Exempla Gratis (E.G.): Code Examples for Free

Celeste Barnaby
Facebook Inc.
U.S.A.
celestebarnaby@fb.com

Koushik Sen
UC Berkeley
U.S.A.
ksen@berkeley.edu

Tianyi Zhang
Harvard University
U.S.A.
tianyi@seas.harvard.edu

Elena Glassman
Harvard University
U.S.A.
glassman@seas.harvard.edu

Satish Chandra
Facebook Inc.
U.S.A.
schandra@acm.org

ABSTRACT

Modern software engineering often involves using many existing APIs, both open source and – in industrial coding environments – proprietary. Programmers reference documentation and code search tools to remind themselves of proper common usage patterns of APIs. However, high-quality API usage examples are computationally expensive to curate and maintain, and API usage examples retrieved from company-wide code search can be tedious to review. We present a tool, EG, that mines codebases and shows the common, idiomatic usage examples for API methods. EG was integrated into Facebook's internal code search tool for the Hack language and evaluated on open-source GitHub projects written in Python. EG was also compared against code search results and hand-written examples from a popular programming website called ProgramCreek. Compared with these two baselines, examples generated by EG are more succinct and representative with less extraneous statements. In addition, a survey with Facebook developers shows that EG examples are preferred in 97% of cases.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

API examples, big code, software tools

ACM Reference Format:
Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. 2020. Exempla Gratis (E.G.): Code Examples for Free. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3417052

1 INTRODUCTION

Application programming interfaces (APIs) are becoming a pervasive component of modern software engineering. A core challenge for software engineers in industry is to use existing APIs in idiomatic ways within their organization. In order to do this, developers often search for API documentation and *usage examples* [5, 6, 25]. However, this can be especially challenging in companies where many APIs are proprietary. Because those proprietary APIs are only documented within the company by its engineers, there is no externally crowdsourced documentation or examples posted on sites like StackOverflow.

Usage examples are a key component of API documentation [17]. Examples can refresh a programmer's memory [4], concretize the more abstract components of documentation [25], and support code improvement and adaptation [16, 38]. However, there is some risk that programmers will generalize from or adapt an example incorrectly, e.g., leaving in irrelevant components or leaving out critical ones. One useful trait of an example is succinctness [22]: having minimal details specific to a particular usage situation and few superficial distractions, leaving just what is common across most or all proper usages of the API. Alternatively, multiple examples showing a variety of usages [23] may help programmers infer what may be common and uncommon usage patterns and parameter values. Writing usage examples that have these helpful properties is labor-intensive, especially when there exist multiple proper, consistently used usage patterns within an organization.

Regardless of whether available documentation includes one or more usage examples, many programmers instead use company- or project-wide code search to find API usage snippets. However, code search engines results ranking is difficult and often defaults to showing the most recently edited files first. The task of sifting through myriad code search results in an attempt to glean a common usage pattern can be tedious, time-consuming, and unproductive [31]. If the developer does decide to use code from a code search result, they have no assurance that this code represents a common usage, rather than an atypical, niche way of using a method. In fact, prior work has shown that individual code examples may even suffer from API usage violations [37], insecure coding practices [7], and unchecked obsolete usage [39]. Therefore, without thoroughly inspecting and comparing many examples, developers may leave out critical safety checks or desirable usage scenarios.

Several approaches have been previously proposed to address this challenge of presenting programmers with good API usage examples, whether found through search or automatically generated. A number of approaches cluster and rank similar examples to reduce the cognitive load of reading through individual examples [5, 11, 12].

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3417052

However, these clustering techniques rely on pre-defined similarity metrics and do not help users understand why some examples are clustered together, e.g., the commonalities and variations among those examples. Buse et al. presented a synthesis technique to generate a single example from multiple similar examples [5], but the synthetic example only demonstrates a common skeleton, without showing possible variations. In contrast, Examplore is capable of visualizing an entire distribution over API usage features in a large number of API usage examples [8], but the analysis requires a predefined API skeleton and concrete usage patterns are best revealed through additional interaction with the visualization.

In this paper, we present EG, a tool that mines codebases and shows multiple common, idiomatic usage examples for API methods. EG assumes access to a large repository of Python programs from many projects. Given a query API method, EG first searches all methods in the repository containing at least one usage of the API method. It then computes the parse tree of each such method and finds the maximal subtree that a) contains the query API, and b) is part of a meaningful proportion of methods. EG then serializes the subtree to create a common idiomatic usage pattern of the API method. EG repeats this process multiple times to find n diverse idiomatic usage patterns.

Once EG has generated several common usage patterns, it displays the patterns to users in an easy-to-use interface. For each usage pattern, EG displays a concise and representative code snippet to serve as an example of that usage pattern. In each example, EG emphasizes the code parts that are part of the common usage pattern in bold texts, while graying out the uncommon parts – referred to in this paper as "filler". Further, EG displays how many times each usage pattern appears in the repository. This interface allows users to efficiently understand the common usage of an API method, and relieves the cognitive load of manually looking through code results in an attempt to discern a common pattern. In additional, EG relieves the burden of manually curating examples for API methods, and automates the task of keeping API examples up-do-date and relevant as a codebase changes.

EG has several properties that are particularly advantageous for its scalability and generality. First, EG is language agnostic: to generate EG examples for a new programming language, one need only implement a new parser. Second, EG does not require mining coding patterns ahead of time, and can retrieve new and idiomatic usage patterns on-the-fly. Third, EG is fast enough to use in real time, and can generate examples from a large corpus containing millions of methods within a couple of seconds on a multi-core server machine. On average, EG takes 1.0 seconds to generate examples for a query method on a 24-core CPU.

We have implemented EG in C++ for Hack and Python. We have also integrated EG into Facebook's internal code search website, where it is used daily by developers. We report our experimental evaluation of EG for Python. We have used EG to index 1,900,911 Python methods obtained from open source GitHub projects. We performed our experiments for Python because it is a language that is widely used at Facebook, as well as in open source projects. We evaluated EG against code search results and examples from ProgramCreek, a website providing code examples of Python methods. We found that developers preferred EG examples to code search results in over 99% of cases, and that a majority of developers found the main features of the EG interface useful. We also found that EG examples were shorter, more relevant, and more representative than code search results or ProgramCreek examples.

The rest of this paper is organized as follows. Section 2 motivates the design of EG with insights and lessons learned from deploying another code search and recommendation tool in Facebook. Section 3 describes a usage scenario of learning APIs with EG. Section 4 describes the pattern mining and example generation algorithms in EG. Section 5 describes the evaluation of EG, including a survey with Facebook developers, a quantitative analysis of examples generated by EG and two other tools, and a summary of EG's usage metrics after its deployment in Facebook. Section 5.4 discusses the challenges we encountered when evaluating EG. Section 6 discusses related work and Section 7 concludes this paper.

