The Charming Code that Error Messages are Talking About

Joshua Charles Campbell[1] **and Abram Hindle**[1]

[1]**Department of Computing Science, University of Alberta, Edmonton, Canada**

ABSTRACT

The intent of high test coverage is to ensure that the dark nooks and crannies of code are exercised and tested. In a language like Python this is especially important as syntax errors can lurk in unevaluated blocks, only to be discovered once they are finally executed. Bugs that present themselves as error messages mentioning a line of code which is unrelated to the cause of the bug can be difficult and time-consuming to fix when a developer must first determine the actual location of the fault.

A new code metric, charm, is presented. Charm can be used by developers, researchers, and automated tools to gain a deeper understanding of source code and become aware of potentially hidden faults, areas of code which are not sufficiently tested, and areas of code which may be more difficult to debug. Charm quantifies the property that error messges caused by a fault at one location don't always reference that location. In fact, error messages seem to prefer to reference some locations far more often than others. The quantity of charm can be estimated by averaging results from a random sample of similar programs to the one being measured by a procedure of random-mutation testing. Charm is estimated for release-quality Python software, requiring many thousands of similar Python programs to be executed. Charm has some correlation with a standard software metric, cyclomatic complexity. 21 code features which may have some relationship with charm and cyclomatic complexity are investigated, of which five are found to be significantly related with charm. These five features are then used to build a linear model which attempts to estimate charm cheaply.

Keywords: Mutation Testing, Software metrics, Complexity

1 INTRODUCTION

Complexity measures for software have always enticed software engineers [8, 21]. The idea that code can be ranked by its complexity and thus prioritized for testing and code inspection has always been interesting.

This interest later motivated automatic test-case generation, random testing, and mutation testing. One of the issues with random search and testing of source code is not all mutations or mutation sites are created equal.

This work introduces the idea of code charm. Code charm is the ability of certain kinds of code to attract errors and error messages. Sometimes an error in another part of the file will induce an error message that reports the wrong location. It turns out

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

these wrong locations follow a certain distribution, which can be measured. Some of the wrong locations that error messages report are surprisingly common.

> Charming code is code that attracts the error messages of other code to itself. Charming code aggregates error messages, some of which do not belong to the charming code.

Knowing about the properties of both charmless and charming code can help mutation testing search for good locations to elicit failures in testing. These general properties can be used to provide guidelines to developers about code that should be tested due to its ability to hide errors.

1.1 Conceptual Charm

The following code illustrates an extremely simple Python program which sets two variables and then sets a third variable to the sum of the first two variables.

```
x     =  1
y     =  2
total =  x + y
```

If the program has a fault on line 1, for example an incorrect type, then the error message will be produced on line 3, as in the following example.

```
x     = "1"    # Oops, wrong type! Charmless!
y     =  2
total =  x + y # TypeError here! Charmed!
```

However, if the program had, for example, a misspelled variable on line 3, then the error message would also appear on line 3.

```
x     =  1       # should x be ex? Charmless!
y     =  2
total =  ex + y  # Fault and error here! Charmed!
```

From these two examples, we can see that there is something different about line 3 compared to line 1. This difference is apparent even if one does not know what the code actually does.

Charm is a quantification of this property of source code. It does not require careful static analysis either by a human or a machine other than executing the software in question. Additionally, it is a single number which expresses a property that arises from the complex interactions of many pieces in a program and the environment in which it is executed.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

1.2 Technical Introduction

Charm is a measure, in the mathematical sense, of a particular piece of code. That code can be a line, a block, a class, a file, or any other subset of a larger code base. It is the difference between the number of errors a piece of code would ideally produce under mutation testing (one for each mutation) and the number of errors that same piece actually produces under mutation testing, relative to the average piece of code. If charm is computed per line, then it is relative to the average line. If charm is computed per block, then it is relative to the average block. In the latter scenario, charm is measured directly by applying the following formula:

$$\frac{\text{errors this block} - \text{mutations this block}}{\text{average mutations per block}}.$$

In this formula, mutations per block divided by the average mutations per block is determined by the estimation procedure used but approaches, in the limit, the number of syntactic elements in this block divided by the average syntactic elements per block.

- charm $> 0 \longrightarrow$ Positive charm: for example, if a line has 5 charm, it produces 5 times as many errors as a typical line would have by being mutated. This implies that the errors caused by other lines being mutated are showing up on this line.
- charm $= 0 \longrightarrow$ Neutral charm. For example, if a line has 0 charm, it produces errors as often as it is mutated.
- charm $< 0 \longrightarrow$ Negative charm. For example, if a line has -2 charm, it produces few errors and is longer or more complex than an average line of code. This indicates that the mutating this line either caused errors on another line, or that mutating this line did not always cause an error.

2 CONTRIBUTIONS

- A new software metric, *charm*, used to indicate the error response under mutation testing of a subset of program.
- Comparison of charm with another standard software metric, *cyclomatic complexity*.
- Comparison of charm with easily obtainable software features, such as indentation and keywords.
- Analysis and interpretation examples of charm results.
- A method for estimating charm cheaply.
- Recommendation for software engineers on how to use charm when testing their software.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

The data in this paper summarizes the output from 880 000 *different* programs. These programs are each mutations of one of 22 Python programs, but by treating them as individual programs we enable exploration of the space of all possible programs.

Each data point can be analyzed individually, tracing the chain of cause and effect through the system, from the code of that individual program to its eventual output. This analysis would undoubtedly yield valuable information and it is the type of analysis that the field of computer science is based on.

However, in comparison with the traditions of static program analysis and debugging, this work presents an attempt to empirically quantify a property of software that is relevant to everyday software engineering. Though this property is produced by the deterministic behaviour of a system, randomness is introduced *artificially* and *automatically* in order to characterize Python error messages by their *average* behaviour.

Each program written by a human represents a significant cost, in terms of that person's time and effort if not a direct monetary cost. Therefore, the number of programs that humans have written is infinitesimal compared to the infinite number of all possible Python programs. Inside that set of all possible Python programs, there are programs which are extremely close to each program written by a human programmer. In fact, there are a practically unexhaustable number of such near-by programs. It makes sense to define what it means for two programs to be close or similar to each other by using a standard metric such as *edit distance*, also known as the Levenshtein distance. This metric is not only applied in the field of computer science, but by geneticists to determine the similarity of genetic code [24] and by linguists [27] to determine the similarity of natural language text.

