A. Artifacts Available for Review

At submission, we made the following artifacts available for reviewing. Along with the final version of this work, we will make these artifacts publicly available.

Datasets A zip file containing the datasets will be available in the final version of this paper. For each of our six fine-tuning tasks, the zip file contains three JSON files – one each for the train, validation, and test splits. Thus, there are a total of 18 JSON files for the six tasks and their three folds each. The labels for the variable-misuse localization and repair task are pointer values (i.e., boolean vectors over the length of the tokenized function). We therefore provide the subword-tokenized sequence of the input functions. For all other tasks, we provide the untokenized source-code of the functions for better readability. Each example consists of the input function, the label and information about the origin of the example. Each example is either 1) in the unmodified form and the information identifies the GitHub project, file and function name where it came from, or 2) in a modified form and the information identifies the nature of the modification along with the original source of the function. In the case of the negative function-docstring examples, the information also tells us where the wrong docstring was obtained from. For the negative examples which involve modifying the original function, the information tells us the nature of the modification.

The zip file also contains a JSON file with the list of all files that were included in the pre-training corpus after deduplication. The url field of each dictionary entry in this file is a unique URL to unambiguously identify the file version that went into pre-training. The pre-training dataset can be extracted using these URLs and GitHub APIs.

The zip file comes with a Python script print_stats. Upon executing this, it loads each of the JSON files in the folder and prints the fields in the dictionaries contained in those files, and the total number of such dictionaries in the file. These numbers can be tallied against the split-wise numbers provided in Table 1 of the main paper.

Model checkpoints We provide the following model checkpoints:

- Pre-trained checkpoint.
- Fine-tuned checkpoint for the function-docstring classification task. This is a sequence-pair classification task.
- Fine-tuned checkpoint for the exception-type classification task. This is a multi-class classification task.
- Fine-tuned checkpoints for the variable-misuse, swapped operand, and wrong binary operator classification tasks. These are binary classification tasks. We only provided the variable-misuse classification checkpoint at submission time, due to size, but will make all available with the final version of the paper.
- Fine-tuned checkpoint for the variable-misuse localization and repair task. This is the multi-headed pointer prediction task.

The checkpoints can be loaded with the code of the publicly released BERT model (https://github.com/google-research/bert). The extension of this model to the multi-headed pointer prediction problem will be open sourced. We will make checkpoints available for all lengths, train dataset sizes, and epochs.

Subword vocabulary The file (to be released in the final version) is the list of subword tokens that are part of the vocabulary used for tokenizing the pre-training and fine-tuning examples. It can be used with the Tensor2Tensor’s subword tokenizer https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/data_generators/text_encoder.py.
B. Data Preparation for Fine-Tuning Tasks

B.1. Label Frequencies

All four of our binary-classification fine-tuning tasks had an equal number of buggy and bug-free examples. The Exception task, which is a multi-class classification task, had a different number of examples per class (i.e., exception types). For the Exception task, we show the breakdown of example counts per label for our fine-tuning dataset splits in Table 1.

B.2. Fine-Tuning Task Datasets

In this section, we describe in detail how we produced our fine-tuning datasets (Section 3.4 of the main paper).

A common primitive in all our data generation is splitting a Python module into functions. We do this by parsing the Python file and identifying function definitions in the Abstract Syntax Tree that have no other function definition between themselves and the root of the tree. The resulting functions include functions defined at module scope, but also methods of classes and subclasses. Not included are functions defined within other function and method bodies, or methods of classes that are, themselves, defined within other function or method bodies.

We do not filter functions by length, although task-specific data generation may filter out some functions (see below). When generating examples for a fixed-length pre-training or fine-tuning model, we prune all examples to the maximum target sequence length (in this paper we consider 128, 256, and 512 subtokenized sequence lengths). Note that if a synthetically generated buggy/bug-free example pair differs only at a location beyond the target length (say on the 600-th subtoken), we still retain both examples. For instance, for the Variable-Misuse Localization and Repair task, we retain both buggy and bug-free examples, even if the error and/or repair locations lie beyond the end of the maximum target length. During evaluation, if the error or repair locations fall beyond the length limit of the example, we count the example as a model failure.

