A Simple, Yet Effective Approach to Finding Biases in Code Generation

Spyridon Mouselinos
University of Warsaw
s.mouselinos@uw.edu.pl

Henryk Michalewski
University of Warsaw, Google
henrykm@google.com

Mateusz Malinowski
DeepMind
mateuszm@deepmind.com

Abstract

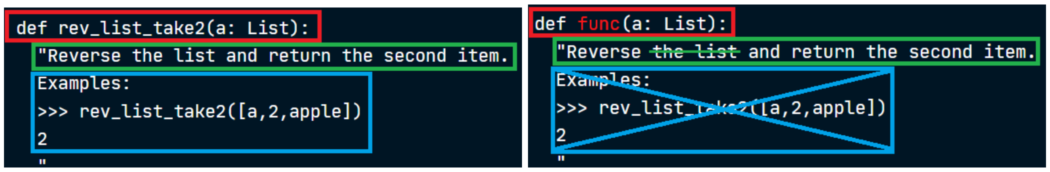
Recently, scores of high-performing code generation systems have surfaced. As has become a popular choice in many domains, code generation is often approached using large language models as a core, trained under the masked or causal language modeling schema. This work shows that current code generation systems exhibit biases inherited from large language model backbones, which might leak into generated code under specific circumstances.

To investigate the effect, we propose a framework that automatically removes hints and exposes various biases that these code generation models use. We apply our framework to three coding challenges and test it across top-performing coding generation models. Our experiments reveal biases towards specific prompt structure and exploitation of keywords during code generation. Finally, we demonstrate how to use our framework as a data transformation technique, which we find a promising direction toward more robust code generation.

1 Introduction

Large language models (LLM) have recently demonstrated their ability to generate code (Li et al., 2022; Wang & Komatsuzaki, 2021; Black et al., 2021; Brown et al., 2020; Wang et al., 2021) or solve challenging programming/math tasks on par with human coders (Li et al., 2022; Lewkowycz et al., 2022b; Chowdhery et al., 2022a); these models are trained with the data-driven paradigm. On the other hand, an increasing body of work also questions whether the data-driven approach leads to acquiring reasoning skills (Piekos et al., 2021; Zhang et al., 2022; Mouselinos et al., 2022), showing that if left alone, it might not be sufficient for achieving truly human-level performance on tasks such as logical or visual reasoning. In many studied cases, models still rely on various hints in their "reasoning" process. This work extends the results above, i.e., the lack of reasoning capabilities, to the code generation domain. More specifically, we devise a framework that automatically identifies cues that a code generation model might exploit. Changes or removal of those cues stands as a reasoning test towards the generational capabilities of the model at hand. Imagine the following scenario: a code generation model is presented with the prompt "*def add_two_nums(a,b): ...*", and the directive "*Generate a function that adds two numbers and returns their sum*". The expected answer should resemble something like, "*return a + b*." Now, if we decide to rename the function name to a generic "*func*" token: "*def func(a,b): ...*" but maintain the same directive, any inability of the model to generate a correct solution can be attributed to cues related to the function name itself; indicating that the tested model "copied-pasted" a solution from training data and it has not learned how to program.

We presume that the reasoning process of code generation models should remain invariant under changes that still provide enough context or pose little if any, additional challenge to a human coder. To this end, we propose an automatic and model-agnostic framework that modifies the following: (1) function names, (2) keywords in a problem specification, or (3) examples provided in the problem prompt. We refer to these three parts as Blocks-Of-Influence; see Figure 1. Each block contributes



```
def rev_list_take2(a: List):
    "Reverse the list and return the second item.
    Examples:
    >>> rev_list_take2([a,2,apple])
    2
    "
```

(a)

```
def func(a: List):
    "Reverse ~~the list~~ and return the second item.
    Examples:
    >>> rev_list_take2([a,2,apple])
    2
    "
```

(b)

Figure 1: **Left - Blocks Of Influence:** Function / argument names (red), problem specification (green), and examples (blue). **Right - Examples:** We demonstrate three possible transformations, one for each block: Swap the function name with ”func”, remove keywords, and remove examples from the red, green and blue block, respectively. In our framework, blocks can be targeted both individually and in a joint fashion.

partially to the context needed for correct completion. We show that minor modifications of these blocks are sufficient to “fool” LLM-based code generation methods. Our results reveal that keyword preference and memorization bias can be identified across multiple models and coding challenges. Note that our modifications maintain the same semantics. We refer the reader to the supplementary material (SectionA.4) for details.

