an oval.



We can create an oval on a canvas c with the following method:

```
id = C.create_oval ( x0, y0, x1, y1, option, ... )
```

This method returns the object ID of the new oval object on the canvas C.

The following script draws a circle around the point (75,75) with the radius 25:

```
from tkinter import *

canvas_width = 190
canvas_height =150

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

w.create_oval(50,50,100,100)

mainloop()
```

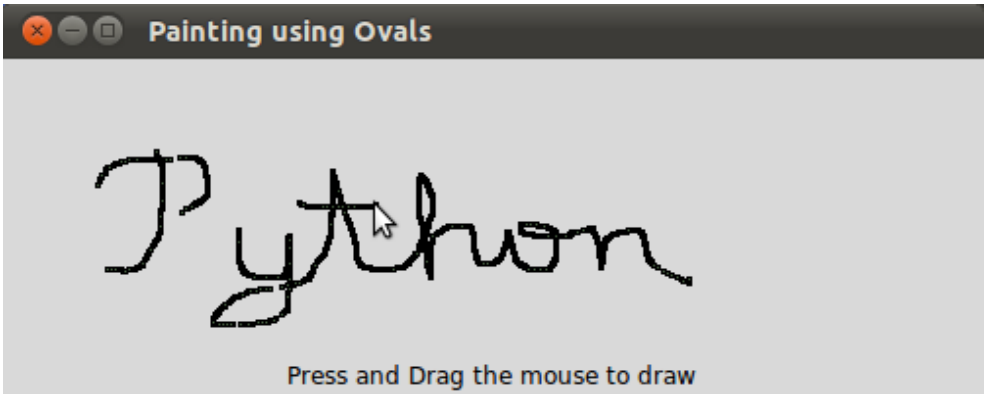
We can define a small function drawing circles by using the create_oval() method.

```
def circle(canvas,x,y, r):
   id = canvas.create_oval(x-r,y-r,x+r,y+r)
   return id
```

PAINTING INTERACTIVELY INTO A CANVAS

We want to write an application for painting or writing into a canvas. Unfortunately, there is no way to paint just one dot into a canvas. But we can overcome this problem by using a small oval:

```
from tkinter import *

canvas_width = 500
canvas_height = 150

def paint( event ):
   python_green = "#476042"
   x1, y1 = ( event.x - 1 ), ( event.y - 1 )
   x2, y2 = ( event.x + 1 ), ( event.y + 1 )
   w.create_oval( x1, y1, x2, y2, fill = python_green )

master = Tk()
master.title( "Painting using Ovals" )
w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack(expand = YES, fill = BOTH)
w.bind( "<B1-Motion>", paint )

message = Label( master, text = "Press and Drag the mouse
to draw" )
message.pack( side = BOTTOM )

mainloop()
```



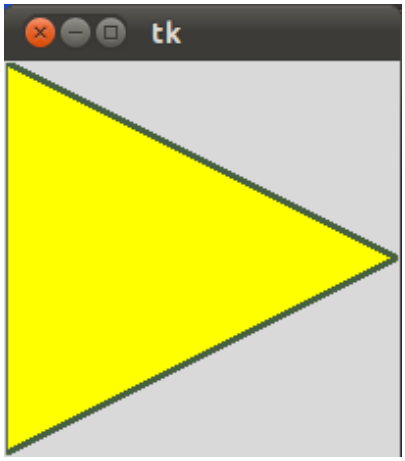
DRAWING POLYGONS

If you want to draw a polygon, you have to provide at least three coordinate points:
create_polygon(x0,y0, x1,y1, x2,y2, ...)

In the following example we draw a triangle using this method:

```
from tkinter import *

canvas_width = 200
canvas_height =200
python_green = "#476042"

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

points = [0,0,canvas_width,canvas_height/2, 0, canvas_hei
ght]
w.create_polygon(points, outline=python_green,
            fill='yellow', width=3)

mainloop()
```

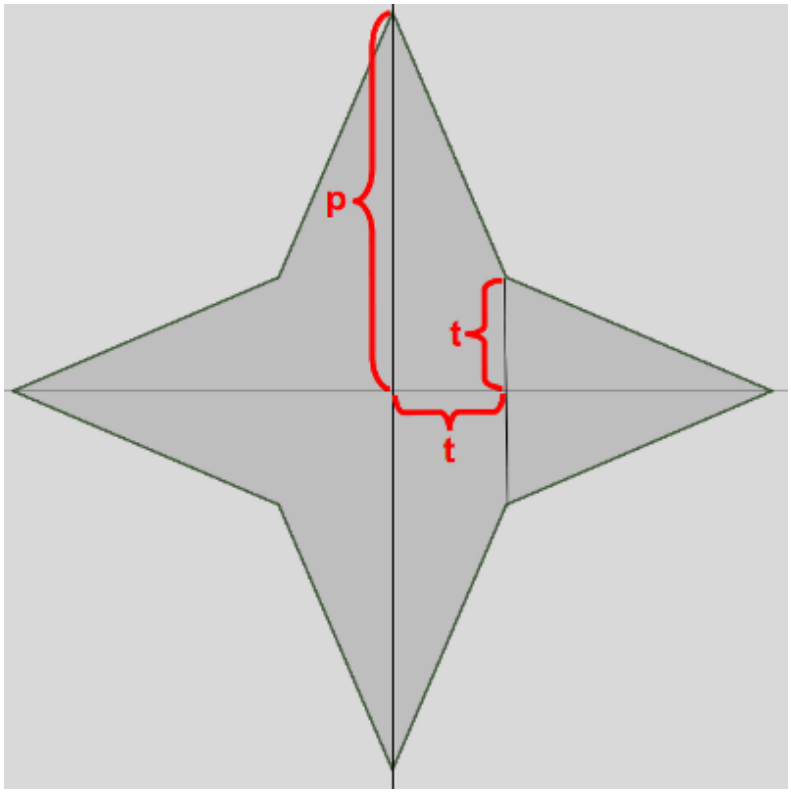
It looks like this:



When you read this, there may or not be Christmas soon, but we present a way to improve your next Christmas with some stars, created by Python and Tkinter. The first star is straight forward with hardly any programming skills involved:

```
from tkinter import *

canvas_width = 200
canvas_height =200
python_green = "#476042"

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

points = [100, 140, 110, 110, 140, 100, 110, 90, 100, 60,
90, 90, 60, 100, 90, 110]

w.create_polygon(points, outline=python_green,
            fill='yellow', width=3)

mainloop()
```

As we have mentioned, this approach is very unskilful. What if we have to change the size or the thickness of the star? We have to change all the points manually, which is of course an error-prone and tedious task to do. So, we present a new version of the previous script which involves more "programming" and programming skills. First, we put the creation of the star in a function, and we use an origin point and two lengths p and t to create the star:

