The Pythonist Manifesto

About the revolutionary project of Pythonism to abolish the totalitarian regime of inappropriate programming languages and techniques

Jonas Schulte-Coerne

February 16, 2015

Contents

1 Software Architecture

1.1 Maybe a list of common slogans

TODO
explain what these slogans mean in practical use (or just remove this section)

- ▷ Do only one thing and do it well
- ▷ Don't program a mathematical function yourself
- ▷ If in doubt, leave it out
- ▷ Program to an API, not an implementation
- ▷ Explicit is better than implicit

1.2 Global state

TODO

- ▷ global constants are good
- ▷ global variables are bad
- ▷ mutable singletons are bad, since they are global variables
- ▷ modules are singletons, so don't change their state. Or don't rely on their state (e.g. the atexit module has to be changed when used).
- ▷ IO is usually a global state

- ▷ something about testability/provability and functional programming languages. (A function is expected to yield the same results for the same parameters. If it also depends on some global state, this expectation is not met.)

1.3 Mutable vs. Immutable data types

A mutable object is an object, whose data can be modified after the object has been fully initialized. Immutable objects on the other hand get all their data through their constructor and cannot be changed afterwards.
Since immutable objects cannot be modified, processed versions of those objects have to be stored in new objects. In-place processing of an immutable object is not possible. The performance penalty for this is often far lower than expected, so choosing an object to be mutable, that could (or should) be immutable, might only be necessary on systems with extremely tight memory restrictions. Immutability also avoids the necessity to copy the object, each time it is passed to another part of the software, because the object cannot be modified unexpectedly and therefore passing a reference is sufficient. This usually gives tremendous speedups and much lower memory consumption compared to a mutable implementation.

Generally, an object should be mutable, if the object is defined by its instance, rather than its data. A character in a computer game is still the same character, even after it has been moved to a different position (the position is part of the character object's data). So it is a sensible choice to make the character object mutable.

If an object is only defined by its data, like for example a signal, it should be implemented as an immutable object. Consider the following example for an explaination why:

```
# mutable
loaded_signal = load_file(filename)
normalize(loaded_signal)
# immutable
loaded_signal = load_file(filename)
normalized_signal = normalize(loaded_signal)
```

In the implementation, where signals are mutable objects, a signal is loaded from a file and then normalized in-place. This has the result, that the variable `loaded_signal` does not contain the signal that has been loaded from the file, as one would expect because of the name, but the normalized signal. This becomes especially dangerous, when the referene to the signal is passed to different parts of the software.
In this case, it would be necessary to either copy the signal every time it is passed, or to assume that the signal is never processed. The copying solution costs a lot

of memory and the copying is considerably slower than the passing of a reference. Doing without processing the signal will cripple the main functionality of the software (which is most likely signal processing) and if the signal is processed anyways, the poor programmer goes to debugging hell, because the program will not crash where the error has been made by modifying the signal, but at another part, where the signal is referenced. Maybe the program will not even crash, but just compute rubbish data, which is especially malicious, as it entrusts the detection of the error to the user.
In the immutable implementation, the loaded and the normalized signal are stored in separate variables. And the references to the implementation can be passed to other parts of the software without the danger of unexpected modifications.

The question of mutability is not only important for the implementation of classes. It must also considered for any other data that is passed through a programming interface, like arguments and return values of functions. So it is always recommended to use tuples instead of lists for such data.

Returning data by calling a function with a reference to a mutable variable (return-by-call-by-reference) is a peculiar habit, that sprouted in the environment of software written in C. In C, this is the easiest way to return multiple values with one function. Furthermore, the explicit notation of pointers in C made it obvious that the variable would be modified by the function call.
In Python it is rather convenient to return multiple values with a tuple, while pointers are not exposed to the programmer. So it is absolutely not necessary for a function to have the side effect of modifying one of its parameters. If a mutable object shall be modified, it is cleaner to implement this modification in a method of its class, rather than an external function. This way, the method can encapsulate the implementation of the class, while an external function would rely on exposed internals of the class.

1.4 Design Patterns

Design patterns are generic solutions to problems, that occur regularly during the design and implementation of software. A surprising amount of research has been done to investigate the advantages and drawbacks of certain design patterns, so the established design patterns are often faster, easier to maintain and easier to extend than naive implementations. So it is always recommended to know the basic design patterns and to develop a feeling for recognizing, when such a pattern might be applicable.

1.4.1 The Observer Pattern

The Observer Pattern is used, when one part of the software needs to retrieve data from another part, but does not know, when this data becomes available.

Consider the example of a user interface, that wants to show the value of a sensor reading, but does not know, when the sensor is delivering new data.

A naive solution would be simple polling:

```
retrieved = 0.0
while not timeout:
    new_value = sensor_driver.GetSensorReading()
    if new_value != retrieved:
        retrieved = new_value
        user_interface.Update(retrieved)
```

It becomes immediately obvious, that this implementation calls the GetSensorReading-method far too often, which might result in a horrible performance of the software.

A slightly more sophisticated (but even worse) solution would be to call the user interface's Update-method from the sensor driver.

```
def OnNewSensorReading(new_value):
    """
    This function is located in the sensor driver
    and processes the new value from the sensor.
    """
    user_interface.Update(new_value)
```

Compared to the polling implementation, this implementation reverses the dependencies, by making the sensor driver depend on the user interface, while a user interface could be instantiated without a sensor driver. The advantage of this implementation is, that all necessary information is available in the sensor driver; it knows the value of the sensor reading, it knows the time when the reading changes, and it knows about the user interface, that wants to do something with the sensor value. This way, it is possible to do without polling and the performance penalty that comes with it.
The drawback of this implementation is that the order of dependencies is now very peculiar and counterintuitive, which will lead to a non-extensible and unmaintainable codebase. The user interface, which has the purpose of displaying sensor results, can now be instantiated without specifying any sensors, which does not make sense, but it is at least not dangerous. A sensor driver, on the other hand, can now only be instantiated with a specified user interface, which leads to an infinte wealth of problems.

The sensor driver can no longer be instantiated without a user interface, which makes automated testing of the driver almost impossible. If many different sensors are used (which is likely), the update of the user interface has to be implemented in every single one of them. If the user interface changes (which is likely), every single driver has to be adapted to these changes. If the sensor drivers shall be used for a similar software, that writes the sensor readings into a data base instead of displaying it in a user interface, the already available drivers cannot be reused, because they are only for updating a user interface. If the user interface raises an error (which is likely, because testing user interfaces is hard), the crash also affects the sensor driver, which can make debugging a nightmare.
In essence, this implementation is a clear violation of the ”Do only one thing and do it well”-paradigma. The update of the user interface must not be a responsibility of such a low level part as a sensor driver.

