The Pythonist Manifesto

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

Jonas Schulte-Coerne

July 9, 2015

Contents

1 Software Architecture **3**
- 1.1 The right attitude towards programming . . . 3
- 1.2 Programming slogans . . . 4
- 1.3 Mutable vs. Immutable data types . . . 7
- 1.4 Global state . . . 9
- 1.5 Design Patterns . . . 11
  - 1.5.1 The Observer Pattern . . . 11
  - 1.5.2 Other Patterns . . . 14
- 1.6 Code Smells . . . 15
  - 1.6.1 Circular dependencies . . . 15
  - 1.6.2 Side effects of imports . . . 16
  - 1.6.3 Excessively long functions . . . 16
  - 1.6.4 Copy and paste . . . 16
  - 1.6.5 The necessity to run the garbage collector . . . 17

2 Tipps for the advanced Python-beginner **19**
- 2.1 Exception handling . . . 19
- 2.2 Neat modules in Python's standard library . . . 22
  - 2.2.1 logging . . . 22
  - 2.2.2 collections . . . 23
  - 2.2.3 functools . . . 24
  - 2.2.4 inspect . . . 25
- 2.3 Special Python expressions . . . 26
  - 2.3.1 with . . . 26
  - 2.3.2 yield . . . 27
  - 2.3.3 lambda . . . 27
- 2.4 Advanced Python tricks . . . 28
  - 2.4.1 Features of the interpreter . . . 28
  - 2.4.2 Decorators . . . 29
  - 2.4.3 Profiling . . . 32
- 2.5 Differences between Python2 and Python3 . . . 33
  - 2.5.1 print . . . 33
  - 2.5.2 Imports . . . 34
  - 2.5.3 String / Unicode . . . 35
  - 2.5.4 Integer divisions . . . 36
- 2.6 Coding style . . . 37

3 Python Pitfalls **39**
3.1 Python's object model 39
3.1.1 Imports 39
3.1.2 Methods 40
3.1.3 Public and private properties 41
3.1.4 Duck typing 41
3.1.5 Multiple inheritance 43
3.1.6 The __del__ method 43
3.2 Object References and Garbage Collection 44
3.3 Threading, Multiprocessing, the GIL and GUIs 46

4 Software tools to facilitate collaborative programming **49**
4.1 Sphinx - Automated generation of documentation 49
4.2 unittest/pytest - Automated software testing 49
4.3 Version control systems 49
4.3.1 git 51
4.3.2 Subversion 52
4.3.3 A rant about centralized version control systems 53

Bibliography **55**

1 Software Architecture

1.1 The right attitude towards programming

It seems that some people (mostly those, who are new to programming) are afraid of exploiting their nearly unlimited power over the small virtual world of the program, they write. This makes them overly cautious, which slows down their development and prevents mistakes, that they could have learned from. So this section is about being fearless. Because, when programming, being fearless is easy, as this is an affair between only the programmer and her/his computer, which usually is not overly judgemental about some raised exceptions.

The first hint is for those pity souls, who have suffered the terror of ancient compiled languages, embedded systems without debug interface, JavaScript[1] or, the most abusive of all, MATLAB: Print out debug information! Not only in Python, but in any reasonable language, this is actually easy.
For a quick and temporary debug output, one can easily use the `print` function. While debug messages, which are meant to stay in the source code, should be implemented with the excellent functionalities of the `logging` module (as explained in section 2.2.1). Printing debug messages is very handy to check, if a certain piece of code is executed, or if it is skipped due to maybe a condition or an exception. It can be used to check if the computed data is as expected, by printing the data directly or some meta information like the output of `len`, `numpy.size` or `dir`. And of course debug messages can print status information like a progress bar, which indicates that the program is still alive.
Some programmers claim that nowadays, when debuggers are available even for high level languages like Python, printing messages for debugging is obsolete or even bad style. Admittedly, the use of a debugger can replace a few debug messages, but by far not all of them (e.g. status messages or the logging messages that stay in the code). And sometimes it is easier to just print a variable, then search for it in the memory dump of a debugger. So debug messages are still a very helpful tool and one should not be afraid to use it.