2 MOTIVATIONS FROM FACEBOOK

Aroma is a code-to-code search and recommendation tool. It has been integrated into Facebook's IDE and internal code search website in December 2018 [16]. Given a code snippet as input and a large code corpus, Aroma returns a set of idiomatic extensions to the input code clustered together from similar code snippets in the corpus. Aroma produces code recommendations for Hack, Python, Java, and JavaScript. Here, we summarize how the lessons we learned from Aroma informed the design of EG.

We expected that developers would query Aroma with multi-line code snippets, to get recommendations for how they should modify or improve their code. However, we found that in practice, most Aroma queries were for single API methods. Furthermore, most of these queried APIs were Facebook-specific APIs for which there was little existing documentation and no hand-written examples. We concluded, then, that developers at Facebook were using Aroma to obtain API usage examples.

Since Aroma was not designed for generating examples of API usage, recommendations created from querying a single API method had several shortcomings. First, we found that across many different methods, APIs, and libraries, Aroma recommendations consistently cut out the arguments passed into a function call. For example, in Figure 1, the example generated by Aroma does not include any arguments to the `assert_frame_equal` method in `pandas.testing`. This is because Aroma is designed to prune out code that is different among multiple snippets in a cluster, while retaining code that is commonly shared among them. Since different calls to this method tend to contain different arguments, the arguments are pruned out in the recommendation. Second, examples generated by Aroma include many extraneous statements. In Figure 1, this example contains several lines that are not strictly relevant to the `assert_frame_equal` call, such as the function header, and the initialization of the `query` variable.

For API learning, these shortcomings are detrimental. When learning the common usage of an API method, it is helpful to see its common arguments, and usually unhelpful to see a lot of extraneous context. Prior work has shown that conciseness is an important feature of code examples, and that the median length of hand-written examples is five lines [22]. Aroma's ability to perform fuzzy searches also goes under-utilized when the query is a single method.

Table 1: EG code examples for a variety of Python methods.

Query	Examples	Notes

Case A: `json.dump`	*This usage pattern is found in 29 out of 336 samples.* [1]
	`with open(self.output_path, 'w') as f:`
	`  json.dump(data, f)`
	This usage pattern is found in 17 out of 336 samples. [2]
	`with open(out_filename, "w") as f:`
	`  json.dump(info, f, indent=2)`
	This usage pattern is found in 17 out of 336 samples. [3]
	`with open(Path(self.rnn_dir) / "cli_args.json", "w") as f:`
	`  json.dump(self.cli_args, f, indent=4, sort_keys=True)`
		The first example shows that the following is idiomatic:
		• Opening a file before calling `json.dump`
		• Passing `'w'` as the second argument to open
		• Passing `f` as the second argument to `json.dump`
		The second and third examples show that it is also idiomatic to pass an integer to the optional parameter `indent`, and to pass `True` to the optional parameter `sort_keys`

Case B: `os.makedirs`	*This usage pattern is found in 103 out of 1699 samples.* [4]
	`output_dir = os.path.join(args.output_dir,`
	`  "checkpoint-".format(global_step))`
	`if not os.path.exists(output_dir):`
	`  os.makedirs(output_dir)`
	This usage pattern is found in 110 out of 1699 samples. [5]
	`base_dir = os.path.dirname(fname)`
	`if not os.path.exists(base_dir):`
	`  os.makedirs(base_dir)`
	This usage pattern is found in 116 out of 1699 samples. [6]
	`year_dir = os.path.join(save_dir,`
	`  url.split('/')[-1].split('.')[0])`
	`if not os.path.isdir(year_dir):`
	`  os.makedirs(year_dir)`
		The first example shows that the following is idiomatic:
		• Calling `os.path.join` and `os.path.exists` before calling `os.makedirs`.
		• Calling `os.makedirs` on the condition that the directory you are making does not already exist.
		The second example shows an alternate idiom where `os.path.dirname` is called instead of `os.path.join`, while the third example calls `os.path.isdir` instead of `os.path.exists`.

Case C: `range`	*This usage pattern is found in 213 out of 2000 samples.* [7]
	`for i in range(3):`
	`  img[:,:,i] = (img[:,:,i] - mean[i]) / std[i]`
	This usage pattern is found in 150 out of 2000 samples. [8]
	`columns=[str(col) + '_%d' % (i, )`
	`  for i in range(len(sum_contrast_matrix.column_suffixes))]`
	This usage pattern is found in 123 out of 2000 samples. [9]
	`for j in range(start, i):`
		The first example shows that the following is idiomatic:
		• Calling `range` in the condition of a for loop.
		• Naming the for loop variable `i`.
		The second example shows a common idiom for list comprehension using range, while the third example shows that two variables may be passed to `range`.

Case D: `csv.writer`	*This usage pattern is found in 11 out of 160 samples.* [10]
	`with open(filename, 'a+', newline='') as file:`
	`  writer = csv.writer(file)`
	`  writer.writerow(fieldnames)`
	This usage pattern is found in 11 out of 160 samples. [11]
	`writer = csv.writer(csvfile, delimiter=',',`
	`  quotechar='|', quoting=csv.QUOTE_MINIMAL)`
	`writer.writerow([hike_name, url, trailhead_name, ...])`
	This usage pattern is found in 11 out of 160 samples. [12]
	`with open(ntf.name, "w") as f:`
	`  ntf_writer = csv.writer(f, delimiter=",")`
		The first example shows that the following is idiomatic:
		• Opening a file before calling `csv.writer`
		• Passing `'a+'` as the second argument to open, and passing `''` as the argument to the optional parameter `newline`
		• Calling `writer.writerow` after `csv.writer`
		The second example shows an alternate idiom where a list of items is passed to `writer.writerow`, while the third example shows an idiom where an argument is provided for the optional parameter `delimiter`

Case E: `requests.post`	*This usage pattern is found in 67 out of 1019 samples.* [13]
	`response = requests.get(url, timeout=10)`
	This usage pattern is found in 48 out of 1019 samples. [14]
	`try:`
	`  response = requests.get(url)`
	`except requests.HTTPError as error:`
	This usage pattern is found in 48 out of 1019 samples. [15]
	`url = 'https://kyfw.12306.cn/otn/passcodeNew/...'`
	`r = requests.get(url)`
		The first example shows that the following is idiomatic:
		• Naming the variable assigned to `requests.get` `"response"`
		• Passing two arguments to `requests.get`
		The second example shows an alternate idiom where the call to `requests.get` is wrapped in a try-catch block, while the third example shows that it is common to initialize a string variable names `url` before calling `requests.get`.

```
def test_should_properly_handle_nullable_longs(self, project_id):
    query = """SELECT * FROM
    /* ... */
    df = gbq.read_gbq(
        query,
        project_id=project_id,
        credentials=self.credentials,
        dialect="legacy",
    )
    tm.assert_frame_equal(
```

Figure 1: Aroma recommendation for assert_frame_equal[16]