B.2.1. Reproducible Data Generation

We make pseudorandom choices at various stages in fine-tuning data generation. It was important to design a pseudorandomness mechanism that gave (a) reproducible data generation, (b) non-deterministic choices drawn from the uniform distribution, and (c) order independence. Order independence is important because our data generation is done in a distributed fashion (using Apache Beam), so different pseudorandom number generator state machines are used by each distributed worker.

Pseudorandomness is computed based on an experiment-wide seed, but is independent of the order in which examples are generated. Specifically, to make a pseudorandom choice about a function, we hash (using MD5) the seed and the function data (its source code and metadata about its provenance), and use the resulting hash as a uniform pseudorandom value from the function, for whatever needs the data generator has (e.g., in choosing one of multiple choices). In that way, the same function will always result in the same choices given a seed, regardless of the order in which each function is processed, resulting in reproducible dataset generation.

To choose among multiple choices, we hash the function’s pseudorandom value along with all choices (sorted in a canonical order) and use the digest to compute an index within the list of choices. Note that given two choices over different candidates but for the same function, independent decisions will be drawn. We also use such order-independent pseudorandomness when subsampling datasets (e.g., to generate the validation datasets). In those cases, we hash a sample with the seed, as above, and turn the resulting digest into a pseudorandom number in [0, 1], which can be used to decide given a target sampling rate.

B.2.2. Variable-Misuse Classification

A variable use is any mention of a variable in a load scope. This includes a variable that appears in the right-hand side of an assignment, or a field dereference. We regard as defined all variables mentioned either in the formal arguments of a function definition, or on the right-hand side of an assignment. We do not include in our defined variables those declared in module scope (i.e., globals).

To decide whether to generate examples from a function, we parse it, and collect all variable-use locations, and all defined variables, as described above. We discard the function if it has no variable uses, or if it defines fewer than two variables; if there is only one variable defined, the problem of detecting variable misuse is moot. For any function that we do not discard, we generate a buggy and a bug-free example, as described next.

To generate a buggy example from a function, we choose one variable use pseudorandomly (see above how multiple-choice decisions are done), and replace its current occupant with a different pseudorandomly-chosen variable defined in the function (with a separate multiple-choice decision).

B.2.3. Wrong Binary Operator

This task considers both commutative and non-commutative binary operators (unlike the Swapped-Argument Classification task). See Table 2 for the full list, and note that we have excluded relatively infrequent operators, e.g., the Python integer division operator //.
Table 1. Example counts per class for the Exception Type task, broken down into the dataset splits. We show separately the 100% train dataset, as well as its 33% and 66% subsamples used in the ablations.