In this paper we sample randomly from the neighborhood of programs around 22 Python programs written by human authors. Then that neighborhood can be characterized by averaging results from a simple test: whether or not each sampled program returns an error on a particular line or block. After averaging, a vector of real numbers are obtained that represents a property of that gigantic neighborhood which surrounds a single valuable program.

The procedure used in this paper is not just about characterizing one program. Instead, the procedure used in this paper characterizes a huge number of programs, and acquires information which is not obtainable through statically analyzing a single program. This information is relevant to the original program, because the original program at the center of the huge set of programs being characterized, and each one of those programs is extremely similar to the original program.

It is most often the case that a program that is written is close to or extremely similar to a desired program that produces the desired results. Closing this gap is one of the fundamental goals of software engineering research, and this paper works to close it, even if just by one edit, in a novel and hopefully helpful way.

This paper attempts to achieve that goal by providing a method for software engineers to become aware of the average behavior of Python error messages. Error messages are relevant to the software engineer because they are an important indication of a fault in software. Misleading error messages, such as error messages that show the wrong location in the code, require the software engineer to find the correct location as well as debug the code in that location.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

Another situation that can occur is that no error message is produced at all, even though the program has a fault at some location. Charm also captures this information in a simple, numerical, way. It is often the case that the original program does produce the desired results and that those results should ideally stay the same even while the program is developed or extended further. In this situation, Charm can assist software engineers by characterizing the pieces of code that could have faults added to them and go undetected, even under test suites that achieve 100% coverage.

3 MOTIVATION

In Campbell *et al.* [Campbell et al., 3] a software engineering tool, UnnaturalCode, was presented along with a testing framework for that tool and similar tools. This paper does not discuss or apply to UnnaturalCode, but it does make use of the random-edit mutation testing tool which was developed to test the performance of UnnaturalCode. The random-edit mutation testing tool is a Python implementation of the algorithm in figure 1.

While investigating Python error reporting performance, it was noticed that Python often likes to report error messages on certain lines far more often than others. These errors were caused by mutations that were made at random locations in the source code. Therefore, it was expected that Python would report errors randomly throughout the source code, but this was not the case.

After informally investigating where Python commonly reported error messages in the source code, a new metric was created to characterize this effect, the *charm*. Lines which Python reports more errors on have a high *charm*, and lines which Python reports fewer errors on have a negative *charm*.

4 PRIOR WORK

This work is situated between the worlds of error detection (discussed in the prior section), software complexity metrics, source code structure (indentation), coverage based testing, search based software engineering, mutation-based testing and genetic programming.

4.1 Complexity

Complexity has been a popular topic since the 1970s as authors tried to characterize code complexity automatically [8, 21]. McCabe's cyclomatic complexity [21] is still in use today, although some argue it is irrelevant to object-oriented code. Essentially McCabe counts all branch points in a block of code. What is a branch point depends on the implementer of the metric. While if-statements are clearly branch points, sometimes a virtual dispatch operator is a branch, sometimes not. McCabe argues that the more branch points in the graph of control flow in a block code, the more complex it is. In general, there is a probability that a line of code will contain a branch, thus the number of branching lines will often be correlated with the total lines of code. In fact McCabe's cyclomatic complexity correlates with lines of code, across numerous language and projects, at 0.407 (Spearman's correlation) [12].

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

Halstead metrics [8] were posed by Maurice Halstead. These metrics count tokens and provide insight into size, vocabulary, variable counts, operator count, entropy, and information content [25] of code. Halstead metrics have been criticized for being ill-formulated, but metrics such as Halstead volume are measures of information density versus the number of tokens in a block [25]. Halstead metrics, such as volume, were found to correlate with source code readability.

McCabe's and Halstead metrics are combined to form the Maintainability index [23], which is commonly used to estimate the maintainability of software. There is some skepticism that these metrics and other metrics like the OO CKJM metrics [4] actually are meaningful [28] and do not overly correlate with size [5].

4.2 Indentation

Indentation is an interesting indicator of blocks. In Python indentation is semantically meaningful and marks the existence of a block. Hindle et al. [12, 11, 10] found that variance in indentation was correlated (Spearman's $\rho = 0.462$) with McCabe's cyclomatic complexity across many programming languages. Hindle et al. found that while LOC and McCabe's often correlate, variance of indentation was more rank correlated with McCabe's cyclomatic complexity than LOC (0.407).

4.3 Search-Based Software Engineering

Search based software engineering (SBSE) applies search techniques such as genetic algorithms and simulated annealing to software engineering [9]. Genetic algorithms and genetic programming applied to source code, testing, and source code generation fall under the search-based software engineering umbrella.

In terms of the scope of search, which this work is concerned with, prior work by the authors on corporae-based error location are quite relevant as they engage in mutation testing to evaluate error-location [Campbell et al., 3]. Tu et al. [29] find that biasing search methods to local scopes improves the performance of some search-based techniques.

4.3.1 Random Mutation Testing

Mutation testing [16] is a method of checking test-cases and the program itself against corruption of the program. If a test-case cannot detect when a program is mutated, that is modified arbitrarily, then perhaps the test-case is not achieving the appropriate path coverage, furthermore it might indicate that the computation in part of the program has limited dependency on its state and output. In both cases it can indicate that faults are present. In this paper, charmless code hides these mutations, faults and errors, better than charming code. In terms of concrete tools Jester [22] is one of the oldest and most famous mutation testing tools. Many tools followed it and compared against Jester itself and improved on mutation testing [26, 20, 15].

Just et al. [17] asked, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" Indeed, they found "73% of real faults are coupled to the mutants generated by commonly used mutation operators." This provides support for the methodology described in the methodology section that uses mutation operations.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

4.3.2 Genetic Algorithms and Genetic Programming

Genetic algorithms are algorithms that represent the search state of a system as set of genes (a vector of features). These features act as input to a function that is evaluated against a utility function. The gene inputs (vector of features) that perform the best against the utility function are selected and mutated further to find possibly better performing candidates. Genetic programming is the mutation of source code or ASTs of source in a fashion inspired by genetic mutations that happen to the DNA of living organisms. Genetic programming differs from genetic algorithms in the sense that it works on trees (ASTs) rather than on vectors (genetic algorithms).

Mutation-based Program Repair In the presence of errors it has been suggested that possible programs can be mutated and searched to provide mutations that repair faulty software. Forrest et al. [6] and Weimer et al. [30] were credited with successfully demonstrating that fix-patches could be generated by mutation or genetic programming. Programs are mutated and tested against test-cases until they are found to have passed the tests. A patch is extracted as the fix-patch. Dongsun et al. [18] found that by biasing the search function with prior information, they could make templates that had higher immediate performance in certain cases of fix patches.