For these reasons, we decided that, while Aroma is still a powerful code recommendation engine with other potential uses, it is not suitable as a generator of API usage examples. Thus, we created EG to allow developers to see succinct, idiomatic usage examples for an API method. Figure 2 shows the top example generated by EG for learning `assert_frame_equal`. This example includes the arguments of the queried method and removes those extraneous statements. The additional code serves only to further illuminate the use of `assert_frame_equals`, as it shows how to initialize its arguments. Further, EG shows code elements that are common in black text, and code elements that are unique to a single snippet in gray text. This allows users to understand what is common and what is atypical, while still seeing a complete, readable example.

```
df = DataFrame(arr)
result = df.astype(dtype)
expected = DataFrame(arr.astype(dtype))

tm.assert_frame_equal(result, expected)
```

Figure 2: EG's example for `assert_frame_equal` [17]

3 USAGE SCENARIO

This section describes a usage scenario of learning Python APIs with EG. While we find EG to be most useful for proprietary libraries with few hand-written examples, we cannot release such proprietary code in this paper for confidentiality reasons. Thus, for the purposes of this scenario, we assume that hand-written examples for the libraries mentioned are not widely available.

Suppose Harry is a novice Python developer. He needs to write code that creates a directory and then writes some text to a file in that directory. He is aware that there is a `makedirs` function in the `os` package, but he is not sure how to use it. He searches for `os.makedirs` in EG. Figure 3 shows the top example generated by EG. This example shows that among 1699 snippets that call `os.makedirs`, 103 followed the same API usage pattern. The bolded code in this example shows the idiomatic usage of `os.path.exists`. Harry finds that it is common to check whether the directory exists before creating it. Further, he finds that it is idiomatic to call `os.path.join` together with `os.makedirs` to safely construct a file path across platforms. A link to the file containing the code snippet used in this example is displayed above the code snippet, which Harry can use if he wants to see additional context.

Harry clicks "Show More Examples" to view additional usage examples of `os.makedirs`, as shown in Figure 4. He sees that the

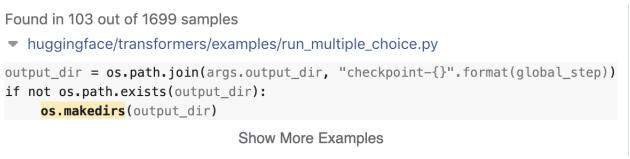


Figure 3: EG's interface showing an example for `os.makedirs`. When code search results initially load, the top EG example is presented as the first result.[4]

third example calls `os.path.isdir` in the `if` statement instead of `os.path.exists`. The text above this code example indicates that this is a common usage pattern appearing in 116 out of 1699 snippets, giving Harry confidence that this is another standard check before calling `os.makedirs`. Harry copies this code from the EG example and replaces `year_dir` with the name of his directory. Since these examples generated by EG have already summarized distinct API usage in hundreds of examples in the codebase, Harry feels he does not look at any additional code search results.

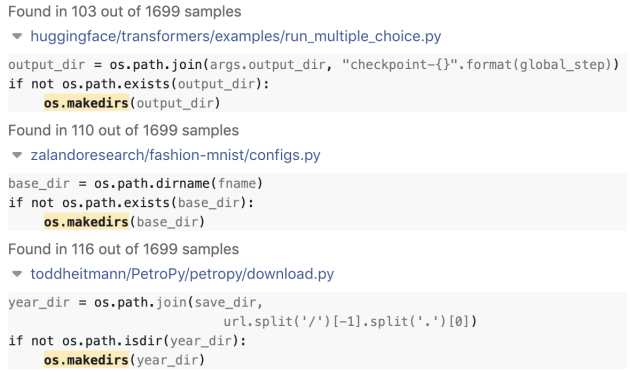


Figure 4: The "Show More Examples" button displays the top three common usage examples.[456]

Harry now needs his code to write text to a file, so he queries `write` in EG, without including a package name. Figure 5 shows the top example generated by EG. He sees that this common usage pattern is found in 150 methods out of 2000 snippets, indicating that it is idiomatic to open a file before calling `write`. He also sees that, in this code snippet, the second argument to `open` is `"w"`. Further search shows that `"w"` means write-only. This is exactly what Harry needs. By stitching this example with the previous example, Harry successfully writes the desired code.

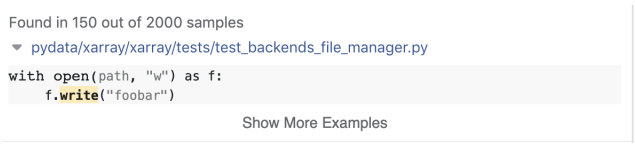


Figure 5: EG's example for `write`.[18]

4 EXAMPLE GENERATION ALGORITHM

In this section, we describe several notations and definitions to compute the simplified parse tree of a program. The terminologies and notations are similar to that in Aroma [16]. We reintroduce the definitions to keep the paper self-contained.

4.1 Formal Definitions

Definition 4.1 (Keyword tokens). This is the set of all tokens in a language whose values are fixed as part of the language. Keyword tokens include keywords such as `while` and `if`, and symbols such as {, }, ., +, *. The set of all keyword tokens is finite for a language.

Definition 4.2 (Non-keyword tokens). This is the set of all tokens that are not keyword tokens. Non-keyword tokens include variable names, method names, field names, and literals.

Examples of non-keyword tokens are `i`, `length`, `0`, `1`, etc. The set of non-keyword tokens is non-finite for most languages.

Definition 4.3 (Simplified Parse Tree). A simplified parse tree is a data structure to represent a program. It is recursively defined as a non-empty list whose elements could be any of the following:

- a non-keyword token,
- a keyword token, or
- a simplified parse tree.

Moreover, a simplified parse tree cannot be a list containing a single simplified parse tree.

We picked this particular representation of programs instead of a conventional abstract syntax tree representation because the representation only consists of program tokens, and does not use any special language-specific rule names such as `IfStatement`, `block` etc. As such, the representation can be used uniformly across various programming languages. Moreover, one could perform an in-order traversal of a simplified parse tree and print the token names to obtain the original program. We use this feature of a simplified parse tree to show the common usage examples.

Definition 4.4 (Label of a Simplified Parse Tree). The label of a simplified parse tree is obtained by concatenating all the elements of the list representing the tree as follows:

- If an element is a keyword token, the value of the token is used for concatenation.
- If an element is a non-keyword token or a simplified parse tree, the special symbol # is used for concatenation.

For example, the label of the simplified parse tree `["x", ">", ["y", ".", "f"]]` is `"#>#"`.

Figure 6 shows a code snippet and its simplified parse tree. In the figure, each internal node represents a simplified parse tree, and is labeled using the tree's label as defined above. Since keyword tokens in a simplified parse tree become part of the label of the tree, we do not create leaf nodes for keyword tokens in the tree diagram—we only add leaf nodes for non-keyword tokens. We show the label of each node in the tree, and add a unique index to each label as subscript to distinguish between nodes with similar labels.