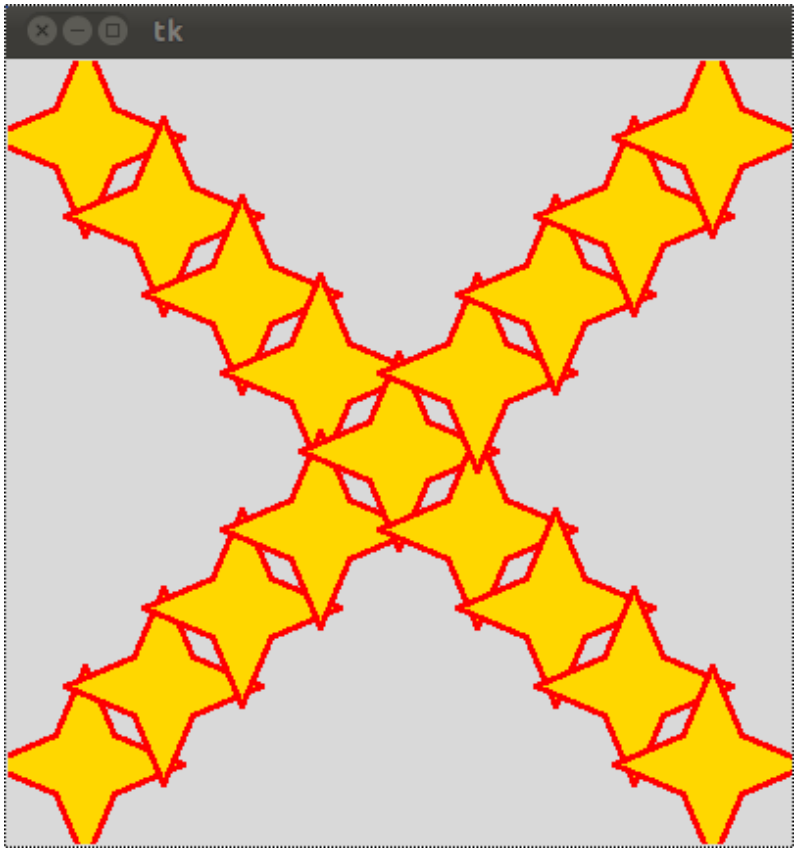


Our new improved program looks like this now:

```
from tkinter import *

canvas_width = 400
canvas_height =400
python_green = "#476042"

def polygon_star(canvas, x,y,p,t, outline=python_green, f
ill='yellow', width = 1):
   points = []
   for i in (1,-1):
      points.extend((x,        y + i*p))
      points.extend((x + i*t, y + i*t))
      points.extend((x + i*p, y))
      points.extend((x + i*t, y - i * t))

   print(points)
```

```
    canvas.create_polygon(points, outline=outline,
                          fill=fill, width=width)

master = Tk()

w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
w.pack()

p = 50
t = 15

nsteps = 10
step_x = int(canvas_width / nsteps)
step_y = int(canvas_height / nsteps)

for i in range(1, nsteps):
   polygon_star(w,i*step_x,i*step_y,p,t,outline='red',fil
l='gold', width=3)
   polygon_star(w,i*step_x,canvas_height - i*step_y,p,t,o
utline='red',fill='gold', width=3)

mainloop()
```

The result looks even more like Xmas and we are sure that nobody doubts that it would be hell to define the polygon points directly, as we did in our first star example:



BITMAPS

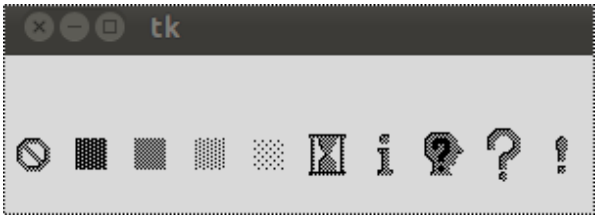
The method create_bitmap() can be be used to include a bitmap on a canvas. The following bitmaps are available on all platforms:
"error", "gray75", "gray50", "gray25", "gray12", "hourglass", "info", "questhead", "question", "warning"

The following script puts all of these bitmaps on a canvas:

```
from tkinter import *

canvas_width = 300
canvas_height =80

master = Tk()
```

```
canvas = Canvas(master,
           width=canvas_width,
           height=canvas_height)
canvas.pack()

bitmaps = ["error", "gray75", "gray50", "gray25", "gray1
2", "hourglass", "info", "questhead", "question", "warnin
g"]
nsteps = len(bitmaps)
step_x = int(canvas_width / nsteps)

for i in range(0, nsteps):
   canvas.create_bitmap((i+1)*step_x - step_x/2,50, bitma
p=bitmaps[i])

mainloop()
```

The result looks like this:



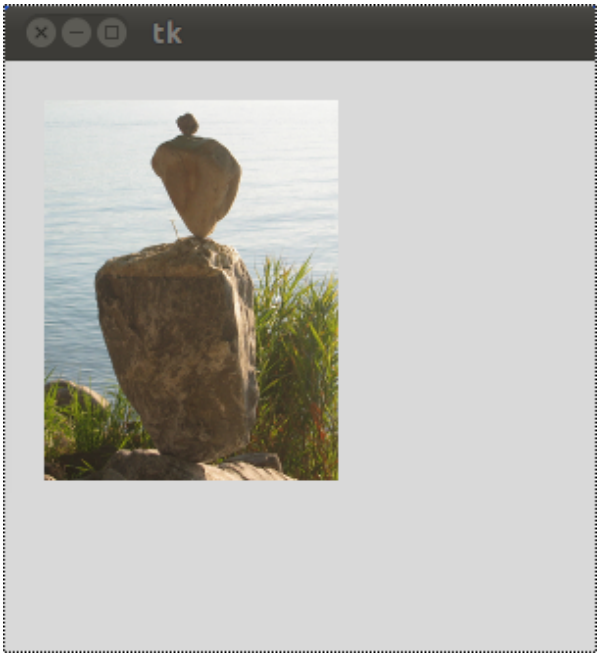
THE CANVAS IMAGE ITEM

The Canvas method create_image(x0,y0, options ...) is used to draw an image on a canvas. create_image doesn't accept an image directly. It uses an object which is created by the PhotoImage() method. The PhotoImage class can only read GIF and PGM/PPM images from files

```
from tkinter import *

canvas_width = 300
canvas_height =300

master = Tk()
```

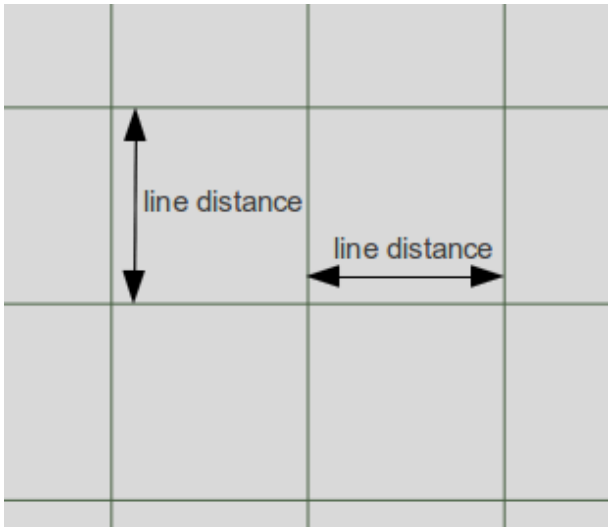
```
canvas = Canvas(master,
           width=canvas_width,
           height=canvas_height)
canvas.pack()

img = PhotoImage(file="rocks.ppm")
canvas.create_image(20,20, anchor=NW, image=img)

mainloop()
```

The window created by the previous Python script looks like this:



EXERCISE

Write a function, which draws a checkered pattern into a canvas. The function gets called with checkered(canvas, line_distance). "canvas" is the Canvas object, which will be drawn into. line_distance is the distance between the vertical and horizontal lines.

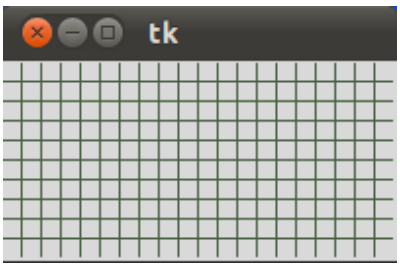


SOLUTION

```
from tkinter import *

def checkered(canvas, line_distance):
   # vertical lines at an interval of "line_distance" pix
el
   for x in range(line_distance,canvas_width,line_distanc
e):
      canvas.create_line(x, 0, x, canvas_height, fill="#4
76042")
   # horizontal lines at an interval of "line_distance" p
ixel
   for y in range(line_distance,canvas_height,line_distan
ce):
      canvas.create_line(0, y, canvas_width, y, fill="#47
6042")


master = Tk()
canvas_width = 200
canvas_height = 100
w = Canvas(master,
           width=canvas_width,
           height=canvas_height)
```

```
w.pack()

checkered(w,10)

mainloop()
```

The result of the previous script looks like this:



SLIDERS

INTRODUCTION

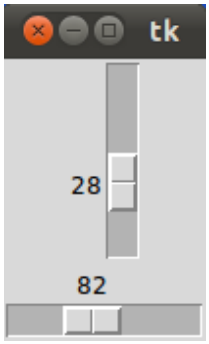
A slider is a Tkinter object with which a user can set a value by moving an indicator. Sliders can be vertically or horizontally arranged. A slider is created with the Scale method().