4.4 Code Coverage Testing

Measuring code coverage of a test suite, the proportion of lines executed while testing, is a respected and common practice among practitioners. There are many tools [31] that measure and extract code coverage from a test-suite. The general belief is that a high quality test-suite has high coverage.

Inozemtseva et al. [14] found that low-moderate correlations between test suite coverage and effectiveness. They controlled for the number of tests and different kinds of coverage. They suggest that this indicates test-code-coverage is a poor indicator of test-suite quality. This emphasizes previous results about the effectiveness of coverage-based testing [13].

Gligoric et al. [7] investigated code coverage and how to evaluate test-suites. They evaluated many criteria and found branch coverage and an intra-procedural acyclic path coverage had good performance. This is relevant to software metrics such as McCabe's cyclomatic complexity because it implies that code that has high McCabe's cyclomatic complexity should be covered.

Andrews et al. [1] have combined code coverage and mutation analysis/testing. They found a clear linear correlation ($R^2 > 0.90$) between mutation detection ratios and coverage, as well as fault detection ratios and coverage. Essentially code coverage of test cases correlated with the fault detection ratio from mutation testing. Our findings somewhat complicates and complements this result, because charmless code is less likely to report errors even if covered.

5 METHODOLOGY

RQ1 "Is charming code related to another software metric, McCabe's Cyclomatic Complexity, for each block?" An experiment was performed to determine if the charm was correlated with the McCabe Cyclomatic Complexity (MCC) of the source. Since

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

both MCC and charm are quantitative measurements based on subsets of some piece of source code, this experiment is designed to determine whether they are capturing the same properties of that piece of source code.

RQ2 "What easily extractable features of Python are relevant to charming code?" An experiment was performed to determine what features of a line of Python code were significantly related to the charm, such as language keywords, indentation level, line length, and block length. A 21-way non-factorial analysis of variance was performed using 21 different easily extractable features of each line of code. Each independent feature tested is listed in Table 1. Features were selected as having a significant relationship with charm if their p-value was less than 0.01.

RQ3 "Can charm be estimated with minimal computational cost?" An experiment was performed to determine whether charm could be estimated using easy-to-compute features and measures of source code. In order to investigate how easily extractable features such as keywords and indentation related to charm, a 21-way non-factorial analysis of variance was performed. Additionally, a linear model was constructed using ordinary least-squares regression. The independent variable was taken to be is charm, and the dependent variables are easily extractable features.

An additional 21-way non-factorial analysis of variance was performed with MCC as the independent variable was also performed. Each independent feature tested is listed in Table 2. Features were selected as having a significant relationship with charm or MCC if their p-value, computed with the F-test, was less than 0.01.

5.1 Direct Estimation

Direct estimation of charm for each line of a program follows the procedure outlined in figure1. This procedure returns a charm value for each line that approximates the true charm of that line by using a basic random sampling method. A similar algorithm is applied to measure subsets of a program other than lines.

Unfortunately the direct estimation procedure takes a large amount of computational effort. For example, in order to obtain precision to 0.01 charm for a short program consisting of 1000 lexical tokens, the procedure must iterate $1000 * 100^2 = 10$ million times. Precision of the direct estimate is taken to be the standard error of the mean used in the estimate. Thus, 10 million slightly different programs must each be executed! While this is an expensive procedure, it can be easily parallelized using map-reduce techniques because each iteration is independent apart from the stopping condition.

5.2 Experiments

Release-quality Python software was used for the experiments presented in this section. The data presented in this paper was obtained from 22 Python 2.7 programs [1] consisting of 9807 lines of Python code and 449 code blocks. The data consists of the results of 880 000 program executions. Every estimated charm results for every line and block is significant to 0.1 charm. Gathering this data required one week on a single Intel Core i7 3770K CPU.

[1] The files/programs used shared at `http://github.com/orezpraw/CharmedDataSet`

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

procedure CHARMPERLINE(p, δ_{max}) ▷ Measure charm directly for a program p.
    $p_{1...l}$ ▷ Treat program p as a vector of lines.
    $T_{1...n} \leftarrow \text{lex}(\vec{p})$ ▷ Lexically analyse program p into n lexemes.
    $m_{1...l} \leftarrow \vec{0}$ ▷ Initialize the count of mutations made to each line to 0.
    $P_{1...l} \leftarrow \vec{0}$ ▷ Initialize the count of errors reported on each line to 0.
    $M \leftarrow 0$ ▷ Initialize mutations made to the program overall to 0.
    $\delta \leftarrow \infty$ ▷ Initialize the upper bound of the standard error in the mean charm to infinity.
    while $\delta > \delta_{max}$ **do** ▷ Iterate until the standard error of the sampled charm falls below threshold δ_{max}.
        $i \leftarrow \text{U}(\{1,2,...,n\})$ ▷ Choose a uniformly random position of a lexeme in the program.
        $j \leftarrow \text{U}(\{1,2,...,n\})$
        $T' \leftarrow \{T_1, T_2, ..., T_{i-2}, T_{i-1}, T_j, T_{i+1}, T_{i+2}, ..., T_n\}$
            ▷ Replace the lexeme at position i with the lexeme at position j.
        $p' \leftarrow \text{join}(T')$ ▷ Reconstruct an executable program p'.
        $e \leftarrow \text{executeAndReturnLineOfError}(p')$ ▷ Run p' and record the line number of the error, if any.
        $m_{\text{lineof}(T_i)} \leftarrow m_{\text{lineof}(T_i)} + 1$ ▷ Record the location of the mutation.
        $P_e \leftarrow P_e + 1$ if $e \in \{1,2,...,l\}$ ▷ Record the location of the error reported, if any.
        $M \leftarrow M + 1$ ▷ Increment the total number of mutations made by one.
        $\overline{c_k} \leftarrow \frac{P_k - m_k}{M/l} \forall k \in \{1,2,...,l\}$ ▷ Update the mean charm.
        $\delta \leftarrow \frac{1 \cdot n}{\sqrt{M}}$ ▷ Update the estimate of the standard error in the mean charm.
    end while
    return $\vec{c}$ ▷ Return mean charm for all lines.
end procedure

Figure 1. Direct estimation algorithm

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

The 22 Python programs are release-quality components of the standard Python library distributed with Python 2.7 itself. These were collected from the 2.7.8 release of Python from `https://www.Python.org/downloads/release/Python-278/`.