The Observer Pattern offers a much more elegant solution, that does without polling, but keeps the sensor driver independent of the user interface. This avoids all drawbacks of the aforementioned implementations at the cost of a slightly higher programming effort.
The idea of the Observer Pattern is, that a part of a software, that wants to be notified about an event (like a new sensor value), passes a callback object to the part of the software, that generates the event. This way, the event generating part has all necessary information to handle the event properly, so polling is not necessary. In addition, the event generating part does not depend on any other parts of the software, as it can just discard the event, if no callback objects are specified.

The following code shows a rudimentary implementation of the sensor-user-interface-example, that uses the Observer Pattern:

```
class UserInterface(object):
    def __init__(self, sensors):
        for s in sensors:
            # inform all sensors that the user
            # interface wants to be notified
            s.AddObserver(self.Update)

    def Update(self, sensor, sensor_value):
        print("Sensor %s has new value: %f" % (str(sensor), sensor_value))

class SensorDriver(object):
    def __init__(self):
        self.__observers = []

    def AddObserver(self, callback):
        """
        Stores the given callback functions
```

```
        """
        self.__observers.append(callback)

    def OnNewSensorReading(new_value):
        """
        Is magically called, when a new sensor reading is available
        """
        for o in self.__observers:
            # notify all interested program parts
            # by calling every callback object
            o(self, new_value)

sensor = SensorDriver()
user_interface = UserInterface(sensors=sensor)
```

If many sensor drivers have to be implemented, the programming overhead of the Observer Pattern can be reduced, by sharing the necessary code trough inheritance. The advantages of this implementation become obvious, when the performance, the maintenance and possible extensions are considered:

- ▷ No polling
- ▷ A sensor driver can be instantiated and tested without any observing object
- ▷ Adding a new front end (like a data base that stores the sensor data) requires no changes in the sensor drivers' code
- ▷ It is even possible to use multiple front ends simultaneously (user interface and data base, maybe even more) without changing the sensor drivers

When the observing objects (in the example, this is the user interface) are destroyed, it is necessary to delete the callbacks that have been passed to the AddObserver-method aswell. So it is necessary to implement a RemoveObserver-method aswell.

```
class SensorDriver(object):
    ...
    def RemoveObserver(self, callback):
        self.__observers.remove(callback)
    ...
```

Due to the very dynamic nature of Python, it is necessary to generate new objects from the methods in certain cases. Therefore it is not guaranteed, that different references to the same method (in the example self.Update) link to the same object, which is a mandatory requirement for the lists remove-method (as used in the RemoveObserver-method) to work. This pitfall can be avoided by storing a weak reference to the

method and passing that to the AddObserver- and RemoveObserver-methods. It has to be a weak reference, so the automatic deletion of unused objects is not annulled by circular references (more on this in Chapter 3 about the common pitfalls when using Python).

```
import weakref

class UserInterface(object):
    def __init__(self, sensors):
        self.__update_method = weakref.proxy(self.Update)
        for s in sensors:
            s.AddObserver(self.__update_method)
    ...
```

1.4.2 Other Patterns

The Observer Pattern is so cool, you can conquer the world just with that. But still, some more patterns would be helpful, so: TODO

2 Tipps for the advanced Python-beginner

This manual is aimed at people, who already know the syntax to define a class or a function in Python. With this it is possible to solve most programming problems, so that certain functionalities of Python, that might facilitate this task, are sometimes overlooked. This chapter introduces some of these functionalities.

2.1 Exception handling

Never trust a program, that doesn't crash, when it is about to enter an invalid state!

Exceptions can be raised, when the program is about to enter a state, which the programmer cannot or does not want to implement. Normally this happens, when invalid or incompatible data is given, like an unparseable string or a division by zero. Another reason to raise an exception is, when the program and its dependencies are not properly set up. This could mean missing libraries or missing configuration files. Sometimes, exceptions are raised because of algorithmic problems, for example when the maximum recursion depth is exceeded. And then there are hardware related exceptions, for when a device is not responding or when the main memory is full.

Exceptions are not only for stopping the program, before it destroys anything. They are also for facilitating the debug process by giving a hopefully informative error message and by showing where the error occured. Nothing is harder to debug than an error, that happened at a completely different place, than where the exception has been raised. So it is generally a good idea to raise an exception instead of further interpreting the data, as soon as something looks vaguely like an error.

It is also possible to raise warnings, that do not stop the program. This can be used for example to inform about deprecated features of the program.

Here is a short list of good practices in raising and handling exceptions:

- Do not guess!

If choosing the proper action for a given piece of data is not a deterministic and reliable process, raise an error about ambiguous data. A potentially wrong interpretation of the data will lead to a misbehaving program, where the errors are spotted way further down the processing chain, than the place, where the errors have been made.
Also a Complicated maze of conditional expressions, that just interprets some data, is tedious to program, hard to understand and unlikely to be maintained properly.

- Raise meaningful exceptions!
  In order to facilitate proper exception handling and maybe debugging aswell, do not just raise an `Exception`. Please bother to at least raise a `RuntimeError`, if it is not clear, which exception shall be raised. But if possible, raise a more specific exception like an `IOError`, a `ValueError`, an `IndexError`, a `ZeroDivisionError` or a `NotImplementedError`.
  There are many more predefined exceptions listed in the Python documentation. If they are not specific enough, it is possible to define custom errors by implementing a class that inherits from `BaseException`.

- Catch exceptions with precision!
  Don't just program a general `try ... except` block to mute some error message. Maybe , there are unsuspected exceptions, that are raised inside the `try` block, which are then masked, because the `try` block just catches any exception. This again leads to errors, that occur later than when they happen.
  So at least specify, which type of exception shall be caught, by naming the class in the `except` block. It is possible to catch different types of exceptions, by appending multiple `except` blocks to one `try` block.
  One can also get the object of the raised exception by typing "`except EXCEPTION_CLASS as e`". This allows a further disambiguation of the caught exception, for example by parsing its error message, and reraising the exception again, when it shall not be caught.

2.2 Neat modules in Python's standard library

The Python standard library offers some very useful modules, that are often overlooked by programmers, who have just started with Python. Some of these modules are not only targeted for very specific applications, but they are just helpful in general. These modules shall be introduced in this section.

2.2.1 logging

The logging module offers an alternative to using `print` for status messages. The advantage of the logging module is, that it offers some functionality for specifying the handling of the status messages without modifying the code that generates the status messages.