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Test</th>
<th>Validation</th>
<th>Train 100%</th>
<th>66%</th>
<th>33%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueError</td>
<td>2324</td>
<td>477</td>
<td>4058</td>
<td>2715</td>
<td>1344</td>
</tr>
<tr>
<td>KeyError</td>
<td>2240</td>
<td>453</td>
<td>4009</td>
<td>2566</td>
<td>1271</td>
</tr>
<tr>
<td>AttributeError</td>
<td>1657</td>
<td>311</td>
<td>2895</td>
<td>1896</td>
<td>876</td>
</tr>
<tr>
<td>TypeError</td>
<td>913</td>
<td>187</td>
<td>1747</td>
<td>1175</td>
<td>564</td>
</tr>
<tr>
<td>OSError</td>
<td>891</td>
<td>164</td>
<td>1641</td>
<td>1106</td>
<td>543</td>
</tr>
<tr>
<td>IOError</td>
<td>865</td>
<td>168</td>
<td>1560</td>
<td>1046</td>
<td>560</td>
</tr>
<tr>
<td>ImportError</td>
<td>776</td>
<td>202</td>
<td>1372</td>
<td>935</td>
<td>471</td>
</tr>
<tr>
<td>IndexError</td>
<td>694</td>
<td>153</td>
<td>1197</td>
<td>813</td>
<td>408</td>
</tr>
<tr>
<td>DoesNotExist</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>KeyboardInterrupt</td>
<td>287</td>
<td>67</td>
<td>590</td>
<td>408</td>
<td>223</td>
</tr>
<tr>
<td>StopIteration</td>
<td>307</td>
<td>69</td>
<td>488</td>
<td>302</td>
<td>155</td>
</tr>
<tr>
<td>AssertionError</td>
<td>177</td>
<td>32</td>
<td>397</td>
<td>276</td>
<td>158</td>
</tr>
<tr>
<td>SystemExit</td>
<td>139</td>
<td>23</td>
<td>264</td>
<td>173</td>
<td>101</td>
</tr>
<tr>
<td>RuntimeError</td>
<td>128</td>
<td>36</td>
<td>299</td>
<td>203</td>
<td>104</td>
</tr>
<tr>
<td>HTTPError</td>
<td>59</td>
<td>13</td>
<td>119</td>
<td>80</td>
<td>35</td>
</tr>
<tr>
<td>UnicodeDecodeError</td>
<td>151</td>
<td>24</td>
<td>251</td>
<td>173</td>
<td>82</td>
</tr>
<tr>
<td>NotImplemented</td>
<td>127</td>
<td>27</td>
<td>222</td>
<td>136</td>
<td>52</td>
</tr>
<tr>
<td>ValidationError</td>
<td>95</td>
<td>15</td>
<td>172</td>
<td>121</td>
<td>58</td>
</tr>
<tr>
<td>ObjectDoesNotExist</td>
<td>105</td>
<td>17</td>
<td>213</td>
<td>142</td>
<td>64</td>
</tr>
<tr>
<td>NameError</td>
<td>95</td>
<td>19</td>
<td>197</td>
<td>124</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 2. Binary operators.

If a function has no binary operators, it is discarded. Otherwise, it is used to generate a bug-free example, and a single buggy example as follows: one of the operators is chosen pseudorandomly (as described above), and a different operator chosen to replace it in the same row of Table 2. So, for instance, a buggy example would only swap == with is, but not with not in, which would not type-check if we performed static type inference on Python.

We take appropriate care to ensure the code parses after a bug is introduced. For instance, if we swap the operator in the expression 1==2 with is, we ensure that there is space between the tokens (i.e., 1 is 2 rather than the incorrect 1is2), even though it was not needed before.

B.2.4. SWAPPED OPERAND

Since this task targets swapping the arguments of binary operators, we only consider non-commutative operators from Table 2.

Functions without eligible operators are discarded, and the choice of the operator to mutate in a function, as well as the choice of buggy operator to use, are done as above, but limiting choices only to non-commutative operators.

To avoid complications due to format changes, we only consider expressions that fit in a single line (in contrast to the Wrong Binary Operator Classification task). We also do not consider expressions that look the same after swapping (e.g., a - a).

B.2.5. FUNCTION-DOCSTRING MISMATCH

In Python, a function docstring is a string literal that directly follows the function signature and precedes the main function body. Whereas in other common programming languages, the function documentation is a comment, in Python it is an actual, semantically meaningful string literal.

We discard functions that have no docstring from this dataset, or functions that have an empty docstring. We split the rest into the function definition without the docstring, and the docstring summary (i.e., the first line of text from its docstring), discarding the rest of the docstring.

We create bug-free examples by pairing a function with its own docstring summary.

