Bachelor Thesis

September 21, 2019

Investigation of Python Documentation Comments in Open-Source Projects

Matej Jakovljevic

of Urdorf, Switzerland (15-922-917)

supervised by

Prof. Dr. Harald C. Gall

Dr.-Ing. Sebastian Proksch

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

Investigation of Python Documentation Comments in Open-Source Projects

Matej Jakovljevic

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Matej Jakovljevic, matej.jakovljevic@uzh.ch

Project period: 21.03.2019 - 21.09.2019

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgement

First of all, I would like to thank Dr.-Ing. Sebastian Proksch for supervising me during my bachelor's thesis. I am especially thankful for the regular meetings where he provided constructive feedback and guided me in the right direction whenever I faced difficulties.

Furthermore, I would like to thank Prof. Dr. Harald Gall for allowing me to write my thesis with the Software Evolution and Architecture Lab at the University of Zurich.

Finally, I would like to thank my family for their support and believing in me throughout my bachelor's thesis.

Abstract

Source code comments are ubiquitous in the field of software engineering. Developers use them to describe their intentions and improve the comprehensibility of the associated source code. The reusability of existing source code increases the importance of documentations comments, which provide information about the reused source code entity to end-users. However, the benefits of comments heavily depend on their content. To help developers write comments, and thus add value to the them, it is necessary to understand what content they usually contain and how they are structured. Since this has not been explored deeply for documentation comments in Python yet, this thesis provides a basis for future researchers by investigating whether Python documentation comments in open-source projects follow a particular syntax, and what content they contain.

Our investigation revealed that most Python documentation comments in open-source projects contain only one line of text, and that more than every third comment that contains more than one line of text follows a certain syntax. Our empirical study on the contents of documentation comments showed that a general description of the documented source code element is dominant, but descriptions of parameters and return values are also common to occur.

Zusammenfassung

Quellcode Kommentare sind allgegenwärtig im Bereich der Software-Entwicklung. Entwickler benutzen diese, um ihre Absichten zu beschreiben und die Verständlichkeit des dazugehörigen Quellcodes zu verbessern. Durch die Wiederverwendbarkeit von bereits bestehendem Quellcode haben dokumentations Kommentaren, die dem Benutzer Informationen über den beschriebenen Quellcode zur Verfügung stellen, an Bedeutsamkeit gewonnen. Dennoch hängen die Vorteile, die Kommentare mit sich bringen, stark von deren Inhalt ab. Um Entwickler beim Schreiben von Kommentaren zu unterstützen, und somit einen Mehrwert der Kommentare zu generieren, ist es notwendig zu verstehen, welchen Inhalt diese normalerweise mit sich bringen und wie sie strukturiert sind. Da dies für dokumentations Kommentare in Python noch nicht tiefgründig untersucht wurde, stellt diese Arbeit eine erste Grundlage für zukünftige Forscher, indem wir untersuchen, ob Python dokumentations Kommentare in Open-Source Projekten einer bestimmten Syntax folgen, und welche Inhalte in den Kommentaren vorkommen.

Unsere Untersuchungen haben gezeigt, dass die meisten Python dokumentations Kommentare in Open-Source Projekten nur eine Textzeile enthalten und das mehr als jeder dritte Kommentar, der mehr als eine Zeile Text enthält, einer gewissen Syntax folgt. Eine empirische Studie über die Inhalte hat ergeben, dass eine generelle Beschreibung des dokumentierten Quellcode Elements dominant ist, jedoch auch Beschreibungen von Parametern und Rückgabewert häufig zu finden sind.

Contents

Chapter 1

Introduction

Source code comments are a crucial part of software systems. As program languages do not provide a construct to provide information about the implementation in an informal manner, developers make use of source code comments to provide necessary information about the computational intent in informal linguistic [8].

Several existing studies that deal with code comments pointing out their importance. According to Woodfield et al. [29], code comments do, next to indentation, and meaningful variable names, increase the source code comprehension. Their contribution to comprehension can even be higher than those of the other two factors since they can be more detailed and can explain the intention of the described part of code. Yet, there is a trade-off the between benefits and disadvantages of comments, as for example comments that have not been adapted during an update of a module can lead to confusion of the reader [20]. Hartzman et al. [18] found that rewriting code documentations when updating a module is essential for maintaining software systems.

However, the benefits and drawbacks that source code comments bring with them affect different groups of people depending on the type of the individual comment. Two main types of code comments exist, which differ in their purpose and content [25]. On the one hand there are the documentation comments, which are written to provide information about the documented code to end-user developers, and, on the other hand, the inline comments which are written for other system internal developers, or oneself.

Haefliger et al. [17] showed that available code is actively reused by developers of open-source systems to solve technical problems. Reusable code increases the importance of documentation comments since developers rely on them to understand the reused piece of code.

Researchers have recognized the importance of documentation comments and have investigated them from multiple perspectives. For example, existing studies investigated mismatches and defects between source code and the appropriate documentation [26, 30]. Nevertheless, the structure and the notation of documentation comments in the programming language Python have not been explored deeply yet. Moreover, Python does not provide an official standard format for documentation comments, or short docstrings [10], which leads to the co-existence of multiple style formats. Our research has revealed that Google format, Numpydoc format, reStructuredText and Epytext, are the common style formats for python docstrings that have evolved [3, 4, 10, 13]. Where the reStructuredText format and the Epytext format only address the notation of a docstring, the Google format, as well as the Numpydoc format provide additional information about the content that should be described. In addition to the style formats, the docstring conventions, described in PEP 257 [9], make recommendations on the content of docstrings. Nonetheless, it is not explored yet, whether docstrings in practice follow one of the style formats, nor whether their content adheres to the recommendations, given by either an appropriate style format or the docstring conventions.

This lack of understanding hinders research that wants to support developers in their coding tasks. While a full body of research investigates how developers can be supported while writing source code, it is not clear how developers can be supported best, when writing comments. In this thesis, we will take the first step towards filling this gap. In the scope of this thesis, we are going to investigate whether docstrings in practice are repetitive concerning their content and the structure of the individual parts they describe.

To investigate the structure of docstrings, we first have to understand the different format styles and their appropriate structures. To be able to treat and analyze all docstrings equally we first need to build a unified meta-model from the existing style formats, and to convert the docstrings to it. This leads to our first research question:

RQ1 How does a unified meta-model for Python docstrings look like, and how can docstrings be converted to it?

The first research question is answered by investigating the common style formats of Python docstrings, and check them for similarities to build a unified meta-model. Afterwards, we are going to create parsers for the common style formats, which can be used to parse docstrings and convert them to the created meta-model. The results from the first research question can further be used to examine how the meta-model can be applied to docstrings in open-source projects, and therefore reveal whether these docstrings contain structured content or not. Here we set our second research question:

RQ2 How is the meta-model applicable on docstrings in open-source projects?

To answer the second research question we are going to analyze the results that we obtain from applying our conversion approach on docstrings in open-source projects. Further, we would like to investigate the contents that are contained in docstrings from open-source projects. Therefore, we formulate our third research question as follows:

RQ3 What do Python docstring in open-source projects describe?

To answer the third and last research question, we are firstly going to build a taxonomy, which enables the classification of docstring contents. In the next step, we let developers classify a set of Python docstrings in an empirical study. Then, we are going to analyze the obtained data, to be able to deduce inferences.

The results of our experiments show that a unified meta-model can be created, and that a syntactic approach exists, which enables us to convert docstrings, that are written in a particular style format, to the created meta-model. Further, the experiments show that there is a majority of one-liner docstrings in open-source projects, and therefore only a small percentage share of them follow a particular style format. Regarding only multi-line docstrings, we observed a significant increase in the percentage share. However, we found that parts of our meta-model are well applicable to the formatted docstrings in open-source projects. The content analysis revealed that a general description of what the documented code element does is present in the big majority of the docstrings, and that categories which are included in our meta-model are likely to occur.

Overall, this thesis present the following main contributions:

- A conceptual meta-model and an approach on a syntactic level that enables converting docstrings to the meta-model to consider them homogeneously.
- A first step in investigating whether certain parts of docstrings in open-source projects are structured and repetitive.
- An empirical study on docstring contents, as well as a taxonomy for them.

All developed tools, analysis files, used data files, and all other artifacts we used in this thesis can be found in our online appendix [24].

Chapter 2

Related Work

The existence of several studies dealing with source code comments underlines their importance in the field of software engineering. In this chapter we are going to present some studies which investigated source code documentation.

Source code comments represent a significant part of a software project [6] but why do developers comment their code? Several studies examined the advantages of using source code comments. Woodfield et al. [29] found in an empirical study, where they presented code in four different modularization types, each once with comments, and once without, to the participants, that source code comprehension can be improved by the addition of comments. It is to mention that they removed indentations and meaningful variable names in their case study, what possibly affected their results positively. Nonetheless, the study shows the advantages of source code comments and their importance, which validates further studies in this field.

Hartzman et al. [18] explain in their work that source code comments contribute to the maintainability of a software system, and that they have to be rewritten when source code is updated, to keep the quality level of the maintainability stable. [12] confirmed the importance of source code comments for maintainability. By surveying maintainers the found that source code and the comments it contains are the most important artifacts to help understanding a system.

Even tough source code comments often have a positive affect on comprehension, and maintainability, they may affect the system quality negatively. Jiang and Hassan [20] show in their study that correct, as well as up to date comments help developers to understand source code, but also that wrong, misaligned, and outdated comments can mislead developers, or even introduce bugs. The study underlines the importance of comment maintenance. As we investigate Python docstrings it is important to be aware of that they may introduce bugs, as the documented piece of code is reused by several end-user developers.

In the study from Fluri et al. [15] it is examined whether source code and the associated comments are changed at the same time. They found that newly added source code is rarely commented. Nevertheless, almost every comment change that is performed, is done during the change of the associated source code. In a later study Fluri et al. [16] investigated the co-changes of source code and comments over the history of a software system. To find the associations between comments and source code, they assumed that the proximity between them indicates an association, and that the comment describes the source code element it is associated to. They found that API changes and the associated comments do not co-evolve, but that they are rewritten in later versions. Generalizing this finding, as well as the findings from [20] indicate that docstrings can confuse, and mislead end-user developers. As the actuality of the comments is not assessed in the scope of this thesis, it is important to keep this in mind.

Other studies deal with the quality, and the content of source code comments. Steidl et al. [28] performed a quality analysis on them. The comments in their sample were categorized into seven

categories, which served as the base for their quality model. To measure the quality of source code comments they defined quality attributes for every category. They provide metrics to asses the quality attributes that detect quality defects in source code comments of a specific category.

Pascarella, and Bacchelli [25] defined a more detailed taxonomy with a two level hierarchy, to classify source code comments in Java open-source software systems. They manually classified a sample of source code comments, with the aid of a built web application, based on their taxonomy. As we are going to create a taxonomy, as well as a web application for the content classification, this is highly related to the third research question of this thesis. They additionally provided an approach for automatically classifying comments according to the created taxonomy by employing supervised machine learning [19], which inferred the classification function from the manually classified classifications. They found that an approach should start with some supervised data, since project specific terms are key for the classification.

Maalej et al. [22] examined the contents of API documentation in .NET 4.0 and Java SDK 6. With a grounded approach, followed by analytical reasoning, they created a taxonomy, which is used to rate randomly-sampled documentation units. Their results from the classification show that a significant amount of documentation units attached to API class members provide only little, or no value. Even though they propose a taxonomy for API documentation, the taxonomy created in the scope of this thesis may differ from it, as for example Python does not require type declarations for parameters, what leads to a need of information through the documentation of the source code entity.

Zhou et al. [30] analyzed directive defects in API documentation. They propose an automatical approach to detect defects in API documentation, by employing an constraint solver based on first-order logic which is based on the obtained results from their API document analysis. The focus in their study is set to parameter constraints, and exception throwing declarations. The subject of another study [26] is the investigation of type declaration mismatches in Python docstrings. They focused on docstrings that were written in the numpy style format, and categorized them into `Complete` (When all parameters are described in the docstring), `Partial` (When at least one, but not all parameters are described in the docstring), and `Missing` (When no parameters are described in the docstring). By manually inspecting a set of public methods, they found that the methods in their method sample were well aligned with less than 1% inconsistent documentations. In addition to the manual inspection, they designed the tool `PyID` which should help developers to keep their documentation and code well aligned. The tool uses a parser that collects the data type of each method, what seems to be related to the first part of this thesis, since we are going to use parsers for the convresion of a docstring into our meta-model. Nevertheless, the parser they used in their work recognizes Numpydoc docstrings, and fulfils a very specific task, where in the scope of this thesis we built parsers for all common style formats, which recognize more general components of a docstring.

Chapter 3

Python Docstrings

To enable an investigation of Python docstrings in open-source projects, we have to convert existing docstrings into a unified meta-model, which allows us to treat all docstrings in the same way, even tough they are written in different style formats. To create a meta-model we have to understand the syntax and the structures of the common style formats, which are introduced in this chapter. After investigating the style formats, we compose a meta-model and provide an approach to convert docstrings to it.