When writing code, that generates a status message, the programmer has to decide, with which log-level the message shall be generated. A reasonable set of log-levels is already implemented in the logging module, but it is also possible to specify custom ones. The already implemented log-levels are `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`. They can be used with convenient functions from the logging module.

```
import logging
logging.debug("This is a debug message")
logging.info("Consider yourself informed")
logging.warning("I warned you not to use the print statement")
logging.error("Not obeying the rules in Handout is an error")
logging.critical("This is a message of critical importance")
```

With the logging module it is possible to specify which messages shall be displayed and which shall be skipped. The following line of code sets up, that all debug- and info-messages are dropped, while everything with the priority of a warning and more is displayed.

```
logging.getLogger().setLevel(logging.WARNING)
```

Furthermore it is easily possible to redirect the logger's output to a file or a widget in a graphical user interface.

```
handler = logging.FileHandler("file.log") # writes to a file
handler = logging.StreamHandler()         # writes to a stream, like the console
                                          # output or a text field in a GUI
handler = logging.NullHandler()           # discards everything
logging.getLogger().addHandler(handler)
```

Those handlers can be customized with formatters, to specify, which information shall be logged in addition to the log message. This information can be for example the log-level, the file or the line number in which the logging has been invoked.

It is also possible to create a separate logger for each module. This way, the logging can be customized in a very fine grained way, by specifying a log level for each module individually. All loggers inherit their log-level from the root-logger, so a common use for this is to set the root-Logger to a very high log-level, while switching on the debug

messages in one specific module. The following code shows how to create and use a module-specific logger.

```
import logging
# create Logger object
logger = logging.getLogger(__name__)
# send a logging message
logger.info("Hello World")    # "logger" not "logging"
```

The logger object can now be set up independently from the root logger, by specifying a log-level or adding a Handler.

Please remember that logging is just for sending status messages. It is not an alternative to raising an exception or even giving an exception a meaningful error message.

2.2.2 collections

The collections module defines some interesting classes for data containment. One of them is the `OrderedDict`, which behaves like a normal dictionary, but when iterating over its keys, the iteration will be performed in the order in which the items have been added to the `OrderedDict`. This is a very convenient mapping type for data, whose order is important.

Other classes that are provided by the collections module are general abstractions, that can be used with the `isinstance` function to check, if an object has a certain functionality. The following code lines test, if the object "`a`" is iterable or callable:

```
import collections
i = isinstance(a, collections.Iterable) # True, if a is a list,
                                        # a tuple, a set, a dict,
                                        # or anything else, that
                                        # can be iterated in a
                                        # for loop.
c = isinstance(a, collections.Callable) # True, if a can be called
                                        # like a function or a
                                        # method.
```

This behavior of `isinstance` shows again, that Python uses duck typing (see section 3.1.4 for more about this). The compared object's type does not have to have inherited from an iterable class, so `isinstance` recognizes it as a subclass of

`collections.Iterable`. It just has to have the functionality of an iterable object. The behavior is accordingly with `collections.Callable`.

2.2.3 functools

The functools module provides functions and decorators to enhance the capabilities of callable objects like functions or methods.

One interesting function, provided by this module, is `partial`. It can be used to create a new function by defining certain arguments from a base function. This technique is sometimes referred to as "Currying" or "Schönfinkeln"[1]. An example for using this function could be to create an `increase`-function from an `add`-function:

```
import functools

def add(a, b):    # define an add-function, that adds two values
    return a + b

increase = functools.partial(add, b=1)    # this function adds 1 to a

y = increase(4)    # y = 4+1
```

In Python3 there exists a special function `functools.partialmethod` for creating methods with a simplified signature: This function is not available in the Python2 version of functools.

```
class Light(object):
    def __init__(self):
        self.__switched_on = False

    def Switch(self, switch_on):
        self.__switched_on = switch_on

    SwitchOn = functools.partialmethod(Switch, switch_on=True)
    SwitchOff = functools.partialmethod(Switch, switch_on=False)
```

Another useful feature of the functools module is the decorator[2] `lru_cache`. This decorator is also only available in Python3. This decorator can be used to cache return values of a function, so it needs not run again, if it has already run for the

[1] After its inventors Gottlob Frege, Moses Schönfinkel and Haskell Brooks Curry. So sadly "Currying" does not come from "spicing up a function".
[2] read more about decorators in subsection 2.4.2.

given set of arguments. This can save computing time on functions that take a long time to run, but are only called with a small variety of arguments.

```
import functools
import time

# "maxsize" is the maximum number of cached values. Calling the
# function with a new combination of parameters will result in the
# deletion of the result for least recently used combination.
# "typed" specifies, if the arguments' types shall be considered.
# If False, intense_computing(a=1.0, b=2.0) will be mapped to the
# same result as intense_computing(a=1, b=2).
@functools.lru_cache(maxsize=2, typed=False)
def intense_computing(a, b):
    time.sleep(3)    # simulate some really hard work!
    return a + b

v = intense_computing(9.0, 6.0)  # this will do some hard work
w = intense_computing(2.3, 4.1)  # this will do some hard work, too
x = intense_computing(9, 6)      # this will just use the cached value
y = intense_computing(3.5, 7.1)  # this will do some hard work again
z = intense_computing(2.3, 4.1)  # this will do some hard work, because
                                 # the cached value has been deleted
                                 # after the last function call
```

2.2.4 inspect

The inspect module provides functionalities to get information about objects like functions or classes. For example, it allows retrieving the arguments of a function (including their default values) or it can deliver the inheritance hierarchy of a class. The features of this module are so extensive, that they cannot be discussed in the briefness of this manual. But they are thoroughly explained in the official documentation.

Object introspection can be cumputationally expensive, so it is advised not to use the inspect functionalities in performance critical code sequences. Most programming tasks can do without it, anyway.
But in certain situations, the inspect module can be extremely helpful. For example, when serializing an object in order to save it to a file. Or for producing valuable debug information, when raising an error.

2.3 Special Python expressions

Python provides some expressions, for which the need is not immediately obvious, since their functionality can be emulated with basic elements of the Python language. Nevertheless, they are extremely helpful for writing cleaner code, that might run faster and is easier to maintain and understand.

2.3.1 with

Certain objects need to be destroyed manually after using them. For example a filestream has to be closed after extracing all necessary information from the file. The manual destruction can be tedious, especially when it has to be ensured with `try` and `finally` blocks.
So Python allows to specify the initialization and destruction methods `__enter__` and `__exit__`, which can be invoked with the `with` statement. The `with` statement creates a block, in which an object can be used. It takes care of calling the `__enter__` method when entering the block and of calling the `__exit__` method when leaving the block. The `__exit__` method is always called, even when an exception is raised.