Python programs were run in a manner analogous to executing `python program.py`. This parses, evaluates, and executes the python source file. Data was acquired by applying the direct estimation algorithm with 40 000 iterations per program. 40 000 iterations was enough to achieve a charm precision of 0.1 or better in every case. Each iteration, first, reads the original file, p. Once p is read, it is lexically analyzed and a vector of lexemes (lexical tokens) are produced. Associated with each lexeme is information such as line number, character position in the line.

Retaining line numbers and positions of tokens within a line is done in order to prevent tokens from being on a different line after the mutation process. This is important so that when charm is estimated, error messages can be mapped to the correct line and block.

Then, a program p' is produced by copying p, then choosing one lexeme at random from p' and replacing it by another lexeme chosen at random from p'. The replacement token is chosen at random from the same program to maximize the similarity of p' to p. This matches, in some instances, the behavior of tools such as Jester [22], which replaces operators such as `++` with `--`. However, the mutation employed in this paper is more flexible.

Next, p' is put into a Python file on disk and executed. p' must be executed in a separate process, with a separate memory space in order to prevent it from crashing the software collecting the data. Additionally, the software collecting the data waits for at most 10 seconds for p' to crash with an error or exit without an error. Once execution time reaches 10 seconds, the program is forcibly killed. This is a rare occurrence, but because random programs are being executed, they may not necessarily halt.

At the end of each iteration, information about the line on which p' produced an error, if any, is recorded along with the position of the mutation and various other statistics. These results are ready to be summarized into per-line and per-block data.

Each Python program is analyzed by the `radon`[19] Python analysis tool which reports which line each block starts on, ends on, and each block's MCC. This process takes less than 1 second for each file.

Each Python program is also analyzed by a short program to extract the 21 features used for analysis of variance and linear regression modeling. This program is comprised of a collection of regular expressions (of the Perl variety, not the theoretical variety) and computes values for the 21 features per line and per block in less than 1 second.

Finally, mutation data, the output of `radon`, and the features extracted in the previous step are combined and summarized in a per-line and per-block format for easy analysis.

Overall, the most time-consuming part of this process is running each mutant, since it involves `fork()`ing and the Python interpreter process, `import`-ing and executing Python code.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

6 RESULTS

6.1 RQ1: McCabe's Cyclomatic Complexity

Block-level charm had a linear (Pearson's r) correlation with MCC, of -0.13 ($p < 10^{-8}$) and a rank correlation (Spearman's ρ) of -0.44 ($p < 10^{-15}$). Thus, MCC and (negative) charm capture some of the same information about software, but are mostly independent measures. Furthermore, it was discovered that using block charm and number of lines to estimate MCC with a linear model provided minimal improvement over estimating MCC using line count alone. R^2 improved from 0.459 to 0.463. Both models have p-values less than 0.01. A plot of MCC vs block charm is shown in figure 4. Charm is somewhat negatively correlated with MCC, though this correlation is not linear.

The range of the line-level charm, which is simply the difference between the charm of the most charming line and the charm of the least charming line, as computed in Figure 5, within a block had a stronger correlation with MCC of 0.45 ($p < 10^{-15}$, Spearman's ρ). A plot of MCC versus charm range is shown in figure 6.

> Complicated blocks often have less average charm than simple blocks, in terms of cyclomatic complexity, but a greater range of charm when considered line-by-line.

6.2 RQ2: Line Structure Relationship

The most significant impact on charm is created by lines which contain statements which are continued on the next line by employing the Python line-continuation character '\'. An example of this is shown in Figure 3. This is likely due to the fact that in Python, new lines usually begin new statements, the fact that blocks are begun by increasing indentation and ended by decreasing indentation, and the specific implementation of the Python 2.7 parser. Python often reports errors from a multi-line statement on the first line of the statement, but this effect is several orders of magnitude too small to explain the extremely high charm of lines ending with the line-continuation character observed.

After filtering out the nine multi-line statements in the data, the most significant relationship with charm was whether or not a line contained any commas (`,`), assignment operators(=, +=, etc.), attribute references (`thing.property`), or square brackets (`[]`), all of which had a negative correlation with charm.

> Charm is related, on a line-by-line basis, to simple syntactic elements such as line-continuation characters, commas, assignment operators, attribute references and square brackets.

6.3 RQ3: Estimating Charm

The features used in the 21-way non-factorial analysis of variance and their significance to both charm and MCC is listed in Table 2. Then, the significant factors revealed by the analysis of variance were used to construct a linear model by ordinary least-squares regression. The coefficients for that model are listed in the last column of Table 2, along

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

Table 1. Comparison of significant predictors for per-line charm with and without multi-line expressions

Property	Charm (all lines)	Charm (filtered)
Line continuation	significant	N/A
Colons		
Square brackets `[]`		significant
Long strings	significant	
Numbers		
Strings	significant	
Commas	significant	significant
Line length		
Indentation		
Indentation increased		
Indentation will increase		
Keywords		
Branching keywords (if/then/else/try/catch)		
Assignment operators		significant
Attribute references		significant
Tests		
Special variables		
Arithmetic Operators		
Comments		
Parentheses `()`		
Braces `{}`		

with the intercept (fixed offset) and R^2 value. Unfortunately, a usable model for estimating charm would ideally have a R^2 near 1, yet the model obtained from the analysis of variance for charm has a R^2 of only 0.60. The models obtained during experimentation do indicate that software metrics such as charm can be estimated cheaply to some degree.

The linear model presented in Table 2 was constructed by first obtaining per-line charm values by following the procedure in figure 1 for each file in the data set. Importantly not only the final charm value was retained, but the number of mutations per line, m, the total number of mutations for each file, M, and number of Python errors for each line, P, was also retained.

Then, the `radon` tool was run in `cc` mode for each file. The options `-js` were passed to `radon` so that `radon` produced output in JSON format (`-j`) and included numerical complexity values (`-s`). The JSON output by `radon` is easily parse-able and intuitively structured data which includes the name of each block, which line each block starts on, which line each block ends on, and each block's complexity value.

Then, a routine was created to read the JSON output of `radon` and construct a map from program line to program block by considering the start and end lines of each block.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

Using this map, each block's number of mutations and number of Python errors was computed by summing the number of mutations and number of Python errors for each block's constituent lines. Then a final charm value was computed by subtracting each block's total number of mutations from the total number of Python errors and dividing that difference by the average number of mutations per block. The average number of mutations per block was computed by dividing the total number of mutations per file by the total number of blocks per file.