3.1 Python Recommendations

A documentation comment in Python or, short docstring, is marked differently than inline comments. Where inline comments are marked with a hashtag (i.e #), a docstring is wrapped into three quotes (i.e. """or ’’’). Python docstring conventions recommend to use double quotes, but it is also possible to use three single quotes [9].

Developers who would like to write more expressive plaintext docstrings are recommended to use reStructuredText markup. The reStructuredText markup language can not replace pure text blocks, nor is it its intention; it is intended as an alternative for more expressive documentation [10].

reStructuredText The reStructuredText markup language is developed in the Docutils project. The markup is easy to read, not only in its processed form but also in its source form. Furthermore, it can be easily typed in any standard editor and contains enough information to be converted to any reasonable markup format. Therefore, reStructuredText meets the generally accepted goals of the Python Documentation Special Interest Group (Doc-SIG). Additionally to that, reStructuredText can be extracted and processed into high-quality documentation. It is to mention, that reStructuredText is not the standard markup for Python docstrings, it is a recommendation since no standard exists [10].

Further, the docstring conventions differentiate between two types of docstrings referring to the line count. They provide recommended syntax for both types.

One-liner docstrings: One-liner docstrings, as the name suggests, fit on one line. Opening quotes, content, and ending quotes should be located on the same line. The content of one-liner docstrings should rather be written in imperative form, than in the descriptive form [9].

Multi-line docstrings: Multi-line docstrings include a summary, which is fits on one line. The summary is followed by a blank line, which followed by a more detailed description [9].

Besides information about a docstring's syntax, the docstring conventions provide recommendations for its semantics. Addressing semantics, a distinction between docstrings for modules, classes, and functions must be made [9].

Module docstrings: Module docstrings should contain a list of classes, exceptions, and functions, which are exported by the module [9].

Class docstrings: Class docstrings should summarize the class' behaviour, list its public methods and instance variables. In case the class has an additional interface for subclasses, the interface should be listed separately [9].

Function and method docstrings: Docstrings for functions and methods should summarize the behaviour and describe all existing arguments, return values, side effects, exceptions, and restrictions [9].

3.2 Style Formats

The absence of an official standard for documentation in Python has led to the emergence of several style formats. During our research, we came across four style formats which are often used in practice; some describe only the syntax; others additionally describe the semantic. In the following subsections, we will take a closer look at the found format styles. We are going to provide examples for the different docstring formats in the following sections. When providing examples, we refer to the example function listed in Listing 3.1. Further, the term "function" is used for both, function and methods.

```
1 def division(a, b):
2     return a // b
```

Listing 3.1: Example function

3.2.1 reStructuredText Format

To write more expressive docstrings, PEP-287 [10] recommends the usage of the reStructuredText markup language. The reStructuredText style format can be rendered by the documentation tool Sphinx,[1] what enables end-users to see the documentation comment in its processed form. As reStructuredText is a markup language, developers may use any available elements to describe specific parts of their docstring. Therefore, we had problems to finding a style guide for this format. Nevertheless, the Python docstring conventions provide us the information about what should be contained in a docstring, and by reading the Sphinx documentation, we found that certain aspects are declared by using field lists, which are heavily used in the reStructuredText style format [9, 11].

A field list is used to map field names and field bodies. The field name is wrapped by two colons, followed by the field body which describes the field name. Inside the two colons, which wrap the field name, a suffixed role can be defined [11]. An example of a field list is shown in Listing 3.2:

```
1 :param a: An example parameter.
```

Listing 3.2: Example of a reStructuredText field list

[1] http://www.sphinx-doc.org/en/master/

Hence, the reStructuredText style format uses field lists to capture arguments, variables, exceptions, returns, and the appropriate types. The following reStructuredText field lists are recognized and formatted nicely by Sphinx [5]:

param, parameter, arg, argument, key, keyword: Describe a parameter of the documented object.

returns, return: Describe the return value of the documented object.

raises, raise, except, exception: Describe exceptions, that are likely to occur when using the documented object.

var, ivar, cvar: Describe variables of the described object. There is no difference between var, ivar, and cvar, all are represented as `Variable` by Sphinx.

type: Describes the type of an parameter.

rtype: Describes the type of the return.

vartype: Describes the type of an variable.

We illustrate an example by writing a reStructuredText docstring for our example function:

```
1  def integer_division(a, b):
2    """Returns the rounded quotient of two integers
3
4    ...here could be a more detailed description
5
6    :param a: The dividend.
7    :type a: float
8    :param float b: The divisor.
9    :return: The rounded result of the division.
10   :rtype: float
11
12   """
```

Listing 3.3: ReST Docstring Example

As shown in Listing 3.3 the type annotation can also be included directly in the field list of the parameter.

3.2.2 Google Format

Google provides an own docstring style format. Regarding the indicators of a docstring, the format adheres to the docstring conventions described by PEP-257 [9]. Multi-line docstrings that follow the Google style format should have one single line at its beginning containing a short summary, followed by a blank line, followed by the rest of the docstring, which matches the docstring conventions as well. According to the Google style format, the one-line summary should be descriptive, what diverges from the docstring conventions [3].

The Google docstring style distinguishes between docstrings for functions, classes, and modules. Not only the syntax is addressed, but also the semantic. Certain aspects should be documented in a special section, which begins with a heading line that ends with a colon, followed by an indented section body on the new line [3].

Module docstrings are only required to include a licence boilerplate, whereas specific sections are required for function and class docstrings. The following sections should be contained in a docstring for functions [3]:

Args: Every parameter of the function should be listed in the `Args` section. The parameter name should be followed by a colon, which is followed by the parameter description. In cases where the source code does not contain a corresponding type annotation, the required type should be included in the parameter description. Descriptions, which do not fit on a single 80-character line should be continued on the next line with an indent of either two or four spaces. Functions that accept `*args` and `**kwargs` should contain them in the function's docstring.

Returns (Yields): This section describes the type and the semantics of the return value. It can be omitted in case the docstring starts with "Returns" or "Yields", as well as in cases where the function does not return anything.

Raises: In this section all relecant exceptions are listed.

The described sections are illustrated by using our example function Listing 3.4:

```
1  def integer_division(a, b):
2    """Returns the rounded quotient of two integers.
3
4    ...here could be a more detailed description
5
6    Args:
7       a: The dividend of type float.
8       b: The divisor of type float.
9
10   Returns:
11     The rounded result of the division of two numbers.
12     Returns a float
13
14   Raises:
15     ZeroDivisionError: Occurs when b is equal to 0.
16
17   """
```

Listing 3.4: Google Style Docstring Example

Classes should contain a docstring which describes the class. In case the class has public attributes they should be documented in an `Attribute` section, which is similar to a function's `Args` section, expect for the heading line [3].

3.2.3 Numpydoc Format

The Numpydoc style format uses the reStructuredText markup syntax for docstrings and can be rendered by Sphinx as well. Since the Numpydoc style format uses sections to document certain aspects in a docstring, one could say, that it is like a mixture of the Google, and the reStructured-Text style formats. Every section in the Numpydoc format, except the deprecation and warning section, is separated by a heading, which is underlined by hyphens. The format provides a clear

list of sections and the order they should be used in if they are applicable. The most information provided by Numpydoc is for function docstrings; thus, we start listing the sections that should be contained in a docstring for a function [4].

Like in the docstring conventions and the Google style format, Numpydoc function docstrings start with a short summary. Then, before providing a more detailed summary, the deprecation section is interposed if applicable. The deprecation section is not signalized by a heading like the other sections; it uses the deprecated directive from Sphinx (i.e. .. deprecated:: 'version') [5]. The expanded summary is followed by required sections, which should occur in the following order [4]:

Parameter: The parameters of a function are listed in this section. A single parameter is represented by putting the parameter name, and optionally its type separated by a colon on the first line, and its description on the next line, which is indented relatively to the parameters name.

Returns: The return values of a function are listed in this section. The form of this section is similar to the parameters section, except that names of the return values are not required, but the types are.

Yields: This section is only relevant for generators, and is built similar to the returns section.

Receives: In this section parameters, that are passed to a generator's `.send()` are explained. Therefore, this section is relevant for generators only too. If a docstring contains a receives section, it must contain a yields section as well.

The required sections can be followed by optional sections. Those optional sections are listed in the recommended order with an brief description of each [4]:

Other Parameters: This section should be used in case a function has a large number of keyword parameters to avoid cluttering the parameters section.

Raises: This section should be used to explain errors which are non-obvious or have a large change to get raised. Single raises are listed similar to parameters and returns, by putting the exception on the first line, followed by the description on the second line which is indented.

Warns: The warns section describes which warnings get raised under which conditions. It is formatted similar to the raises section.

Warnings: Different then the other sections the warnings section is not separated by a header. It can be written in free text or reStructuredText markup and contains cautions to the users.

See Also: This section is used to reference other functions from within a docstring. In case the referenced function is located in the same sub-module, the name of the function and the description are put on the same line and separated by a colon. The description can be omitted if the functionality is clear from the function name. In cases where the referenced function is located in another sub-module or even in another module, it should be prefixed by the module and the sub-module. A function `func_a`, which is located in another module, let say `examplemodule`, under the sub-module `examples`, would be listed as `examplemodule.examples.func_a`.

Notes: This section can be used to provide additional information about the code and discussion about the algorithm. It is possible to include mathematical equations written in latex format, or images by using the appropriate directives (i.e. '..math::' and '..image::').

References: In this section references, used in the notes section can be listed.

Examples: In this section the usage of a function can be illustrated by using the doctest[2] format. This section is optional but strongly recommended by Numpydoc. It is to point out that the example section is not intended as a test framework.

Listing 3.5 illustrates a Numpydoc example of our example function:

```
def integer_division(a, b):
  """Returns the rounded quotient of two integers.

  ...here could be a more detailed description

  Parameters
   ----------
  a : float
    The dividend.
  b : float
    The divisor.

   Returns
   -------
   float
    The rounded result of the division of two floats.

   Raises
   ------
  ZeroDivisionError
    Occurs when b is equal to 0.

  """
```

Listing 3.5: Numpy Style Docstring Example

Even if the raises section is not required by the Numpydoc standard, we put it into our example for consistency, since we put it into out example docstrings for ReStructuredText, and Google format as well.

For class docstrings according to Numpydoc format the same sections should be used as described for function docstrings, except for the return section, which is not applicable on classes. Additionally, an attribute section `Attributes` should be inserted after the parameter section to document a class' attributes. The constructor method should be documented in the class docstring as well. Usually, it is not necessary to document the public methods of a class, but in some cases it might be useful to provide the most relevant methods in a method section [4].

Every module should contain a docstring including at least a summary line. Other sections are optional and should be used in the following order: *Extended summary, Routine listings, item See also, Notes, References, Examples* [4].

[2] `https://docs.python.org/3/library/doctest.html`

3.2.4 Epytext Format

Epytext is a lightweight markup language that enables developers to format and structure their docstrings. Docstrings that use Epytext markup can be procesed by Epydoc to produce well formatted API documentation. As Epytext is intentionally very lightweight, it is recommended to use reStructuredText markup for even more expresive docstrings [13].

Epytext uses fields to describe specific properties of the documented object. Fields are marked by a field tag which starts with an '@'-symbol followed by the field name and ends with a colon. The field name can optionally be followed by a space and a field argument. Additional information about the field name or field argument follow after the field tag. The provided information can be style with other Epytext markup elements [13].Listing 3.6 illustrates how an Epytext field looks like:

```
1 @param a: An example parameter.
```

Listing 3.6: Example of a Epytext field

In Listing 3.6 `param` represents the field name, and `a` the field argument. Similar to reStructuredText field lists, Epytext fields can describe parameters, variables, return values, types, raises, and much more[3]. Listing 3.7 shows the docstring of our example function using the Epytext style format:

```
1  def integer_division(a, b):
2    """Returns the rounded quotient of two integers
3
4    ...here could be a more detailed description
5
6    @param a: The dividend.
7    @type a: float
8    @param b: The divisor.
9    @type b: float
10   @return: The rounded result of the division.
11   @rtype: float
12
13   """
```

Listing 3.7: Epytext Docstring Example

Similar as for the reStructuredText format we did not find any semantic information for Epytext docstrings. However, knowing how to represent specific properties in the Epytext format enables us to verify whether a docstring, that follows the Epytext notation, contains components that are recommended by the Python docstring conventions, the Google format, and the Numpydoc format.

3.3 Conceptual Meta-Model

All docstring formats provide information about the syntactic properties of a docstring. However, only the Google, and the Numpydoc style formats provide additional information about what should be described in it. Therefore, we refer to these two formats, as well as to the information from the docstring conventions to decide what should be contained in our meta-model. The

[3] `http://epydoc.sourceforge.net/manual-fields.html`

documented properties and sections differ depending on the component they are described in and can not be represented in the same meta-model. Therefore, we decided to restrict our meta-model to represent function docstrings, as functions occur more often than classes and modules, and we therefore assume that we will find more docstrings describing them than other source code elements.