The following example shows how to use the `with` statement to access a file:

```
with open("path_to_file", "r") as f:     # f is the variable name, by
    for l in f.readlines():              # which the file object can
        print(l.strip())                 # be accessed
# no need to call f.close(). "with" will do that.
```

To open multiple files, simply separate them with commas in the with statement. In order to avoid overly long lines, a backslash can be used to state that the following line belongs to the current command.

```
with open("path_to_file1", "r") as f1,\
     open("path_to_file2", "w") as f1,\
     open("path_to_file3", "wb") as f3:
    pass    # do something with the files
```

2.3.2 yield

`yield` is an alternative to the `return` statement for functions or methods that return sequences of items. First creating the whole sequence and then returning it can consume a lot of memory, so `yield` allows to return the sequence element wise. This

way, it is possible to iterate over the return value of a function, while only having the current element in memory. The future elements are not loaded yet and the past elements can be (or have been) discarded.
The return value of such a function is called a "generator".

The `yield` statement is typically located inside a loop. The following example shows a function that returns a list and the equivalent implementation, employing the `yield` statement.

```
def with_return(maximum):
    result = []
    for i in range(maximum + 1):
        result.append(i)
    return result

def with_yield(maximum):
    for i in range(maximum + 1):
        yield i
```

Since the elements of the returned sequence of a function utilizing the `yield` statement are discarded, when advancing to the next element, random access to the sequence's elements is not possible. For accessing the elements randomly, they have to be loaded into the memory by casting the result value into a randomly accessible data type like a tuple or a list.

```
a = with_return(maximum)[3]          # this will work
b = with_yield(maximum)[3]           # this will crash
c = tuple(with_yield(maximum))[3]    # this will work
```

Many functions of python return a generator or generator like objects. For example the `range` function or the `readlines` method of a filestream.

2.3.3 lambda

The `lambda` statement allows defining functions like a variable. This functionality is not absolutely necessary, since Python allows to define functions even inside other functions (closures) and the handling of function pointers is easy. However for simple functions, that just calculate a value in one line of code, a brief `lambda` statement clutters the code less than the definition of a closure.

Lambda functions are created by assigning a `lambda` statement, which is followed by the function's arguments and its return value, to a variable. The return value is

just stated by the formula that calculates it; the `return` statement is not issued. The following example shows how to define a lambda function:

```
# the usual way:
def add(a, b):
    return a + b

# with lambda:
add = lambda a, b: a + b
```

2.4 Advanced Python tricks

This section introduces some cool aspects of Python, that increase its usability tremendously, although they might not be necessary to accomplish everyday programming tasks.

2.4.1 Features of the interpreter

The Python interpreter makes use of two important environment varibles [3].

- `PYTHONPATH` can be used to specify an additional path, in which Python will search for modules, when trying to import something. The environment variable is usually empty and is used for making libraries available, that are not installed system wide, when executing a specific program that needs them. This is useful for testing programs or libraries that are still in development. Usually such programs are started through a Bash/Batch-script, that sets the `PYTHONPATH` accordingly before executing the program.
  `PYTHONPATH` can contain multiple paths. On Windows they are separated by semicolons, on Linux with colons.

- `PYTHONSTARTUP` can be used to specify the path to a script, that is executed, when the Python interpreter is started in interactive commandline mode. This can be used to import modules that are commonly used like `math` or `os`.
  The following lines are a startup script, that enables autocompletion for the interactive Python interpreter:

  ```
  import readline
  import rlcompleter
  ```

[3] Environment variables can be set with the commands `set` or `setx` on Windows and `export` on Unix and its derivates.

```
readline.parse_and_bind("tab: complete")
```

It might be necessary to install the readline package before using this script. When this script has run, hitting the tab-key no longer enters a tab, but it requests the autocompletion. Hitting tab once extends the currently entered start of a command by the letters of a command that can be determined unambigously. Hitting tab twice shows a list of all possible extensions to the currently entered start of a command.

A program can be started in interactive mode with the commandline parameter `-i`.

```
python -i my_program.py
```

With this, program is run normally and after it has finished, the Python interpreter switches to the interactive command line mode. This can be very useful for debugging, as all global variables, that are instantiated by the program, can then be investigated interactively.
Of course global variables should have been avoided, so one will rarely start a whole program in interactive mode. But if the software parts are written in a nicely modular way, one can write a script that instantiates the part that shall be debugged, and then run it in order to test that software part interactively.

2.4.2 Decorators

Decorators are a special feature of Python to enhance functions, methods and classes with extra functionality. They are functions themselves, that operate with the decorated function as a parameter. To apply a decorator to a function, the decorator is noted with an `@` as a prefix.

The following example shows a decorator that ensures that the return value of a function is never zero.

```
def non_zero(function):
    def new_function(*args, **kwargs):
        result = function(*args, **kwargs)
        if result == 0.0:
            return 1e-12
        else:
            return result
    return new_function

@non_zero
```

```
def subtract(a, b):
    return a - b
```

When interpreting this code, Python will replace the `subtract`-function with the return value of `non_zero(subtract)`. This will be the closure `new_function`, whose variable `function` is set to `subtract`. A function call of `subtract` will be replaced like the following:

$$subtract(a, b) \Rightarrow non\_zero(subtract)(a, b) \Rightarrow new\_function(a, b) \tag{2.1}$$

The decorator separates the securing of a nonzero result from the actual calculation. And it can of course be applied to other functions aswell. This way, decorators can be a very elegant vehicle to comply with the "Do only one thing, and do it well" mantra. But decorators are often hard to understand, so be careful, not to program in accordance to the Monty-Python slogan "surprise your friends, amuse your enemies".

The function, with which the original function is replaced needs not be created directly by the decorator function. When using a closure, for creating the replacing function, decorators with parameters become possible. The following example shows a decorator that limits a functions return value to a specifiable maximum.

```
def limit(maximum):
    def generator(function):
        def new_function(*args, **kwargs):    # did you watch Inception?
            result = function(*args, **kwargs)
            if result > maximum:
                return maximum
            else:
                return result
        return new_function
    return generator

@limit(5)
def add(a, b):
    return a + b
```

A decorator only needs to be a callable object, so decorators with parameters are usually much easier to understand, if they are implemented in form of a class:

```
class Limit(object):
    def __init__(self, maximum):
        self.__maximum = maximum
        self.__function = None
```

```
    def __ReplacementFunction(self, *args, **kwargs):
        result = self.__function(*args, **kwargs)
        if result > maximum:
            return maximum
        else:
            return result

    def __call__(self, function):   # defining this makes
        self.__function = function  # the object callable
        return self.__ReplacementFunction
```

Of course, decorators can be stacked:

```
@Limit(5)
@non_zero
def add(a, b):
    return a + b
```

In this example, the input function for calling the `Limit`-object is the replaced function from the `non_zero`-decorator.