[1] Despite the lack of a usable print function and the carefree inclusion of every thinkable feature or paradigma make JavaScript seem unusable and confusing to even the most intrepid Perl programmers, JavaScript can be a fun language that actually has some virtues, as Douglas Crockford explains [2].

Secondly, "Try and error" is a valid way of developing software. Of course, doing something right, on the first attempt is very impressive and satisfying, but often, it is quicker to try someithing out and react to the error messages. One might even learn a new thing or two, when hazarding making a mistake.
And usually, the risk of running erroneous development code is negligible, as long as it does not overwrite important data or crash programs, that run for productive purposes. So as long as the developed software (including its data and other ressources like network servers) is sufficiently modularized, so that the development test runs can be separated from the productive execution of the software, "Try and error" is a formidable approach to solving problems in software development.
Be fearless.

And thirdly, have fun! When programming, a programmer is constantly solving small problems, which on its own is hugely motivating. But when she or he takes a step back and recognizes the complexity and elegance, that emerges from the combination of all these small solutions, the joy is almost beyond words.
On the other hand, one gets easily used to this motivation of steadily repeating successes, so that a problem, that cannot be solved rapidly, easily becomes discouraging. In such a case, it is important to avoid spending too much time on feeling unproductive while trying to figure out a seemingly impossible solution. First, one should check, if the problem can be divided into smaller subproblems, whose solution is more obvious. If that is not possible, one should go on to solve a different problem and come back to the current problem later. Sometimes before going on, it is necessary to program a workaround or an improvised solution for the current problem. If a proper solution feels out of reach, the extra effort for such a workaround is worth the effort. This break often allows to find a different approach to the problem or it at least gives some time to think about it, while not being in front of a keyboard and feeling urged to hack in a solution immediately.

1.2 Programming slogans

Just like in any other field, programming does also have its slogans, proverbs and milkbottle wisdoms, that are meant to convey the knowledge of an experienced programmer in the form of a catchphrase, that is easy to rembember. Consequently, the person, who recites these slogans in the most self-indulgent manner, must clearly be the wisest and most experienced programmer, which is why this manual has to have a slogan section, too. The following list only contains slogans, that are applicable to programming in Python, which allows to do without the all time classic "Goto, considered harmful".

Do only one thing and do it well

This is basically the software formulation of the ”divide and conquer” proverb. It means, that a complex system is best divided into separate components, in a way that each of these components has exactly one purpose, which it fullfills in the best way possible. Of course it is possible to apply this division recursively and divide a full system into subsystems, which ultimately consist of such components. And evidently, the subsystems must be focused on a certain area of tasks aswell.
The advantage of this strategy is, that it makes the components easier to program, as they are targeted at a very specific problem. Furthermore, since a component is the best available solution for a problem, there is no need to write a different component with the same purpose. And since all subsystems and components have their own ”area of interest”, on which they work, in order to fullfill their purpose, they can be programmed in a way, that they are independent of other subsystems or components. This makes it possible to instantiate them without instantiating too many other components as dependencies. Without this independent instantiation, reusing or testing a component would not be practically possible. And of course, once a better solution for a given problem is found, the respective component can be easily replaced without affecting other components.
Although it is not a single piece of software, the family of UNIX operating systems is a great example for the virtues of this slogan. With its modular set of exchangeable kernels, libraries and tools, the UNIX operating systems have achieved a versatility and efficiency, that is unparalleled by any other operating system so far.

Program to an API, not an implementation

As the name implies, APIs are the interfaces between different pieces of code. Usually, there has been made at least a modest attempt to make the API versatile and easy to use and to document it sufficiently. And because APIs are so well thought through, they are unlikely to change. So when using a piece of software, it is reasonable to rely on the API and trust its documentation. All the aforementioned points mostly do not apply to the implementations behind APIs. That code tends to change regularly and its documentation is usually appalling, which provides unpromising conditions to depend on, when developing a software component.