Since the docstring conventions and the two docstring formats give similar information about what should be described in a documentation comment, the creation of a conceptual meta-model is straightforward. A short summary as well as the descriptions of a function's parameters, and return values are listed in every of the three references and are therefore included in the meta-model. The description of raises is optional according to the Numpydoc format but should be included according to both, Google style format, and the docstring conventions. As additional information about possible exceptions that can be thrown when using a function are important, we decided to include the description of exceptions in the meta-model as well. Additionally we included a description part, which contains running text and sections from the common style formats that are not explicitly defined in our meta-model. Thus, the meta-model looks like follows:

Short Summary: A docstring should contain a short summary that fits on a single line. We do not differentiate between imperative and descriptive summaries, as this thesis regards the syntactic level of docstrings.

Description: A docstring may contain a more detailed description that is separated from the summary. Components that are defined in any of the common style formats but not included in the meta-model belong to the description of the meta-model docstring.

Parameter Descriptions: If a function has input parameters, a docstring should contain descriptions of these.

Return Value Description: A docstring of a function that has an return value should describe this return value.

Exception Descriptions: Exceptions that can be raised by a function should be described in the docstring.

All components of the meta-model, except the description component have a defined representation in every common style format what enables us to use the respective syntax from the style formats for the conversion.

3.4 Conversion Approach

To convert the content of a docstring into the meta-model parsing the docstring is inevitable. To cover as many Python docstrings as possible we have created parsers for each of the common style formats. To create the parsers we used the tool Antlr4 [1].

3.4.1 Antlr4

ANother Tool For Language Recognition, short Antlr, is a parser generator that can be used to build languages, tools, or frameworks. After defining and compiling a grammar, Antlr generates a parser and a lexer in the target programming language [1]. An Antlr grammar consists of rules which define whether an input can be parsed or not. Antlr distinguishes between two types of rules, the lexer rules, which define tokens that are recognized by the parser, and the parser rules,

which are defined by lexer rules and other parser rules to specify valid input. The type of a rule is recognized by the first letter. Parser rules start with a lower-case letter, where lexer rules start with an upper-case letter. Additionally Antlr4 grammars allow to define actions, which are code blocks written in the target language, inside the rules. The use of actions makes it possible to call functions when a special tokens or parser rules occurs. Tokens that are not associated to any lexer rule, and therefore have no definition, can also be created in a grammar. Actions may be used to emit such tokens to the parser on defined events. Tokens that are defined by a lexer rule are recognized by a parser and have to be included in a parser rule. In case a token is recognized by the parser but not explicitly allowed in any of the allowed parser rules an exception is raised. The generated parser checks the lexer rules, respectively tokens in order of their definition. Therefore, a parser will raise an exception in case a not allowed token, which is defined before an allowed token, is recognized even if the allowed token is applicable as well. We illustrate the functionality of a parser with a simple example of an Antlr4 grammar shown in Listing 3.8 [2].

```
1  grammar Calculator;
2
3  calculus: NUMBER OPERATION NUMBER ANY_CHAR? EOF;
4
5  SPACES: [ \t\r\n] -> skip;
6
7  NUMBER: [0-9]+;
8
9  OPERATION: '+' | '-' | '*' | '/';
10
11 BIGLETTER: [A-Z];
12
13 ANY_CHAR: .;
```

Listing 3.8: Simple Grammar Example

The example shows a grammar for a simple calculator that takes two numbers and performs one of the four basic operations. The name of the grammar is `Calculator`. The only parser rule in the grammar is `calculus`, where `NUMBER`, `OPERATION`, `BIGLETTER`, `ANY_CHAR`, and `SPACES` represent lexer rules. The '+' in the `NUMBER` and `BIGLETTER` lexer rules indicates that one or more digits respectively upper-case letters may occur in the corresponding token. The `OPERATION` token has four alternatives which are separated by '|', and the skip command in the `SPACES` token means that spaces will be ignored by the parser. The '.'-character That defines the `ANY_CHAR` token indicates that every character is accepted. The defined parser rule in Listing 3.8 expects a number, followed by an operation-sign and a number, which are optionally followed by an arbitrary character, indicated by the question mark after `ANY_CHAR`. The predefined token `EOF` indicates the end of the input. To illustrate which inputs are parsable and which are not we provide examples in Table 3.1

- The input in the first row of the table is parsable since `ANY_CHAR` is optional.
- As spaces are ignored by the parser, the second input example is parsable as well.
- The 'A' in the third example is contained in the `ANY_CHAR` token but the input is not parsable since the parser reads the defined tokens from top to bottom, and therefore first recognizes the `BIGLETTER` token which is not expected by the parser rule. Hence, the parser raises an exception.

Table 3.1: Input Examples for Parser generated from Listing 3.8

Input	Parsable
1 + 1	Yes
1 * 1	Yes
1 + 1 A	No
1 / 1	No
1 - 1 #	Yes
1 + 1 ##	No

- The input example in the fourth row is not parsable as well, since the parser again first recognizes the `NUMBER` token, which is not expected at this position.
- The fifth example is parsable as the '#'-symbol is only contained in the `ANY_CHAR` token which is expected by the parser.
- The last example is not parsable since either one or none occurrence of the `ANY_CHAR` token is expected.

3.4.2 Parser Specification

We created a parser for each common docstring format style. While dealing with the individual formats and the question of how to convert a docstring into our meta-model, we were able to define properties that our parsers have to fulfil.

- To enable converting a docstring to the meta-model, properties defined in the meta-model should be recognized by every parser referring the particular syntax of each format.
- As documentation comments are written by humans and humnas are error-prone, the parsers should also parse docstrings that differ slightly from the format to intercept docstrings that should be written in a certain format but do not strictly follow the format due to small mistakes in the syntax.
- As text blocks are valid in every docstring format they should be parasble by every parser.
- To prevent that the defined properties from the meta-model are erroneously moved to the description part, what would happen in case a parser parses a docstring written in another style format, docstrings that follow a particular style format should only be parsable by the parser created for the appropriate format.
- Docstrings containing syntactic properties of two or more formats should not be parsable by any parser, because every parser would only recognize parts of the docstring written in the corresponding format what leads to faulty results.

During the parser creation we oriented ourselves to the defined properties and tried to implement them. In the following we are going to explain how we achieved these properties.

Recognize Meta-Model Components To ensure that a docstring can be converted into the meta-model we defined a parser rule for each of its components in every style format grammar. The parser rules are defined by the indicator of the component and the expressions for the allowed syntax inside the component. An example of the parser rule of the arguments component in the Google format style grammar is shown in Listing 3.9, where the deeper nested parser rules are omitted. As shown in the example the parser rules contain other parser rules, this enables us to access the single elements of a component. While parsing an input the parser rules can be extracted from the parser as rule context, what enables us to check which part of the input is parsed by the defined parser rule. This allows us to assign the input which is parsed by a certain parser rule to the corresponding component in the meta-model.

```
1 argsComponent : ARGS (NEWLINE+ INDENT)? (args NEWLINE*)* DEDENT?;
2 args : argName argDescription;
3 ...
```

Listing 3.9: Google Format Argument Component Parser Rules

Allowing Deviations To make the syntax of the individual format parsers looser, we decided to add optional indents and dedents to some parser rules, make the first letter of section indicators in the Google and the Numpydoc formats case insensitive, and to ignore the order in which the defined components occur, except for the short summary which has to occur on the first line.

```
1  """
2     Calculates sum of two parameters.
3     Args:
4
5        param1: first parameter
6
7
8        param2: second parameter
9
10    Returns: Sum of the input parameters
11    """
```

Listing 3.10: Input that diverges from Google Format Syntax

Listing 3.10 illustrates an example of a docstring that does not strictly follow the Google style format, since the arguments in the `Args` section have multiple line breaks among them, and the `Return` component is not followed by a line break after the section header. Nevertheless, this docstring input should be parsable by our Google format parser as it contains only small deviations.

To enabled the parser to recognize indentations, we had to define the tokens `INDENT` and `DEDENT` which are not associated with any lexer rule. The created tokens are emitted by an action that is called after every line break. The action checks how many indentations the previous line had, counts the indentations of the current line, and compares them. In case the indentation count is the same no token is emitted. If the indentation count is greater than the indentation count on the previous line the `INDENT` token is emitted. Otherwise if it is smaller than on the previous line the `DEDENT` token is emitted. Therefore, indents and dedents have to be explicitly included in a parser rule at the position they are expected, otherwise the parser will raise an exception when an indent or a dedent occurs. Hence, we added optional indents and dedents to some parser rules, even tough they do not expect any. For example the Numpydoc docstring in Listing 3.11 should

be parsable by our Numpydoc parser, although the format does not expect the parameters to be indented relative to the section header.

```
"""
Parameters
 ----------
   param1 : int
      The first parameter.
   param2 : str
      The second parameter.
"""
```

Listing 3.11: Indented Parameters in Numpy Format

Parse Text Blocks As text blocks are valid in every docstring format they should be parsable by every parser. To ensure that text blocks are parsable, we defined a parser rule `description` that accepts all tokens that are not specifically defined for one of the parser rules that represents a component of the meta-model. When the parser starts parsing it is first checked whether other, more specific, rules are applicable, if not, it tries to apply the description rule. Using the description rule implies that other components of a format, which are not defined by a parser or lexer rule, are parsed and moved to the description. This is intended since not all components are explicitly defined in the meta-model, but the content should remain complete.

Recognize Style Formats Parameters, return values, and exceptions which are written in a particular style format must not be parsable by another parser's description rule since in this case they would be moved to the description part of the meta-model, what would falsify the results. To ensure that a docstring following a certain format can only be parsed by the corresponding parser, every parser contains a token that defines the characteristics from the other style formats, like for example section headers. This token is defined but not allowed in any of the parser's parser rules, therefore the parser recognizes the token if it occurs in the input and raises an exception.

3.4.3 Docstring Conversion

The defined parsers can be used to convert docstring into our meta-model. Every time a parser enters a rule context defined by a parser rule that represents one of the meta-model components it adds the content of the rule context to the meta-model object. When entering the rule context for the parameter, return, or the exception section, the parser adds every single parameter, return value, or raise to the corresponding list in the meta model, as well as a boolean for the corresponding component which indicates that the component was found in this docstring. If a description rule context is entered the whole context is added to the description of the meta-model object and the boolean which indicates whether the docstring contains a description component is set to true. The short summary and the corresponding boolean are just added to the short summary component in all style format parsers except the one for the Google style format. Since the return component may be omitted in Google style docstrings if the short summary describes what is returned, we check whether the summary contains the words 'return' or 'yield', in which case the parser would add the short summary to the short summary and the return component of the meta-model object. If the summary does not describe what is returned the context is just added to the short summary component.

Chapter 4

Docstrings in Practice

To investigate how our created meta-model is applicable on docstrings in practice, we are going to gather docstrings from open-source projects and apply our conversion approach on them. After converting the docstrings into the meta-model form we are going to analyze the obtained data to be able to deduce the applicability of the meta-model.

4.1 Docstring Gathering

To obtain a set of docstrings we first had to procure open-source Python projects. To do so, we used the code hosting platform Github[1] which hosts over 100 million repositories and is used by more than 40 million people. Since not every project contains documentation comments, we restricted our selection to the 100 top projects according to their star rating. We assume that code from projects with a high star rating is reused more often by other developers, and therefore should contain more docstrings than code from projects with lower star ratings. The Github API[2] enabled us to clone the projects directly from source code. Since we cloned the repositories directly from source code, we were able to automatically add the author and the default branch, which are obtained through the Github API, to the paths of the repositories. Adding the author and the default branch to the repository path enables us to recreate the url of any source file in the repository. After cloning the Python repositories the docstrings had to be extracted from the Python source files.

Extract Docstrings from Python Files To extract the docstrings from their Python source file we used the predefined Antlr4 grammar for Python3.[3] As docstrings are string literals, which belong to the expression atoms [9, 27], we had to check for expression atoms in the source file. The Python3 grammar defines a parser rule for expression atoms which is deeply nested into the parser rule for statements. Since docstrings may only occur in the first statement of a module, class, function, or method definition [9], we iterated only through the first statement of each source code component that could possibly contain a docstring. To find the docstrings in a source file, we defined conditions which are checked every time the generated parser enters certain rule contexts which represent the parser rules. At the beginning of a file it is checked whether the first matched entered rule context is a statement context. If yes it is further checked whether the statement contains a simple statement[4] by checking if a simple statement context is present in the

[1] https://github.com/
[2] https://developer.github.com/v3/
[3] https://github.com/antlr/grammars-v4/blob/master/python3/Python3.g4
[4] https://docs.python.org/3/reference/simple_stmts.html

Table 4.1: Old and New Folder Structure

Source File Path	/python/cpython/blob/master/Lib/code.py
Docstring Path	/python/cpython/blob/master/Lib/code/functions/78.txt

children list of the statement context. If a component's first statement contains a simple statement the first child of the simple statement is taken and iterated through. During the iteration it is checked whether every entered rule context has exactly one child since a docstring is a standalone expression atom, and therefore the only element in a simple statement. In case every deeper nested rule context has exactly one child and an atom rule context is found at the end of the iteration, it is check whether the found atom expression starts with three double or single quotes, to verify whether a docstring is found. When the parser enters a class or a function definition, the first statement of the class or function suite is checked for docstrings in the same way as it is done in the first statement of the file. When a docstring is found, the part of the input that matches the definition of the atom parser rule is extracted from the atom rule context which in this case represents the content of the docstring. In order to know where, and in which source code entity the docstring can be found in the source file we extract additional information about the position of the docstring by checking on which line the first token of the atom rule context occurs, and information about the type of the component in which the docstring is defined.