When decorating a method, the decorators will be called with the unbound methods, since they are executed when creating the class and not an object of that class. Often, when decorating a method, the decorator shall modify a bound method, so it needs information about the object, when it is created. For this, it is possible to define a class with a `__get__`-method, which is executed, when the method of an object is accessed. These classes are called "descriptors". This method gets the instance and the class as parameters and shall return the replacement function. Be careful when storing a reference to the instance, as this might result in cyclic references (see section 3.2 for more on this). The following example shows the usage of a decorator for a method:

```
class Descriptor(object):
    def __init__(self, unbound_method):
        pass

    def __get__(instance, instance_type):
        def replacement_function(*args, **kwargs): # changes nothing
            return self.__unbound_method(instance, *args, **kwargs)
        return replacement_function

def decorator(method):    # this can be used with @decorator
    return Descriptor(method)
```

The replacements that are going on in this example are shown below.

When interpreting the class:

$$\begin{aligned} & MyClass.Method \\ \Rightarrow & decorator(MyClass.Method) \\ \Rightarrow & Descriptor(MyClass.Method) \end{aligned} \tag{2.2}$$

When calling a method of an object:

$$\begin{aligned} & myObject.Method(arg) \\ \Rightarrow & Descriptor(MyClass.Method).\_\_get\_\_(myObject, MyClass)(arg) \\ \Rightarrow & replacement\_function(arg) \end{aligned} \tag{2.3}$$

2.4.3 Profiling

Python has inbuilt capabilities for profiling a Python program. Profiling means, running the program and logging the time, that the interpreter spends on any given command. These logs reveal performance issues and parts, where performance optimizations would be most beneficial.

The profiling modules can be imported in the software's source code for a very fine grained setup of which commands shall be logged. For more about this, please read the manual page about this subject [3], as documenting these extensive capabilities would be beyond the scope of this manual.

There is also a way of profiling a complete program, without modifying it. This can be done by loading the `cProfile` module at the startup of the interpreter, before executing the program.

```
python -m cProfile -o profile_file my_program.py
```

In this example, the path where the profiling log is stored is given by `-o profile_file`.

Profiling can be used to compare the performance of different implementations. For the sake of fairness, the programs, that are to be profiled for this purpose, must only differ in that implementation. This also includes, that the programs have to be deterministic, which means:

- ▷ no user interaction
- ▷ no random numbers, that are not seeded to be equal in both programs
- ▷ no interaction with hardware, whose performance might vary a lot (e.g. network communication).

The profiling logs can then be evaluated in a Python script with the help of the `pstats` module.

```
import pstats

# load the profiling log file
p_loaded = pstats.Stats("profile_file")
# sort by the time spent in the given function
p_sorted = p_loaded.sort_stats("time")
# print information about the 20 functions that used the most time
p_sorted.print_stats(20)
```

In this example, the functions are sorted by the time that has actually been spend in that function. When a function called another function, only the time of calling the other function is added to the first function's time. If the runtime of the other function shall also be added to the time of the calling function, the profile must be sorted by the cumulative time.

```
...
p_sorted = p_loaded.sort_stats("cumulative")
...
```

2.5 Differences between Python2 and Python3

Programming in Python2 and Python3 is very similar. The differences are mainly details and the underlying C implementation. This is the reason, why many libraries for Python2 have not been ported to Python3, yet.
It is however possible to write Python code that runs with both Python2 and Python3. This section shall give an overview about the main differences and how to achieve compatibility with both versions.

2.5.1 print

Probably the most obvious difference between the two versions is, that `print` is a function in Python3. Of couse typing the additional parentheses is annoying, but this is also a much cleaner programming interface, which can be advantagous in many situations. For example a pointer to the `print` function can be passed as a handler, that specifies what to do with some data.

```
def do_something(function, data):
    return function(data)

do_something(function=print, data=(1, 2, 3))
```

In order to use the function version of `print` in Python2, one has to import it from (the) `__future__`.

```
from __future__ import print_function
print("Hello World")
```

Imports of `__future__` features, that have been backported to Python2, are ignored in Python3, so that code which uses them, runs with both versions.

2.5.2 Imports

Python3 tries to distiguish more between modules, that have been installed in some system directory and modules that are part of the software's source code. This is similar to the `#include` statement in C/C++, where `#include "header.h"` specifies, that the header file shall be included from the local source code, while `#include ⟨header.h⟩` includes a system wide installed header. In this context, "system wide" means everything, that can be found through `sys.path`, so even self written and locally installed libraries are considered to be installed "system wide" here. Since the working path, in which the Python interpreter has been started (independent of later calls of `os.chdir`), can also be found through `sys.path`, the modules in this path are also handled as if they were installed system wide.

In Python2 all modules are being directly imported by importing their name, while in Python3, local modules are imported through the modules "." and "..". System wide installed libraries are still imported by their name in Python3.
Maybe an example can illustrate these confusing sentences. Assuming, one wants to import the function `helperfunction` from the module `helper`, which is defined in the file "helper.py" in the current working directory, the following lines would be used to import this function.

```
# import a system wide library (works the same way with both versions)
import os

# Python2
# import a module into its own namespace
import helper
# import a function into the local namespace
from helper import helperfunction
```

```
#Python3
# import a module into its own namespace
from . import helper
# import a function into the local namespace
from .helper import helperfunction
```

The Python3 way of importing modules has some advantages.

- Python can now distinguish between local and system wide modules with the same name.
- It is possible to import modules that are in the parent directory of the current module (with `from .. import module`)
- It also works with Python2.7, so it is advised to use it for portable code.

2.5.3 String / Unicode

In Python2, strings are usually ascii strings, while prepending a `u` before the quotes defines, that the string shall be encoded as unicode (e.g. `u"Hello World"`). Python3 on the other hand always uses unicode.

Due to its fixation on unicode, Python3 can do without some functions, that are available in Python2. These are among others the `unicode` function, that is used to cast objects into unicode strings, and the `BaseString` class, that is used to check, if a variable is a (unicode) string, with `isinstance`. These have to be avoided, when writing code, that shall run with both versions.
Also note, that the `str` function behaves differently, as its Python2 variant converts its parameter to ascii, while in Python3 it is converted to unicode. This can be utilized for portable code, where an object shall be casted to the most common string format in the respective version of Python.

Furthermore, Python3 is rather picky about its input. For example, when reading a text file, one has to specify the file encoding, when opening the file, if it is not unicode:

```
with open("file.txt", "r", encoding="ISO-8859-15") as f:
    # do something with the file
```

In Python2, the `open` function does not accept an `encoding` parameter, so it is not possible to open a non-unicode file with the same line of code in both versions of Python. Since it is hard write the `with` line in a conditional way, that depends on the

Python version, without duplicating the code inside the `with` block, it might be best, to create the parameters for the `open` function inside an `if` block.

```
import sys

kwargs = {}
if sys.version_info.major >= 3:
    kwargs["encoding"] = "ISO-8859-15"
with open("file.txt", "r", **kwargs) as f:
    for line in f:
        if sys.version_info.major <= 2:
            line = line.decode("ISO-8859-15")
        # do something with the lines of the file
```

If the encoding is not specified for a non-unicode file, Python3 crashes, while Python2 only reads peculiar special characters from non-ascii files. Actually Python3's behavior is a good thing, because it crashes, where the error happens. Python2 masks the error by going on, until the string is interpreted somewhere.