Using the Scale widget creates a graphical object, which allows the user to select a numerical value by moving a knob along a scale of a range of values. The minimum and maximum values can be set as parameters, as well as the resolution. We can also determine if we want the slider vertically or horizontally positioned. A Scale widget is a good alternative to an Entry widget, if the user is supposed to put in a number from a finite range, i.e. a bounded numerical value.

A SIMPLE EXAMPLE

```
from Tkinter import *

master = Tk()
w = Scale(master, from_=0, to=42)
w.pack()
w = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w.pack()

mainloop()
```

If we start this script, we get a window with a vertical and a horizontal slider:



ACCESSING SLIDER VALUES

We have demonstrated in the previous example how to create sliders. But it's not enough to have a slider, we also need a method to query it's value. We can accomplish this with the get method. We extend the previous example with a Button to view the values. If this button is pushed, the values of both sliders is printed into the terminal from which we have started the script:

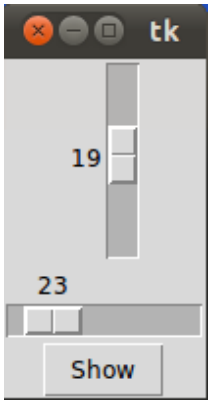
```
from Tkinter import *

def show_values():
    print (w1.get(), w2.get())

master = Tk()
w1 = Scale(master, from_=0, to=42)
w1.pack()
w2 = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w2.pack()
Button(master, text='Show', command=show_values).pack()

mainloop()
```

INITIALIZING SLIDERS

A slider starts with the minimum value, which is 0 in our examples. There is a way to initialize Sliders with the set(value) method:

```
from Tkinter import *

def show_values():
    print (w1.get(), w2.get())

master = Tk()
w1 = Scale(master, from_=0, to=42)
w1.set(19)
w1.pack()
w2 = Scale(master, from_=0, to=200, orient=HORIZONTAL)
w2.set(23)
w2.pack()
Button(master, text='Show', command=show_values).pack()

mainloop()
```

The previous script creates the following window, if it is called:



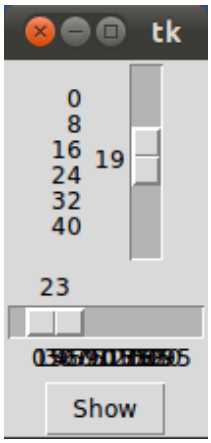
TICKINTERVAL AND LENGTH

If the option tickinterval is set to a number, the ticks of the scale will be displayed as multiples of that value. We add a tickinterval to our previous example.

```
from Tkinter import *

def show_values():
    print (w1.get(), w2.get())
```

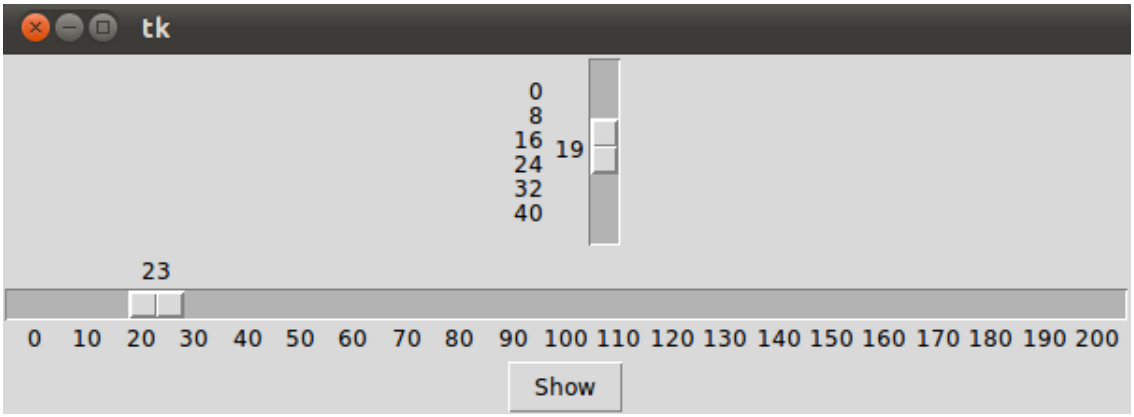
```
master = Tk()
w1 = Scale(master, from_=0, to=42, tickinterval=8)
w1.set(19)
w1.pack()
w2 = Scale(master, from_=0, to=200,tickinterval=10, orien
t=HORIZONTAL)
w2.set(23)
w2.pack()
Button(master, text='Show', command=show_values).pack()

mainloop()
```

If we start this program, we recognize that the vertical slider has the values 0, 8, 16, 24, 32, 40 added to its left side. The horizontal slider has also the numbers 0,10,20, 30, ..., but we can't see them, because the get smeared on top of each other, because the slider is not long enough:



To solve this problem we have to increase the length of our horizontal slider. We set the option length. length defines the x dimension, if the scale is horizontal and the y dimension, if the scale is vertical. So we change the definition of w2 in the following way:
w2 = Scale(master, from_=0, to=200, length=600,tickinterval=10, orient=HORIZONTAL)

The result looks like this:


tk
0
8
16
24
32
40
19
23
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
Show

TEXT WIDGETS

INTRODUCTION AND SIMPLE EXAMPLES

A text widget is used for multi-line text area. The tkinter text widget is very powerful and flexible and can be used for a wide range of tasks. Though one of the main purposes is to provide simple multi-line areas, as they are often used in forms, text widgets can also be used as simple text editors or even web browsers.

Furthermore, text widgets can be used to display links, images, and HTML, even using CSS styles.

In most other tutorials and text books, it's hard to find a very simple and basic example of a text widget. That's why we want to start our chapter with a such an example:

We create a text widget by using the Text() method. We set the height to 2, i.e. two lines and the width to 30, i.e. 30 characters. We can apply the method insert() on the object T, which the Text() method had returned, to include text. We add two lines of text.

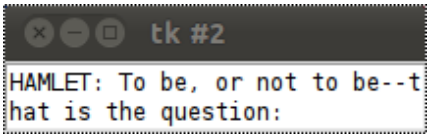
```
import tkinter as tk

root = tk.Tk()
T = tk.Text(root, height=2, width=30)
T.pack()
T.insert(tk.END, "Just a text Widget\nin two lines\n")
tk.mainloop()
```

The result should not be very surprising:

Let's change our little example a tiny little bit. We add another text, the beginning of the famous monologue from Hamlet:

```
import tkinter as tk

root = tk.Tk()
T = tk.Text(root, height=10, width=30)
T.pack()
quote = """HAMLET: To be, or not to be--that is the quest
ion:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
Or to take arms against a sea of troubles
And by opposing end them. To die, to sleep--
No more--and by a sleep to say we end
The heartache, and the thousand natural shocks
That flesh is heir to. 'Tis a consummation
Devoutly to be wished."""
T.insert(tk.END, quote)
tk.mainloop()
```

If we start our little script, we get a very unsatisfying result. We can see in the window only the first line of the monologue and this line is even broken into two lines. We can see only two lines in our window, because we set the height to the value 2. Furthermore, the width is set to 30, so tkinter has to break the first line of the monologue after 30 characters.



One solution to our problem consists in setting the height to the number of lines of our monologue and set width wide enough to display the widest line completely.