Contributions. The main contributions of our work can be summarized in three points.
First, we propose a novel automated framework that identifies possible biases in a coding style dataset. Our framework removes subtle hints, introducing minimal changes in the form of keyword replacement or partial code-block omission, ultimately acting as an adversarial test. Suggested changes resemble typical errors of a human coder and challenge the generational capabilities of the model under test. Since the framework operates on a data level, it is agnostic to the model’s structure and internal workings. Finally, the framework can be easily adjusted to any input format or programming language.
Second, we introduce the ”*Blocks of Influence*” concept. We suggest that every instance of a typical coding challenge can be analyzed into three parts (blocks). Each part is correlated with a different method of hinting and is used as a target of our transformations. A model’s reasoning process is informed by all three blocks, making them perfect analyzing tools for cases of failing code generation.
Third, we explore new ways of achieving robustness during code generation. In Section 6.2, we study the effects of adversarial training against our proposed perturbations and the benefits of including examples with longer descriptions during finetuning. Our results show that combining these techniques leads to higher model resilience.

2 Related Work

Our approach is inspired by works of various research directions, which we briefly describe here.
Solving coding and math challenges. The emergent abilities of large language models to generate, summarize and translate textual information, have recently sparked interest in their aptitude for math, logic, and programming challenges. Tasks such as code-completion (Chen et al., 2021; Shin et al., 2019; Hendrycks et al., 2021a; Li et al., 2022), code summarization and code translation (Lu et al., 2021) have been proposed, with models constantly progressing towards near-human performance, as recently shown in Li et al. (2022). Similarly, Hendrycks et al. (2021b); Saxton et al. (2019); Ling et al. (2017); Amini et al. (2019) have proposed tests measuring a model’s ability to perform math and logic, ranging from trivial school problems to competition-grade challenges. Decoder-only works such as Brown et al. (2020); Chen et al. (2021), trained with causal language modeling, have managed to achieve impressive results in code generation tasks over multiple programming languages. Feng et al. (2020); Kanade et al. (2020) utilized the encoder-only architecture and provided strong pre-training baselines that excel in code understanding, and can be further finetuned for tasks such as code repair and code retrieval. Fried et al. (2022) created the first generative model to perform infilling, with the use of a novel masking objective that leverages the benefits of bidirectional context in an autoregressive setting. Open-source projects such as Black et al. (2021); Wang & Komatsuzaki (2021); Tunstall et al. (2022a); Mitchell et al. have demonstrated the value of good pre-processing, deduplication, and careful hyperparameter tuning, achieving impressive results, especially in lightweight architectures. Finally, massive-scale models such as Chowdhery et al. (2022b) demonstrated breakthrough capabilities in language, reasoning,

and code tasks achieving state of the art performance in multiple domains simultaneously. In a follow-up work, Lewkowycz et al. (2022a) incorporated advanced prompting and evaluation techniques to better solve mathematical questions.

Bias in large language models. There is some existing research towards discovering biases in large language models. In the area of ethics, Wallace et al. (2019) shows that generative models can be conditioned to produce toxic content, with the use of nonsense, adversarial prefixes. Similarly Liang et al. (2021) suggest that models might adopt biases and social stereotypes found among their training data and provide ways to apply fairness during generation. Countermeasures in the form of filtering or calibration, have been proposed by Zhao et al. (2021); Liu et al. (2022), claiming that sanitized zero shot examples contribute to mitigating biases during generation.

Probing reasoning through biases. There have been notable attempts to systemize intelligence and reasoning as concepts (Legg, 2008; Chollet, 2019), yet a few recent works try to approach reasoning, through the analysis of failure modes in deep learning models. Glockner et al. (2018) suggest that natural language inference systems can be easily fooled with a single hypernym / hyponym word swap. Similarly, Lin et al. (2020) prove that numerical commonsense reasoning in LLMs is heavily biased by adjectives describing the object of interest. Concerns against the current data-driven methods have been expressed by Razeghi et al. (2022), pointing out that LLMs are more accurate on mathematical challenges that involve terms significantly more frequently in their pre-training dataset. Piekos et al. (2021) claim that LLMs can answer math and logic questions without an understanding of the rationale behind them. They introduce a novel task of correctly classifying the order of reasoning steps and achieve better outcomes than purely data-driven baselines.

Adversarial methods and Language Processing. NLP community developed excellent methods to prepare adversarial tasks, including the TextAttack framework Morris et al. (2020) and sophisticated techniques to elicit adversarial examples from humans, as in Talmor et al. (2022), though our work seems to be the first focused on the disciplined construction of adversarial examples for code.

3 Benchmarks

Name	#Problems	#Tests per Problem	Avg. desc. length	Avg. keywords
HumanEval (Chen et al., 2021)	164	8	449	4
MBPP (Austin et al., 2021)	1000	3	235	4
DMCC (Train / Python3) (Li et al., 2022)	8139	85	1480	9

Table 1: Datasets used in experiments. We present the number of problems, number of tests per problem, average length of the challenge description and average distinct keywords identified by our framework.