To obtain the simplified parse tree of a code snippet, EG relies on a language-specific parser. For example, EG utilizes the `lib2to3` Python parser to produce the simplified parse tree for a Python program. Once the simplified parse tree of a code snippet has been created, the rest of EG's algorithm is language-agnostic.

```
with open(self.output_path, 'w') as f:
        json.dump(data, f)
```

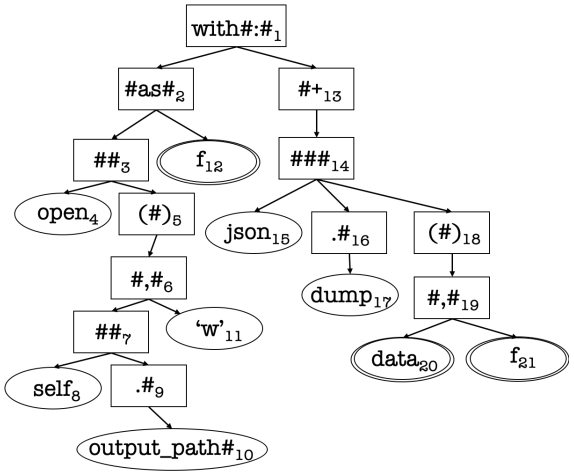


Figure 6: A simplified parse tree of a code snippet. Variable nodes are highlighted in double circles.

We will represent a simplified parse tree t using the tuple (N, L, E), where

- N is the set of nodes of the tree,
- L is a function that maps a node to the label of the subtree rooted at the node,
- E is a children function. If n_2 is the i^{th} direct child of the node n_1, then $E(n_1, i) = n_2$. If the i^{th} child of a node n does not exist, then $E(n, i) = \perp$.

For example, `with#:#`$_1$ and `self`$_8 \in N$ are sample nodes in the tree shown in Figure 6. $L($ `with#:#`$_1) =$ `with#:#`. $E($`#as#`$_2, 2) =$ `f`$_{12}$.

A **subtree** of a tree t is a tree rooted at some node in t and contains all the descendants of the node in t. Formally, $t' = (N', L, E')$ is a subtree of $t = (N, L, E)$ if the following conditions hold:

- $N' \subseteq N$,
- for all $n_1 \in N'$ if there exists $n_2 \in N$ and an $i \in \mathbb{N}$ such that $E(n_1, i) = n_2$, then $n_2 \in N'$ and $E'(n_1, i) = n_2$,
- for all $n \in N'$, if there exists $i \in \mathbb{N}$ such that $E(n, i) = \perp$, then $E'(n, i) = \perp$.

For example, the subtree rooted at `##`$_3$ in Figure 7 is highlighted in red.

A **context** tree of a tree t is the tree with some of its subtrees removed. Formally, if $t' = (N', L, E')$ is a context tree of $t = (N, L, E)$, then the following conditions hold:

- $N' \subseteq N$,
- for all $n_1 \in N'$ if there exists $n_2 \in N$ and a $i \in \mathbb{N}$ such that $E(n_2, i) = n_1$, then $n_2 \in N'$ and $E'(n_2, i) = n_1$.

In Figure 7, we highlight a context of the tree in green.

A **context subtree** can be obtained from a tree first by picking a subtree of the tree and then picking a context tree of the subtree. We also use the term **pattern** to refer to a context subtree.

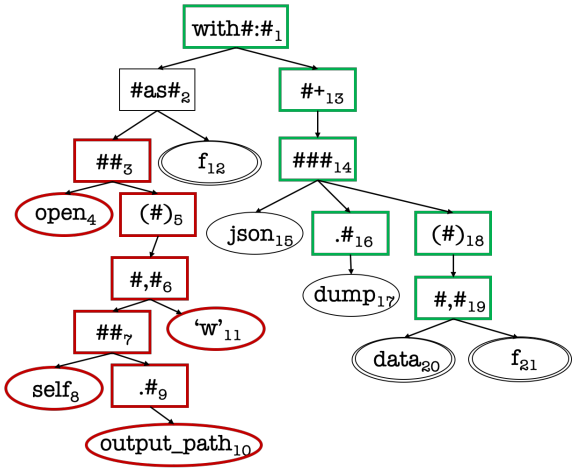


Figure 7: A simplified parse tree with a subtree highlighted in red and a context highlighted in green.

4.2 EG Algorithm

We assume that we are given a set of trees T and a query tree q. Note that a query – for example, `json.dump` – is parsed as a code construct as well. EG works in two steps to create a common usage pattern. First, it finds a pattern t such that q is a subtree of t and t is a pattern in each tree in a subset T' of T. The pattern denotes a partial code snippet that is common and contains the query snippet. Second, EG finds a completion of the pattern by picking a subtree from a suitable tree in T'. The subtree must contain the pattern as a context. The subtree denotes a common usage code snippet of q.

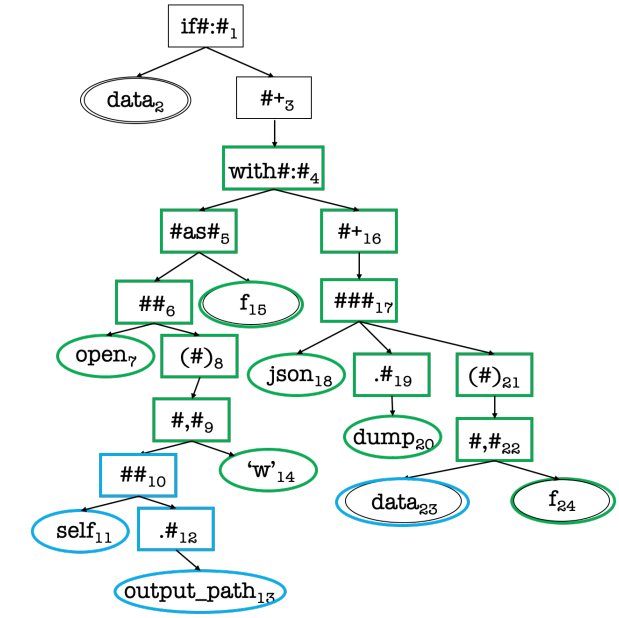
Phase 1. EG starts with the pattern q and grows it iteratively by adding nodes to the pattern as shown in Figure 9. Let us assume that after some iteration the current pattern is $t_c = (N, L, E)$ and it is present exactly in $T_c \subseteq T$ trees. Then a suitable neighboring node n_1 is added to the pattern to obtain a new bigger pattern as described in Figure 10 . The tuple (l, i, n_1, n_2, b) denotes that a node n_1 is added to the tree t_c where l is the label of n_1, n_2 is the node in t_c connected to n_1, and b is a Boolean which if true means $E(n_2, i) = n_1$, and $E(n_1, i) = n_2$ if b is false. The **support** of a tree added to the pattern is the number of trees in T_c that contain the new pattern (see Figure 11). In an iteration, EG adds a node to t_c such that the new pattern has the highest support. At the end of an iteration, EG updates t_c with the new pattern and the set of all the trees in T_c containing the new pattern becomes the new T_c.