Prepare Docstrings for Analysis After extracting the docstrings from their source files, we wrote every docstring to a `.txt` file named after the line number on which it occurs. The created `.txt` file is stored in a folder which has the same relative path as the docstring's source file extended with the name of the source file and the type of the source code entity which is documented by the docstring. Naming the file after the line number on which the docstring occurs in the source file ensures that docstrings from the same source file never have the same name.

Table 4.1 illustrates the relative path of a Python source file after cloning a repository from Github with the additional information about author and default branch, and the new relative path of the created `.txt` file. Due to this folder structure we know that the project `cpython` from the author `python` contains a functions docstring on line 78 of the `copy.py` file. Since the information from the relative paths and information about the line counts of the docstrings may be useful in the data analysis we add them to the docstring objects. As all docstrings now are contained in a seperate `.txt` file they can be parsed and converted into our meta-model by using our conversion approach.

After every parser has tried to parse all docstrings we merge the obtained meta-model objects. To ensure that the dataset of docstrings does not contain the same docstring multiple times we add only docstrings whose relative path, which is unique due to our folder structure, is not already contained in the dataset. The resulting dataset contains all converted docstrings that are parsable by any of the style format parsers. To allow an analysis we write the meta-model objects from the obtained dataset to a `.json` file which can exported be and further processed.

4.2 Analysis of Docstrings in Practice

We used Python and the data analysis toolkit Pandas[5] to analyze the obtained data. To allow an analysis on the obtained meta-model objects we read in the created `.json` file, and added the objects to a dataframe.

[5] `https://pandas.pydata.org/pandas-docs/stable/`

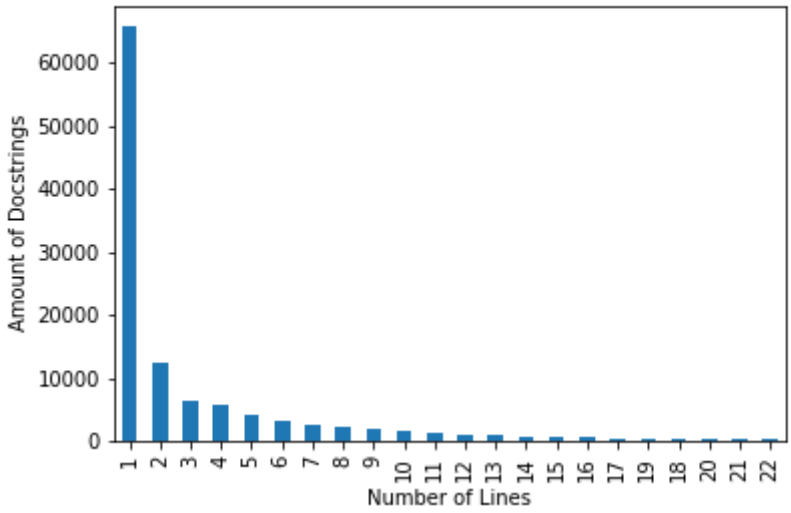

Figure 4.1: Line Count Distribution of Function Docstrings

In a first step of the analysis we examined how many docstrings could be parsed by our parsers by reading the number of objects contained in the dataframe. From the total of *151,740* docstrings we gathered from Github, *148,764* were parsable, leading to *98.04%*. The high percentage was expected as text blocks and docstrings that do not follow a specific format style, in general are parsable by our parsers where their content is added to the description part of the meta-model. By reading the distinct project names out of the dataframe we could figure out that the parsable docstrings come from *91* distinct projects.

Since text blocks are parsable by all of our parsers it is possible that docstrings that document classes or modules are converted to the meta-model as well. In this case their whole content would reside in the description component, or in the short summary component if it contains only one line of text. To obtain a meaningful analysis we decided to remove all class and module docstrings from the dataframe as our meta-model represents function and method docstrings. Removing the class and module docstrings resulted in a dataframe with *115,733* docstrings. Therefore, we can state that function docstrings with a percentage share of *76.27%* of all parsed docstrings represent the large majority of docstrings in open-source projects, what validates the restriction of the meta-model to represent function and method docstrings. After investigating from how many different projects the function docstrings in our dataset originate, we noticed that the number of projects shrunk from *91* to *87* what implies that *4* projects do not contain function docstrings.

4.2.1 Style Format Occurrence

Since we are looking for consistencies in the structure of docstrings we examined the occurrence of docstrings with a style format. This could be done since in case a style format was recognized by the appropriate parser, it is marked in the meta-model object. The investigation revealed that *20,727* of the docstrings in the sample follow a particular style format. Compared to the total of all parsable function, or method docstrings, about *17.91%* are written in one of the common style formats. As we assumed that the majority of the one-liner docstrings do not follow the notation of

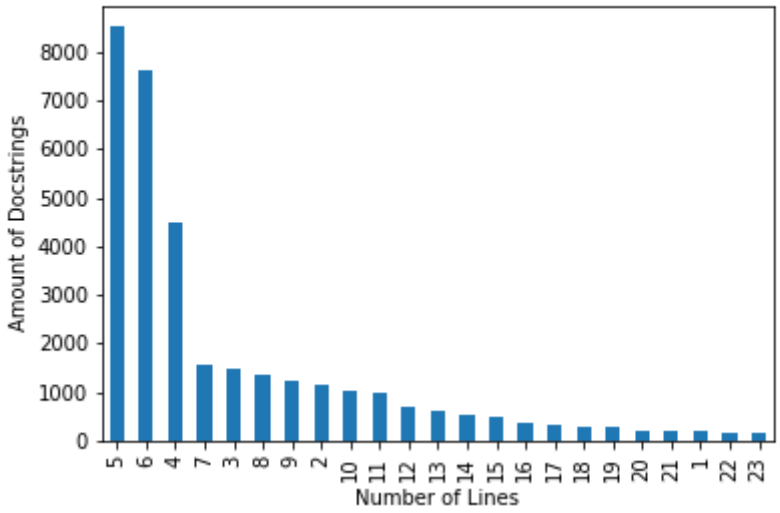

Figure 4.2: Line Count Distribution of Docstrings that follow a particular Style Format

a particular standard we investigated the distribution of line counts contained by docstrings. Figure 4.1 shows the line count distribution of the lower 97.5% of the parsable function and method docstrings. As the figure shows the majority of the docstrings in the set have only one line of text. To examine whether our assumption about one-liner docstrings is correct we extracted all docstrings that follow a particular style format and analyzed their line count distribution which is illustrated in Figure 4.2. The figure contains only line counts from docstrings that follow a particular style format and represents the lower 90% of the line counts since plotting all line counts that occur would result in an unclear figure with too many values. As the figure illustrates only few docstrings that follow a particular style have a line count of one (*211* in our dataset).

Occurrence in Multi-Line Docstrings To investigate the presence of formatted docstrings in multi-line docstrings we made the same distinction as the Python docstring conventions do [9]. We separated our dataset of docstrings into one-liner docstrings and multi-line docstrings, where every docstring that contains one line of text belongs to the one-liner docstrings regardless the position of the beginning and end quotes. For the further analysis we focused on multi-line docstrings since the great majority of the docstrings that follow a particular format contain more than one line of text. To keep our data consistent we removed the *211* one-liner docstrings from the dataframe that contains the docstrings with a recognized format.

Regarding only multi-line function docstrings, we observed that the number of projects in the dataframe further shrunk from *87* to *84* what shows that in *three* of the projects not a single multi-line function docstring can be found. Nevertheless, the number of formatted docstrings remains almost unchanged. We report *20,516* multi-line docstrings written in one of the four common style formats in our dataset, whereas the total number of function docstrings has more than halved. From initially *115,733* function docstring in the dataset, we obtained *49,816* multi-line docstrings, which leads to an increase from *17.91%* to *41.18%* of docstrings with a certain format, what indicates that more than every third multi-line docstring in open-source projects follows a particular format. Table 4.2 summarizes the distribution of docstrings with a particular format.

Table 4.2: Amount of Docstrings with a particular Format

	No of Docstrings	% of Docstrings with Format
All Function Docstrings	*115,733*	-
Docstrings with Format	*20,727*	*17.91%*
Multi-Line Docstrings	*49,816*	-
Multi-Line Docstrings with Format	*20,516*	*41.18%*

4.2.2 Distribution on Projects

Since it is possible that many docstrings in large projects follow a particular format, and therefore the number of docstrings following a format may be influenced heavily by only few projects, we investigated the distribution of docstrings written in any of the common formats on the different projects. To determine the distributions on the projects we used the dataframe containing all the multi-line docstrings that are written in a style format. Since we added the information about the source project to the meta-model object, we could simply read this value to determine which docstring comes from which project. To check whether the docstrings in the dataframe come from few projects or whether they are spread over many projects, we calculated their occurrences in every project from the dataframe. Investigating the occurrence of formatted docstrings on the project level revealed that *9,267*, which is about *45.17%*, belong to only three open-source projects. Regarding the five project with the most formatted docstrings we found that *58.24%* of the docstrings belong to these projects. The percentage further increases to *77.23%* when we consider the top ten projects. Additionally we report that docstrings that are written by using one of the common style formats are contained in only *73*, instead of the initial *84* open-source projects, that contain multi-line docstrings for their functions. Therefore, we report that developers in *11* open-source projects with multi-line function docstrings do not follow a particular style format.

The results show that only few open-source projects contain the large majority of formatted docstrings, and that the total number of formatted docstrings in the remaining projects has to be relatively small. Nevertheless, it is possible that the projects with many formatted docstrings contain a lot more docstrings in general and therefore the percentage share of formatted docstrings is possibly not greater then in the smaller open-source projects. Therefore, we examined the relative occurrence of formatted docstrings on the project level. To calculate the percentage shares we used the dataframe containing all parsed multi-line function docstrings and the dataframe that only contains multi-line docstrings that follow a style format. We calculated the occurrence of every single project in both dataframes by grouping them by the project, and obtained a new dataframe that contains the project name, the total count of docstrings in the project, and the count of docstrings that follow a style format, by merging the two dataframes. To analyze the occurrence of formatted docstrings in the individual projects we added a new column which contains the percentage share of docstrings that follow a particular style format to the new obtained dataframe by dividing the number of formatted docstrings by the number of all docstrings in a project. The boxplot in Figure 4.3 illustrates the results of the investigation. The results indicate that the percentage of docstrings written in a particular style format in 50% of the remaining projects is between *8.26%* and *67.97%*, which represents the first quantile, and the third quantile respectively. The median of the percentages of formatted docstrings is *37.50%*, which means that 50% of the projects contain more than *37.50%*, respectively less than *37.50%* formatted function docstrings.

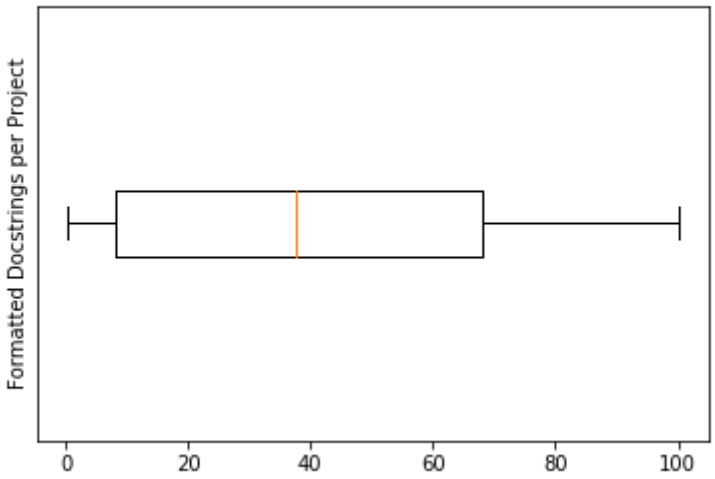

Figure 4.3: Distribution of the Percentage Shares of Formatted Docstrings in Projects

Table 4.3: Percentage of Meta-Model Components Present in the Docstrings

	Amount	Contained in % of Docstrings with Style Format
Docstrings with Style Format	21972	100%
Summary	*14,395*	*70.16%*
Parameter Description	*14,198*	*69.20%*
Return Value Description	*11,754*	*57.29%*
Exception Description	*1,528*	*7.45%*

4.2.3 Meta-Model Validity

The findings that *41.18%* of all multi-line docstrings in our dataset follow the notation of a particular style format as well as that they are present in *73* of *84* projects where the median and the mean of the formatted docstring percentage shares is similar to the overall percentage, does not provide information about the validity of our meta-model. To verify whether our meta-model can be applied on docstrings in practice, we investigated how often docstrings, that follow the notation of a particular format, provide information which is not converted into the description component of the meta-model, but into either summary, parameter, return value, or exception description. Here again we investigate the multi-line docstrings that document functions.