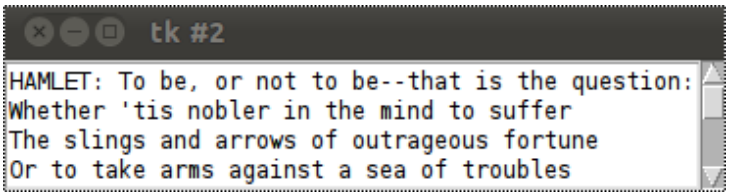
But there is a better technique, which you are well acquainted with from your browser and other applications: scrolling

SCROLLBARS

So let's add a scrollbar to our window. To this purpose, Tkinter provides the Scrollbar() method. We call it with the root object as the only parameter.

```
import tkinter as tk

root = tk.Tk()
S = tk.Scrollbar(root)
T = tk.Text(root, height=4, width=50)
S.pack(side=tk.RIGHT, fill=tk.Y)
T.pack(side=tk.LEFT, fill=tk.Y)
S.config(command=T.yview)
T.config(yscrollcommand=S.set)
quote = """HAMLET: To be, or not to be--that is the quest
ion:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
Or to take arms against a sea of troubles
And by opposing end them. To die, to sleep--
No more--and by a sleep to say we end
The heartache, and the thousand natural shocks
That flesh is heir to. 'Tis a consummation
Devoutly to be wished."""
T.insert(tk.END, quote)
tk.mainloop()
```

The result is a lot better. We have now always 4 lines in view, but all lines can be viewed by using the scrollbar on the right side of the window:

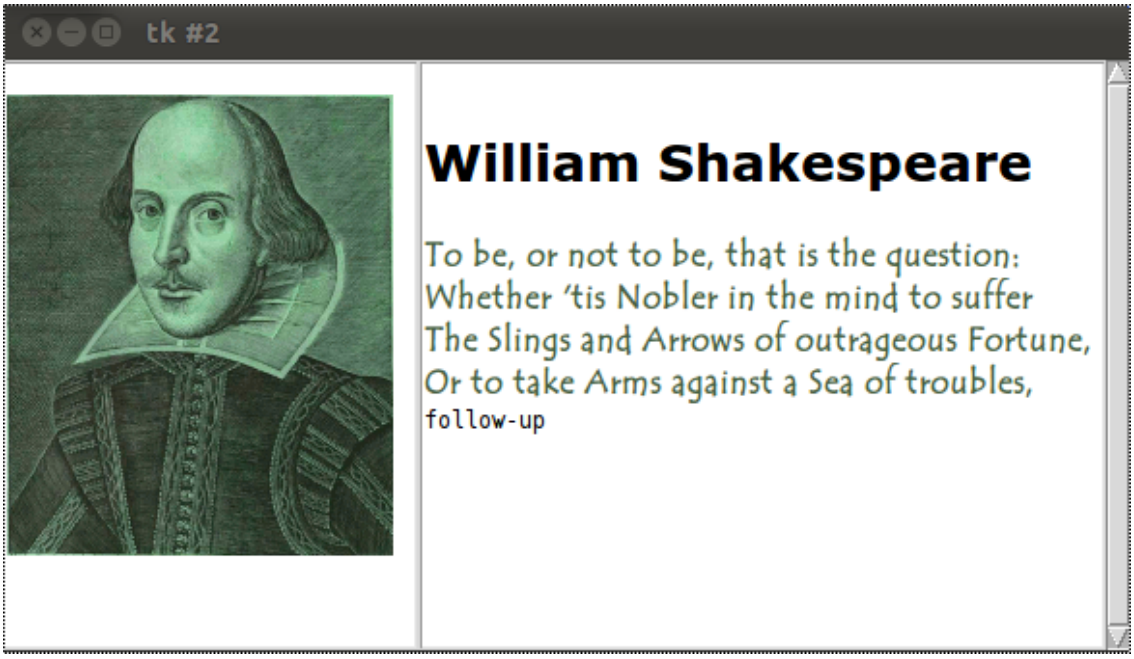


TEXT WIDGET WITH IMAGE

In our next example, we add an image to the text and bind a command to a text line:

```
import tkinter as tk
```

```
root = tk.Tk()

text1 = tk.Text(root, height=20, width=30)
photo = tk.PhotoImage(file='./William_Shakespeare.gif')
text1.insert(tk.END, '\n')
text1.image_create(tk.END, image=photo)

text1.pack(side=tk.LEFT)

text2 = tk.Text(root, height=20, width=50)
scroll = tk.Scrollbar(root, command=text2.yview)
text2.configure(yscrollcommand=scroll.set)
text2.tag_configure('bold_italics', font=('Arial', 12, 'b
old', 'italic'))
text2.tag_configure('big', font=('Verdana', 20, 'bold'))
text2.tag_configure('color',
                    foreground='#476042',
                    font=('Tempus Sans ITC', 12, 'bold'))
text2.tag_bind('follow',
               '<1>',
               lambda e, t=text2: t.insert(tk.END, "Not n
ow, maybe later!"))
text2.insert(tk.END,'\nWilliam Shakespeare\n', 'big')
quote = """
To be, or not to be that is the question:
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune,
Or to take Arms against a Sea of troubles,
"""
text2.insert(tk.END, quote, 'color')
text2.insert(tk.END, 'follow-up\n', 'follow')
text2.pack(side=tk.LEFT)
scroll.pack(side=tk.RIGHT, fill=tk.Y)

root.mainloop()
```

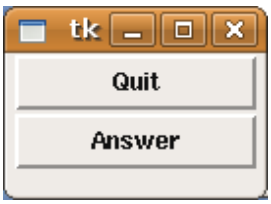

tk #2
William Shakespeare
To be, or not to be, that is the question:
Whether 'tis Nobler in the mind to suffer
The Slings and Arrows of outrageous Fortune,
Or to take Arms against a Sea of troubles,
follow-up

DIALOGUES AND MESSAGE BOXES

INTRODUCTION

Tkinter (and TK of course) provides a set of dialogues (dialogs in American English spelling), which can be used to display message boxes, showing warning or errors, or widgets to select files and colours. There are also simple dialogues, asking the user to enter string, integers or float numbers.

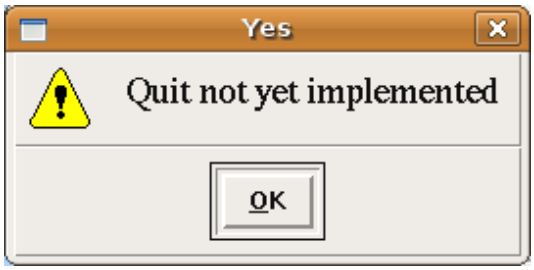
Let's look at a typical GUI Session with Dialogues and Message boxes. There might be a button starting the dialogue, like the "quit" button in the following window:



Pushing the "quit" button raises the Verify window:

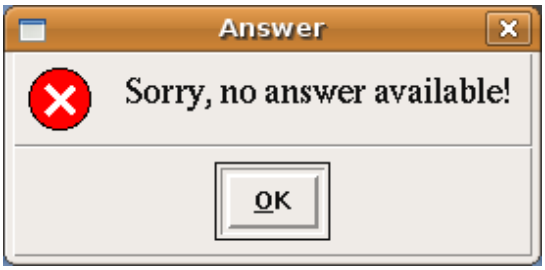
Let's assume that we want to warn users that the "quit" functionality is not yet implemented. In this case we can use the warning message to inform the user, if he or she pushes the "yes"

button:



If somebody types the "No" button, the "Cancel" message box is raised:

Let's go back to our first Dialogue with the "quit" and "answer" buttons. If the "Answer" functionality is not implemented, it might be useful to use the following error message box:



Python script, which implements the previous dialogue widges:

```
import tkinter as tk
from tkinter import messagebox as mb
```

```
def answer():
    mb.showerror("Answer", "Sorry, no answer available")

def callback():
    if mb.askyesno('Verify', 'Really quit?'):
        mb.showwarning('Yes', 'Not yet implemented')
    else:
        mb.showinfo('No', 'Quit has been cancelled')

tk.Button(text='Quit', command=callback).pack(fill=tk.X)
tk.Button(text='Answer', command=answer).pack(fill=tk.X)
tk.mainloop()
```

MESSAGE BOXES

The message dialogues are provided by the 'messagebox' submodule of tkinter.