EG continues the iterations until the number of nodes in t_c exceeds a configurable threshold γ (usually set to 100) or the cardinality of T_c divided by T goes below a configurable threshold α (usually set to .05). Threshold γ ensures that the generated example is not too long, while threshold α ensures that the generate example is a common snippet.

Figure 8 shows the maximal pattern computed for the query `json.dump` from two simplified parse trees. The nodes in the pattern are highlighted in green.

Phase 2. Once EG has computed a pattern contained in several trees, it tries to complete the pattern by adding the missing subtrees

```
if data:
    with open(self.output_path, 'w') as f:
        json.dump(data, f)
```




```
print("Writing to %s." % json_path)
with open(json_path, 'w') as f:
   json.dump(scan_json_results, f)
```

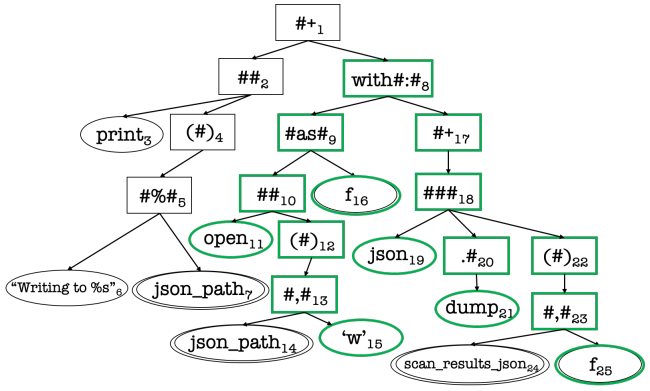


Figure 8: The maximal pattern computed for the query `json.dump` from two different simiplified parse trees. The nodes in the pattern are highlighted in green. The filler code is highlighted in blue.

in the pattern. In EG our goal is to show a real code snippet instead of a synthetic one, because we have found that programmers feel more confident with real code snippets. This means that we need to pick a minimal subtree from the final set T_c such that the subtree contains the pattern. We focus on a few properties of the tree which makes the common usage example code snippet short yet common.

Phase 1:
Input: a set of simplified parse trees T
Input: the query tree q
$T_c \leftarrow \{t \in T \mid t \text{ contains } q \text{ as a subtree}\}$
$t_c \leftarrow q$
while no. of nodes in $t_c \leq \gamma$ and $|T_c| > |T| * \alpha$ **do**
  if $\exists(l, i, n_1, b)$ and $\exists n_2$ in nodes of t_c such that $\textsc{Support}(\textsc{Extend}(t_c, l, i, n_1, n_2, b), T_c) \geq \textsc{Support}(\textsc{Extend}(t_c, l', i', n_1', n_2', b), T_c)$ for all (l', i', n_1') and n_2' in nodes of t_c **then**
    $t_c \leftarrow \textsc{Extend}(t_c, l, i, n_1, n_2, b)$
    $T_c \leftarrow \{t \in T_c \mid t \text{ contains } t_c\}$
  end if
end while
return t_c, T_c

Figure 9: Phase 1 algorithm.

$\textsc{Extend}(t_c, l, i, n_1, n_2, b)$
Let $t_c = (N, L, E)$
$L \leftarrow L \cup \{n_1 \mapsto l\}$
$N \leftarrow N \cup \{n_1\}$
if b **then**
  $E \leftarrow E \cup \{(n_2, i) \mapsto n_1\}$
else
  $E \leftarrow E \cup \{(n_1, i) \mapsto n_2\}$
end if
return t_c

Figure 10: $\textsc{Extend}(t_c, l, i, n_1, n_2, b)$ adds the node n_1 with label l to the node n_1 in t_c and adds the edge $E(n_2, i) = n_1$ if b, and $E(n_1, i) = n_2$ otherwise.

$\textsc{Support}(t, T)$
return $|\{t' \mid t' \in T \text{ and } t' \text{ contains t as a pattern}\}|$

Figure 11: $\textsc{Support}(t, T)$ computes the number of trees in T that contain t as a pattern.

First, the code snippet should be short. Second, it should have more commonality to the code snippets in T_c.

In order to find the best subtree that extends the pattern found in Phase 1, we assign a **score** to each subtree obtained from the trees in T_c. Let us call a subtree that can fill a missing subtree in the pattern a **filler tree**. Let us also call the location at which a filler subtree is missing in the pattern a **hole**. Therefore, a pattern has a fixed finite number of holes. For a given tree t in T_c and a hole in the pattern, let f be the filler that fills the hole in t. The score of f is then the number of trees in T_c where f is the filler of the hole. If the filler has more than β_t of tokens (usually set to 5) or more than β_c characters (usually set to 50), we set the score of f to 0. This ensures that the code snippets are concise. We take the sum of all the fillers of the pattern in t to compute the score of t. We then pick the tree in T_c which has the highest score and show its minimal subtree containing the pattern as the common usage example. EG reconstructs the example with an in-order traversal of the subtree. Figure 8 shows the filler code computed for the query `json.dump`. The nodes in the filler code are highlighted in blue.

4.3 Creating Multiple Common Usage Examples

The algorithm above describes the process of generating a single common usage example for an API method. However, in many scenarios a user may be interested in multiple yet diverse usage examples. EG generates distinct usage examples for a single query as follows. EG generates the first common usage example using the regular EG algorithm described above—however, EG maintains a set of all the nodes added to the pattern in Phase 1. Let us call this set *used_nodes*. EG then saves that pattern, and begins the example generation again with the same initial set of trees. However, if at any point the *second* most common adjacent node is not in *used_nodes* and has at least half as many occurrences as the most common adjacent node, we add that node to the pattern instead. We then finish the example generation as normal.

EG repeats this process n times to create n distinct usage examples. In the EG interface, we display the top three usage examples.

5 EVALUATION

We designed the following experiments to evaluate EG. In each experiment, we compared a code example of an API method generated by EG against a randomly selected code snippet containing the method from the code corpus. The random example displays the line of code where the method is called, as well as the two lines of code preceding and following the method call. This random example serves as a reasonable stand-in for an arbitrary code search result, as code search engines typically display 2-4 lines of additional context by default. We use this as a comparison point since code search is the de facto way developers learn APIs in real-world programming workflows—especially for proprietary APIs where no hand-written examples exist [3, 26].

We aim to answer the following research questions:

RQ1. Do developers prefer EG code examples to code search results?
RQ2. How does EG perform against comparable tools on several quantitative metrics measuring code example quality?
RQ3. If EG is made accessible, will developers incorporate EG code examples into their workflows?