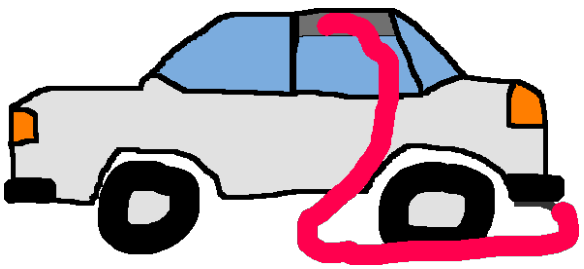

Figure 1.1: When heating a car, the decision to rely on the implementation detail, that the catalytic converter removes all carbon monoxide from the exhaust gasses, usually yields results which are both dramatic and undesired. So it is preferable to stick to the documented features and use the car’s heater.

So take for example a function, whose
documentation claims that it returns a
unique identifier. When retrieving such an identifier, one can expect it to be unique, but all other assumptions are not valid, as they are not documented. Neither can one assume, that the identifier values are produced in the same order, after the program has been restarted. Nor is it allowed to make any assumptions about the type of the returned identifier.
If one relies on internal details of an API implementation, the program will work at first, but it is very likely, that it will behave peculiarly, once this implementation has changed. Such errors are usually hard to find, as they often do not crash the program completely and the defective code is in a different area than the recent changes.

Explicit is better than implicit

This is one of the famous Python mantras. It basically means, that one should produce clean and readable code by avoiding implicit assumptions. The simplest example for this is passing arguments to a function as keyword arguments rather than positional arguments, when the number of arguments exceeds two.
But this mantra also refers to API design. It is usually better to ask for every value explicitely, rather than guessing it from another source. A good example for this is the implementation of a class. A class should ask for every necessary parameter in its constructor, so that objects of that class are always fully initialized.
If a parameter is not passed via the constructor, but later given via a setter method, the programmer of that class can decide between two evils. Either the object is not fully initialized, before that setter has been called, or the constructor tries to guess a reasonable value for that parameter. Such guessing code is usually messy, not documented and incapable of guessing always right, which will eventually result in an error. And, as the keen reader might have guessed by now, ”eventually” is much worse in this context than ”always”, because such irregular errors are harder to find, to reproduce and it is more difficult to verify their absence through automated tests.
The clean way would be to ask for that parameter in the constructor and give it a default value. Those default values are part of the API of a class, so it is specified explicitely, what value that parameter will have by default, rather than guessing the value implicitly.

Don’t program a mathematical function yourself

Especially for Python, there are good libraries for almost any general problem available. So it is not necesseary to program something as general as a common mathematical function on one’s own.
With mathematical functions, the usual drawbacks of homebrew implementations are

just suboptimal performance and maybe a bad API for that function. This is not good, but certainly not critical.
The topic becomes more severe, when it comes to security relevant components like cryptography algorithms. Here it is absolutely mandatory, to use an open implementation, that has been reviewed and tested by as many people as possible. The "security by obscurity" approach of using a custom cryptography algorithm has nearly always proven to be insecure (and if there is an application, where it hasn't yet, it is just a matter of time until it does prove to be insecure...).

If in doubt, leave it out

If one is not sure, that a certain feature is needed, this feature should not be added to the software. After a feature has been introduced, there is nearly always a component or a user that relies on it and thus makes it impossible to remove it, even if the feature is faulty or deprecated. So adding a requested feature later on is usually easier than removing an unused feature, that one does not want to maintain anymore.

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 explanation 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 makes 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 Global state

Global state is when mutable objects are accessible from several independent components of a software's source code. The problem with such objects is, that changing one of them has consequences, that reach beyond the scope of the current component. So the risk of undesired side effects is greatly increased. To cope with these side effects, it is usually necessary to perform extensive checks, when accessing such a global object. And when these checks fail, the resulting error is often hard to find and a lot of work to fix, because it involves many components of the software.

When thinking about global state, one is usually reminded of global variables and that they are considered to be evil. They actually are, but also other malicious constructions exist, which create global state:

- ▷ Mutable singletons[2]
- ▷ Modules (since they are singletons)
- ▷ File access

The classification of modules as harmful global state might need some explanation. Most modules just provide functionality and are not meant to be modified, which means, that they do not share state between the components which use the same module, so all is well. Other modules, like the `atexit` module, do have functions, which change the state of that module, which requires some caution. In case of the `atexit` module, this is usually not critical, as other software components are not affected, when one component registers a function, that shall be executed, when the program terminates.
But there actually are negative examples for global state in modules. One of these is the `random` module, which allows a reinitialization of the global random number generator with its `seed` function. This design decision has probably been made, to facilitate the writing of simple scripts, but in more complex software, this function can easily be misused. For example, when one component seeds the generator with a special value and expects a specific sequence of generated random numbers (this assumption is reasonable, as the generated numbers are only "pseudo random", which means that their generation follows a deterministic law). If another component changes the state of the random number generator (e.g. by retrieving a number or by calling the global `seed` function again), the assumptions of the first component about the generated sequence will be wrong. To circumvent this problem, one should always create a local instance of the random number generator (`random.Random()`), when the actual values of the generated numbers are of importance.