First we examined how often the components occur by counting the booleans which we added to the meta-model object during the conversion. The investigation showed that summary, parameter, and return description are present in more than *the half* of the docstrings, where the exception description is occurs only in *7.45%*. Table 4.3 Shows how often each component occurs in our dataframe.

Since the percentage of the component occurrence from all docstrings is strongly affected by few big projects, we examined the component occurrences on the project level too. To calculate

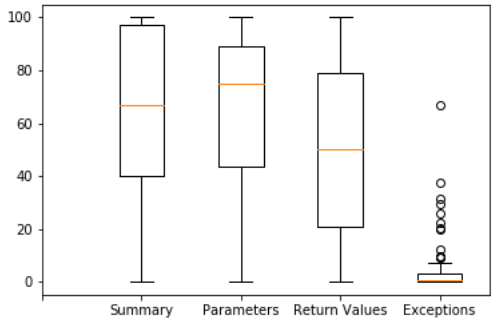

Figure 4.4: Boxplots of Component Distribution among Projects

the percentage shares of the component occurrences per project, we again grouped the dataframe containing the multi-line docstrings by their source project and defined a function, that counts the occurrence of a specific component by checking the corresponding boolean flag from the meta-model object. The function returns the percentage share of every component by dividing its occurrence by the total amount of docstrings in the project.

The statistical key figures from the results are illustrated in form of boxplots and can be found in Figure 4.4. The medians for the percentage shares of short summary, return value, and exception component in the individual projects are slightly smaller than the obtained percentage share from all formatted docstrings, what indicates that docstrings in the big projects generally describe these components more often. The median for the occurrence of the parameter description is higher than the overall occurrence, what suggests that structured docstring in bigger projects describe less parameters than the structured docstrings in the smaller projects.

After checking how often the multi-line docstrings describe the individual components, we are interested in the combination in which they usually occur. To investigate in which combinations and how often the components can be found, we used a frequent itemset miner from `mlxtend`.[6] Through the frequent itemset miner we were able to obtain information about which components of the meta-model are described together and how frequently the combinations occur. We provided the booleans, that describes whether a docstring contains a component or not, as input and obtained results about the frequency of the component pairs which are illustrated in Table 4.4.

As Table 4.4 shows, the summary, the parameter description, and the return description relatively occur often in any combination. All three together occur in *35.66%* of all docstrings that follow a particular style format. Since expection descriptions occur rarely in the formatted docstrings from our dataset, combinations including the exception description component occur barely as well. Therefore, the combination of all four components is reported in only *4.72%* of the multi-line docstrings.

Since the frequent itemsets could be strongly affected by the few big open-source projects in our dataset as well. We determined the frequent itemsets on the project level too. For the analysis of the itemsets on the project level, we focused on itemsets that contain the short summary, parameter description, and return value description components, since these components occur frequently and are therefore more likely to be present in the individual projects. To obtain the results we again grouped the docstrings by their source project. Afterwards, we provided the booleans that indicate whether a component is present or not as input for the itemset miner.

[6] `http://rasbt.github.io/mlxtend/`

Table 4.4: Percentage of Meta-Model Component Groups Present in the Docstrings

Component Combination	How Often in %
Summary, Parameter Description	**52.76%**
Parameter Description, Return Value Description	**45.56%**
Summary, Return Value Description	**44.03%**
Summary, Parameter Description, Return Value Description	**35.66%**
Parameter Description, Exception Description	**6.76%**
Summary, Exception Description	**6.66%**
Summary, Parameter Description, Exception Description	**6.15%**
Return Value Description, Exception Description	**5.53%**
Parameter Description, Return Value Description, Exception Description	**5.19%**
Summary, Return Value Description, Exception Description	**4.99%**
All	**4.72%**

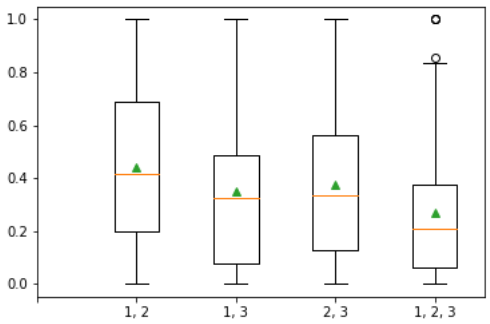

Figure 4.5: Boxplots of frequent Itemsets among Projects

The statistical key figures from the frequent itemset distribution among the individual projects are illustrated in Figure 4.5, where `1` represents the short summary component, `2` the parameter description, and `3` the return value description. The investigation has shown that the mean of the occurrences of every itemset is smaller than the overall occurrence of the itemsets. As evident from the illustrated boxplots in Figure 4.5, the median of the pairs is smaller than the overall occurrence as well. This indicates that these itemsets occur more often in big open-source projects with many formatted docstrings than in smaller projects, since they have a bigger affect on weighted mean (which corresponds to the overall occurrence), which is in this case greater than the unweighted one. Nevertheless, the key figures of the itemset distribution among the projects in our dataset indicate that these itemsets do not only occur in the few big projects, but are present in the smaller projects as well.

Chapter 5

Docstring Contents

To investigate the contents of docstrings from open-source projects, we created a classification tool that allows several people to rate the contents of docstrings at the same time. To enable a classification we first had to create a taxonomy, which served as the basis for the classification, and to define a set of docstrings for it. After letting participants classify the docstring contents, we analyzed the resulting data to understand what is usually described in docstrings.

5.1 Taxonomy

To enable classification we firstly had to define a taxonomy for docstring contents. As card-sorting is often used for problems with multiple possible solutions, we decided to use the card-sorting method to create the taxonomy for docstring contents [7]. The card-sorting method asks users to group given terms which are written on cards. the groupings of several users represent the result for a problem [23]. We used a slightly different approach. To create a meaningful taxonomy, we asked eight developers to provide us information about what they usually write into their documentation comments. In order to receive meaningful and unbiased input, we made sure that the surveyed developers have various demographics

Developer Demographics Three of the eight participant were still students, two bachelor students, and one master student but they all had already over one year of professional programming experience. Two other participants were full-time developers with an academic degree in computer science and three, respectively five years of professional experience. The other three surveyed developers had no academic degree. They all finished their apprenticeship as software developers and have been working for 2, 5, and 11 years respectively. All participants, except two, who were oriented to javadoc, could give information about contents referring to Python docstrings since they have already used Python and have written docstrings in their professional career.

We wrote all the obtained information on cards in form of general terms to enable the application of the card-sorting method. As we have dealt intensively with docstrings during this thesis, we added our own input to the set of cards as well. After grouping similar inputs together we were able to obtain the following taxonomy:

General Description: Part of a docstring that describes what the documented component does, or what it can be used for.

Technical Documentation: Part of a docstring that describes in a technical way how something is achieved by the documented component.

Algorithmic Details: Part of a docstring that refers to the used algorithm and explains, discusses, or provides background to it.

Special Cases: Part of a docstring that describes what happens in special cases, like for example if input parameters have wrong types.

Input Parameter(s): Part of a docstring that contains information about input parameter(s). Everything related to input parameters is included in this category. Therefore, descriptions of input parameters, input parameter types, defaults if no input parameters are given, examples for input parameters, or descriptions of requirements which have to be fulfilled by an input parameter, belong to this category.

Return Value: Part of a docstring that contains information about the return value of the documented function, or method. Everything related to the return value is included in this category. Hence, descriptions of what is returned, return types, examples for returns, or also guaranteed returns belong to this category.

Error Cases: Part of a docstring that describes which exceptions can occur and how they arise.

Example Usage: Part of a docstring that shows how to use the documented component. This can either be done by using code snippets, or doctests.[1]

Metadata: Part of a docstring that contains information that does not directly refer to the documented component or its usage. For example, author, creation date, version, etc.

Upgrade Instructions: Part of a docstring that describes what to do in case the documented component is deprecated.

Structural References: Part of a docstring that references or links to other source code components.

Copyright/Licence: Part of a docstring that contains information about the copyright, and/or licence.

To ensure that every docsrtring can be classified, we have additionally added the option other. In cases where other is selected, participants can specify what is contained in the docstring's content.

5.2 Definition of the Dataset

The used dataset for the content classification of docstrings is a subset of the docstrings we extracted from open-source Python projects. To define the set of docstrings for the classification we first defined the properties a sample for a single participant should fulfil to obtain meaningful data for the analysis. The knowledge of how a single sample is composed allows us to randomly select a multiple of the individual components from the extracted docstrings to obtain our dataset for classification. To define a representative sample we defined the following properties that a single sample should fulfil, sorted in descending order of importance:

[1] `https://docs.python.org/3/library/doctest.html`

- A sample for a single participant should contain docstrings that provide enough information so that their content can be classified.
- The docstrings in a sample for a single participant should reflect the docstrings obtained from the Github projects referring to the distribution of the length and the source code entities in which the docstrings are defined, so that the whole dataset for the classification is representative for docstrings in open-source projects.
- Docstrings in a single sample must not be too long so that the classification does not consume too much time, since this could result in potential participants being lost.
- Since test functions are not reused by end-user developers, docstrings of tests should not be included in a sample.

The defined properties may contradict each other which leads to compromises that have to be made. Contradictions may be found between the first and the second as well as between the first and the third specified properties in the list. Containing enough information for a meaningful content classification while reflecting the length distribution of docstrings in practice may contradict since many developers write short docstrings to document their code, as we saw in the static analysis in the previous chapter. The contradiction between the first and the third property comes from the fact that we believe that longer docstrings tend to have more content and allow a more accurate content classification.

To determine the composition of a single sample, we first explored how to divide the docstrings according to the component in which they are defined, and the length distribution length distribution of their content. To examine whether there are differences between the length distributions of docstrings that are defined in different components, we first divided them into groups according to the component they are described in. After dividing the docstrings into the groups, we first investigated how many docstrings reside in which group to find out how many docstrings from which components should be included to the sample. The investigation showed that the great majority of the docstrings belong to the function group with *118,635* docstrings, which is about *78.18%* of all docstrings, where the class, and module group contain *12.10%*, respectively *9.72%* of all docstrings.

To investigate the length distributions of the docstrings in the different groups, we analyzed the line counts of their docstrings. In a first step we removed the outliers to avoid too long docstrings. After removing the outliers we removed docstrings that document test entities of the source code since tests are not reused by end-user developer. In order to not remove too many docstring, we only checked whether the name of the source file of every docstring contains the word `'test'`, where case sensitivity does not matter. Since we assume that one-liner docstrings may not contain much information for the classification, we decided that they should not be present to the same extent as they are in open-source projects. Nevertheless, they cannot be omitted completely, because they represent the majority of docstrings. Therefore, they are included in the sample to a lesser extent. To determine the length distribution of the remaining docstrings in the sample, we removed the one-liner docstrings as well, since these represent the large majority of the docstrings, and would therefore heavily influence the results. After the removal we started to examine the length distributions of the remaining docstrings per group. For each group we determined the first and the third quantile as well as the median. These key figures of the content length distribution are represented in Table 5.1. On basis of these statistical key figures, we assigned every docstring to a category, depending on its number of text lines and the group it belongs to. The first category is the one-liner category, and is the same for every group of docstrings. Hence, every docstring that contains exactly one line of text belongs to the one-liner category. The other categories are determined by the key number listed in Table 5.1 and the type of the docstring's definition component. docstrings that have a smaller or equal number of

Table 5.1: Key Numbers of Lines per Docstring for each Group

	% of All Docstrings	1st Quantile	Median	3rd Quantile	Max Nr of Lines
Function	78.18%	3	5	8	20
Class	12.10%	2	4	7	22
Module	9.72%	3	4	9	27

text lines than the first quantile of the corresponding definition component belong to the category short, where docstrings whose number of lines is greater than the first quantile and smaller than or equals to the third quantile are assigned to the category medium. Comments with a line count greater than the third quantile but smaller than or equal to the maximum number of lines belong to the category long. Docstrings that contain more lines of text than the maximum number of lines (there are such when regarding the whole set we gathered from Github since we removed the outliers before obtaining the maximal line count) belong to the category very long and are not taken into account in the content classification.