2.5.4 Integer divisions

This difference between Python2 and Python3 deserves special attention, as it is unlikely to lead to a crash, but it can easily lead to unintended calculation results. In Python3, the normal division with "/" is always a floating point division, while in Python2, dividing two integers performs an integer division, which discards the remainder. The following code gives an example for this:

```
a = 10.0 / 4.0 # 2.5 in both versions (as expected)
a = 10 / 4     # 2 in Python2, 2.5 in Python3
a = 10 / 2     # 5 in Python2, 5.0 in Python3 (a float although both
               #                               operands are integers
               #                               and the remainder is 0)
```

In both versions of Python the "//" operator is used for divisions, that discards the remainder of the division. If both operands are integers, using this operator also returns an integer. So in order to write code that is portable between the two versions, use "//" to perform an integer division.

```
a = 10.0 // 4.0  # 2.0 in both versions
a = 10 // 4      # 2 in both versions
```

2.6 Coding style

This is the point, where this manual becomes inconsistent. Although it proudly claims on its title page, that it aims to abolish a totalitarian regime, it promotes the usage of a programming language that is pretty totalitarian about its syntax.
While truly libertarian languages like Perl execute anything from latin up to a demonic mixture of special characters and burnt out brain cells, Python has adopted the idea, that there should be only one solution for each problem. Because of this, it imposes many restrictions how the code has to be written (e.g. its strict indentation rules).
The reason to choose such a strict syntax specification for Python becomes clear, when thinking about the main benefit of a totalitarian system: Even stupid people understand, what's going on. So reading and understanding a piece of Python source code, that someone else has written, is most of the times reasonably simple. The art is to achieve this simplicity, without oppressing programmer in his way to think, talk, act and, of course, design a program. And Python practices this art rather admirably.

Even Python did not dare to impose a fully unified coding style on the programmer, which means that certain code constructions can be written in different ways. Some best practices for programming in Python are compiled in PEP8 [2].
The reasons for most of the recommendations in PEP8 are thoroughly explained, so it is best to follow them. Other recommendations are just a matter of taste. Since many other projects are programmed in compliance with PEP8, it is good to follow these recommendations aswell, just for the sake of a recognizable programming style.

But there are two issues, in which this manual recommends a different coding style than PEP8:

- Feel free to exceed a line length of 79 characters, if it improves the readability of the code.
  Since many programmers no longer use command line editors on small screens with a small resolution, the 79 character limit seems a bit strict and old fashioned. It is still a good guideline to split a function call or a string to multiple lines, when it exceeds this threshold, but in some cases (like heavily indented blocks), a relaxed adherence to the 79 character rule can significantly improve the readability of the code.

- Please use tabs for indentation!
  With a tab, the indentation is only one character per level, which is easy to maintain. When manually indenting or unindenting code with whitespaces, it happens easily, that one whitespace is forgotten or too much. This mistake cannot be discovered as easily as with tabs, where one character is much wider. Since Python uses the indentation level for assigning code to code blocks, an indentation mistake will break the program.

Furthermore most editors allow specifying the width of a tab in their settings. This way, each programmer can decide independently, how steeply the indentation shall be displayed.

This manual even recommends using tabs for indentation and whitespaces for aligning code, which can result in a mixture of tabs and spaces at the beginning of a line of code. PEP8 will probably not agree with this, but this case is not mentioned there. The following code gives an example, how this is meant (the arrows and u-shaped-thingys are tabs and spaces):

```
def␣my_function(with_many_arguments,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣that_are_split,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣into_multiple_lines):
    →x␣=␣(with_many_arguments,
    →␣␣␣␣␣123,
    →␣␣␣␣␣456)
```

In this example, the indentation is done with tabs, while the alignment of the function arguments of `my_function` and the tuple elements of `x` are aligned with spaces. There are two advantages of the alignment with spaces. First, a perfect alignment is always possible, while with tabs, the alignment can only be done in coarse steps. And second, the alignment becomes independent of how wide a tab is set to be in the respective editor.

3 Python Pitfalls

Python is a very good programming language, but it still has some quirks, that one has to know about, when debugging a Python program. This chapter shall explain some Python specifics, which might behave in an unexpected manner for some programmers.

3.1 Python's object model

3.1.1 Imports

Importing modules works slightly differently in Python compared to C/C++, as the specification of an imported module is not done by its file path (like when including a C/C++-header), but by its module path. Since most folders and files, that contain Python source code, are also modules, the file path and the module path are mostly equivalent. Consider an example, where folder "a" is a module, that contains another file or folder, that defines module "b". "a" is in the working directory of the Python interpreter, so it can be imported directly.

```
import a    # a can be imported directly
import a.b  # b can be imported through a
```

In this example, the similarity between file and module paths becomes obvious, as they only differ in the fact, that file paths are concatenated with a file delimiter like "/" or "\", while module paths use a "." for accessing the properties of the module objects. But in module paths, it is not possible to access parent modules, like in file paths with the ".." folder name. Python3 offers a ".." module to access the direct parent module, but it cannot be concatenated to access that module's parent module.

It is generally recommended to do without accessing the parent module, as this practice avoids circular dependencies, where one module depends on another, which in turn depends on the first. While this works well with modules, that act as software libraries, it can be counter intuitive when it comes to inheritance, as it is not recommended to

sort subclasses into subfolders, because this would require to access the subfolders' parent folder.

Folders can be made to a Python module, by implementing an ___init___.py file inside them, which is executed when importing the module. After importing a folder's module, all py-files inside the folder are accessible as submodules. In order to avoid inconveniently long module paths, where every object is loaded from its own submodule, the ___init___.py file can be used to import objects into the module's namespace. When such an imported object has the same name as a py-file, the module's property that can be accessed through the given name, becomes ambiguous, as it is not clear, if the submodule or the imported object is meant.
This hazard has to be avoided by naming conventions. This manual recommends to write all source code filenames and folder names in lower case letters, while using CamelCase[1] for class names. In most cases, function names are defined in lower case letters as well, which can clash with the lower case file names. So it must be avoided, that a file name has the same name as a function, for example by combining multiple functions in one file and choosing a meaningful name for the group of functions as the file name. For very small functions (and very, very small numbers of functions), it might even be okay to define them in the ___init___.py file.