In order to extract the values for the independent variables, the number of colons, literal numbers, attribute references and parentheses were counted in each block. This was done by first removing comments from the block of code and then counting the number of colon characters and parentheses characters. Regular expressions were used to count the number of literal numbers and attribute references. The regular expression for matching attribute references, also known as dot operations, is simply a name followed by a full stop (`.`) followed by a name. The regular expressions matching names and numbers in Python are defined in `lib/tokenize.py` in the Python source distribution.

Ordinary least-squared regression was then employed by combining the final charm values for each block and the counts of colons, parentheses, numbers and attribute references into a single data file with one record per block. Then this data file was loaded into the R statistical computing environment. Once in R, the regression was performed by the built-in `lm` function by passing the formula:

```
lm(charm ~ mle + attributerefs +
           parentheses + numbers + colons)
```

as the only argument, where `mle` is a multi-line expression. Table 2 describes the coefficients and intercepts used in this model resulting in an R^2 of 0.60.

> Charm can be estimated to some degree using easily obtained code features such as the number of colons in the code.

7 DISCUSSION

7.1 Charming Code

Apart from multi-line statements, charming code is often simple statements that introduce a new block such as `try` or `if`, and other features which indicate the beginning of a block of code, such as colons, keywords, and equality or inequality tests.

> Charming lines of code are often short, simple that is commonly sprinkled throughout every Python program and given little thought.

This pattern can be clearly seen in figure 2. Consider, for example, the line of code in figure 2 that simply says "`finally:`." What could possibly go wrong with that line of code? Before the experiments presented in this paper, the authors assumed that such a line of code would not produce many error messages, even under mutation. However,

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

Table 2. Comparison of significant predictors for MCC and per-block charm

Property	MCC ANOVA	Charm ANOVA	Charm LM Coeff.
Multi-line Expressions		significant	6.1
Colons	significant	significant	-0.028
Square brackets `[]`	significant		
Long strings	significant		
Numbers	significant	significant	0.51
Strings	significant		
Commas			
Lines			
Average indentation			
Indentation increased			
Indentation will increase	significant		
Keywords	significant		
Branching keywords (if/then/else/try/catch)	significant		
Assignment operators			
Attribute references	significant	significant	-0.026
Tests	significant		
Special variables	significant		
Arithmetic Operators			
Comments	significant		
Parentheses `()`	significant	significant	-0.013
Braces `{}`			
Intercept			-0.021
R^2			0.60

the data shows that as far as Python error messages are concerned, a lot can go wrong with that line of code.

However, despite this pattern, other surface-level features of the code are better predictors of charming code, as was determined by the analsys of variance detailed in the previous section. Therefore, on a line-by-line basis, it is not completely clear how to *best* identify charming code quickly and easily by eye. It is possible that a factorial analysis of variance would clarify this, but it would involve 51 quintillion factors. It is clear, however, that line continuation plays a huge role for Python 2.7. On a block-by-block basis, blocks containing multi-line statements and many hard-coded number literals are the two best indicators of charming code.

7.2 Charmless Code

Indications of hard-coded data such as square brackets (which are used for array data in Python), commas, and assignment operators become significant indicators of negative charm on a line-by-line basis. Another indicator that becomes significant on the line-by-line level, once multi-line statements are removed, are attribute references. All of

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

these features are present in larger quantities on longer, more complicated lines of code. Consider, for example, the line of code in figure 2 with the least charm (most negative charm):

```
def __setitem__(self, key, item): self.data[key] = item
```

This single line of code defines an entire function with three arguments. There are a lot of things that can go wrong on this line, such as misspelled identifiers, arguments being in the wrong order, etc.

Long, complicated statements with many syntacic elements are often charmless.

Complexity and length are both correlated with negative charm, but they are poorer predictors than other easily countable features such as colons, attribute references and parentheses. This indicates that while many features may correlate with negative charm, it is often the simplest features that perform the best in a linear model.

For example, high MCC, keywords such as `if`, `then`, and `else`, indentation increasing on the following line are all associated with negative charm. However, they can all be replaced by simply counting colon and parentheses characters.

7.3 Practical Applications for Software Engineering

- Extremely charming lines of code pose a threat to the test-ability of the nearby code.
- Large tables of hard-coded numerical values or strings exhibit negative charm.
- Areas of code which exhibit non-zero charm may have an impact on maintenance costs.
- Charm intuitively corresponds with maintainability since its far easier to debug a fault once a software engineer knows what line the fault is on (0 charm).
- Charm is a relationship between code and related code; including tests covering that code.

Extremely charming lines of code pose an extreme threat to a system's ability to locate a fault. These lines should be rewritten if possible. For example, based on the results presented here, the line-continuation character should be avoided in Python 2 code.

Consider the following example code. It contains a constant specification of the first 10 Fibonacci numbers. This code is difficult to check "by eye" and will never cause an error on the same line if it is faulty due to a simple mistake such as transposed numbers, unless it is tested. Lines such as these exhibit negative charm. Therefore, estimating charm line-by-line may be a viable, if expensive, way of locating such lines.

```
fibonacci = [  1,   1,  2,  5,  3,
              13, 21, 34, 55, 89  ]
```


PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

When authoring software, in some situations, it may be desirable to mitigate the effects of areas of code with non-zero charm. Intuitively, because it takes more time to locate the actual source of a fault when the fault is reported on the wrong line, areas with non-zero charm distribution may cost more to debug.

Charm should be estimated by executing all relevant tests to a piece of code being characterized. Charm can be re-estimated as new tests are introduced to evaluate some of their efficacy. This is in line with standard procedures used with Jester [22] and similar tools.

7.4 Recommendations for Genetic Programming

Genetic programming solutions should consider that errors reported in charming blocks might not be caused by those blocks. This is important when multiple mutations are applied. The order of mutation application might be irrelevant given the existence of charming blocks and mutation in charmless code.

Dynamic languages like Python pose difficulties, such as allowing syntactically incorrect blocks to exist in a program. The existence of a string eval adds many layers of difficulty. These difficulties might cause many practitioners to avoid genetic programming with dynamic languages. Yet programming languages like Python and Javascript are very popular and lots of software is written in them. Without the safety of an oracle who knows all valid programs [Campbell et al.], such as a compiler, what can one do for safe mutations? The concept of code charm allows genetic programmers to leverage known risks of charmless and charming code in the absence of an oracle of valid programs.