'messagebox' consists of the following functions, which correspond to dialog windows:

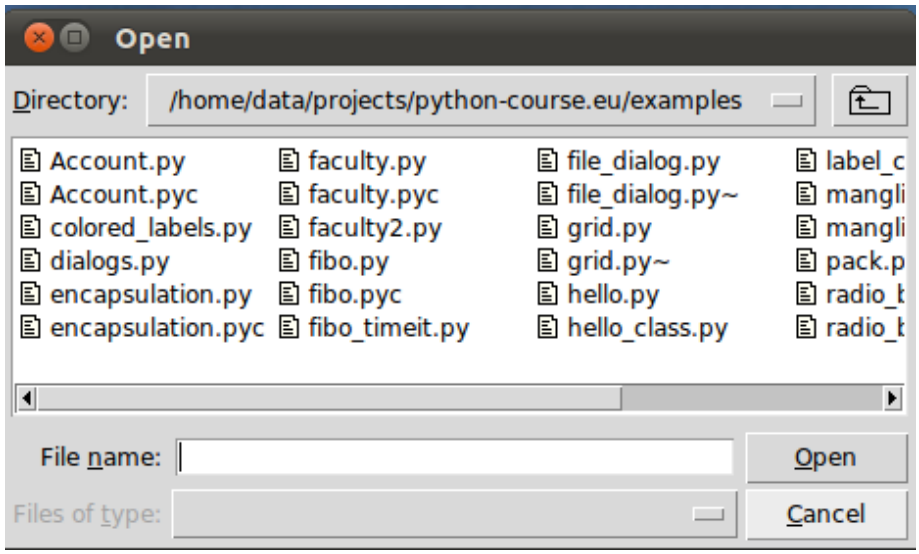
- askokcancel(title=None, message=None, **options)
  Ask if operation should proceed; return true if the answer is ok
- askquestion(title=None, message=None, **options)
  Ask a question
- askretrycancel(title=None, message=None, **options)
  Ask if operation should be retried; return true if the answer is yes
- askyesno(title=None, message=None, **options)
  Ask a question; return true if the answer is yes
- askyesnocancel(title=None, message=None, **options)
  Ask a question; return true if the answer is yes, None if cancelled.
- showerror(title=None, message=None, **options)
  Show an error message
- showinfo(title=None, message=None, **options)
  Show an info message
- showwarning(title=None, message=None, **options)
  Show a warning message

OPEN FILE DIALOGUE

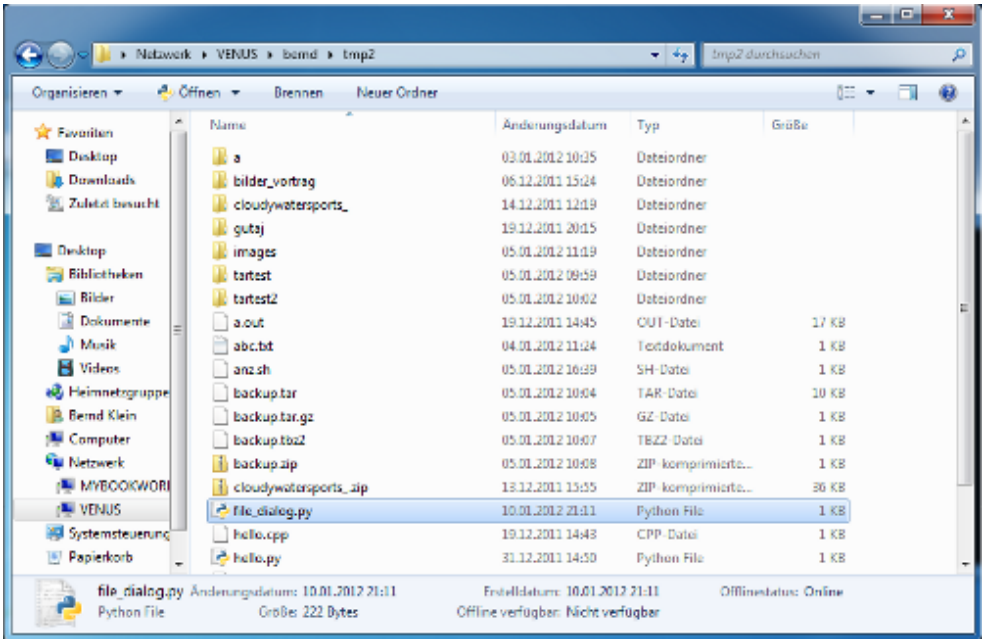
There is hardly any serious application, which doesn't need a way to read from a file or write to a file. Furthermore, such an application might have to choose a directory. Tkinter provides the module tkFileDialog for these purposes.

```
import tkinter as tk
from tkinter import filedialog as fd

def callback():
    name= fd.askopenfilename()
    print(name)

errmsg = 'Error!'
tk.Button(text='File Open',
       command=callback).pack(fill=tk.X)
tk.mainloop()
```

The code above creates a window with a single button with the text "File Open". If the button is pushed, the following window appears:



The look-and-feel of the file-open-dialog depends on the GUI of the operating system. The above example was created using a gnome desktop under Linux. If we start the same program under Windows 7, it looks like this:



CHOOSING A COLOUR

There are applications where the user should have the possibility to select a colour. Tkinter provides a pop-up menu to choose a colour. To this purpose we have to import the 'tkinter.colorchooser' module and have to use the method askColor:

```
result = tkinter.colorchooser.askcolor ( color, option=value, ...)
```

If the user clicks the OK button on the pop-up window, respectively, the return value of askcolor() is a tuple with two elements, both a representation of the chosen colour, e.g. ((106, 150, 98), '#6a9662')
The first element return[0] is a tuple (R, G, B) with the RGB representation in decimal values (from 0 to 255). The second element return[1] is a hexadecimal representation of the chosen colour.
If the user clicks "Cancel" the method returns the tuple (None, None).

The optional keyword parameters are:

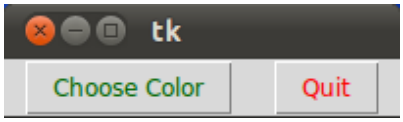
color	The variable color is used to set the default colour to be displayed. If color is not set, the initial colour will be grey.
title	The text assigned to the variable title will appear in the pop-up window's title area. The default title is "Color".

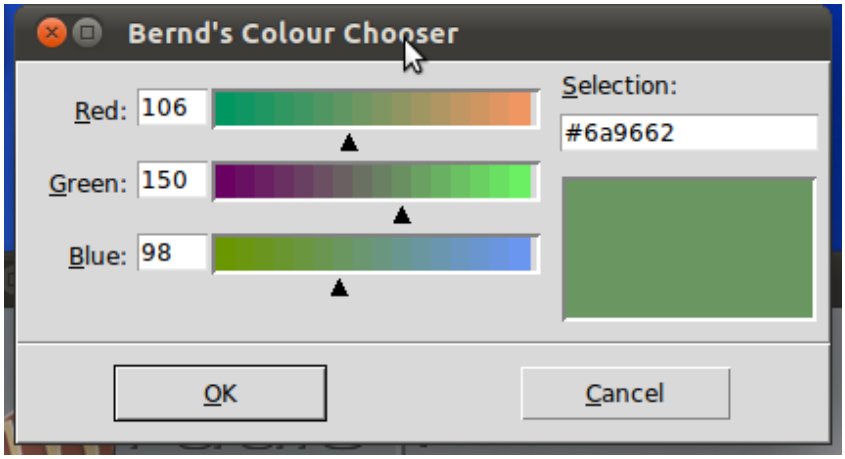
parent	Make the pop-up window appear over window W. The default behaviour is that it appears over the root window.