In this section, we describe the datasets used in our experiments; We employed a widely used, complex coding challenge (HE), a simpler one with everyday python problems (MBPP), and finally, a dataset that uses long and analytic descriptions instead of docstrings (DMCC).

HumanEval (HE). This is a human-curated problem-solving dataset described in Chen et al. (2021). It consists of 164 original programming challenges assessing language comprehension, algorithms, and simple mathematics, with some comparable to simple software interview questions. Each problem is presented as an incomplete function, accompanied by a docstring. The docstring contains the task, directives for its completion, and a few examples. For each task, we are provided with a set of unit tests. A task is considered solved when all unit tests are passed. This requires both a syntactically and functionally correct Python program to be generated. The dataset contains no training splits, expecting models to solve it in a ”zero-shot” fashion.

Mostly Basic Python Problems (MBPP). The dataset was introduced in Austin et al. (2021). It contains 974 short Python functions designed to be solved by entry-level programmers. Contrary to HumanEval, the task is given through a text description rather than a docstring. Additionally, there are no input-output examples in the prompt. Test cases for each problem are provided, assessing the functional correctness of the model's completion. This dataset consists of a mixture of crowd-sourced and hand-crafted questions. MBPP challenges models to perform tasks of imperative control flow, requiring loops and conditionals in its solutions. Although there is no official train/test split on the dataset, its creators have experimented with fine-tuning experiments on it.

Deepmind Code Challenges (DMCC). This refers to the highly challenging dataset proposed by Li et al. (2022). The dataset includes problems, solutions and test cases scraped from the Codeforces platform (Mirzayanov, 2020), along with existing public competitive programming datasets scraped from from Description2Code (Caballero, 2016), and CodeNet (Puri et al., 2021). The dataset contains problems from multiple programming languages and training, validation, and test splits. For our purpose, we focused on python3 programs in all three splits. DMCC format is slightly different from the previous datasets, with each problem being presented in three parts. The first part is a long description of the problem alongside examples and some notes for its solution. The second part consists of multiple solutions, both correct and incorrect, that correspond to the particular problem instance. Finally, unit tests are provided that can be classified into public tests, visible to a human programmer. Moreover, hidden tests are used during the final evaluation of the submitted code.

4 Evaluation

Models. In our experimental setup, we test four models representing a different approach to code generation. The CodeParrot (Tunstall et al., 2022a), has a typical GPT-2 (Radford et al., 2019) architecture, and exhibits very good performance given its size. We also challenge the Incoder (Fried et al., 2022) model, which is trained under a novel bi-directional causal objective, being able to handle context more efficiently than its causal counterparts. Bloom (Mitchell et al.) is a recent model that excels in multiple domains. We are interested in testing its coding capabilities even though that was not the main goal of the model; however, due to its scale, the model can generate code. Finally, we have the powerful Codex model, able to tackle most of the proposed coding challenges in the HumanEval and MBPP datasets. A list of the tested models with their sizes, as well as KeyBert (Grootendorst, 2020) that is used in our framework, can be found in Table 2.
Finetuning and evaluation. During our experiments, we ensured that every coding instance was properly formatted and contained all the necessary parts (*Blocks of Influence*) for our tests. Specifically, in the case of DMCC, a snake-case equivalent of the problem name was generated. This enabled us to apply the proposed *Anonymize* transformation to its examples during our ablation study in Section 6.2. Finally, problems without prompt examples are given two (randomly generated) valid ones in all datasets. Regarding code execution, we modify the existing evaluation suite that the authors of HumanEval provide to support input/output from external files and add logging functionalities to it. The experiments are done in a zero-shot fashion for all datasets and models.
Performance metrics. Whereas the syntactic correctness of a Python program can be easily evaluated, functional correctness is currently studied under the pass@k metric, introduced in Kulal et al. (2019). This metric serves as an estimator of the real model generational capabilities under a specific budget. In Chen et al. (2021), authors propose an updated estimator formulation, empirically proving that for each budget k, ten times more generations should be used for an unbiased estimate. Finally, the modification of 10@k is introduced in Li et al. (2022), testing only the ten best solutions out of k attempts. Note here that k is tested at magnitudes of 10^4-10^6 instead of the typical 10^0-10^2 used in most code generation works. This setup's purpose is to demonstrate their method's filtering effectiveness. Thus, to avoid confusion, our pass@k metric is calculated at exactly k attempts. An average of five runs with different seeds is presented for all experiments. Sampling temperatures are 0.2 / 0.8 for pass@1 / pass@100 respectively, which is the optimal values across the tested models.