5.1 RQ1: Survey with Facebook Developers

We first conducted a survey to measure the quality of code examples generated by EG, compared with code search results. The survey first displayed six common Python libraries, and asked participants to select two libraries they were most familiar with. The survey then showed ten API methods in each of the two selected libraries. For each method, two code examples were listed: the top-ranked example generated by EG (Option A) and a random example from the code search result (Option B). Participants were asked three questions about these two kinds of examples. Table 2 shows the questions in the survey.

We sent out the survey to 21 Facebook developers. 18 developers completed the survey (86% response rate). Overall, the examples generated by EG were preferred over the random examples 97% of

the time. In addition, 66% of participants agreed that it is helpful to see the number of code examples that follow the same API usage pattern. 100% of participants agreed that it is helpful to color-code and distinguish code parts that are commonly shared among many examples. When asked to describe what they liked and disliked about the two kinds of examples, participants expressed a markedly positive sentiment towards EG: one said, "*the usage count is super useful especially to make sure that the code you are looking at is consistent with the rest of the codebase.*" Another participant said, "*I think that the formatting (color) makes it easier to quickly compare a few examples...and find the most relevant example for your use case.*"

Table 2: Questions asked in the Facebook survey.

1. Suppose you were learning to use this library. Which code examples would you prefer to see? Select one: • Strongly prefer A • Prefer A • Somewhat prefer A • Somewhat prefer B • Prefer B • Strongly prefer B
2. To what extent do you agree with this statement: It is helpful to see the count of methods that contain a common usage pattern (e.g. "Common usage pattern found in 120 out of 2000 methods"). • Strongly agree • Agree • Somewhat agree • Somewhat disagree • Disagree • Strongly disagree
3. To what extent do you agree with this statement: It is helpful for a code example to be formatted so I can see what is common and what is unique to a specific use case (e.g. common part in black, unique part in gray). • Strongly agree • Agree • Somewhat agree • Somewhat disagree • Disagree • Strongly disagree

5.2 RQ2: Quantitative Evaluation with Metrics

In addition to the qualitative survey with real developers, we conducted a quantitative analysis of the quality of examples generated by EG. We defined several metrics to measure example quality:

- ***Succinctness***: How many lines of code are in the example?
- ***Relevancy***: How relevant is the surrounding code in the example w.r.t. understanding the usage of the queried API?
- ***Representativeness***: How frequently do other examples in the code corpus follow the same pattern in the example?

Succinctness is measured by counting the number of lines in an example. We did not count empty lines or code comments. *Relevancy* is measured as the ratio of relevant lines in an example to total lines. A relevant line is a line whose meaning and connection to the query method is clear without additional explanation or context. Figure 12 illustrates this metric by showing random code search results and EG examples for two methods, with relevant lines bolded and the query methods highlighted. In the code search example for `np.array`, the first line does not show how or why `reshape` is called, so this line is deemed irrelevant. Without additional context, we also do not know what `discretize.EntropyMDL` does, so lines 4 and 5 are not relevant. In the EG example for `np.array`, lines 1 and 2 show calls to `np.array`, while lines 3 and 4 show the returned values of `np.array` being passed to `fit_transform` – so all of these lines are relevant. In the code search example for `pd.concat`, it is not clear what `df1` on lines 4 and 5 is used for, and how or if it pertains to `pd.concat` – so these two lines are irrelevant. In the EG example for `pd.concat`, lines 1 and 2 show the initialization of variables passed to `pd.concat` in line 3 – making all 3 lines relevant to understanding how `pd.concat` is used. *Representativeness* is measured by the ranking score that EG assigns to an example. Recall that this score is the sum of the number of occurrences of each filler option in the example. In this way, this score reflects how representative this example is of a common use of the query method. To measure the representativeness of the comparison baseline (i.e., code examples randomly selected from the original search result), we first check whether the method containing this random example is one of the methods containing the EG common usage pattern. If it is, we take that method's ranking score as the representativeness score. Otherwise, we assign the random example a representativeness score of 0. Notice that relevancy is measured by manually assessing the code snippets, while succinctness and representativeness are computed automatically.

For this experiment, we considered four popular Python libraries: `Pandas`, `os`, `Numpy`, and `TensorFlow`. For each library, we selected the ten most used methods in GitHub – forty methods total. We compared average succinctness, relevancy, and representativeness of the top EG example, a random code search result, and the top example from ProgramCreek [2]. ProgramCreek is a website where users can query a Python library method and see functions from open source GitHub projects that call that method. Users vote on which functions represent the best example of a method. The top ProgramCreek example was taken to be the method with the most upvotes. Since the example was a complete method, relevancy was not a meaningful measurement; however, we were still able to measure length.

Table 3 shows the quality of code examples generated by EG, randomly selected from code search results, and selected from ProgramCreek [2]. Compared with examples from EG and ProgramCreek, random examples contained many more irrelevant lines of code, as well as long, uninformative identifier names. Meanwhile, ProgramCreek examples were on average over five times longer than EG examples.

We also collected 100 Hack API methods that had been queried in Facebook's code search website most frequently over a 30 day period. Hack is a programming language created by Facebook as a dialect of PHP [1]. These 100 Hack methods were queried an

Code search examples

```
        ).reshape((100, 1))
Y = np.array([0] * 25 + [1] * 75)
table = data.Table.from_numpy(None, X, Y)
disc = discretize.EntropyMDL()
dvar = disc(table, table.domain[0])
```

```
Ozone1 = pd.concat([df.Ozone] * K)
print(Time1.shape, Ozone1.shape,
        Time1.describe(), Ozone1.describe())
df1 = pd.DataFrame();
df1['Time'] = Time1.values;
```

EG examples

```
X = np.array(['a', 'b', 'c'])
y = np.array([1, 0, 1])
out = encoders.JamesSteinEncoder(model='binary')
                .fit_transform(X, y)
```

```
sparse1 = pd.SparseSeries(val1, name='x')
sparse2 = pd.SparseSeries(val2, name='y')
res = pd.concat([sparse1, sparse2], axis=1)
```

Figure 12: Random code search examples and EG examples for several methods, with relevant lines bolded.[19]

Table 3: The Quality of Code Examples for 40 Popular Methods in Python

Type of Example	Length	Relevancy	Representativeness
EG	2.675	.996	59.6
Code Search	3.9	.640	.2
ProgramCreek	13.8	–	–

average of 8.6 times, ranging from 5 times to 26 times. For these 100 methods, we measured the average length and representativeness of EG examples and random code search results. Since these 100 methods are proprietary API methods in Facebook, we were not able to find curated examples from ProgramCreek. As a result, we are not able to compare EG with ProgramCreek. Table 4 shows the quality of code examples generated by EG and randomly selected from code search results. Similar to the results on open-source libraries, examples generated by EG were significantly more concise and representative than examples selected from the original code search results.