With the concept of charm, genetic programming can tune itself to be more careful and require more testing of mutations to charmless code while perhaps worrying less about unnoticed behaviour in charming code. Charm highlights the issue of mutations that testing suites might have a hard time exercising. Search algorithms can leverage the concept of charm to avoid combining charming and charmless mutations, thus hiding the effect of the charmless code. These result emphasize the need to test charmless code thoroughly before genetic programming in order to alleviate the hiding or shedding of errors and error messages.

7.5 Future Work

This work investigated charm in Python. Python has interesting properties such as deferring type checking until execution, thus even simple errors may lie dormant until execution. This is not the case for statically compiled code with compilers, including languages like Java and C++. What is very charming in Python code might be completely charmless in other languages. One possible research question is "how similar and different is charm in other languages, especially compiled statically typed languages?" One language that might be interesting to investigate is Fortran 90, since it is compiled and statically typed but has multi-line statements with a line-continuation character just like Python.

Charm may be useful as a feature or predictor of bugs, but this has not been evaluated empirically. If a block of code has very low charm, it is both complex and unlikely

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

to produce error messages. Intuitively, such blocks may be more likely to contain undiscovered bugs.

Charm-based test-suite coverage may improve test suite effectiveness, but this has not been evaluated empirically. Compared with test suite effectiveness and code coverage, what improvements can charm bring to test suite quality measurement?

Cheaply extractable features of source code, that have not been tested here, may provide more accurate charm models. Only 21 such features were evaluated in this paper. However, it is possible to extract many other features or combinations of features.

One possible application of this work which has not yet been investigated is employing charm to recover the structure of code which cannot be viewed directly. As an example, code could hypothetically be encrypted in a way that the encrypted ciphertext version of the code can be modified and any error messages produced are publicly visible. In that situation, the procedure outlined in this paper can still be employed, though not on a line-by-line basis. The results may give a clear indication of some code structures, such as multi-line statements in Python.

8 THREATS TO VALIDITY

Construct validity Construct validity is hampered by the use of mutations rather than actual faults, although there is some prior work to suggest that mutants are good representatives of faults [17].

Internal Validity Internal validity is threatened by the choice of systems to test and choice of mutations to execute, these can result in selection bias. In terms of confounding this work focuses on the confounding effect of charming code because it motivates the entire concept.

External validity External validity is hampered by the use of a small set of Python programs and the sole use of Python. Charm probably is quite relevant to other languages dynamic or statically typed, but this work focused solely on Python. External validity can also be threatened by the number of mutations tested.

9 CONCLUSIONS

Charming code attracts errors and error messages, while charmless code hides errors or passes its error on to other code to report. This concept of charming code is relevant to mutation testing, code coverage based testing, and genetic programming as it indicates that some mutations in charmless areas of the code could be inducing errors that are not reported, or reported in the wrong location. The concept of charming code allows us to talk about the properties of blocks that we might use in random testing, mutation testing and genetic programming.

Charmless code blocks often have higher MCC than charming or neutral code blocks. This makes intuitive sense since it is harder to debug cause and effect relationships in complex code with branches and loops than simpler code. Linear models can be produced that estimate code charm, though not to an ideal degree. Interestingly, charming code blocks are not big or small, they are not moderately or strongly corre-

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

lated with lines of code (LOC). Code charm was demonstrated on a corpus of Python software using the Python language.

Programmers, practitioners, testers, and researchers should consider charming code and charmless code when debugging, testing, or mutation testing code. Not only does charmless code imply that code-coverage is not enough, but that more work has to be done via mutation to tease out errors in charmless code. Furthermore if testers are aware of charming code they can second guess error messages and consider other possible error generating locations that have been misreported by compilers and interpreters. Extremely charming lines of code pose an extreme threat to an interpreter or compiler's ability to locate a fault. Python's line continuation character '\' should be avoided because it creates extremely charming lines of code.

Thus in this work, the concept of charming code and charmless code has been discussed. Its ramifications regarding mutation based testing, and test coverage have been explored, and models that estimate charming code have been provided.

REFERENCES

[1] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624.

[Campbell et al.] Campbell, J. C., Hindle, A., and Amaral, J. N. Python: Where the mutants hide or, corpus-based coding mistake location in dynamic languages. `http://webdocs.cs.ualberta.ca/~joshua2/python.pdf`.

[3] Campbell, J. C., Hindle, A., and Amaral, J. N. (2014). Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 252–261. ACM.

[4] Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[5] El Emam, K., Benlarbi, S., Goel, N., and Rai, S. (2001). The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650.

[6] Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM.

[7] Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M. A., and Marinov, D. (2013). Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA. ACM.

[8] Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA.

[9] Harman, M. and Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14):833–839.

[10] Hindle, A., Godfrey, M., and Holt, R. (2008a). From indentation shapes to code structures. In *8th IEEE Intl. Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

[11] Hindle, A., Godfrey, M., and Holt, R. (2008b). Reading beside the lines: Indentation as a proxy for complexity metrics. In *Proceedings of ICPC 2008*.

[12] Hindle, A., Godfrey, M. W., and Holt, R. C. (2009). Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming*, 74(7):414 – 429. Special Issue on Program Comprehension (ICPC 2008).

[13] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press.

[14] Inozemtseva, L. and Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*.

[15] Irvine, S., Pavlinic, T., Trigg, L., Cleary, J., Inglis, S., and Utting, M. (2007). Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 169–175.

[16] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

[17] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? In *Proceedings of the Symposium on the Foundations of Software Engineering*.

[18] Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press.

[19] Lacchia, M. (2015). Radon 1.2. `https://github.com/rubik/radon/tree/v1.2`.

[20] Madeyski, L. (2010). Judy – a mutation testing tool for java. *IET Software*, 4:32–42(10).

[21] Mccabe, T. J. (1976). A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320.

[22] Moore, I. (2001). Jester-a junit test tester. *Proc. of 2nd XP*, pages 84–87.

[23] Oman, P. W. and Hagemeister, J. (1994). Construction and testing of polynomials predicting software maintainability. *J. Syst. Softw.*, 24(3):251–266.

[24] Ossowski, S., Schneeberger, K., Clark, R. M., Lanz, C., Warthmann, N., and Weigel, D. (2008). Sequencing of natural strains of arabidopsis thaliana with short reads. *Genome research*, 18(12):2024–2033.

[25] Posnett, D., Hindle, A., and Devanbu, P. (2011). A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 73–82. ACM.

[26] Schuler, D. and Zeller, A. (2009). Javalanche: Efficient mutation testing for java. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 297–298, New York, NY, USA. ACM.