Model Name	Usage Type	Sizes Used
KeyBert (Grootendorst, 2020)	model access	2M
Codeparrot (Tunstall et al., 2022a)	model access	110M / 350M*/ 1.5B
Incoder (Fried et al., 2022)	model access	1.6B / 6B
Bloom (Mitchell et al.)	model access & API	560M* / 1.7B / 176B
Codex (v1 / v2) (Chen et al., 2021)	API	~175B

Table 2: Models used in experiments. Entries marked with * refer to finetuned or trained from scratch models. The size of Codex models is an estimate, since its not officially disclosed

5 Method

5.1 Blocks of Influence

We propose that each coding challenge should be treated as a sequence of three, distinct but complementary blocks, rather than a single, homogeneous input. We refer to them as "*Blocks of Influence*", and correlate each with different hinting methods during code generation. Taking Figure 1 as an example, the model is challenged to complete a function that reverses a list, and then returns its second item. All the datasets that we use have all three blocks of influence; therefore changes made in one such block will still keep enough contextual information in the remaining blocks.

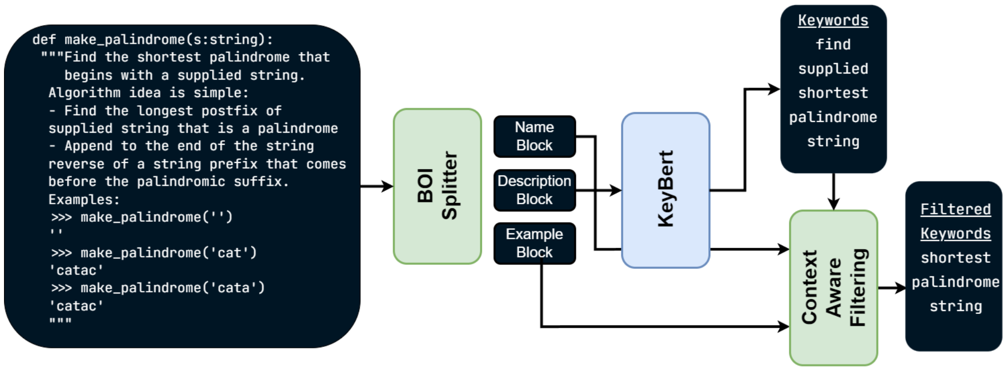
Name Block. The first block of influence, marked in red, informs the model about the function name and the names and expected types of the input arguments. Let us assume that initially, a model generates correct solutions to a problem. However, the model fails when we rename the function name to something unrelated to the task e.g *"fun"*. This failure mode indicates that neither the problem description was understood nor the model could extract a reasoning pattern out of the given examples. We associate such cases with memorization bias, where the model relies heavily on replicating snippets from its training dataset with the same or similar function name.

Description Block. The description of the problem, stands as the second block, marked in green. The model is expected to form a solution strategy for the presented task by utilizing its natural language understanding capabilities. In this block, we observe that removing specific keywords from the problem description can lead to catastrophic results in model performance. The most interesting cases involve keywords the lack of which, does not degrade the context quality of the problem description. For example in Figure 1, the removal of the word pair "the list" creates a description that is still well understandable by a human coder if the remaining blocks are unaltered. The model should be able to deduct it from the existence of the word "list" in the function name and the list type of input in the example given. We associate such effects with limited abilities of the model to truly understand the given task, and instead rely on inherent preference bias - frequency of token(s) in training set - to fill the missing context.

Example Block. As the final block, we consider examples or notes given after the problem description. These act as demonstrations, guiding the model to specific reasoning patterns. Let's consider a scenario where some models cannot generate correct code when examples are absent. Arguably, the model has exhausted its capacity to understand the task and given inputs. In this failure mode, the provided examples act as a "reasoning tie-breaker" between proposed solutions the model can generate. This effect is especially interesting in the case of composite tasks. The model can easily solve each task (e.g Figure 1: reverse a list / return the second item of an iterable). However, the request of combining those tasks seems odd enough, that the model needs extra examples to internally filter out faulty strategies. We associate such effects with poor generalization.

5.2 Framework

Transformations in blocks enable us to repurpose a coding challenge into a test for reasoning failures and biases. In our method, we extend our tests to modify not only a single but two blocks at a time so that additional combinations of hints/cues can be explored. Here, we avoid performing changes in all the *Blocks of Influence* at the same time; a model stripped of any necessary information cannot generate a proper solution. Our framework is composed of three modules that operate in two steps.



```
def make_palindrome(s:string):
  """Find the shortest palindrome that
     begins with a supplied string.
   Algorithm idea is simple:
   - Find the longest postfix of
   supplied string that is a palindrome
   - Append to the end of the string
   reverse of a string prefix that comes
   before the palindromic suffix.
   Examples:
    >>> make_palindrome('')
   ''
    >>> make_palindrome('cat')
   'catac'
    >>> make_palindrome('cata')
   'catac'
   """
```

Figure 2: Keyword extraction step: The coding challenge is initially processed by the *Blocks of Influence* splitting module (**BOI splitter**). Keybert, then recieves the *Description Block* and suggests possible hinting keywords. Those are subsequently passed through the context-aware filtering module, leaving eligible keywords finally ready to be used in the transformation step.