[2]Singletons are objects that are only instantiated once. When attempting to create another instance, one will only obtain the same instance, that was created before.

Another example of global state in modules is appending to `sys.path`, which might be tolerable in trivial scripts, but is absolutely prohibited in even small software projects.

File access is also a common source of errors due to global state. Especially with configuration files, it is likely, that different software components modify the same file, which can lead to race conditions or inconsistencies of loaded configuration parameters. So it is advised to encapsulate the access to a certain file in a single component, as this makes it easy to avoid race conditions. Furthermore, functions and classes should ask for all their parameters as arguments, rather than retrieving them from a global data source like the configuration. This encourages a software design, in which all external parameters are loaded only once (maybe at the beginning of the software's execution).

There is even a more theoretical justification for avoiding global state, which has to do with provability. Consider the following logical conclusion as an example for this:

$$\left.\begin{array}{l} A \Rightarrow B \\ B \Rightarrow C \end{array}\right\} \Rightarrow \quad A \Rightarrow C \tag{1.1}$$

This conclusion is only valid, when all objecs A, B and C are the exactly the same throughout the whole set of formulas. This requirement seems trivial, but with global state, it is easy to dissatisfy, as the following piece of code shall demonstrate:

```
threshold = 0    # a global variable
def A(parameter):
    return parameter >= threshold
```

When the value of the global variable changes, `A` might still be the same instance, but from a mathematical point of view, it has become a different function, since it might yield a different result for the same parameter. So testing, if the software works correctly, becomes extremely hard, because theoretically all conclusions, that are drawn from the test runs, have to be validated for every possible combination of values for the global variables, which is usually impossible.
The remedy for this problem is simply to make the global variable a parameter of the function.

```
def A(parameter, threshold=0):
    return parameter >= threshold
```

This way, the function always returns the same result for a given combination of arguments, so it remains the same function also in the mathematical sense. This allows it to test the function like a mathematical function, which can usually be done without brute force testing all possible parameter combinations.

Functional programming languages like Haskell go to great lengths to avoid global state. This makes programming in such languages very counterintuitive for programmers, who are only familiar with imperative programming languages like Python. But since the lack of state makes functional programming languages very similar to mathematical formulas, it is much easier to apply the methods of proving mathematical theorems on programs in these languages. Sometimes it is even possible to formally prove the correctnes of a given implementation.

1.5 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.