Let's have a look at an example:

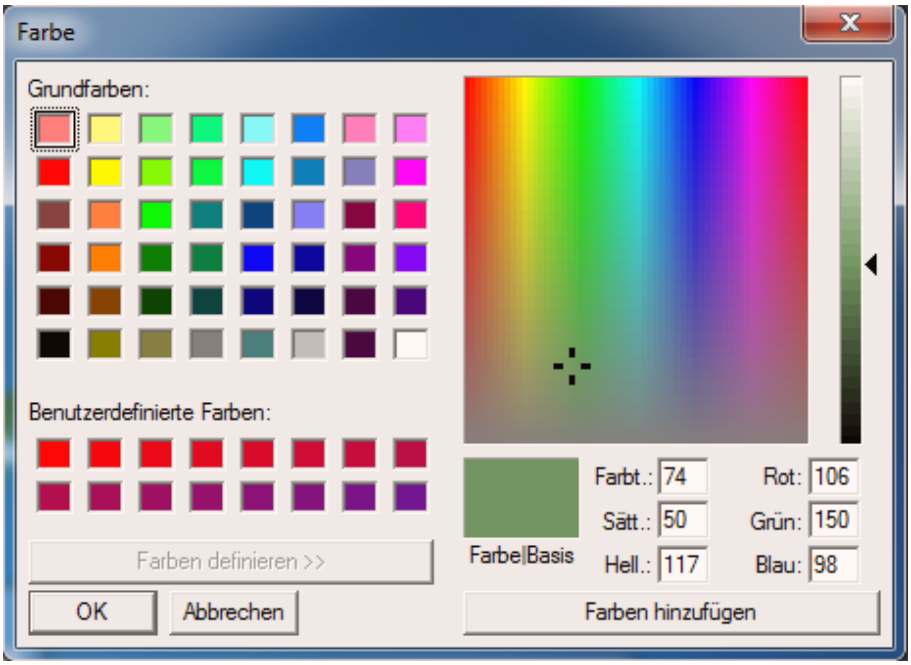
```
import tkinter as tk
from tkinter.colorchooser import askcolor

def callback():
    result = askcolor(color="#6A9662",
                      title = "Bernd's Colour Chooser")
    print(result)

root = tk.Tk()
tk.Button(root,
          text='Choose Color',
          fg="darkgreen",
          command=callback).pack(side=tk.LEFT, padx=10)
tk.Button(text='Quit',
          command=root.quit,
          fg="red").pack(side=tk.LEFT, padx=10)
tk.mainloop()
```

The look and feel depends on the operating system (e.g. Linux or Windows) and the chosen GUI (GNOME, KDE and so on). The following windows appear, if you use Gnome:





Using the same script under Windows 7 gives us the following result:



LAYOUT MANAGERS / GEOMETRY MANAGER

INTRODUCTION

In this chapter of our Python-Tkinter tutorial we will introduce the layout managers or geometry managers, as they are sometimes called as well. Tkinter possess three layout managers:

- pack
- grid
- place

The three layout managers pack, grid, and place should never be mixed in the same master window! Geometry managers serve various functions. They:

- arrange widgets on the screen
- register widgets with the underlying windowing system
- manage the display of widgets on the screen

Arranging widgets on the screen includes determining the size and position of components. Widgets can provide size and alignment information to geometry managers, but the geometry managers has always the final say on the positioning and sizing.

PACK

Pack is the easiest to use of the three geometry managers of Tk and Tkinter. Instead of having to declare precisely where a widget should appear on the display screen, we can declare the positions of widgets with the pack command relative to each other. The pack command takes care of the details. Though the pack command is easier to use, this layout managers is limited in its possibilities compared to the grid and place mangers. For simple applications it is definitely the manager of choice. For example simple applications like placing a number of widgets side by side, or on top of each other.

Example:

```
import tkinter as tk

root = tk.Tk()

w = tk.Label(root, text="Red Sun", bg="red", fg="white")
w.pack(fill=tk.X)
w = tk.Label(root, text="Green Grass", bg="green", fg="bl
ack")
w.pack(fill=tk.X)
w = tk.Label(root, text="Blue Sky", bg="blue", fg="whit
e")
w.pack(fill=tk.X)

tk.mainloop()
```

FILL OPTION

In our example, we have packed three labels into the parent widget "root". We used pack() without any options. So pack had to decide which way to arrange the labels. As you can see, it has chosen to place the label widgets on top of each other and centre them. Furthermore, we can see that each label has been given the size of the text. If you want to make the widgets as wide as the parent widget, you have to use the fill=X option:

```
import tkinter as tk

root = tk.Tk()

w = tk.Label(root, text="Red Sun", bg="red", fg="white")
w.pack(fill=tk.X)
w = tk.Label(root, text="Green Grass", bg="green", fg="bl
ack")
w.pack(fill=tk.X)
w = tk.Label(root, text="Blue Sky", bg="blue", fg="whit
e")
w.pack(fill=tk.X)
```

```
tk.mainloop()
```

PADDING

The pack() manager knows four padding options, i.e. internal and external padding and padding in x and y direction:

External padding, horizontally

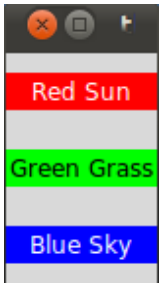
The code for the window above:

padx

```
import tkinter as tk

root = tk.Tk()
w = tk.Label(root, text="Red Sun", bg="red", fg="whit
e")
w.pack(fill=tk.X, padx=10)
w = tk.Label(root, text="Green Grass", bg="green", fg
="black")
w.pack(fill=tk.X, padx=10)
w = tk.Label(root, text="Blue Sky", bg="blue", fg="whit
e")
w.pack(fill=tk.X, padx=10)
tk.mainloop()
```

pady

External padding, vertically



The code for the window above:

```
import tkinter as tk

root = tk.Tk()
w = tk.Label(root, text="Red Sun", bg="red", fg="white")
w.pack(fill=tk.X, pady=10)
w = tk.Label(root, text="Green Grass", bg="green", fg="black")
w.pack(fill= tk.X, pady=10)
w = tk.Label(root, text="Blue Sky", bg="blue", fg="white")
w.pack(fill=tk.X, pady=10)
tk.mainloop()
```

ipadx Internal padding, horizontally.

In the following example, we change only the label with the text "Green Grass", so that the result can be easier recognized. We have also taken out the fill option.


```
import tkinter as tk

root = tk.Tk()
w = tk.Label(root, text="Red Sun", bg="red", fg="white")
w.pack()
w = tk.Label(root, text="Green Grass", bg="green", fg="black")
w.pack(ipadx=10)
w = tk.Label(root, text="Blue Sky", bg="blue", fg="whit
```

```
e")
w.pack()
tk.mainloop()
```

Internal padding, vertically

We will change the last label of our previous example to ipady=10.

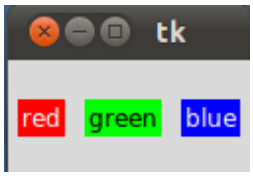
ipady

```
import tkinter as tk

root = tk.Tk()
w = tk.Label(root, text="Red Sun", bg="red", fg="whit
e")
w.pack()
w = tk.Label(root, text="Green Grass", bg="green", fg
="black")
w.pack(ipadx=10)
w = tk.Label(root, text="Blue Sky", bg="blue", fg="whit
e")
w.pack(ipady=10)
tk.mainloop()
```

The default value in all cases is 0.

PLACING WIDGETS SIDE BY SIDE

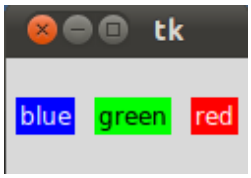
We want to place the three label side by side now and shorten the text slightly:



The corresponding code looks like this:

```
import tkinter as tk

root = tk.Tk()

w = tk.Label(root, text="red", bg="red", fg="white")
w.pack(padx=5, pady=10, side=tk.LEFT)
w = tk.Label(root, text="green", bg="green", fg="black")
w.pack(padx=5, pady=20, side=tk.LEFT)
w = tk.Label(root, text="blue", bg="blue", fg="white")
w.pack(padx=5, pady=20, side=tk.LEFT)
tk.mainloop()
```

If we change LEFT to RIGHT in the previous example, we get the colours in reverse order:



PLACE GEOMETRY MANAGER

The Place geometry manager allows you explicitly set the position and size of a window, either in absolute terms, or relative to another window. The place manager can be accessed through the place method. It can be applied to all standard widgets.