The first step involves splitting the code instance into the *Blocks of Influence*. For this, we utilize a Regex-based module that fully decomposes the code snippet. This is followed by identifying possible hinting keywords in the *Description Block*. Ideally, we are interested in unigrams or bigrams that provide excess information towards completing the coding task. Secondary targets include keywords that specifically describe a niche task and can act as memorization cues (e.g., if the task of finding a palindrome is nicknamed "tacocat", it is possible that functions with that name exist in the training set of the model. Such keywords should be removed so that no verbatim code is generated). We experimented with three different methods for keyword extraction: Keybert (Grootendorst, 2020) which is an LM-based approach, RAKE (Rose et al., 2010), a text-mining algorithm, and finally, a combination of a Named Entity Recognition module and hand-crafted dictionary. We opted for the KeyBert solution, finetuned on code embeddings, since it was relatively fast, even in CPU inference, and provided the most insightful findings from the other alternatives. However, carelessly removing words from the code prompt can lead to a non-interesting drop in performance, associated with poor prompt quality rather than hinting effects. In order to focus on those effects, we introduce a special filtering stage. Keywords absent from the *Name Block* and *Example Block*, or correspond to adjectives and adverbs non-related to coding, are of no interest to us. This is achieved with a mixture of hand-crafted rules and weighted embedding similarity of each keyword with the set of words: [Python, Programming, Code] again achieved through KeyBert. Words with similarity under 0.7 are considered unrelated and thus discarded. The first step is presented in Figure 2.
As a second step, we implemented a selector that chooses between the following transformations, and modifies the coding challenge accordingly (Figure 1):
Drop one. Removes one of the provided keywords from the *Description Block*. This is repeated N times where N is the number of identified keywords. The performance drop is reported as an average of the N generated code instances.
Drop all. Removes all the provided keywords simultaneously from the *Description Block*
Drop examples. Removes all the provided examples from the *Example Block*. In the case of any notes existing alongside the examples, the experiment is repeated twice, with and without the notes, and the average performance drop is reported.
Anonymize. Replaces the function name with an arbitrary token. We use *"func"* in our experiments. Note that the function name is also replaced in the provided examples as well so no leak of information takes place.

We are also interested in combining anonymization with the other modes to provide even more challenging versions of the available transformations. For instance, the *Anonymize + Drop Examples* transformation pushes the model to form a solution strategy purely by utilizing its natural language understanding skills on the *Description Block*. It is important to note here that some combinations are deliberately omitted (e.g., *Drop all + Drop Examples*) since, as mentioned in 5.1, they reduce the input to a meaningless prompt.

6 Experiments

6.1 Results on Block of Influence transformations

The main results of our experiment are presented in tables 3 and 4 (small / large models). Despite their simple nature, our transformations cause consistent drops in performance across different model sizes on both datasets.[1] Mere anonymization causes drops of 19% on average in both Pass@1 and Pass@100 metrics, validating our claims of memorization biases. Single (*Drop One*) and full keyword removal (*Drop All*) reduce models' performance by 15% and 22% on average, suggesting their inability to deduct the missing context from *Name Block* and *Example Block*. Instead, models rely on generating arbitrary, commonly used snippets that are vaguely fit for the task. Especially interesting are the cases of *Drop Examples* and *Anonymize + Drop Examples*, with 15% and 25% average drops. Both transformations remove the information provided by the docstring examples, with the latter having the additional restriction of an anonymized function. With the *Description Block* unmodified in both cases, these transformations target the models' abilities to create solutions based on their natural language understanding. The combination of anonymization with the drop of all keywords (*Anonymize + Drop All*) seems to be the most challenging transformation overall, with drops of approximately 40%. Its primary purpose is to assess the model's capability of deducting

[1] We present a full table of results, including Codeparrot (110M) and Codex(v1) in the Appendix.

the missing context of the *Description Block* by only observing patterns in the examples. These observations suggest a clear model preference over its sources of information, with the task description being the primary one. Thus, when a model exhausts its abilities to understand the task, it tries to exploit similarities of the function name with previously seen code solutions. Simultaneously, the model's reasoning relies on the example demonstrations, which, as seen from (*Anonymize + Drop All*), are not always able to provide clear directives.

Table 3: Small Model results on Human Eval and MBPP.