1.5.1 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 required 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)
    ...
```

1.5.2 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

1.6 Code Smells

Code smells are not really bugs, but indications, that something is probably wrong. A piece of software, which is bug-free, but whose source is contaminated with code smells still runs cleanly. Nevertheless it is likely, that future modifications of the source code will break something.
This section lists examples of common code smells and the damage, they will probably cause. Most of these examples are taken from a great talk by Nina Zakharenko, in which she describes the distress, in which a developer manoeuvres himself by working around an issue, rather than fixing it [7].

1.6.1 Circular dependencies

Often, circular dependencies between modules are not critical in Python. But sometimes, a dependency circle has than to be broken, by importing one of the modules inside a function. The background of this topic is being discussed in section 3.1.1.

This is usually an indication for a bad code structure, overly large modules or a more generally inept distribution of code into multiple source files.

Overly large modules or a inept distribution of code are usually easy to fix. Just make use of Python's powerful module system and group the code elements, that really belong together in a package or even a single source file, while avoiding to mash up independent pieces of code. The benefits of this method are, that this allows the reuse of that code in other parts of the software, facilitates testing of the software and avoids circular dependencies.

Fixing a bad code structure generally requires more work, that does not only involve moving code between source files without modifying it, but it also includes actual refactoring of the source code.
An example for a such bad code structure would be a decorator, that depends on the code elements (e.g. classes), which it decorates. It becomes immediately obvious, that such a decorator cannot be used anywhere else, but for this special purpose. So generally, such code structures sacrifice generality of the code (most often unnecessarily). And this makes testing the software a veritable mess, as the code cannot be tested intependently.

1.6.2 Side effects of imports

Technically, there is no difference between a Python script and a module. But while a script initializes some objects, works with them and then exits, a module is usually imported at the initialization of a software and remains alive, as long as the Python interpreter is running.

Often, projects start as a small Python script, which is merely a proof of concept. These scripts then tend to evolve to more complex programs, whose source code is distributed into multiple modules. When refactoring a script, in order to split it into modules, the hardest part is to avoid the global variables, that have not been a problem in the script. This makes it necessary to replace the global variables by new arguments in functions and new constructor parameters in classes.

Writing a module with global variables is highly discouraged, since this unnecessarily limits the reuse of the code and leads to seemingly non-deterministic behaviour. This is discussed in section 1.4.

1.6.3 Excessively long functions

Usually, there are two reasons for excessively long functions or methods. One is factory code, which is long, because it has to initialize a lot of objects. Factory code is very simple, because it rarely contains contitional executions or loops, so the excessive length here is usually not a problem and often unavoidable. The other reason is bad software design.

The non-factory code is the part, where the program's magic happens. It is likely that this code is very complex, due to loops and the sharing of data between multiple parts of the software. So it is strongly recommended, to divide the code into small bits (functions, classes), which perform only one task (see section 1.2 for more information).

Small functions are easier to write and understand. And their focus on one task, makes them less specific to one part of the software, which might allow using them somewhere else as well.

1.6.4 Copy and paste

Duplicated code is a mess. Changing it, requires modifications in multiple parts of the source code. Reusing components is confusing, because it's unclear, which copy is

to be used. And it makes the source code unnecessarily long.

Copying code is a clear indication, that the source code is not modularized properly. Instead of duplicating code, move that code into a separate function and use that function instead of the duplicates.

1.6.5 The necessity to run the garbage collector

This might be a Python specific code smell. And sometimes it is also very hard to avoid depending on the garbage collector.

Python handles most of its memory management with reference counting, which is simple and efficient. But in some cases, like circular references (see section 3.2), the reference counting hits its limits and the garbage collector has to be invoked. Apart from the performance penalties, that come with the necessity to run the garbage collector, circular references are an indication for incoherent dependencies between objects.

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` statement 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. The code example below shows how to catch only specific exceptions.
  In order to avoid masking unexpected exceptions, it is good practice to keep the `try` block as short as possible and put everything, that depends on the successful execution of the suspicious command, in an `else` block.

The following code example describes the blocks, in which the exceptions are handled.

```
try:
    # exceptions, that are raised while executing this block, are
    # handled by the following blocks, instead of crashing the program.
except:
    # this block is executed, when an exception is raised
    # multiple except blocks are possible, in order to allow specific
    # handling of certain exception classes.
else:
    # this block is executed after the try block, when no exception
    # has been raised.
finally:
```

```
    # this block is executed after all other blocks, independently
    # of any exceptions, that have or have not been raised in any of
    # the other blocks (not only in the try block).
```

It is possible to specify the class of the exception, which shall be handled by an `except` block. It is also possible to store the raised exception in a variable, which allows further disambiguation of the caught exception, for example by parsing its error message, and reraising the exception again, when it shall not be caught.

```
try:
    ...
except RuntimeError:
    # this block is executed, when a RuntimeError has been raised.
except (IOError, ValueError):
    # this block is executed, when an IOError or a ValueError has
    # been raised.
except (IndexError, AttributeError) as e:
    # this block is executed, when an IndexError or an AttributeError
    # has been raised. The variable 'e' contains the object of the
    # exception.
except BaseException as e:
    # this block is executed, when an error of any type (since all
    # exceptions are subclasses of BaseException) has been raised,
    # which has not been handled by the above except blocks. The
    # variable 'e' contains the object of the exception.
except:
    # this is a more concise version of the above block, when no
    # reference to the exception object is needed. In this example,
    # this block will never be executed, since all exceptions are
    # handled by the BaseException block.
```

It is not necessary to catch every possible exception, when writing a `try...except` statement. So general `except` blocks, like the last two of the example above are purely optional. When there is no `except` block, that handles the raised exception, this exception crashes the program, like there was no `try` block around it. This is usually the desired behaviour in the case of an unexpected exception.
It is also possible to write a `try...finally` statement, without any `except` blocks. This can for example be useful, to close open file handles, even when an exception is raised.

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 manual 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)
```