We use the place geometry manager in the following example. We are playing around with colours in this example, i.e. we assign to every label a different colour, which we randomly create using the randrange method of the random module. We calculate the brightness (grey value) of each colour. If the brightness is less than 120, we set the foreground colour (fg) of the label to White otherwise to black, so that the text can be easier read.

```
import tkinter as tk
import random

root = tk.Tk()
```

```
# width x height + x_offset + y_offset:
root.geometry("170x200+30+30")

languages = ['Python','Perl','C++','Java','Tcl/Tk']
labels = range(5)
for i in range(5):
   ct = [random.randrange(256) for x in range(3)]
   brightness = int(round(0.299*ct[0] + 0.587*ct[1] + 0.1
14*ct[2]))
   ct_hex = "%02x%02x%02x" % tuple(ct)
   bg_colour = '#' + "".join(ct_hex)
   l = tk.Label(root,
                text=languages[i],
                fg='White' if brightness < 120 else 'Blac
k',
                bg=bg_colour)
   l.place(x = 20, y = 30 + i*30, width=120, height=25)

root.mainloop()
```

GRID MANAGER

The first geometry manager of Tk had been pack. The algorithmic behaviour of pack is not easy to understand and it can be difficult to change an existing design. Grid was introduced in 1996 as an alternative to pack. Though grid is easier to learn and to use and produces nicer layouts, lots of developers keep using pack.

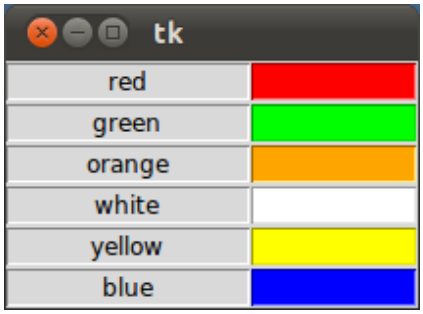
Grid is in many cases the best choice for general use. While pack is sometimes not sufficient for changing details in the layout, place gives you complete control of positioning each element, but this makes it a lot more complex than pack and grid.

The Grid geometry manager places the widgets in a 2-dimensional table, which consists of a number of rows and columns. The position of a widget is defined by a row and a column number. Widgets with the same column number and different row numbers will be above or below each other. Correspondingly, widgets with the same row number but different column numbers will be on the same "line" and will be beside of each other, i.e. to the left or the right.

Using the grid manager means that you create a widget, and use the grid method to tell the manager in which row and column to place them. The size of the grid doesn't have to be defined, because the manager automatically determines the best dimensions for the widgets used.

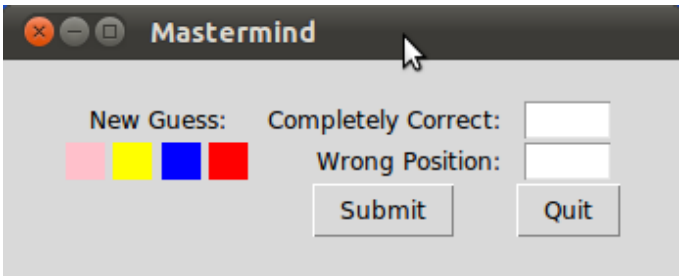
EXAMPLE WITH GRID

```
import tkinter as tk

colours = ['red','green','orange','white','yellow','blu
e']

r = 0
for c in colours:
    tk.Label(text=c, relief=tk.RIDGE, width=15).grid(row=
r,column=0)
    tk.Entry(bg=c, relief=tk.SUNKEN, width=10).grid(row=
r,column=1)
    r = r + 1

tk.mainloop()
```



MASTERMIND / BULLS AND COWS

IMPLEMENTATION IN PYTHON USING TKINTER



In this chapter of our advanced Python topics we present an implementation of the game Bulls and Cows using Tkinter as the GUI. 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.

The Algorithm is explained in detail in our chapter "Mastermind / Bulls and Cows" in Advanced Topics. You can also find the code for the module combinatorics.

THE CODE FOR MASTERMIND

```
from tkinter import *
from tkinter.messagebox import *
import random

from combinatorics import all_colours

def inconsistent(p, guesses):
    """ the function checks, if a permutation p, i.e. a li
st of
colours like p = ['pink', 'yellow', 'green', 'red'] is co
nsistent
with the previous colours. Each previous colour permuatio
n guess[0]
```

```
compared (check()) with p has to return the same amount o
f blacks
(rightly positioned colours) and whites (right colour at
wrong
position) as the corresponding evaluation (guess[1] in th
e
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 evaulation given by the human playe
r makes
sense. 3 blacks and 1 white make no sense, for example.
"""
   (rightly_positioned, permutated) = a
   if (rightly_positioned + permutated > number_of_positi
ons) \
       or (rightly_positioned + permutated < len(colours)
- number_of_positions):
      return False
   if rightly_positioned == 3 and permutated == 1:
      return False
   return True

def get_evaluation():
   """ get evaluation from entry fields """
   rightly_positioned = int(entryWidget_both.get())
   permutated = int(entryWidget_only_colours.get())
   return (rightly_positioned, permutated)

def new_evaluation(current_colour_choices):
   """ This funtion gets an evaluation of the current gue
ss, checks
the consistency of this evaluation, adds the guess togeth
er with
the evaluation to the list of guesses, shows the previous
guesses
and creates a ne guess """
   rightly_positioned, permutated = get_evaluation()
   if rightly_positioned == number_of_positions:
      return(current_colour_choices, (rightly_positioned,
permutated))
```

```
   if not answer_ok((rightly_positioned, permutated)):
      print("Input Error: Sorry, the input makes no sens
e")
      return(current_colour_choices, (-1, permutated))
   guesses.append((current_colour_choices, (rightly_posit
ioned, permutated)))
   view_guesses()

   current_colour_choices = create_new_guess()
   show_current_guess(current_colour_choices)
   if not current_colour_choices:
      return(current_colour_choices, (-1, permutated))
   return(current_colour_choices, (rightly_positioned, pe
rmutated))


def check(p1, p2):
   """ check() calcualtes the number of bulls (blacks) an
d 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


def new_evaluation_tk():
   global current_colour_choices
   res = new_evaluation(current_colour_choices)
```

```
   current_colour_choices = res[0]

def show_current_guess(new_guess):
    row = 1
    Label(root, text="   New Guess:   ").grid(row=row,
                                        column=0,
                                        columnspan=4)
    row +=1
    col_count = 0
    for c in new_guess:
         print(c)
         l = Label(root, text="    ", bg=c)
         l.grid(row=row,column=col_count,  sticky=W, padx
=2)
         col_count += 1

def view_guesses():
    row = 3
    Label(root, text="Old Guesses").grid(row=row,
                                         column=0,
                                         columnspan=4)
    Label(root, text="c&p").grid(row=row,
                                 padx=5,
                                 column=number_of_positio
ns + 1)
    Label(root, text="p").grid(row=row,
                               padx=5,
                               column=number_of_positions
+ 2)
    # dummy label for distance:
    Label(root, text="         ").grid(row=row,
                                       column=number_of_p
ositions + 3)


    row += 1
    # vertical dummy label for distance:
    Label(root, text="             ").grid(row=row,
                                           column=0,
                                           columnspan=5)

    for guess in guesses:
      guessed_colours = guess[0]
      col_count = 0
      row += 1
      for c in guessed_colours:
         print(guessed_colours[col_count])
```