	Codeparrot (1.5B)				Incoder (1.6B)				Bloom (1.7B)			
	Human Eval		MBPP		Human Eval		MBPP		Human Eval		MBPP	
Method	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Original	4.1	17.8	6.1	31.2	11.3	24.2	14.6	56.7	4.3	14.6	6.6	37.2
Drop One	3.9	13.2	4.2	26.8	10.5	22.3	11.5	45.4	3.0	12.2	2.7	27.6
Drop All	3.6	11.1	3.9	21.7	9.7	17.6	12.8	42.1	2.4	9.8	2.6	24.2
Drop Ex	3.7	14.3	5.3	27.5	11.3	22.2	14.4	43.8	3.6	12.8	3.1	29.0
Anon	3.8	12.5	4.7	23.2	9.1	21.8	11.3	45.2	3.6	11.6	3.1	27.5
Anon+Drop One	3.3	**9.5**	3.9	20.2	7.4	21.5	10.5	44.9	2.4	9.1	2.4	25.3
Anon+Drop All	2.1	**8.9**	3.9	**17.9**	**6.3**	17.5	**8.0**	41.3	1.8	**8.5**	2.0	23.1
Anon+Drop Ex	3.7	11.8	4.6	22.8	8.7	21.3	11.2	43.5	3.4	11.6	3.0	26.7

Table 4: Large Model results on Human Eval and MBPP.

	Incoder (6B)				Codex (v2)				Bloom (176B)			
	Human Eval		MBPP		Human Eval		MBPP		Human Eval		MBPP	
Method	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Original	15.2	47.0	19.4	65.1	49.4	91.4	60.1	86.3	16.4	57.2	20.8	62.4
Drop One	12.1	35.3	18.9	52.6	36.0	86.2	56.0	79.2	12.8	48.6	15.8	51.4
Drop All	10.2	28.2	15.6	47.0	37.1	73.7	52.1	69.5	11.5	40.2	14.2	44.4
Drop Ex	12.7	29.5	17.4	50.3	41.4	81.0	48.8	70.7	15.2	43.3	15.8	50.1
Anon	11.6	32.9	14.8	50.7	44.5	90.4	57.9	81.7	14.0	48.3	15.1	51.2
Anon+Drop One	**8.1**	30.6	13.5	46.7	**29.8**	74.4	51.2	69.5	**12.8**	41.9	13.6	46.8
Anon+Drop All	**7.5**	**25.2**	**11.2**	**38.9**	**24.2**	**68.7**	**47.2**	**63.8**	**10.3**	**36.8**	**12.6**	**38.4**
Anon+Drop Ex	11.2	28.1	14.5	50.2	34.1	**72.5**	42.6	70.5	14.0	39.8	14.3	47.8

6.2 IMPROVING ROBUSTNESS

Inspired by the field of adversarial training, we decided to investigate the effects of using our framework transformations as training augmentations. To this end, we apply our framework to examples of the MBPP challenge and use them as a finetuning dataset for three different Codeparrot models (110M / 350M / 1.5B). We use HumanEval as our test dataset, which bears no overlap with the MBPP. In this way, our models have not seen examples of the test set during their training or finetuning steps. In Table **??**. We compare the results of our models before and after finetuning. Models benefit from the introduction of augmented examples and partially recover from modes of failure caused by our framework. The larger the model, the more its defensive abilities increase. We believe this effect is closely related to large language models' scaling reasoning capabilities together with their parameter size. The need to rely on hints can be attributed to low data quality or lack of task-specific inductive biases. However, the capacity to properly understand coding tasks is undoubtedly there. In order to robustify the code generation abilities of a model, we suggest to expose them to challenges that push their deductive and reasoning abilities.

	Codeparrot - 110M				Codeparrot - 350M				Codeparrot - 1.5B			
	Pass@1 (T=0.2)		Pass@100 (T=0.8)		Pass@1 (T=0.2)		Pass@100 (T=0.8)		Pass@1 (T=0.2)		Pass@100 (T=0.8)	
Method	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
Drop One	3.0	**3.6**	9.7	**10.3**	3.0	**3.6**	11.5	**12.1**	3.6	**4.2**	13.4	**14.0**
Drop All	2.4	2.4	7.3	**7.9**	3.0	3.0	9.7	9.7	3.0	**3.6**	11.6	**12.8**
Drop Ex	3.6	3.6	9.7	**10.3**	3.6	3.6	12.8	12.8	3.6	3.6	14.6	**15.2**
Anon	3.0	**3.6**	8.5	**9.1**	3.6	3.6	11.5	**12.1**	3.6	3.6	12.2	**14.0**
Anon+Drop One	3.0	**3.6**	7.3	**7.9**	3.0	**3.6**	8.5	**9.7**	3.0	**3.6**	9.7	**10.9**
Anon+Drop All	1.8	**2.0**	6.7	6.7	1.8	**2.4**	8.5	8.5	2.4	2.4	8.5	**9.7**
Anon+Drop Ex	3.0	3.0	8.5	**9.1**	3.6	3.6	11.5	**12.8**	3.6	3.6	12.2	**14.0**

Table 5: Results of fine-tuning Codeparrot Models on the modified MBPP dataset. Each model is tested again on Human Eval, showing slight recovery against the attacks.