[1] After its inventors Gottlob Frege, Moses Schönfinkel and Haskell Brooks Curry. So sadly "Currying" does not come from "spicing up a function".

```
    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 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.

[2] read more about decorators in subsection 2.4.2.

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`.

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

The following lines are a startup script, that enables autocompletion for the interactive Python interpreter:

```
import readline
import rlcompleter
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, owner):
        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 [6], 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 PEP 8 [4].
The reasons for most of the recommendations in PEP 8 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 PEP 8, it is good to follow these recommendations aswell, just for the sake of a recognizable programming style.

Some points of PEP 8 are an issue of discussion though:

- 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 and even the quality of the code, as Raymond Hettinger pointed out [3].

- The indentation with four spaces is messy and, with inappropriate text editors, even painful. Nevertheless, if the written code shall be published, it is better to change to a decent text editor, than to switch to an indentation with tabs. Python is extremely fussy about its indentation characters and since most Python source code is indented with four spaces, it is best to obey that convention. Otherwise, developers, who are new to the project, are likely to introduce errors, by assuming, that the code was indented with four spaces.
  The general opinion in discussions seems to be, that although many people

prefer tab indentation over space indentation, consistency is valued much higher than personal preference on that topic [5]. Jamie Zawinski even gives some technical reasons for avoiding tabs in source code documents [8].

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 the programmer to improve the performance and the stability of the program's implementation.
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 1.5.1). 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 Sphinx - Automated generation of documentation

TODO

4.2 unittest/pytest - Automated software testing

TODO

4.3 Version control systems

Version control systems are mainly used for two purposes. The first is, as the name implies, versioning the state of a project. This is useful to track which feature has been added in which version and for going back forth between different versions of the project. The second purpose of version control systems is to manage the collaboration between multiple developers. For this it provides an easy way to convey the work of one contributor to the others. Furthermore, it has means to resolve conflicts, when two developers have modified the same part of the project.

Version control systems work best on text files, because they are capable of tracking the content of such files. This allows storing the incremental changes to a file, which allows for a more fine grained versioning and more intelligent conflict resolution mechanisms (and it saves some disk space). With binary files on the other hand, version control systems usually cannot interpret their content, which means, that in newer versions, the whole file is replaced, rather than just the modified lines. This can consume a lot of disk space and it does not allow merging the changes of multiple developers.

The workflow with a version control system is generally the following:

1. Retrieve a local copy of the repository or update a fomally retrieved copy
2. Do some changes to that local copy
3. Commit these changes to the version control system
4. Convey the changes by pushing them to a centralized server

Most version control systems fall into one of the two categories, "distributed" and "centralized" version control systems. In centralized version control systems (like CVS or Subversion), the local copy of the project data is merely a working copy and not a repository on its own. Each commit is directly uploaded to the repository on a centralized server, which means, that the last two points of the workflow enumeration are done in one step.
With a distributed version control system, the local copy is a full blown repository, in which the changes are first committed locally, before conveying these commits to other developers. Collaboration between developers is also possible without a global repository, as the changes can also be downloaded from another developer's local repository.
The main advantage of centralized version control systems is, that their lack of features makes them easier to use. Furthermore, the graphical user interfaces and plugins for file managers and IDEs[1] tend to be more mature for common version control systems like Subversion.
Distributed version control systems usually have a steeper learning curve, but their capabilities are definitively worth the effort, as is explained in the ranting section 4.3.3.

Most version control systems allow "branching" a project's repository. This means, that the project's state is copied and changes can be made to this copy, without affecting the main "branch" of the project. Branching is useful for implementing and conveying experimental features, while still keeping and maintaining a stable version of the project. After the feature has been fully implemented, its branch can be merged into the stable branch.
In a distributed version control system, the local copies are technically branches and uploading them to another repository (e.g. one on a centralized server) is a merging operation.