[27] Serva, M. and Petroni, F. (2008). Indo-european languages tree by levenshtein distance. *EPL (Europhysics Letters)*, 81(6):68005.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

633 **[28]** Sjøberg, D. I., Anda, B., and Mockus, A. (2012). Questioning software maintenance metrics: A comparative case study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM
636 '12, pages 107–110, New York, NY, USA. ACM.
637 **[29]** Tu, Z., Su, Z., and Devanbu, P. (2014). On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM.
640 **[30]** Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society.
643 **[31]** Yang, Q., Li, J. J., and Weiss, D. M. (2009). A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

Figure 2. Example charm of 50 lines of code from the Python standard library. Most code is charmless in this example.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

```
"""Python part of the warnings subsystem."""
import linecache
import sys
import types
__all__ = ["warn", "showwarning", "formatwarning", "filterwarnings",
           "resetwarnings", "catch_warnings"]
def warnpy3k(message, category=None, stacklevel=1):
    """Issue a deprecation warning for Python 3.x related changes.
    if sys.py3kwarning:
        if category is None:
            category = DeprecationWarning
        warn(message, category, stacklevel+1)
def _show_warning(message, category, filename, lineno, file=None, line=None):
    """Hook to write a warning to a file; replace if you like."""
    if file is None:
        file = sys.stderr
    try:
        file.write(formatwarning(message, category, filename, lineno, line))
    except IOError:
        pass # the file (probably stderr) is invalid - this warning gets lost.
showwarning = _show_warning
def formatwarning(message, category, filename, lineno, line=None):
    """Function to format a warning the standard way."""
    s =  "%s:%s: %s: %s\n" % (filename, lineno, category.__name__, message)
    line = linecache.getline(filename, lineno) if line is None else line
    if line:
        line = line.strip()
        s += "  %s\n" % line
    return s
def filterwarnings(action, message="", category=Warning, module="", lineno=0,
                   append=0):
    """Insert an entry into the list of warnings filters (at the front).
    import re
    assert action in ("error", "ignore", "always", "default", "module",
                      "once"), "invalid action: %r" % (action,)
    assert isinstance(message, basestring), "message must be a string"
    assert isinstance(category, (type, types.ClassType)), \
           "category must be a class"
    assert issubclass(category, Warning), "category must be a Warning subclass"
    assert isinstance(module, basestring), "module must be a string"
    assert isinstance(lineno, int) and lineno >= 0, \
           "lineno must be an int >= 0"
    item = (action, re.compile(message, re.I), category,
            re.compile(module), lineno)
    if append:
        filters.append(item)
    else:
        filters.insert(0, item)
def simplefilter(action, category=Warning, lineno=0, append=0):
    """Insert a simple entry into the list of warnings filters (at the front).
```

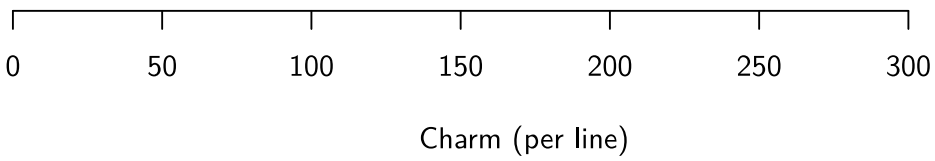

Figure 3. Example charm of each line of code from the Python standard library. This listing shows code which exhibits a line of extremely charming code.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

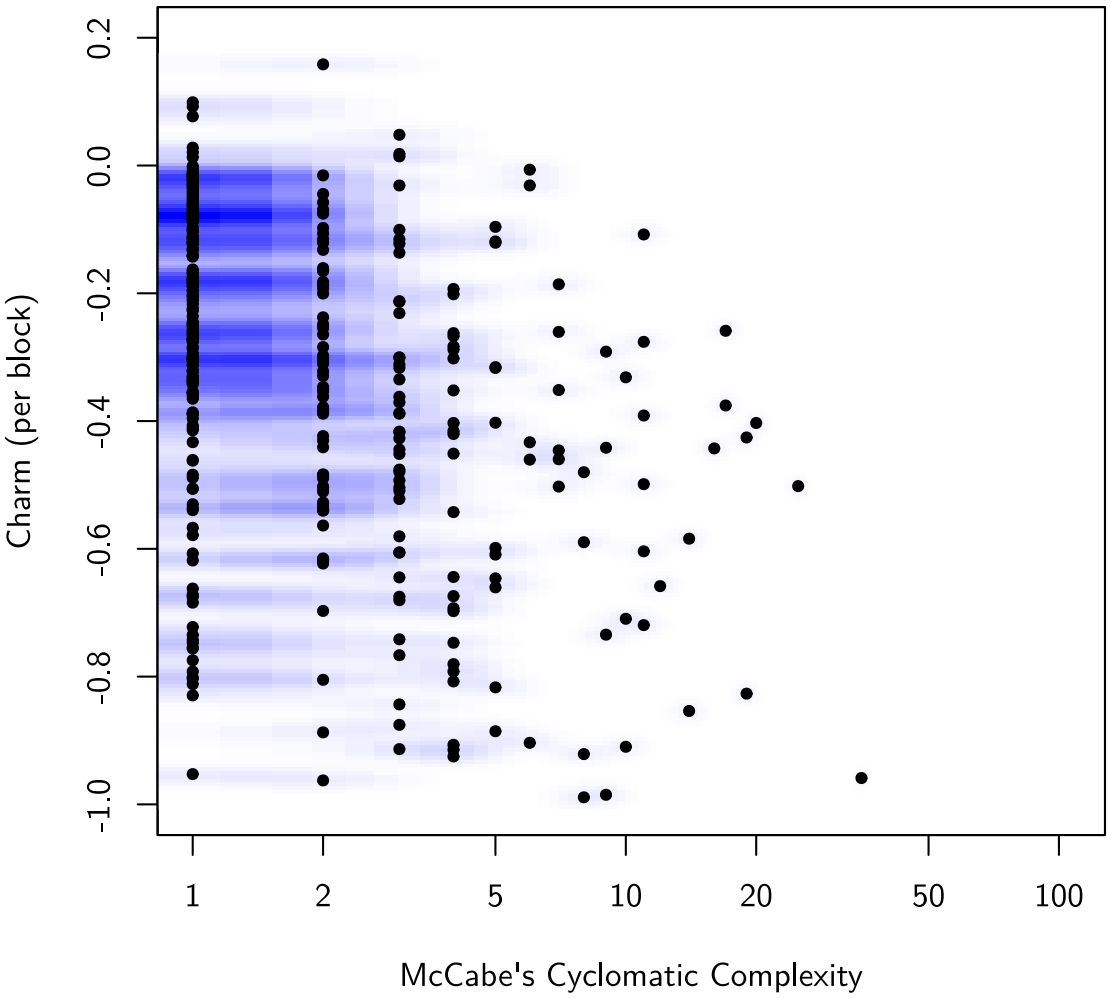

Figure 4. Plot showing Charm vs MCC (density of points corresponds to darkness of blue background).