Table 4: The Quality of Code Examples for 100 Internal Methods in Facebook

Type of Example	Length	Representativeness
EG	3.5	116.6
Code Search	4.6	2.1

5.3 RQ3: Live Usage in Facebook

We have integrated EG into Facebook's internal code search website. When users query a method name in Hack or Python, the top EG example is displayed first, before the standard code search results. There is a link to the full contents of the file containing the code snippet used in the example, and a "Show More Examples" button that displays two additional common usage patterns. EG only shows the three most common usage patterns, ensuring that its interface is compact and easy to use. EG's integration into the code search platform was *frictionless*: developers began to use EG with no prior announcement or tutorial.

EG is deployed on a dedicated set of servers to respond to queries from developers. Our search server has 24 cores, and on average takes 1.0 seconds end-to-end to generate Python code examples for the queries used in Section 5.2. The median response time is .8 seconds and the maximum is 2.3 seconds. EG re-indexes the millions of methods in Facebook's codebase daily. This indexing process works the same as in Aroma [16]. On a 24-core server, this process takes 20 minutes on average. If EG were to be deployed on a larger codebase, it would be possible to implement incremental indexing for only changed files. Since the goal of EG is to provide relevant and up-to-date usage examples, we show examples for only the most recently indexed version of the codebase, and we do not maintain past examples generated from prior versions.

We have been logging the usage of EG in the code search website. We log each time a user copies or selects code from an EG example, clicks the file link, or clicks the "Show More Examples" button. Note that copying and selecting are the only events we log with a clear signal that the user actually reused code in the example.

Over a period of 24 days, from April 20 to May 13, EG was triggered to generate code examples for an average of 1,171 code search queries per day. Facebook developers interacted with EG examples an average of 59 times per day, and copied or selected code from an EG example an average of 30 times per day. While this appears to be a low ratio of interactions to total examples generated, there are several factors to keep in mind. First, developers do not always query method names because they want to see code examples—for instance, a developer may instead be looking for a specific file or class. We have no way to determine what a developer's intentions are when they query a method name. Second, note that we integrated EG into the code search website without any public announcement. Therefore, Facebook developers may not even notice it among the other features of the code search website. The discoverability of EG is an orthogonal problem from its effectiveness, which we will investigate in the future. Finally, because a central feature of EG examples is succinctness, developers may be learning or "mentally copying" from an example without physically interacting with it.

Despite these concerns, these results show that real developers indeed utilize EG in their workflows. A formal A/B test comparing EG examples against code search results remains as future work.

5.4 Discussion

Evaluating example generation is an interesting and complicated problem. We initially attempted to design a study wherein developers receive a comprehensive list of API usage questions. They are asked to answer these questions for one API using EG, and

for another using a realistic baseline of code search. The problem with this approach is that EG's focus is on providing a short list of idiomatic usage examples. EG makes no claim to offer the best or most informative usage example—but a succinct example representing a common usage pattern. Thus, we needed an evaluation that measured the benefit of seeing such a common usage pattern.

We next attempted to design a human study wherein participants complete short programming tasks using EG. A main challenge was devising a control to measure EG against. An obvious candidate is Facebook's internal code search website. However, EG is not intended to replace code search altogether, but rather to be a complementary extension integrated into existing code search tools. Thus, it did not make sense to restrict the use of code search. However, Facebook employees are conditioned to use code search results as a go-to method for API inquiries, so even when the EG example contained salient, time-saving information, they often still wanted to page through code search results. Untangling what the user gained from EG versus what they gained from code search, or from documentation, proved difficult. In addition, success in solving a short programming task is extremely dependent on what background knowledge a developer has.

A tool like EG also runs the risk of identifying and perpetuating common anti-patterns. By indexing Facebook's codebase, we ensure that all code has been reviewed by a developer – however, code can still become out of date or deprecated. A potential solution would be to only index code in a codebase written after a certain threshold date. Another would be to indicate to the user in the UI the date when the code displayed in an example was written. Exploring this issue further is left for future work.

6 RELATED WORK

Developers often search for code in their own codebases or online to fulfill programming needs such as learning new APIs and locating code snippets with desired functionality [4, 21, 26, 28, 34]. For example, Sim et al. conducted a lab study with 36 graduate students to evaluate the effectiveness of different code retrieval techniques [28]. In the demographic survey, 50% of participants reported to search code online frequently and 39% reported to search occasionally. Sadowski et al. analyzed the search logs generated by 27 Google developers over two weeks [26]. They found that developers issued an average of 12 code search queries per weekday.

There is a large body of literature in code search [3, 9, 10, 12–15, 18–20, 24, 27, 29, 30, 32, 33, 35, 36]. These techniques focus on 1) enriching search queries and 2) improving search algorithms. For example, beyond simple keyword descriptions, S^6 [24] and CodeGenie [15] allow users to identify relevant code based on test cases. Prospector [18] supports expressing type constraints such as desired input and output types in a query. Code-to-code search tools such as FaCoY [13] take code fragments directly as input and identify other similar code. Wang et al. represented source code as a dependency graph to capture control-flow and data-flow dependencies in a program, and matched search queries against program dependence graphs [35]. Gu et al. trained a neural network to predict relevant code examples given natural language queries [9].

Unlike our work, the aforementioned techniques provide limited support for browsing and assessing code search results. Previous studies have shown that it is cognitively demanding to navigate through code search results [6, 31]. As a result, developers often rapidly skim through a handful of search results and make a quick judgement about the quality of these results [4]. When browsing search results, they also often backtrack due to irrelevant or uninteresting information in search results [6]. More specifically, Starke et al. show that developers rarely look beyond five examples when searching for code examples [31]. These observations indicate that the code exploration process is often limited to a few search results, leaving a large portion of foraged information unexplored.

Several approaches have been proposed to help developers navigate through code search results. To enable users to explore a large number of code examples simultaneously, Examplore constructs a code skeleton with statistical distributions of individual API usage features in those examples [8]. ALICE allows users to mark several search results as desired or undesired and then automatically filter the remaining search results, so users do not have to manually go through all of them [30]. eXoaDocs employs program slicing to remove extraneous statements in a code example and then clusters sliced code examples based on the similarity of semantic characteristics such as invoked API methods in an example [12]. Buse and Weimer improved eXoaDocs by synthesizing a single concise code example to summarize similar examples in a cluster [5].

Our approach differs from these techniques in several perspectives. While the tools described above rely on the syntax and semantics of the Java language, EG is language agnostic, requiring only a parser for the target language. Examplore requires a pre-defined API usage skeleton to register and align code examples, while EG does not require a pre-defined skeleton. Buse and Weimer's tool generates usage examples for a target class, while EG generates usage examples for API or library methods. Finally, to our knowledge, EG is the only tool designed to generate common usage patterns of APIs that has been integrated into the code search platform of a large software company, and is used by developers daily.