[1] "Integrated Development Environment", feature rich text editors for programming, that have been extended with capabilities for file and project management (e.g. Eclipse or Spyder)

4.3.1 git

git might be the fastest and most mature distributed version control system. It has proven its performance and project management capabilities in the development of the Linux kernel, where it has its origins. But nowadays, git is being used by a huge number of other projects aswell. Its development is also very active.

git is very rich in complicated features, which makes it easy to lose track, of what is currently going on. So one has to decide carefully, when to use a graphical tool and when to fall back to the command line. Plugins for IDEs and file manages are often limited by their user interface, which can be problematic in certain scenarios like resolving merge conflicts.

The documentation for git is very good, so querying the usage of a command via `git COMMANDNAME -help` usually helps a lot. Some tasks (like setting up certain things with the config) are more complicated, but good examples and HowTos can be found in the internet.

The following list introduces some of the most usual commands.

- `git clone URL DIR` makes a local copy of the repository, that is found under `URL`, in the local directory `DIR`.
- `git status` and `git log` print information about the repositories state. Graphical tools for this are `gitk`, `gitg` or `qgit`.
- `git gui` shows a convenient GUI with a diff viewer, a file selector and a comment field, that is very useful, when selecting the modifications, that shall be put into a commit.
- `git pull origin` pulls the commits from the branch, from which the local repository has been cloned. When `origin` is replaced with a different branch name, the commits will be pulled from there.
- `git push origin` uploads the pending local commits to the repository, from which the local repository has been cloned. When `origin` is replaced with a different branch name, the commits will be pushed there.
- `git checkout NAME` switches to the given branch, if `NAME` is a branch. If `NAME` is a file, that file will be checked out, which means, that it will be reset to the state of the last commit.
- `git merge BRANCH` merges the branch with the name `BRANCH` into the current

branch.

- `git branch NAME -t BRANCH` creates a new branch with the name `NAME`, whose origin is the branch `BRANCH`.

- `git stash` is a feature to stash and restore modifications without commiting them. Many git commands require a repository without pending modifications. So it is sometimes necessary to stash modifications, that are not yet complete enough for a commit.

- `git mergetool` is the first aid in case of a merge conflict, that cannot be resolved automatically. It starts a diff viewer with an editor (which has to be specified in git's config), with which the conflict shall be resolved manually.

- `git format-patch -n 13` exports the last 13 commits as patches (the number can be changed). This can be a convenient way to share a few very specific commits with other developers. But usually, it is better to convey modifications with the `push` and `pull` commands.

- `git rebase -i HEAD 13` allows rearranging, combining and deleting the last 13 commits (the number can be changed). Be careful to use this feature only for commits, that have not yet been conveyed to other repositories, as this will likely result into merge conflicts.

4.3.2 Subversion

Subversion is probably the most advanced centralized version control system. It is widely used in many projects and actively developed.

As the capabilities of centralized version control systems are fairly limited, they can be easily represented in a graphical user interface. The plugin for the Windows Explorer "TortoiseSVN" is for example very mature, rich in features and yet easy to use. On Linux, the plugin "RabbitVCS" is well integrated into some common file managers. It has similar features like TortoiseSVN and is also available for other version control systems. For the larger IDEs, there are also plugins like "Subversive" for Eclipse.

When creating a new project with Subversion or migrating an existing project to be controlled with it, special care needs to be taken with the directory hierarchy. Counterintuitively, Subversion represents branches and tags by files and directories, rather than managing them internally. This makes it necessary to allocate one directory for branches and one for tags, when creating a new repository. If this is omitted, the project cannot be branched or tagged.
A common method is to create a directory called "trunk", which contains the main

branch, in which all contributions are combined into a complete and stable version of the project. A directory with the name ”branch” usually contains subdirectories, in which the branches are stored. Similarly, the tagged versions are saved to their respective subdirectory in the directory called ”tag”.
Also it has to be made sure, that the access to the branch directory is sufficiently permissive, in order to avoid, that developers mess up the main branch in trunk with huge or unstable commits, because they cannot create a branch for the development of their feature.