If two or more modules import each other, it is impossible, that each one of these modules is imported completely, before continuing with importing the other modules. This means, that during the import, these circularly dependent modules cannot provide their full functionality.
Circular module dependencies are not a problem in Python, as long as the functionalities from the other module are not required during the import. This is the case, when the functionalities are only used inside functions or methods, that are not executed, when importing the modules. But certain functionalities like base classes for inheritance or decorators have to be available, when importing a module, in which case circular dependencies can lead to serious problems.
Generally circular dependencies are an indication for bad library design, so if they occur it is strongly recommended to consider refactoring the source code.

Section 2.5.2 presents some further details about the differences between the import systems of Python2 and Python3.

3.1.2 Methods

Python's model of methods is similar to the one of C++. In both languages, a method is basically a function, which gets the context of its instance as the first argument. This argument is a pointer, that is needed to access the properties of the object from within

[1] words are separated by capital letters. The first letter is a capital letter.

the method. In C++ the first argument is declared implicitly, so the `this`-pointer has not to be declared when implementing the method. Python, being faithful to its slogan "explicit is better than implicit", demands that the `self`-argument is declared explicitly in the method declaration.

The explicit statement of the `self` argument makes sense, because during declaration, the method is still "unbound", which means that the method is still a function without knowledge about any object. When accessing the method of an object, a "bound" method is created, by using the unbound method as a template and setting the `self`-argument to a pointer to the object. This is the reason, why the `self`-argument has not to be passed, when calling a method.

Bound methods are created on the fly and they are stored only in rare cases (see section 3.2 for more background on this). So comparing a method with itself for equality might fail, because the bound method objects are not the same instances.

3.1.3 Public and private properties

Python does not really have a concept of private or protected properties. But it is general consensus, that variables, functions or methods, whose name starts with an underscore, are considered as part of an internal implementation detail. This way, one can discourage the access of a "to be protected"-property from outside code, by giving it a name that starts with an underscore.

When the name of a property starts with two underscores, the property is considered private. At runtime, this property is given a different name to avoid unwanted side effects, when a derrived class has a property with the same name. This way, private objects are a bit harder to access from outside code, but this is only "security by obscurity" and not meant to be a proper protection.

The underscore naming convention is only for notifying a programmer, that accessing certain properties from outside code is a weird and strongly discouraged idea. If she/he wants to do so anyway, who is Python to judge her/him?

3.1.4 Duck typing

Python is not statically typed. So, the only thing, that matters when accessing a property of an object, is, if the object has that property. It is unimportant, how and where this property is implemented and how the object got it. This model is called

duck-typing[2].

It is even possible to extend certain objects, by assigning a value to a property that has not been there before.

```
# define a class
class MyClass(object):
    pass

# create an object of this class
obj = MyClass()

# extend this object
obj.x = 42
```

This only works for pure Python objects. Objects whose classes are implemented in other programming languages (e.g. instances of `str`) cannot be dynamically extended.

The duck typing of Python seems to work inconsistently, when it comes to the function `isinstance`. In most cases, the `isinstance` function looks into the inheritance graph of an object, instead of comparing the functionality, that is provided by the object, as it would be done when performing duck typing. So an object is not considered an instance of a class, even if it has the same methods, as long as the class of the object is not a subclass of the given class:

```
class A(object):
    def Method(self):
        pass

class B(object):
    def Method(self):
        pass

a = A()
print(isinstance(a, B))    # False, althogh classes A and B are equal
```

On the other hand, the `collections` module offers classes, for which `isinstance` checks for a certain functionality, so it can return `True` even if the object is not an instance of a subclass of the considered class:

```
import collections
```

[2] from a poem of James Whitcomb Riley: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

```
class C(object):
    def __call__(self):
        pass

c = C()
print(isinstance(c, collections.Callable))  # True, although C is
                                            # not a subclass of
                                            # collections.Callable
```

3.1.5 Multiple inheritance

Python supports classes that inherit from multiple different base classes. Implementing such a class is fairly straight forward, as the constructors of the base classes have to be called explicitly in the constructor of the derived class. This is much more intuitive than the implicit order, in which the constuctors of the base classes are called in C++.

Implementing a multi-inheritance class without a constructor is strongly discouraged, because this would implicitly call only the constructor of the first base class. If several base classes implement the same method, the respective method of the derived class will be inherited from the first class, that implements this method.

In Python it is recommended, that classes inherit either from `object` or from a class that is derived from `object`. Critically considered, one might note that multi-inheritance would lead to circular inheritance (the object oriented paraphrase for incest), as all base classes inherit from `object`. Mercifully, Python has the decency, not to communicate its moral judgement about objects with interconnected ancestry, so this does not raise an error.
But again, this only works for certain classes (probably for `object` and for classes with a pure Python implementation). For Example, inheriting from multiple derived classes of `str` raises an error.

3.1.6 The `__del__` method

When defining a class, it is possible to define a `__del__` method, which is called, when an object of that class is being deleted. This must not be confused with a destructor, which is called to tear down an object before its deletion.

Generally it is not necessary to implement the `__del__` method. It is even discouraged, as it comes with some drawbacks. For example, when dealing with circular references,

an explicit call of the garbage collector can find such unused objects and delete them. However, if more than one of the objects, that reference each other, has implemented a `__del__` method, the order, in which the objects are deleted, can be relevant. Since the garbage collector cannot know this order, the objects are not deleted automatically.

The `__del__` method is called, when the reference count of an object reached zero. Therefore it is mandatory, that the method does not create any new references to the object. So passing the `self` reference to another code object is forbidden, since this would make it impossible to delete the object.
An example for such an errant endeavor, would be to pass the `self` reference to a logging framework (or print), to log that the object has been deleted.

3.2 Object References and Garbage Collection

Python uses reference counting to detect, if an object is still in use or if it can be deleted. The idea behind that is very simple: Everytime, a new reference to an object is created, the object's reference counter is increased and everytime, a reference is deleted (e.g. because the reference was used in the local namespace of a function that has returned), the reference counter is decreased. When the reference counter is zero, the object cannot be accesed from within active code, so the object can be deleted safely.

Problems arise with circular references. This happens when one object holds a reference to another object, which again holds a reference to the first object. Even if none of these two objects is reachable from within active code, their reference counters will not reach zero, because they hold references to each other.
In this case, Python's garbage collector has to be run, which detects unreachable objects and deletes them. Garbage collection can be a very complex task, that interferes with a program's performance. So it is advisable to avoid circular references in order to reduce the number of times, that the garbage collector has to be run.

Invoking the garbage collector with the command `gc.collect()` returns the number of deleted objects. If this number is not zero, it shows that the implementation contains circular references and maybe a memory leak. So showing this number in a debug message might help increasing the performance and stability of the program. If the number of deleted objects is always zero, the invokation of the garbage collector can of course be omitted.