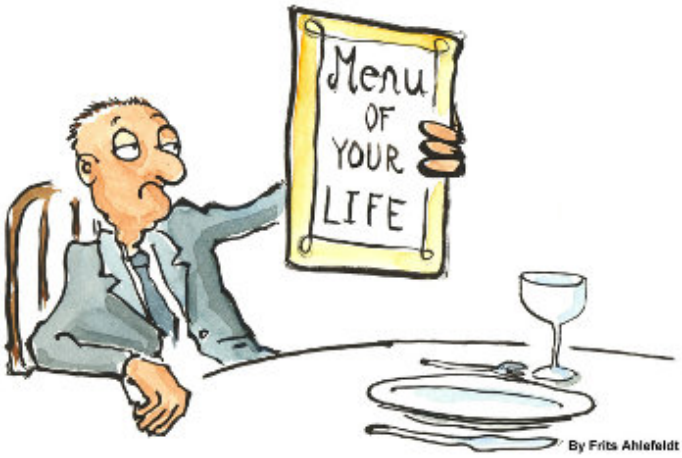
```
            l = Label(root, text="    ", bg=guessed_colours
[col_count])
            l.grid(row=row,column=col_count,  sticky=W, padx
=2)
            col_count += 1
         # evaluation:
         for i in (0,1):
           l = Label(root, text=str(guess[1][i]))
           l.grid(row=row,column=col_count + i + 1, padx=2)



if __name__ == "__main__":
   colours = ["red","green","blue","yellow","orange","pin
k"]
   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) )

   row_offset = 1
   root = Tk()
   root.title("Mastermind")
   root["padx"] = 30
   root["pady"] = 20

   entryLabel = Label(root)
   entryLabel["text"] = "Completely Correct:"
   entryLabel.grid(row=row_offset,
                sticky=E,
                padx=5,
                column=number_of_positions + 4)
   entryWidget_both = Entry(root)
   entryWidget_both["width"] = 5
   entryWidget_both.grid(row=row_offset, column=number_of
_positions + 5)

   entryLabel = Label(root)
   entryLabel["text"] = "Wrong Position:"
   entryLabel.grid(row=row_offset+1,
                sticky=E,
                padx=5,
                column= number_of_positions + 4)
```

```
    entryWidget_only_colours = Entry(root)
    entryWidget_only_colours["width"] = 5
    entryWidget_only_colours.grid(row=row_offset+1, column
=number_of_positions + 5)



    submit_button = Button(root, text="Submit", command=ne
w_evaluation_tk)
    submit_button.grid(row=4,column=number_of_positions +
4)

    quit_button = Button(root, text="Quit", command=root.q
uit)
    quit_button.grid(row=4,column=number_of_positions + 5)
    show_current_guess(current_colour_choices)


    root.mainloop()
```

MENUS

INTRODUCTION

Most people, if confronted with the word "menu", will immediately think of a menu in a restaurant. Even though the menu of a restaurant and the menu of a computer program have at first glance nothing in common, we can see that yet the have a lot in common. In a restaurant, a menu is a presentation of all their food and beverage offerings, while in a computer application it presents all the commands and functions of the application, which are available to the user via the grafical user interface.



Menus in GUIs are presented with a combination of text and symbols to represent the choices. Selecting with the mouse (or finger on touch screens) on one of the symbols or text, an action will be started. Such an action or operation can, for example, be the opening or saving of a file, or the quitting or exiting of an application.

A context menu is a menu in which the choices presented to the user are modified according to the current context in which the user is located.

We introduce in this chapter of our Python Tkinter tutorial the pull-down menus of Tkinter, i.e. the lists at the top of the windows, which appear (or pull down), if you click on an item like, for example "File", "Edit" or "Help".

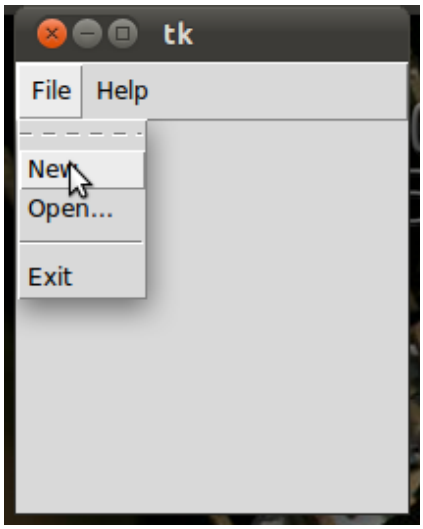
A SIMPLE MENU EXAMPLE

The following Python script creates a simple application window with menus.

```
from Tkinter import *
from tkFileDialog   import askopenfilename
```

```
def NewFile():
    print "New File!"
def OpenFile():
    name = askopenfilename()
    print name
def About():
    print "This is a simple example of a menu"

root = Tk()
menu = Menu(root)
root.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)
filemenu.add_command(label="New", command=NewFile)
filemenu.add_command(label="Open...", command=OpenFile)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=root.quit)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
helpmenu.add_command(label="About...", command=About)

mainloop()
```

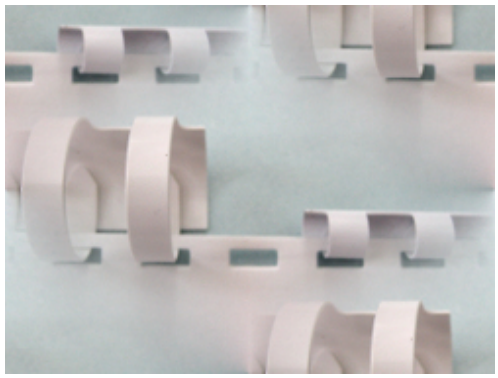
It looks like this, if started:



EVENTS AND BINDS

INTRODUCTION

A Tkinter application runs most of its time inside an event loop, which is entered via the mainloop method. It waiting for events to happen. Events can be key presses or mouse operations by the user.



Tkinter provides a mechanism to let the programmer deal with events. For each widget, it's possible to bind Python functions and methods to an event.

widget.bind(event, handler)

If the defined event occurs in the widget, the "handler" function is called with an event object. describing the event.

```
#!/usr/bin/python3
# write tkinter as Tkinter to be Python 2.x compatible
from tkinter import *
def hello(event):
    print("Single Click, Button-l")
def quit(event):
    print("Double Click, so let's stop")
    import sys; sys.exit()

widget = Button(None, text='Mouse Clicks')
widget.pack()
widget.bind('<Button-1>', hello)
widget.bind('<Double-1>', quit)
widget.mainloop()
```

Let's have another simple example, which shows how to use the motion event, i.e. if the mouse is moved inside of a widget:

```
from tkinter import *

def motion(event):
```

```
    print("Mouse position: (%s %s)" % (event.x, event.y))
    return

master = Tk()
whatever_you_do = "Whatever you do will be insignificant,
but it is very important that you do
it.\n(Mahatma Gandhi)"
msg = Message(master, text = whatever_you_do)
msg.config(bg='lightgreen', font=('times', 24, 'italic'))
msg.bind('<Motion>',motion)
msg.pack()
mainloop()
```

Every time we move the mouse in the Message widget, the position of the mouse pointer will be printed. When we leave this widget, the function motion() is not called anymore.

EVENTS

Tkinter uses so-called event sequences for allowing the user to define which events, both specific and general, he or she wants to bind to handlers. It is the first argument "event" of the bind method. The event sequence is given as a string, using the following syntax:

```
<modifier-type-detail>
```

The type field is the essential part of an event specifier, whereas the "modifier" and "detail" fields are not obligatory and are left out in many cases. They are used to provide additional information for the chosen "type". The event "type" describes the kind of event to be bound, e.g. actions like mouse clicks, key presses or the widget got the input focus.

Event	Description
<Button>	A mouse button is pressed with the mouse pointer over the widget. The detail part specifies which button, e.g. The left mouse button is defined by the event <Button-1>, the middle button by <Button-2>, and the rightmost mouse button by <Button-3>. <Button-4> defines the scroll up event on mice with wheel support and and <Button-5> the scroll down.
	If you press down a mouse button over a widget and keep it pressed, Tkinter will automatically "grab" the mouse pointer. Further mouse events like Motion and Release events will be sent to the current widget, even if the mouse is moved outside the current widget. The current position,