After categorizing the docstrings we executed test runs to determine an appropriate sample size that allows a participant to classify the assigned sample within an appropriate time. As a result of several tests we found that 15 docstrings is a good sample size that can be classified within 10 minutes. The 15 docstrings in a single sample are composed in such a way that they represent the distribution over the different groups as well as their length distribution:

12x Function Docstrings Analyzing the docstrings we obtained from Github has shown that *78.18%* of the docstrings are defined in a function definition. To represent the distribution of docstrings in open-source projects we reflect the result on our sample. Furthermore, we reflect the line count distribution of the docstrings as follows:

- **2x One-Line Docstrings** We believe that one-liner docstrings do not contain much information to be classified. Nevertheless, they represent the majority of the docstrings in open-source projects and can not be completely ignored. Therefore, we decided to include two one-line docstrings in the sample.
- **2x Short Docstrings** Since both, short and long docstrings represent about a quarter of the adjusted docstrings (i.e. those without one-line docstrings, test docstrings, and outliers), we decided to include two short docstrings in the sample instead of three, since long docstrings provide more content, and therefore may provide more information for the classification.
- **5x Medium Docstrings** As medium docstrings represent about 50% of the adjusted docstrings, we include five medium docstrings to reflect the results of the analysis in a sample.
- **3x Long Docstrings** As long docstrings represent a quarter of the adjusted docstrings, we include three long docstrings in a sample to reflect the results from the analysis.

2x Class Docstrings Since docstrings that document a class definition represent *12.10%* of all obtained docstrings, we include two class docstrings in a sample. The whole dataset used for the classification contains the same number of every length category of class docstrings. Since only two class docstrings are included in a sample for one participant, the length categories of the docstrings are randomly chosen.

1x Module Docstring Docstrings that document a module definition represent *9.72%* of all obtained docstrings. Therefore, we include one module docstrings in a sample. Similar to

class docstrings, all length categories of module docstrings are contained to the same extent in whole dataset and the length of the module docstring in a single sample is randomly chosen.

As we obtained the distribution of docstrings in a single sample, we were able to determine the dataset for the classification. To obtain the dataset, the docstring distribution of a single sample was taken and multiplied by a constant so that the number of docstrings with certain properties is a multiple of the number of docstrings with the same properties in one sample. Afterwards, the defined number of docstrings with the corresponding properties were extracted randomly from the set of docstrings we gathered from Github.

5.3 The Classification Tool

In order to obtain more data for the docstring content analysis we created a web application that enables multiple participants to classify docstrings simultaneously. After entering the web application, the potential participant sees a welcome page where we introduce the purpose of the application. After reading the purpose the participant is redirected to a second page where we ask for demographic information in order to know the background of the participants, and therefore be able to assess whether the data resulting from the classification is unbiased or not. After completing the demography form, the participant is redirect to a page, which briefly explains the application, before being redirected to the main page, where docstring contents are classified. The main page of the web application is shown in Figure 5.1. The main page is separated in two sides, the left and the right side.

Left Side The left side of the web application's main page is used to display the docstring to be classified, as well as its context in the source code. The docstring content is displayed in the upper box of the left side, as shown in Figure 5.1, where the docstrings context resides in the lower box. To find the context of the docstring the participants can scroll down or simply click on the button `Find in Context`, what causes that the application automatically jumps to the docstring context, which is highlighted in the source code. Figure 5.2 shows the highlighted docstring in its context.

Right Side The right side of the main page contains all elements for the interaction with the participants. The taxonomy, which is used to classify the docstring contents, is at the very top. The question mark symbols beside every category of the taxonomy show a short description of what is understood by the category on hover. A text box is positioned below the taxonomy to enable participant to leave general comments about the docstring or to specify what is described by a docstring in case the option other is selected. Buttons which redirect the participant to either the next or the previous docstring are located at the bottom of the right side. In case the participant scrolls to view the context of the docstring the entire right side moves with it, so that the classification can be performed while the participant is studying the context.

After classifying 15 docstring contents the participant is redirected to a page, where we ask whether there was a missing category in the provided taxonomy, and provide space for additional comments about the web application, the taxonomy, or docstrings in general. Additionally we ask participant to leave their email address, so that we may contact them in case that their classifications are not clear. On the last page of the web application we thank the participants and give them the opportunity to continue the content classification. In case a participant decides to classify more docstring contents the web application redirects to the main page, where the participant may classify docsrtring contents until all docstrings in the dataset are classified. The web

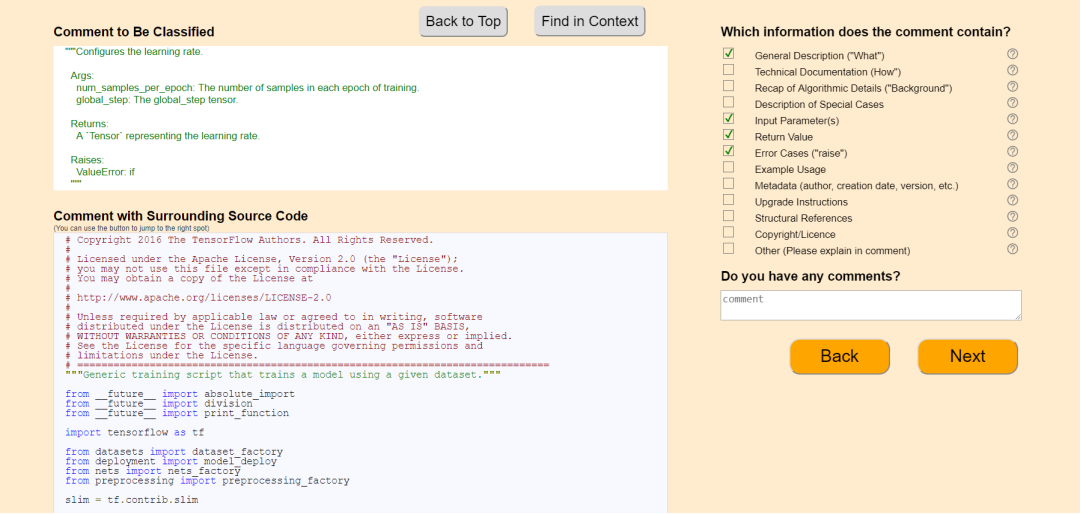

Figure 5.1: The Main Page of the Web Application

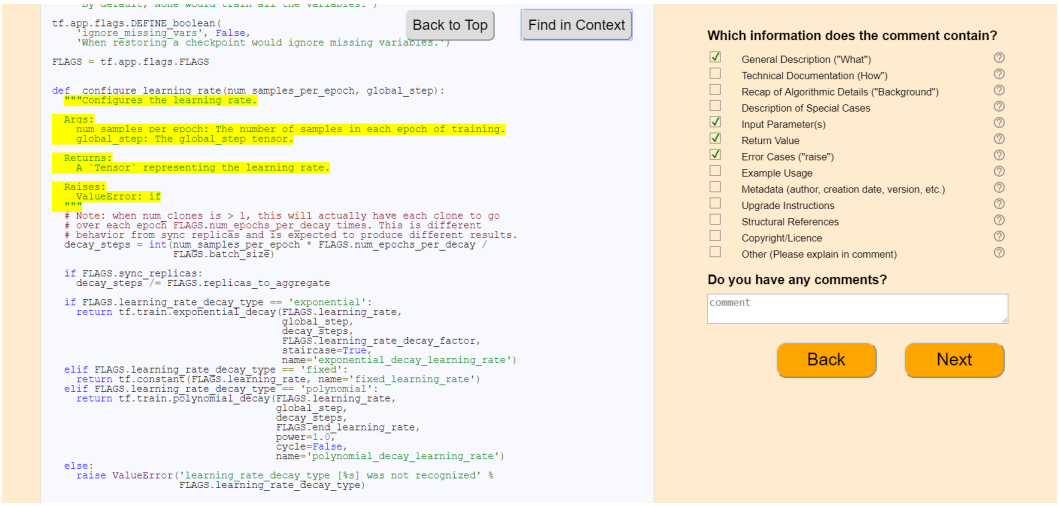

Figure 5.2: The Main Page of the Web Application - Docstring Context

application loads repetitively three new docstrings, one short, one medium, and one long, from the dataset and assigns them to the participant, without interrupting classification process.

5.4 Content Classification Analysis

To investigate the docstring contents we obtained from the classification we started with examining the demographics of the surveyed participants to assess whether the data resulting from the classification is unbiased or not. The obtained data from the demographic form enabled us to investigate the variety of the participants' degrees related to computer science, their professional programming experiences, their Python skills, and their job titles. Additionally we used the demographic form to obtain information about which docstring format is used by a participant to know how many people follow a specific syntax when writing docstrings.

5.4.1 Demographics and Style Preferences

As we started analyzing the demographics of the participants, we noticed that *106* persons filled in the demographic form but only *85* of them have classified at least one comment. Reasons for the difference could be that participants started the survey on mobile phone, filled in the demographic form, and decided to fill in the survey on their PC since the desktop version is clearer and easier to read and navigate through the survey. Additionally it is possible that people left the survey as they saw what they are asked to do. During the investigation of the participants' demographics, we encountered one participant that provided unreal information, as for example an impossible high number for the professional experience. Since the demographic data already seemed suspicious to us, we looked at the classifications of this participant and noticed that every docstring was rated with the category other with an inappropriate comment. In order to obtain meaningful results from the survey, we removed this participant and all the associated classifications.

To examine which docstring formats are used by developers in practice, we used the information we obtained from all the remaining *105* participants that filled in the demographic form. To analyze the demographic variety of the participants, we consider only the information provided by participants that have classified at least one comment.

Used Docstring Formats The question about the used docstring format has the following possible answers: I don't write Python comments, I don't follow a particular style, reStructuredText, Google, Epytext, Numpydoc, No Answer, and Other (please specify). In case Other (please specify) is selected, a textarea appears, where the client can specify the docstring format. The analysis of the provided answers revealed that *64* of *105* participants do not follow a particular style, what makes about *60.95%* of all surveyed people. All participants selected one of the provided answers except for one participant who left a comment that says that the participant uses the default PyCharm style, but does not know the name of the format. PyCharm provides a raw structure of a docstring by typing in three double quotes and pressing enter. The raw structure uses the reStructuredText format style as long as nothing is changed in the settings. As the comment says that the default PyCharm format is used, we decided to count that as reStructuredText. The further analysis showed that reStructuredText markup, and the Google style format are used the most by the participants, as reStructuredText was selected *13* times, and the Google style format *10* times. The Numpydoc style format is used by *5* of the surveyed persons, where the Epytext format is used only by *2* of them. *11* participants stated that they do not write Python comments at all. Counting all participants that selected one of the common style formats shows that *30* participants style their docstrings according to a style format, what makes about *28.57%* of all participants.

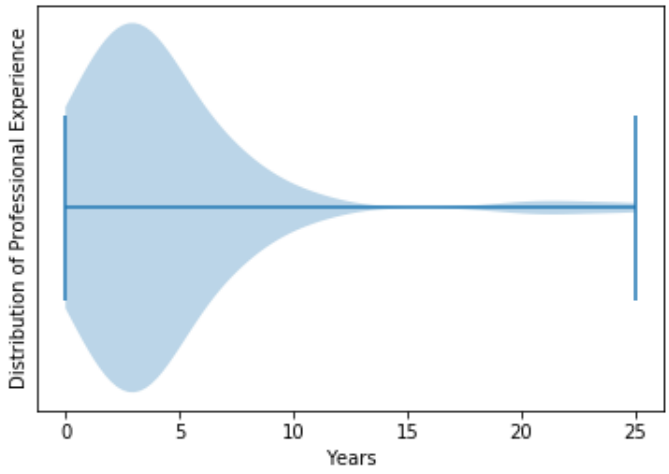

Figure 5.3: Violin-Plot of Participants' Professional Programming Experience

Demographics The more different the demographics of the individual participants are, the more unbiased and therefore more meaningful is the data obtained from their classifications. To determine the variety of the participants we analyzed their professional experience, their self-assessment of their Python skills, their highest degree related to computer science, as well as the different job titles of their current position. We present the distribution of the professional programming experience in a violin plot illustrated in Figure 5.3. The experiences of the participants ranges from *0* up to *25* years. As it is evident in Figure 5.3, most of the participants have less than *five* years of professional experience. Regarding the median of the distribution, reveals that *50%* of all participants have *three* years or less, respectively three years or more of professional programming experience, what we consider to be enough for a qualitative classification.

Investigating the Python skills self-assessments of the participants revealed that *49* out of *84* participants assessed their Python skills as moderate, what makes *58.33%* of all participants. *21* participants indicated that they have high Python skills, where *12* rated their Python skills as low. Only *two* participant had no Python skills at all. Participants with moderate or high Python skills represent the large majority, as they make up *83.33%* of all participants.

Regarding the highest degree related to computer science of the participants we obtained a high variety. *33* participants have a bachelor's degree, *14* a master's degree, college, and high school degree are selected by *10* respectively *12* participants, and *four* participants have indicated that they have a doctor's degree. The remaining *11* participants have no degree related to computer science; they finishing their education, visited a bootcamp, have a degree in another area, or learned programming by themselves. Further we went manually through the different provided job titles of the participants to examine their variety. We found many different job descriptions. Students, PhD students, IT consultants, network engineers, research associates, software developers, data analysts, and CTO (Chief Technology Officer) are only some examples of the provided job titles.