Python has a module called `weakref`, which can be used to create weak references that do not affect an object's reference counter. When using a weak reference, it is not assured, that the reference is still valid, as the object could have been deleted. But since weak references cannot be used to keep unreachable objects alive, replacing

an ordinary (strong) reference with a weak one can break circular references.

A classical case of circular references are methods. An instance has a reference to each of its methods, so the methods can be invoked through the object itself. Also each method has a reference to the instance, so they can access the object's data.
It is not possible to break these reference circles with weak references. If the references to the methods were weak ones, the method's reference counts would always be zero, which meant, that they would be deleted immediately. Replacing the method's reference to the instance with a weak reference, would forbid the invokation of a method without storing a reference to the object. This is shown in the following example:

```
class MyClass(object):
    def MyMethod(self):
        print(self)
        return 28

# this would work if the method's reference to
# the instance (self) was a weak reference:
myObject = MyClass()
retval = myObject.MyMethod()

#this would not work:
retval = MyClass().MyMethod()
```

In the first call of `MyMethod`, the reference count of the instance of `MyClass` is one, because it is stored in `myObject`.
In the second call however, the instance is not stored, so the only reference to it is in the method. If this reference were a weak one, the object would be deleted before the method is executed and therefore the method call would crash.

Python avoids the problem of circular references, that comes with objects and their methods, by creating the method objects on the fly. For this, it distinguishes between unbound and bound methods. Unbound methods are properties of a class and basically functions, which expect an object as a first parameter `self`. When retrieving a method of an object, a bound method is created by predefining the `self` parameter of the unbound method with the object itself.
During a normal method call, the reference to the bound method is not stored anywhere, so that it is deleted after the method call has terminated. This way, circular references are avoided. But also comparing a method with itself for equality might return `False`, because the two bound methods are not the same object.
In certain situations, a reference to a bound method is stored in a variable (for example in some implementations of the Observer Pattern, like in section 1.4.1). Here special care must be taken to avoid memory leaks.

3.3 Threading, Multiprocessing, the GIL and GUIs

Python's threading module offers an easy way to split the program's execution into multiple threads, that run in parallel. It is very fast and simple to share even large amounts of data between the threads.
But this simplicity comes at a cost. In many cases, the Global Interpreter Lock (GIL) interrupts all other threads, when one thread is accessing shared data. This is necessary to ensure the consistency of the data and for automated deletion of unused objects. Since most data can be shared, this locking happens on a very regular basis, which degrades the theoretically parallel threads to a timeslot execution, that does not take advantage of the capabilities of modern multiprocessor computers.
A lot of research is being done on how to avoid the necessity of this global locking, so with newer versions of Python, the efficiency of multithreaded software tends to increase.

Libraries for graphical user interfaces like Qt or wxPython expect that all executions of routines, that affect the graphical display, are issued from within the same thread. Accessing the graphical user interface from within different threads is likely to cause the program to crash, but that is not ensured. Even an implementation that executes cleanly once can crash on another occasion. This unpredictable behaviour makes debugging extremely difficult, so it is best to avoid the multithreaded rendering of a graphical user interface.
In wxPython the wx.CallAfter method runs a given function from inside the user interface's event loop, which can be used to ensure, that this function is run in the correct thread. In Qt the event loop is not as exposed to the programmer as in wxPython, so there a different approach is required[3].

External libraries, especially those written in compiled languages like C, often implement their own memory management and can therefore release the Global Interpreter Lock during their calculations. This way, many calls of external library functions can be executed in parallel to another Python thread.

To make use of the hardware of modern multiprocessor computers, the multiprocessing module provides capabilities for starting a new Python process, that works on its own memory space. As the usual data objects are not shared between processes, the Global Interpreter Lock is not necessary to ensure the consistency of these objects, which allows more of the processes to run in parallel. The API of the multiprocessing module is very similar to that of the threading module.

The drawback of multiprocessing is that sharing data between processes is much more difficult than between threads. It is not possible to just access a variable from within

[3] e.g. https://stackoverflow.com/questions/10991991/pyside-easier-way-of-updating-gui-from-another-thread

another process. Instead, an object has either to be serialized and sent to the process as a message, or the object has to be written into a shared part of the memory. Both the message passing and the shared memory require copying the data, which can be very slow on large data objects.
Furthermore, there are certain requirements, which a shared object has to meet. The shared memory is statically typed, with a limited set of available types, so for exchanging an object over shared memory, it must be possible to convert the object's data into a format that is supported by the shared memory. For sharing an object with message passing, it must be possible to serialize the object, which is sometimes not possible. Especially function pointers are often impossible to serialize, as the function itself is not exchanged and therefore has to be available in the other process.

For sending a function pointer to another process, the referred function has to be available, when the program is imported as a module. The following code shows an example program with some shareable functions and some, whose pointer cannot be serialized.

```
from some_module import function1

def function2(arguments):
    def function3(other_arguments):
        pass
    pass

if __name__ == "__main__":

    from some_other_module import function4

    def function5(completely_different_arguments):
        pass
```

- A pointer to `function1` can be shared. (Given that program is written to the file main.py. Then it would be possible to write `import main` and `main.function1()`.)
- A pointer to `function2` can be shared for the same reasons as for `function1`.
- A pointer to `function3` cannot be shared, as `function3` is only defined in the context of `function2`. A new process would import `main`, but it would obviously not run `function2` to get a definition of `function3`.
- A pointer to `function4` can only be shared, if `some_other_module` is imported somewhere else in the source code, because everything inside the [`if __name__ == "__main__"`]-block is only run, when the Python file is run as the main

process and not when it is imported as a module.

- ▷ A pointer to `function5` cannot be shared, because it is only defined inside the [`if __name__ == "__main__"`]-block

In POSIX compliant operating systems, that implement the fork()-command, the objects that are passed to another process through its constructor are copied with the fork()-command and therefore need not to be serializable.

4 Software tools to facilitate collaborative programming

4.1 Doxypy - Automated generation of documentation

TODO

4.2 Unittest - Automated Testing of software parts

TODO

4.3 Version control systems

TODO

4.3.1 Subversion

TODO

4.3.2 git

TODO

Bibliography

[1] Creative Commons: *Attribution-ShareAlike license version 4.0*. https://creativecommons.org/licenses/by-sa/4.0/, 2013

[2] Rossum, Guido van ; Warsaw, Barry ; Coghlan, Nick: *PEP 8 - Style Guide for Python Code*. https://www.python.org/dev/peps/pep-0008/, 2001-

[3] The official Python documentatin by the Python Software Foundation: *The Python Profilers*. https://docs.python.org/3.4/library/profile.html, 1990-