7 CONCLUSION

We presented EG, a new tool for generating usage example for API methods. EG works by first indexing a large code corpus. Given a query method, it assembles a list of method bodies in the corpus containing that method, then finds the maximal subtree that contains the query API and is part of a meaningful proportion of methods. EG then reconstructs this subtree into a succinct, relevant and representative code example.

To evaluate EG, we indexed a code corpus of 1.9 million Python methods, and designed a survey where we showed developers pairs of EG examples and code search results for commonly used methods in popular Python libraries. We observed that developers preferred EG examples to code search results 97% of the time, and that 100% of developers agreed that the color-coding of the common usage pattern in EG examples is helpful. Further, we defined several metrics to measure example quality, and quantitatively compared EG examples against code search results and ProgramCreek examples using these metrics. We found that across all metrics, EG performs better than these alternatives. Finally, we integrated EG into Facebook's internal code search website. A log of developers' activities shows that developers indeed interact with EG examples.

CODE REFERENCES

[1]This code snippet is adapted from https://github.com/openai/gym/blob/master/gym/wrappers/monitoring/video_recorder.py#L229. Accessed in March 2020.

[2]This code snippet is adapted from https://github.com/scrapinghub/splash/blob/master/scripts/rst2inspections.py#L77. Accessed in March 2020.

[3]This code snippet is adapted from https://github.com/supernnova/SuperNNova/blob/master/supernnova/utils/experiment_settings.py#L161. Accessed in March 2020.

[4]This code snippet is adapted from https://github.com//huggingface/transformers/tree/master/examples/run_multiple_choice.py. Accessed in March 2020.

[5]This code snippet is adapted from https://github.com/zalandoresearch/fashion-mnist/blob/master/configs.py#L49. Accessed in March 2020.

[6]This code snippet is adapted from https://github.com/toddheitmann/PetroPy/blob/master/petropy/download.py#L195. Accessed in March 2020.

[7]This code snippet is adapted from https://github.com/TarrySingh/Artificial-Intelligence-Deep-Learning-Machine-Learning-Tutorials/blob/master/deep-learning/1-pixel-attack/networks/capsnet.py#L41

[8]This code snippet is adapted from https://github.com/scikit-learn-contrib/category_encoders/blob/master/category_encoders/sum_coding.py#L238. Accessed in March 2020.

[9]This code snippet is adapted from https://github.com/waditu/tushare/blob/master/tushare/util/common.py#L40. Accessed in March 2020.

[10]This code snippet is adapted from https://github.com/bboczeng/Nyxar/blob/master/api/coinmarketcap.py#L94. Accessed in March 2020.

[11]This code snippet is adapted from https://github.com/vgpena/next-weekend/blob/master/scraper.py#L82

[12]This code snippet is adapted from https://github.com/baychimo/loto/blob/master/tests/test_loto.py#L134. Accessed in March 2020.

[13]This code snippet is adapted from https://github.com/home-assistant/core/blob/master/homeassistant/components/ohmconnect/sensor.py#L70. Accessed in March 2020.

[14]This code snippet is adapted from https://github.com/zvtvz/zvt/blob/master/zvt/recorders/exchange/china_index_list_spider.py#L85. Accessed in March 2020.

[15]This code snippet is adapted from https://github.com/testerSunshine/12306/blob/master/verify/pretreatment.py#L26. Accessed in March 2020.

[16]This code snippet is adapted from https://github.com/pydata/pandas-gbq/blob/master/tests/system/test_gbq.py#L130. Accessed in March 2020.

[17]This code snippet is adapted from https://github.com/pandas-dev/pandas/blob/master/pandas/tests/frame/test_dtypes.py. Accessed in March 2020.

[18]This code is adapted from https://github.com/pydata/xarray/blob/master/xarray/tests/test_backends_file_manager.py#L197. Accessed in March 2020.

[19]These code snippets have been adapted from https://github.com/renn0xtek9/Arithmos/blob/799fe071ab3a85ea9a0f86b8099548f11be96841/Arithmos/tests/test_discretize.py#L111, and https://github.com/antoinecarme/pyaf/blob/master/tests/perf/test_ozone_long_series.py#L23. Accessed in March 2020.

REFERENCES

[1] 2020. Hack: Programming Productivity Without Breaking Things. https://hacklang.org/.

[2] 2020. Program Creek. https://www.programcreek.com/.

[3] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

[4] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.

[5] Raymond PL Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 782–792.

[6] Ekwa Duala-Ekoko and Martin P Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 266–276.

[7] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 121–136.

[8] Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing api usage examples at scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[9] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[10] Reid Holmes and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA). ACM Press, New York, NY, USA, 117–125. https://doi.org/10.1145/1062455.1062491

[11] Nikolaos Katirtzis, Themistoklis Diamantopoulos, and Charles Sutton. 2018. Summarizing Software API Usage Examples Using Clustering Techniques. In *Fundamental Approaches to Software Engineering*, Alessandra Russo and Andy Schürr (Eds.). Springer International Publishing, Cham, 189–206.

[12] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an intelligent code search engine. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

[13] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 946–957.

[14] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. 2009. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 476–482.

[15] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. CodeGenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 525–526.

[16] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152, 28 pages. https://doi.org/10.1145/3360578

[17] Walid Maalej and Martin P Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[18] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (Chicago, IL, USA). ACM, New York, NY, USA, 48–61. https://doi.org/10.1145/1065010.1065018

[19] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1069–1087.

[20] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 111–120.

[21] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. 2013. Documenting apis with examples: Lessons learned with the apiminer platform. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 401–408.

[22] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 25–34.

[23] MaryKay Orgill. 2012. *Variation Theory*. Springer US, Boston, MA, 3391–3393. https://doi.org/10.1007/978-1-4419-1428-6_272

[24] Steven P Reiss. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 243–253.

[25] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[26] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 191–201.

[27] Naiyana Sahavechaphan and Kajal Claypool. 2006. Xsnippet: mining for sample code. *ACM Sigplan Notices* 41, 10 (2006), 413–430.

[28] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 1 (2011), 1–25.

[29] Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering* 23, 5 (2018), 2622–2654.

[30] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active inductive logic programming for code search. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 292–303.

[31] Jamie Starke, Chris Luce, and Jonathan Sillito. 2009. Working with search results. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. IEEE Computer Society, 53–56.

[32] Jeffrey Stylos and Brad A Myers. 2006. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 195–202.

[33] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM,

204–213.

[34] Medha Umarji, Susan Elliott Sim, and Crista Lopes. 2008. Archetypal internet-scale source code searching. In *IFIP International Conference on Open Source Systems*. Springer, 257–263.

[35] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. 2010. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated software engineering* (Antwerp, Belgium) *(ASE '10)*. ACM, New York, NY, USA, 457–466. https://doi.org/10.1145/1858996.1859091

[36] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 344–354.

[37] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.

[38] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and supporting adaptation of online code examples. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 316–327.

[39] Jing Zhou and Robert J Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 266–277.