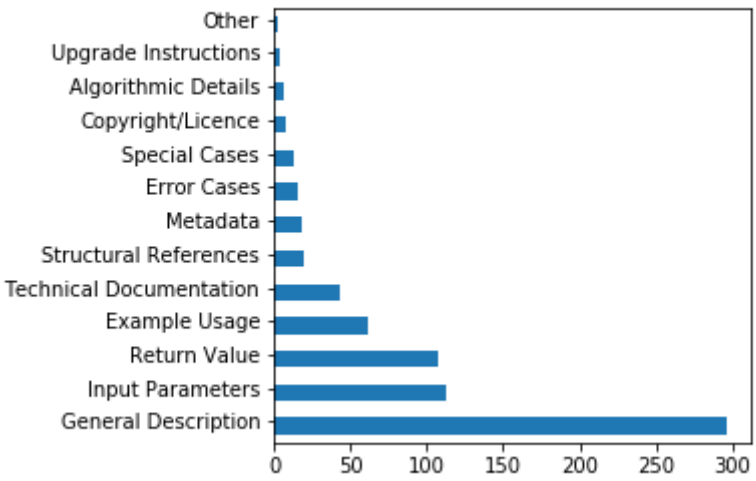

Figure 5.4: Occurrence of Docstring Content Categories

5.4.2 Classification Analysis

As we have ensured that the diversity of the participants is given and thus the classifications are unbiased we started to examine the obtained data from the classification. To obtain an even more meaningful classification we decided to assign the same docstring to three different participants. We defined that the content of a docstring contains a category when the same category was assigned to it by at least two different participants. After extracting the classification data from the database to a `.csv` file and load it into a dataframe using the analysis toolkit Pandas, we could obtain *341* classified docstrings. It is possible that a docstring has more than three classifications, since we check two hours after the assignment of the docstring whether the participant has classified it or not. If yes, and the docstring is assigned, respectively classified, already three times, no further assignments of this docstring are made. In case the docstring has not been classified two hours after the assignment, we allow further assignments of the same docstrings, even if it is assigned already three times, to ensure that the docstring is not only assigned, but also classified three times. For a few docstrings we could only consider 2 classifications, because we had to remove one participant and his classifications from the dataset.

Category Occurrences As we loaded the data from the classification into a dataframe we were able to analyze the obtained data using Pandas. To investigate which categories are assigned to which docstring, we grouped the dataframe by docstrings and the assigned categories. To determine how often a category was assigned to a docstring we performed an aggregation on the dataframe, which counts the participants that assigned a certain category to it. Figure 5.4 shows how often every category was assigned to a docstring by at least two participants. As evident in the figure, the general description of what is done by the documented source code entity occurs by far the most. The category occurs in *297* of totally *337* docstrings to which a category was assigned at least twice, which is about *88.13%*. Input parameter (*113* times), and return value description (*107* times) occur often as well, what underlines our meta-model and the results from the static analysis. Even though the exception description is recommended to be part of the docstring con-

tent by the docstring convetions, and the Google style format, the category was assigned only to *16* docstrings, what makes about *4.75%* of all classified docstrings. The rare occurrence of exception descriptions underlines the results from the static analysis, which delivered similar results. Surprisingly an example of the usage occurs relatively often (*62* times). As we did not expect such a result, we decided to have a closer look at docstrings, containing this category. By manually going through the docstrings we noticed that many of the docstrings that contain an example usage come from one big project which often contains an usage example in its docstrings. Therefore the occurrence of usage examples would be lower if docstrings from this project would not be considered. The remaining results are less surprising, a description of how something is achieved or processed (technical description), occurred in *43* docstrings, references to other source code elements in *19*, metadata *18*, and description of special cases in *16* of the docstring contents. A recap of algorithmic details which provides background information was found in *6* docstrings. Information about the copyright and/or the licence can be observed in *7* docstrings. The reason for the rare occurrence of information about copyright and licence could be that we focused more on method and function docstrings than on class or module docstrings. Upgrade instructions which are given in case of deprecation occurred only in *4* of the classified docstrings and the option other was selected *3* docstrings. We manually read the comments the participants provided to us and afterwards looked at the docstrings to verify whether the comments are appropriate, or possibly one of the provided categories from the taxonomy could apply on the docstring's content. One docstring contained an empty construct of a restructuredText docstrings, which was probably created by *PyCharm* but not filled by the developer. The second docstring just repeated the definition name, and the third docstring provided information for those end-user developer who want to override the method. The provided comments were appropriate in all *three* comments.

Categories that are likely to occur together Besides the total number of occurrences of every category we investigated the most frequent itemsets of the categories to examine whether the appearance of a certain category leads to the occurrence of other categories. to obtain results about which categories are likely to occur together in a docstring's content, we used Pandas and the frequent itemset miner from `mlxtend`[2], as we did for the syntactic analysis of the docstrings. To obtain the itemsets and the frequency in which they occur we used the dataset with docstrings that contain categories which were assigned to them by at least two participants. To bring the categories in the right form for the itemset miner we used the `transactionEncoder` from `mlxtend` and gave the categories of each docstring as input. The `transactionEncoder` creates a new dataframe with all categories as columns and assigns true, respectively false values to every entry, depending on whether a docstring contains the particular category or not. Once we created the dataframe for the frequent item set miner, we gave it as input and were able to obtain the results. The results of the five itemsets that are most likely to occur together in docstring from our dataset are presented in Table 5.2. Regarding the results of the most frequent itemsets it is evident that their order is similar to the order of most frequent itemsets we obtained in the syntactic analysis of docstrings, where the category general description can be seen as either the summary, or the description component. The category itemsets occur less often than the itemsets of the meta-model components. One reason for the difference could be that we removed one-liner docstrings for the syntactic analysis of the docstrings, where we included one-liner docstrings in the classification. The fact that the general description, input parameters, and return value occur most frequently in the classified docstring contents as well as in their itemsets confirms the results obtained from the syntactic analysis and validates the inclusion of these components in the meta-model.

Since the general description is dominating and occurs in the majority of the docstrings, and

[2] `http://rasbt.github.io/mlxtend/`

Table 5.2: Top five frequent itemsets of the classification

How Often in %	Category Combination
General Description, Input Parameter(s)	30.27%
General Description, Return Value	27.89%
General Description, Example Usage	17.21%
Input Parameter(s), Return Value	16.32%
General Description, Input Parameter(s), Return Value	15.13%

Table 5.3: Top five frequent Itemsets of the remaining Categories

How Often in %	Category Combination
Example Usage, Metadata	2.37&
Example Usage, Technical Documentation	1.78%
Special Cases, Example Usage	0.89%
Algorithmic Details, Technical Documentation	0.89%
Special Cases, Technical Documentation	0.59%

the occurrence of input parameter, exception, and return value desriptions can partially be investigated by using our converson approach, we decided to investigate which category pairs occur most frequently when we regard the classifications without these categories. As the remaining categories are not recognized by our syntactic approach, we use the classifications from the participants, who are able to recognize specific categories even in running text, to investigate in which combinations these categories occur. To obtain the most frequent itemsets we used the dataframe, generated by the transactionEncoder and removed the mentioned categories. Subsequently we provided the dataframe as input for the frequent itemset miner and obtained the results.

The five pairs of the remaining categories that occur most frequently are listed in Table 5.3. The results that the category combinations which are not recognized by our approach occur rarely. The combinations that occurs the most, is present in only *2.37%* of the docstrings. This percentage share might seem low but regarding the total occurrence of the category metdata we see that in half of the docstring contents that describe metadata an example usage is described as well.

The Inter-Rater Agreement To enable an assessment of the clarity of docstring contents, as well as the understandability assessment of our taxonomy, we calculated the inter-rater agreement for every classified docstring. The inter-rater agreement measures the classification variability between the participants [21]. To calculate the inter-rater agreement we used Fleiss' kappa [14] which enables us to investigate the agreement between more than two raters. To calculate the Fleiss' kappa for every single docstring we first grouped the dataframe by docstrings, and then by the participants. To perform the Fleiss' kappa we created a method, that created a list for the classification of every participant. The list contains zero and one values, which represent the categories from the taxonomy. The value '1' is set if the participants assigned the particular category to the docstring, otherwise '0' is set. After the representation list was created for every single classification, we were able to perform the Fleiss' kappa between the classifications of every single docstring. To calculate the Fleiss' kappa of every docstring we used agreement module from

Table 5.4: Key Numbers of Classification Inter-Rater Agreement

Mean	0.72
Min	-0.13
1. Quantile	0.57
Median	0.77
3. Quantile	1
Max	1

the Natural Language Toolkit (NLTK).[3] Table 5.4 shows the key numbers of the inter-rater agreements of the individual docstrings. Landis et al. [21] provided a benchmark for the strength of the agreement in their work. In their benchmark they defined that the agreement strength of a kappa statistic less than 0 is poor, between 0 and 0.2 slight, between 0.21 and 0.4 fair, between 0.41 and 0.6 moderate, between 0.61 and 0.8 substantial, and between 0.81 and 1 almost perfect. Regarding the boxplot of the kappa statistics and the benchmark, it is evident that the median, which is *0.77*, is only slightly less than the lower bound of an almost perfect agreement, also the first quantile (value that is higher than 25% of all values) is only slightly lower than the lower bound of a substantial agreement strength. The minimal value of -0.13 belongs to one of the docstring for which we could obtain only two classifications due to the removal of a participant. One participant assigned the category general description, where the other assigned the categories return value, example usage, and other. In this case no agreement between the participants was found. Nevertheless regarding the whole set of classified docstrings, we can say that the agreement between the participants is good what indicates that the contents of docstring can be understood in general, and that our created taxonomy is clear and understood by the participants.

[3] `https://www.nltk.org/`

Chapter 6

Discussion

After investigating the formatted docstrings in open-source projects and the contents of docstrings we are going to pick up our findings, discuss their meaning, and answer our research questions. Additionally, we are going to propose ideas for new researches in this area as well as list the threats to the validity of our findings.

6.1 Discussing Research Questions

We defined *three* research questions in the introduction of this thesis. In this section, we try to answer them by discussing the findings from the previous chapters.

First Research Question We start with the first research question:

> RQ1: How does an unified meta-model for Python docstrings look like, and how can docstrings be converted to it?

To answer this research question, we presented the common docstring style formats as well as the docstring conventions in the second chapter of this thesis. The recommendation comparison of the docstring conventions, the Google style format, and the Numpydoc style format led to our conceptual meta-model for function docstrings that contains a short summary, a parameter description, a return value description, an exception description, and a description component, which contains all parts of a docstring that are not recognized by any of our parsers. To convert docstrings from open-source projects to the specified meta-model, we proposed an approach on a syntactic level, which uses parsers to recognize the defined components that follow the syntax of a common style formats. The parsers allow us to directly convert the recognized elements into the corresponding components of the meta-model.

Previous studies used parsers to recognize parts of source code comments too. Pascarella et al. [26] made use of a parser to recognize type declarations in docstrings that use the Numpydoc format. However, they used the parser for a very specific task and restricted their study to docstrings formatted according to the Numpydoc format, where our approach provides parsers for each of the common style formats and recognizes more general elements of a docstring. Even though in the scope of this thesis, we did not investigate the individual parameters that are described in a docstring, we composed our parser rules for the recognition of the parameter sections from smaller rules, which enable our parsers to recognize single parameters. Therefore, future researchers could use our parsers as a basis for investigating type declaration mismatches regarding all common styles. As it is important to know which parameters are used in a function when investigating type declaration mismatches, the function definition would have to be considered in

such a study. To examine the mismatches between the documented, and the actual parameter types, one could use the static type checker `MyPy`,[1] as Pascarella et al. [26] did in their study.

Further the co-existence of several docstring style formats, which differ in their representation, may lead to confusion of end-user developers. Therefore, future works could explore the effect of a unified representation on docstrings in open-source projects in an empirical study. Our approach could be used as a basis for the conversion of the different style formats into a unified representation. To enable a unified representation, future researcher would have to define a syntax for the representation and extend our meta-model by more components as well as our parsers to recognize the newly added elements.

Second Research Question We applied our approach from the first research question on docstrings from open-source projects to answer the second research question:

RQ2: How is the meta-model applicable on docstrings in Open-Source projects?

To answer the second research question and therefore determine whether docstrings in practice are repetitive and structured, we extracted docstrings from open-source Python projects, that we obtained from Github. Afterwards, we applied our parsers on the extracted docstrings and wrote the obtained docstrings in meta-model form to a `.json` file which was loaded into a data frame to analyze how our meta-model applies to them.

Regarding all docstrings in our dataset, we reported that *17.91%* of them follow a particular style format. The low percentage was to be expected, considering that *56.78%* of the docstrings in our dataset are one-liner docstrings. After the removal of the one-liner docstrings, we observed an increase of the percentage to *41.18%*, which indicates that more than every third multi-line docstring in the set follows a particular style format. Additionally, we found that *11* of the *84* open-source projects from our dataset that contain multi-line function docstrings do not include a single docstring that is written according to a style format.