To create buggy examples, we pair every function with another function’s docstring summary, according to a global
Supplementary Material for “Learning and Evaluating Contextual Embedding of Source Code”

pseudorandom permutation of all functions: for all $i$, we combine the $i$-th function (without its docstring) with the $P_i$-th function’s docstring summary, where $P$ is a pseudorandom permutation, under a given seed. We discard pairings in which $i == P[i]$, but for the seeds we chose, no such pathological permuted pairings occurred.

B.2.6. Exception Type

Note that, unlike all other tasks, this task has no notion of buggy or bug-free examples.

We discard functions that do not have any except clauses in them.

For the rest, we collect all locations holding exception types within except clauses, and choose one of those locations to query the model for classification. Note that a single except clause may hold a comma-separated list of exception types, and the same type may appear in multiple locations within a function. Once a location is chosen, we replace it with a special \_HOLE\_ token, and create a classification example that pairs the function (with the masked exception location) with the true label (the removed exception type).

The count of examples per exception type can be found in Table 1.

B.2.7. Variable Misuse Localization and Repair

The dataset for this task is identical to that for the Variable-Misuse Classification task (Section B.2.2). However, unlike the classification task, examples contain more features relevant to localization and repair. Specifically, in addition to the token sequence describing the program, we also extract a number of boolean input masks:

- A candidates mask, which marks as True all tokens holding a variable, which can therefore be either the location of a bug, or the location of a repair. The first position is always a candidate, since it may be used to indicate a bug-free program.

- A targets mask, which marks as True all tokens holding the correct variable, for buggy examples. Note that the correct variable may appear in multiple locations in a function, therefore this mask may have multiple True positions. Bug-free examples have an all-False targets mask.

- An error-location mask, which marks as True the location where the bug occurs (for buggy examples) or the first location (for bug-free examples).

All the masks mark as True some of the locations that hold variables. Because many variables are subtokenized into multiple tokens, if a variable is to be marked as True in the corresponding mask, we only mark as True its first subtoken, keeping trailing subtokens as False.

C. Attention Visualizations

In this section, we provide sample code snippets used to test the different classification tasks. Further, Figures 1–5 show visualizations of the attention matrix of the last layer of the fine-tuned CuBERT model (Coenen et al., 2019) for the code snippets. In the visualization, the Y-axis shows the query tokens and X-axis shows the tokens being attended to. The attention weight between a pair of tokens is the maximum of the weights assigned by the multi-head attention mechanism. The color changes from dark to light as weight changes from 0 to 1.

References

Supplementary Material for “Learning and Evaluating Contextual Embedding of Source Code”

def on_resize(self, event):
  event.apply_zoom()

Figure 1. Variable Misuse Example. In the code snippet, ‘event.apply_zoom’ should actually be ‘self.apply_zoom’. The CuBERT variable-misuse model correctly predicts that the code has an error. As seen from the attention map, the query tokens are attending to the second occurrence of the ‘event’ token in the snippet, which corresponds to the incorrect variable usage.
**Figure 2.** Wrong Operator Example. In this code snippet, ‘other is not self’ should actually be ‘other < self’. The CuBERT wrong-binary-operator model correctly predicts that the code snippet has an error. As seen from the attention map, the query tokens are all attending to the incorrect operator ‘is’.
Figure 3. Swapped Operand Example. In this code snippet, the return statement should be `model in cls._registry`. The swapped-operand model correctly predicts that the code snippet has an error. The query tokens are paying substantial attention to `in` and the second occurrence of `model` in the snippet.
Figure 4. Function Docstring Example. The CuBERT function-docstring model correctly predicts that the docstring is wrong for this code snippet. Note that most of the query tokens are attending to the tokens in the docstring.
Figure 5. Exception Classification Example. For this code snippet, the CuBERT exception-classification model correctly predicts `__HOLE__` as `OSError`. The model’s attention matrix also shows that `__HOLE__` is attending to `subprocess`, which is indicative of an OS-related error.

```python
try:
    subprocess.call(hook_value)
    return jsonify(success=True), 200
except __HOLE__ as e:
    return jsonify(success=False, error=str(e)), 400
```