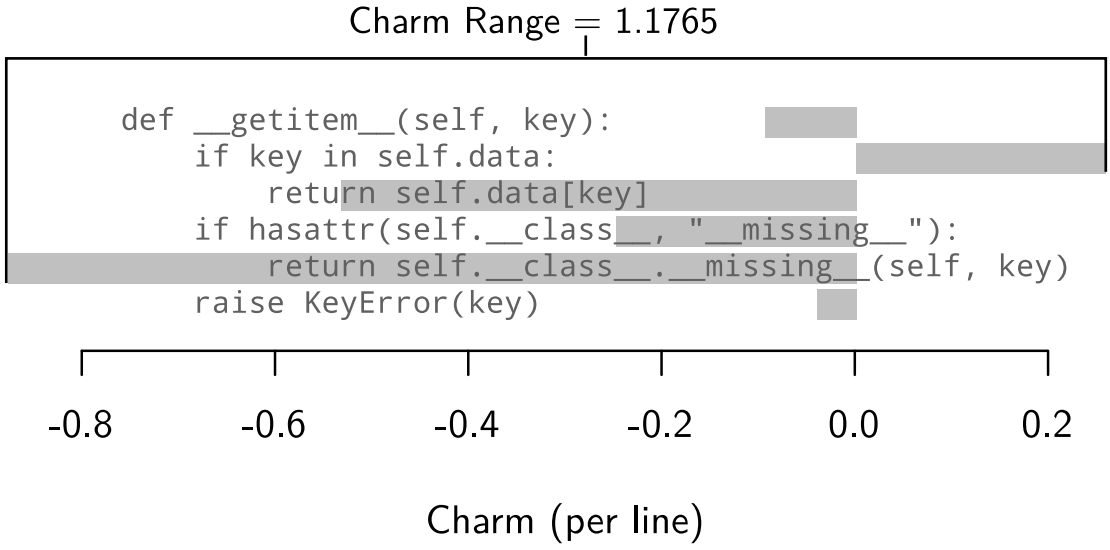

Figure 5. Example charm of one block of code from the Python standard library showing range of charm.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015

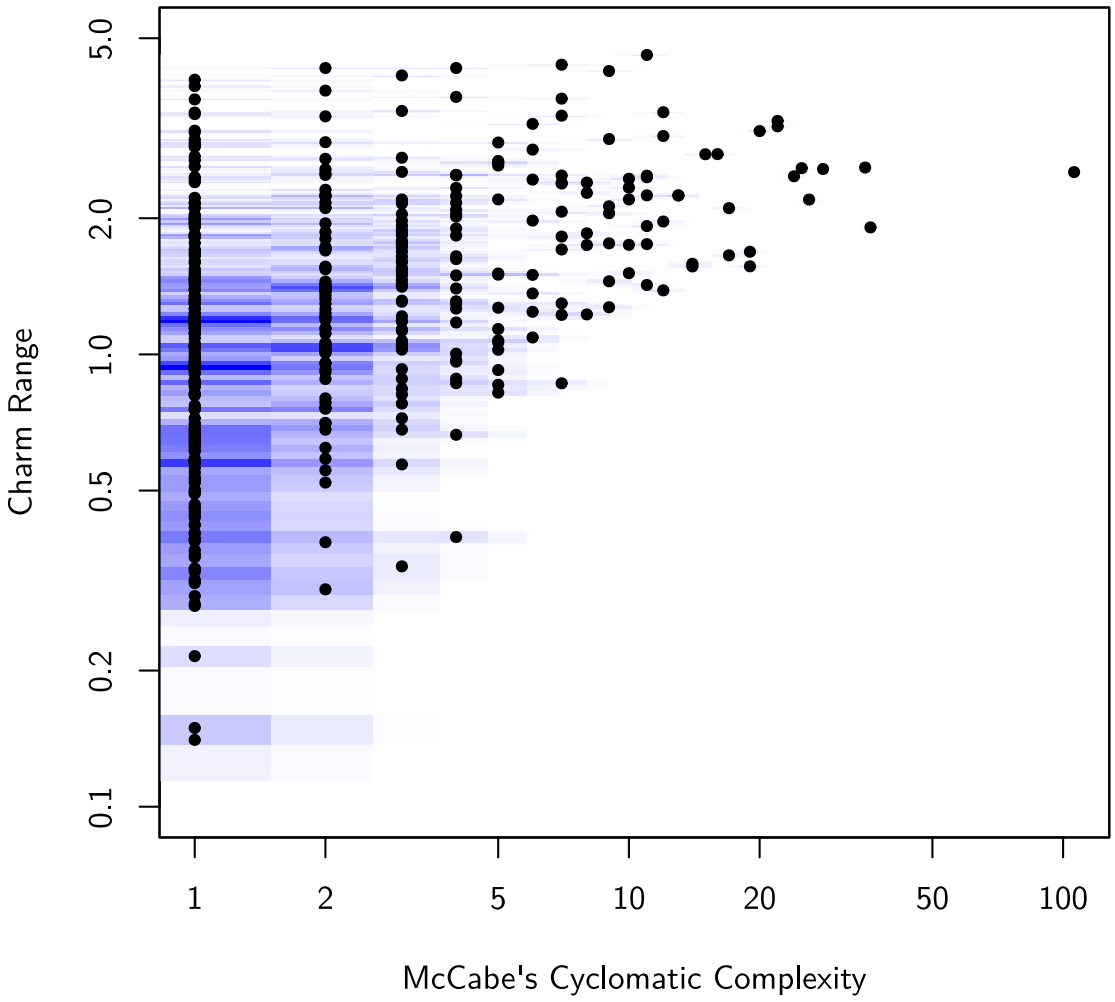

Figure 6. Plot showing Charm Range (maximum line charm minus minimum line charm) vs MCC (density of points corresponds to darkness of blue background).

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1138v1 | CC-BY 4.0 Open Access | rec: 28 May 2015, publ: 29 May 2015