When causally training on coding datasets, models condition on multiple functions and declarations in the same file. The input is a conglomerate of rapidly changing contexts, with each function or class being a self-contained entity. Subsequently, a model is accustomed to localizing its focus when trained on such data. As an extension to our previous experiment, in Table **??**, we measure the effects of using a long description dataset, DMCC, as a finetuning target. By training on long descriptions of natural language, we aim to promote the context-deducting skills of the model under test. A model able to widen its focus can avoid distractions caused by missing keywords since it will not rely heavily on internal biases but on context understanding. We choose Bloom as the model under test since it was not explicitly tuned for code generation but rather general language understanding. In Table **??**, we present results of finetuning Bloom on MBPP, modified by our framework. We observe similar performance improvements as in Table **??**. We experiment again, this time combining both MBPP and DMCC examples. We show that incorporating examples of more extended context leads to even better performance against transformations targeting the *Description Block* and language understanding. Similar experiments were conducted with the CodeParrot variants, but were unfruitful. We attribute this to the restricted focus in terms of training data (CodeParrot is trained extensively on Python3 codebases), and architectural differences between the models. We believe that the merging benefits of our two proposed setups can serve as an interesting direction towards model resilience in code generation scenarios.

	Pass@1 (T=0.2)			Pass@100 (T=0.8)		
Method	Before	+MBPP	+DMCC	Before	+MBPP	+DMCC
Drop One	3.0	3.6	3.6	10.3	10.9	10.9
Drop All	2.4	2.4	**3.0**	9.1	9.1	**9.7**
Drop Ex	3.0	3.0	3.0	11.0	11.5	11.5
Anon	2.4	3.0	**3.6**	10.3	10.9	**11.5**
Anon+Drop One	1.8	2.4	2.4	7.9	9.1	**9.7**
Anon+Drop All	1.8	1.8	**2.4**	7.0	7.0	**8.5**
Anon+Drop Ex	2.4	3.0	3.0	9.7	10.3	**11.5**

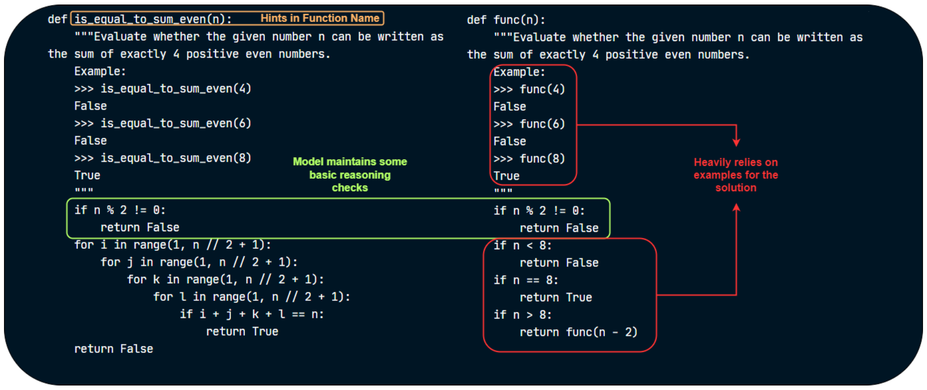
Table 6: Ablation study of combined finetuning methods. We present Bloom (560M) performance on HumanEval before and after the modified MBPP and long-description DMCC examples.

7 Conclusions

We present a simple approach to isolate subtle cues and benchmark the reasoning capabilities of code generation models through input-level transformations. Our method treats code examples as a combination of three blocks, each providing different cues to the model. We show that minor block changes can lead models to failure, signifying the existence of biases. Our framework can automatically identify and remove keywords in the input, responsible for indirect hinting. We show that popular models with solid results on challenging coding challenges are susceptible to our tests, with their performance degrading noticeably. Moreover, we studied the effects of utilizing our proposed transformations during the fine-tuning of a model. Models can benefit from our proposed changes, with the effect proportional to their parameter size. We believe that, despite their success, code generation systems with large language models as their backbone inherit some of their biases and modes of failure. Training on structured and well-documented code, combined with techniques like the ones we propose, is hopefully a promising direction towards robust code generation. Although an ideal fit for competition-style challenges, our method can be extended to support less formatted high-quality codebases (e.g. GitHub repositories). Large files can be broken down into individual functions/classes, each further analyzed into Blocks of Influence. In such codebases, function names should be closely relevant to their purpose. The existence of meaningful docstrings is crucial, the absence of which promotes more memorization and biases as we exhibited. Moreover, the input/output checks contained in function unit tests can be repurposed as function examples. Keywords can be chosen similarly, with the context being co-informed by both local and larger scopes.