4.3.3 A rant about centralized version control systems

Centralized version control systems have some drawbacks when it comes to managing projects with multiple developers, so it is strongly advised to use a distributed version control system whenever possible. To justify this claim, these drawbacks are listed in this section:

Encouragement of huge commits

In a centralized version control system, all commits are directly conveyed to all other developers. This encourages to only commit, when a feature is fully implemented and reasonably stable, which usually leads to huge commits with a lot of merge conflicts. With distributed version control systems on the other hand, the commits are only visible to other developers, after they have been pushed to a shared repository. This way, the developer is encouraged to make small commits, that track the development of the feature in a very fine grained way, so they are easy to merge and give a detailed history of the project's state.
A way to get around this problem, while still using a centralized version control system, is the excessive use of branches, which can be tricky, if the repository is set up carelessly.

Only global sharing commits

A distributed version control system allows pulling changes from any other developer, who has a copy of the respective repository. So it is easy and recommended to share newly developed features first within a small group and convey them only to the global repository, after the feature is reasonably stable.
In a centralized version control system, all commits are global, so the bleeding edge development is easily mixed with product maintenance. This scheme risks the stability of the product, as experimental features are likely to spill over to the production version of the project.
So the product maintenance must be explicitly separated from the development of new features by branching. This is also recommended with distributed version control

systems, but because of the aforementioned ability to share new developments non-globally, the contributions, that are commited to the global repository, tend to have a much higher quality and stability.

Overwriting updates with outdated versions

An inherent advantage of distributed version control systems is, that conflicts only occur between changes, that have been commited to version controlled repositories, which gives the version control system a lot more information, that is helpful for resolving such conflicts automatically.
In a centralized version control system, these conflicts occur between committed changes and uncommitted changes in a non-updated local copy. This usually leads to the current version of the file being overwritten to a non-updated but more recently modified version. So the changes from the developer, who has commited first, are lost. Common remedies for this problem are reducing the number of persons, who are allowed to commit, or asking everybody, if there are uncommited changes, before commiting. Both of them are completely ridiculous, as they undermine the capabilities of version control systems for project management.
The correct way to get around these conflict problems, is to use branches, in a way, that emulates the scheme of distributed version control systems, which means creating a branch for each developer or the development of each feature.

Problems with restrictive administrator privileges

If a version control system shall be used to distribute code between multiple developers, a centralized version control system always needs a server. This server needs an administrator, who does the administration and maintenance of the repository, which can include routine tasks like creating or merging branches, depending on the file access permissions of the server. Usually not every developer has permissive access to the files on the server, which is where the problems start, since they have to bother the administrator on a very regular basis with these routine tasks.
Distributed version control systems do not necessarily need a server, as the developers can pull updates directly from each other. Nevertheless, a central main repository on a server is usually useful[2]. For this central repository, the same problems with sparsely granted administration rights arise, just like with a centralized version control system. But the developer's local copies of that repository offer the full capabilities of the version control system to each developer, which includes unlimited branching and merging. Furthermore, the developers can share their contributions through their local copies, without relying on the access permissions of the central server. This allows to set the administrator privileges much more restrictively, without inhibiting the productivity of the developers.

[2]This server does not need to be a specialized version control server, as a simple shared directory is often sufficent.

Bibliography

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

[2] Crockford, Douglas: *JavaScript: The Good Parts.* https://www.youtube.com/watch?v=_DKkVvOt6dk, 2011

[3] Hettinger, Raymond: *Beyond PEP 8 – Best practices for beautiful intelligible code.* https://www.youtube.com/watch?v=wf-BqAjZb8M, 2015

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

[5] stackoverflow: *Why does Python pep-8 strongly recommend spaces over tabs for indentation?* https://stackoverflow.com/questions/120926/why-does-python-pep-8-strongly-recommend-spaces-over-tabs-for-indentation, 2009

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

[7] Zakharenko, Nina: *Technical Debt - The code monster in everyone's closet.* https://www.youtube.com/watch?v=JKYktDRoRxw, 2015

[8] Zawinski, Jamie: *Tabs versus Spaces: An Eternal Holy War.* http://www.jwz.org/doc/tabs-vs-spaces.html, 2000