To investigate the distribution of formatted docstrings among the remaining projects, we examined how many of them belong to each project and how high the percentage share of them in the individual open-source projects is. Analyzing the total number of formatted docstrings per project showed that the large majority of formatted docstrings came from only few big projects. Nevertheless, to investigate whether our meta-model may be applied on docstrings in practice, we further investigated how many docstrings, relatively to the total amount of docstrings in each project, follow a particular style format. We observed a wide range of the percentage shares. However, the distribution showed that formatted docstrigs can not only be found in large Python projects, but also in the smaller ones, as the median of the distribution is almost as great as the overall percentage share of formatted docstrings. Despite the fact that formatted docstrings occur in smaller projects as well, the overall percentage share of them is greater than the median of the individual percentage shares, which indicates that in general big projects have a larger share of docstrings that follow a particular style format. A reason for that could be, that in big projects, where many developer work simultaneously, and reuse code written by another developer, the way of writing docstrings is regulated to ensure better understanding of the source code.

After investigating how many docstrings contain certain annotations and syntax, we examined how often the components defined by our meta-model appear in them. Regarding the total occurence of the individual components, and the occurrence of the frequent itemsets, we state that the components short summary, parameter description, and return value description are present in a significant share of the formatted docstrings. Only the exception description occurs rarely in docstrings from open-source projects. Regarding the total occurrence of the exception description and its occurrence in any combination with the other components, we determine that an exception description almost never occurs as only component in a docstring. Even the combination of

[1] `http://mypy-lang.org/`

all four components from the meta-model can be found in more than half of the docstrings that contain an exception description. Therefore, we state that docstrings that contain an exception description are very likely to contain other components from our meta-model as well. Despite the rare occurrence, we wouldn't recommend removing the exception description from the meta-model, as information about possible exceptions are very important to end-user developers, who rely on docstrings.

By applying our conversion approach to Python docstrings in open-source projects, we contribute a first step in investigating their structure and annotation. To conclude and answer the second research question about the applicability of the meta-model on docstrings from open-source projects, we state that parts of the meta-model are well applicable on formatted docstrings as well as on all multi-line docstrings, since formatted docstrings represent a significant share of them. Due to the high number of one-liner docstrings, applying our meta-model on all docstrings in open-source projects is difficult, as only *17.91%* of them follow a particular style format.

As in the scope of this thesis we exclusively referred to the contents of docstrings, we did not distinguish between void-functions, functions that do not take any input parameters, and functions that do not throw any exceptions. Future researchers could pick up our findings and additionally consider whether the function takes input parameters, returns a value, or throws an exception to investigate in how many cases the components occur, in which they really make sense.

Part of the docstring contents that describe one of the discussed components are moved to the desccription component of the meta-model if they do not follow the notations of a common style format. Therefore, further work on this topic could use the approach and the meta-model provided by this thesis, and additionally define heuristics to parse the docstring contents which are moved to the description component. With such an approach future researchers could investigate how often the individual components appear in docstrings that do not follow a particular style format.

Third Research Question As we are also interested in the contents of docstrings that are not recognized by our approach, we proposed the third research question:

> RQ3: What do Python docstring in Open-Source projects describe?

To answer this research question, we firstly created a taxonomy which is used as the basis for classification. Afterwards, we created a classification tool that enables multiple participants to classify docstring contents simultaneously in order to obtain more classifications which also delivers more meaningful and more impartial results than classifying all docstrings by ourselves. After creating the tool for the classification, we defined the sample which should be assigned to a single participant by adhered to predefined properties the sample should contain.

To measure the meaningfulness of the gathered classifications, we firstly examined the participants' demographics variety. The results revealed that the demographics of the participants have a wide variety regarding their working area and their highest degree related to computer science which increases the meaningfulness of the classifications. To further increase the meaningfulness of the analysis, we decided to assign every docstring to three different participants and accept a category only if at least two of them assigned it to a docstring. Additionally, we have calculated the agreement of the classifications for each docstring to investigate how precise the docstring contents are and how well the participants understand our taxonomy. According to the benchmark proposed by Landis et al. [21] the median of the classification agreements of every docstring is only slightly smaller than the lower bound of a substantial agreement. The strong agreement between the classifications indicates that our taxonomy as well as most of the docstring contents from our dataset are good understood by the participants.

After analyzing the demographics variety and the inter-rater agreement of the participants, we examined the obtained classifications. Our analysis revealed that the category general description is present in *297* of *337* docstrings that have at least one assigned category which makes about *88.13%*. The second and third most frequent categories are input parameter(s), return value respectively, which are both present in our created meta-model as well. Regarding that a short summary as it is defined in our meta-model could represent a general description, we can say that the three content categories that occur the most are present in our meta-model, what validates their inclusion. Similar as in the syntactic analysis, the category error cases occurs only in few docstring contents.

The investigation has shown that beside the categories which we already defined in our meta-model, an example of the usage and a technical documentation of how something is achieved by the documented source code entity, are likely to occur in docstring contents. Examining the frequent itemsets of docstring contents revealed that combinations of the most present categories are most likely to occur in combination. As the category general description is present in the large majority of the docstring contents, it is not surprising that the percentage share of combinations between it and any other category is almost equal to the percentage share of the other category of the combination. Examining the most frequent itemsets between categories that are not contained in our meta-model revealed *half* of all docstrings that describe metadata, provide an example usage as well. This finding could indicate an association between these two categories.

Surprisingly we found that the category structural references, which occurs the third most of the remaining categories, is not contained in any of the top five combinations, what indicates that references are not likely to occur in combinations with other categories.

In conclusion, we can say that docstring contents in our dataset often contain a general descriptions of what the documented construct does or what it can be used for, input parameter descriptions, return value descriptions, and relatively often an example usages and a technical documentation of how something is done. The consequence is that the most frequent itemsets that are present in docstring contents are made up of these categories. Other categories occur rarely but every category is present in at least *3* docstrings.

It is to point out that the obtained results depend on the number of classified docstrings and the used docstring set for the classification. Therefore, future works on the topic of docstring contents could use another set of docstrings, and/or classify more docstrings by using the taxonomy proposed by this thesis.

6.2 Threats to Validity

Internal Threats The static analysis on docstrings in open-source projects is performed on a dataset we obtained from Github. Since we have selected the 100 Python projects with the highest star rating from Github without considering their size certain big projects may have a large impact on the results.

To reduce the impact of large projects on the results of our static analysis we performed the analysis on project level as well. The analysis on the project level revealed that the percentage share of formatted docstrings is bigger than or equal to *37.50%* what is slightly less than the overall percentage share. Statistical key figures of the investigations regarding the meta-model validity showed only slightly different results than the investigation on the overall dataset. Therefore, the analysis at the project level has shown that large projects have an impact on the overall results but also that the impact is not as large at it seems when just regarding the total number of docstrings per projects without further analysis.

Further, docstrings in the dataset for the classification of docstring contents may not be rep-

resentative for docstrings in open-source projects. To shrink this threat we based the selection of the docstrings for the dataset on characteristics of docstrings that resulted from an investigation of all docstrings that we obtained from Github. Nonetheless, we did not consider the contents of docstrings during the investigation of the selection what leads to the threat still being present.

A threat to the validity of our conversion approach is that the created parsers might not fully reflect the style formats. To reduce this threat we tested our parsers by letting them parse examples of the corresponding style format.

The completeness of our created taxonomy is another threat to validity. To obtain an unbiased and complete taxonomy we performed the card-sorting method. Despite the variety of the surveyed developers we can not ensure that the taxonomy is complete. Therefore, we asked the participants at the end of the survey whether any categories were missing through the classification. By reading the comments we determined that a further category that provides information for developers who want to inherit or override a class could have been added to our taxonomy.

External Threats Humans are error-prone and as docstring are usually written by humans it is possible that docstrings that are intended to be written in a particular style format are not recognized by our parsers due to small mistakes in the syntax. By allowing deviations from the style formats we have tried to intercept as many small mistakes as possible but it is not possible to intercept all mistakes.

By letting several participants classify the contents of docstrings we have been able to collect more impartial data. Nonetheless, the publication of our classification tool also provides a target for people who want to sabotage our analysis or work. This threat can not be reduced and it became reality. As we could read the classifications of the participants at any time we were able to detect the sabotage attempt and remove the participant from the dataset.

Chapter 7

Summary

In the scope of this thesis we investigated the structure, and the contents of Python docstrings. In a first step we created a conceptual meta-model which enables us to treat all docstrings the same during the analysis, without having to take into account in which format style they are written. We provide an approach for converting docstring from Open-Source projects into the form of the meta-model. As our approach uses parsers of the common docstring style formats to recognize certain components of the docstring and convert them into meta-model form, it is restricted to syntactic properties of the common style formats.

After applying our approach on a set of Open-Source Projects, we analyzed the obtained data. The analysis revealed that the majority of docstrings in Open-Source projects contain only one line of text. As we want to examine how our meta-model is applicable to docstrings in practice, we investigated how many docstrings that have more than one line of text have structured content, i.e. contain sections that are defined by one of the common style formats. The investigation revealed that more than every third docstring with more than one line of text contains structured content. By further examining the structured multi-line docstrings, we showed that our meta-model is applicable on them, as all components, except the `exception description` component, occur frequently, no matter whether we regard the occurrence of the single components, or the occurrence of them in any combination.

To investigate docstring contents, we conducted an empirical study. In a first step we created a taxonomy which provides the basis for a classification. After creating the taxonomy using the card-sorting method, we implemented a classification tool that allows multiple participants to classify docstrings at the same time. the subsequent examination of the obtained data revealed that a general description is the dominant category and that it is present in the very large majority of the docstrings. Further we could show that descriptions of input parameters, and return values are likely to occur as well. As a logical consequence, the most frequent category-pairs consist of a combination of these categories. Since these categories can partially be recognized by our syntactic approach, we further investigated the most common itemsets without these categories, what showed that there is no combination between them that occurs unusually often.

Bibliography

[1] Antlr. https://www.antlr.org.

[2] Antlr 4 documentation. https://github.com/antlr/antlr4/blob/master/doc/index.md.

[3] Google styleguide. https://github.com/google/styleguide/blob/gh-pages/pyguide.md.

[4] Numpydoc docstring guide. https://numpydoc.readthedocs.io/en/latest/format.html.

[5] Sphinx documentation. http://www.sphinx-doc.org/en/master/.

[6] O. Arafati and D. Riehle. The comment density of open source software code. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 195–198. IEEE, 16.05.09 - 24.05.09.

[7] A. R. Barrett and J. S. Edwards. Knowledge elicitation and knowledge representation in a large domain with multiple experts. *Expert Systems with Applications*, 8(1):169–176, 1995.

[8] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[9] David Goodger. Pep 257 – docstring conventions. https://www.python.org/dev/peps/pep-0257/, 2001.

[10] David Goodger. Pep 287 – restructuredtext docstring format. https://www.python.org/dev/peps/pep-0287/, 2002.

[11] David Goodger. restructuredtext markup specification. http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html, 2017.

[12] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In S. Tilley and R. M. Newman, editors, *Proceedings of the 23rd annual international conference on Design of communication documenting & designing for pervasive information - SIGDOC '05*, page 68, New York, New York, USA, 2005. ACM Press.

[13] Edward Loper. Epydoc. http://epydoc.sourceforge.net/index.html.

[14] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, 1971.

[15] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79. IEEE, 28.10.07 - 31.10.07.

[16] B. Fluri, M. Würsch, E. Giger, and H. C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, 2009.

[17] S. Haefliger, G. von Krogh, and S. Spaeth. Code reuse in open source software. *Management Science*, 54(1):180–193, 2008.

[18] C. S. Hartzman and C. F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 138–170. IBM Press, 1993.

[19] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: Data mining, inference, and prediction : with 200 full-color illustrations*. Springer series in statistics. Springer, New York, 2001.

[20] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In S. Diehl, H. Gall, and A. E. Hassan, editors, *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, page 179, New York, New York, USA, 2006. ACM Press.

[21] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159, 1977.

[22] W. Maalej and M. P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, 2013.

[23] M. MAGUIRE. Methods to support human-centred design. *International Journal of Human-Computer Studies*, 55(4):587–634, 2001.

[24] mjakov95. mjakov95/bachelorthesis: Initial release, Sept. 2019.

[25] L. Pascarella and A. Bacchelli. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 227–237. IEEE, 20.05.17 - 21.05.17.

[26] L. Pascarella, A. Ram, A. Nadeem, D. Bisesser, N. Knyazev, and A. Bacchelli. Investigating type declaration mismatches in python. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 43–48. IEEE, 20.03.18 - 20.03.18.

[27] Python Software Foundation. The python language reference. https://docs.python.org/3/reference/.

[28] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92. IEEE, 20.05.13 - 21.05.13.

[29] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 215–223, Piscataway, NJ, USA, 1981. IEEE Press.

[30] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 27–37. IEEE, 20.05.17 - 28.05.17.