Acknowledgments

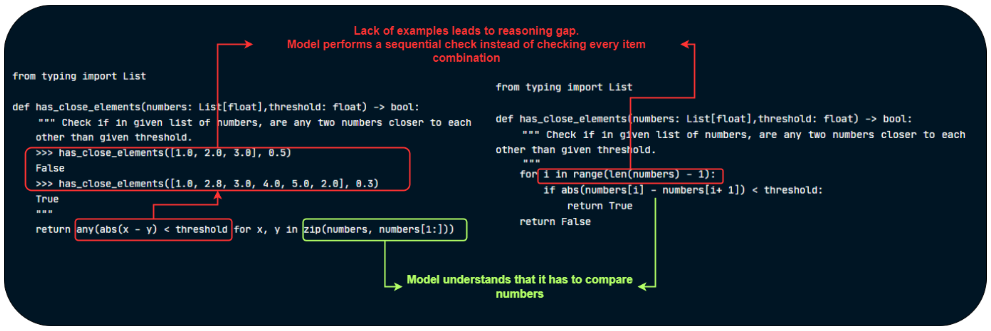
All experiments were performed using the Entropy cluster funded by NVIDIA, Intel, the Polish National Science Center grant UMO-2017/26/E/ST6/00622 and ERC Starting Grant TOTAL. The work of Spyridon Mouselinos and Henryk Michalewski was supported by the Polish National Science Center grant UMO-2018/29/B/ST6/02959. We also thank Taylan Cemgil and Ethan Dyer for their valuable comments on the early draft of the work.


Hints in Function Name
Model maintains some basic reasoning checks
Heavily relies on examples for the solution

```
def is_equal_to_sum_even(n):
    """Evaluate whether the given number n can be written as
the sum of exactly 4 positive even numbers.
    Example:
    >>> is_equal_to_sum_even(4)
    False
    >>> is_equal_to_sum_even(6)
    False
    >>> is_equal_to_sum_even(8)
    True
    """
    if n % 2 != 0:
        return False
    for i in range(1, n // 2 + 1):
        for j in range(1, n // 2 + 1):
            for k in range(1, n // 2 + 1):
                for l in range(1, n // 2 + 1):
                    if i + j + k + l == n:
                        return True
    return False
```

```
def func(n):
    """Evaluate whether the given number n can be written as
the sum of exactly 4 positive even numbers.
    Example:
    >>> func(4)
    False
    >>> func(6)
    False
    >>> func(8)
    True
    """
    if n % 2 != 0:
        return False
    if n < 8:
        return False
    if n == 8:
        return True
    if n > 8:
        return func(n - 2)
```

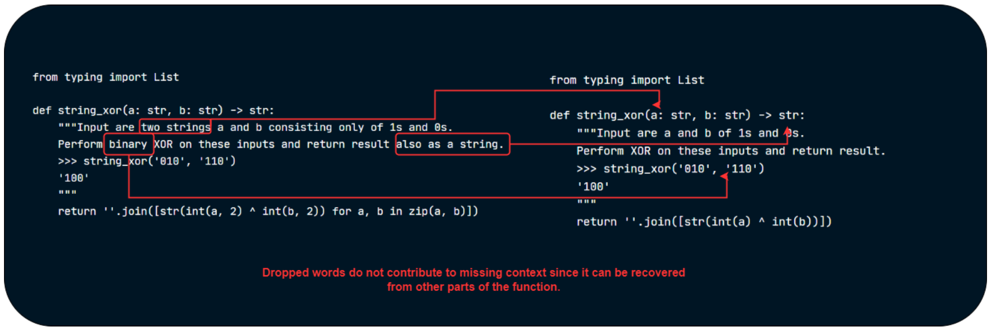
(a) *Anonymize* in Codex (v2): In its completion, the model generates some basic reasoning checks (If the sum is not divisible by two, return False) but relies purely on the examples given for the final solution.


Lack of examples leads to reasoning gap.
Model performs a sequential check instead of checking every item combination
Model understands that it has to compare numbers

```
from typing import List

def has_close_elements(numbers: List[float],threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each
    other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    return any(abs(x - y) < threshold for x, y in zip(numbers, numbers[1:]))
```

```
from typing import List

def has_close_elements(numbers: List[float],threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each
other than given threshold.
    """
    for i in range(len(numbers) - 1):
        if abs(numbers[i] - numbers[i+ 1]) < threshold:
            return True
    return False
```

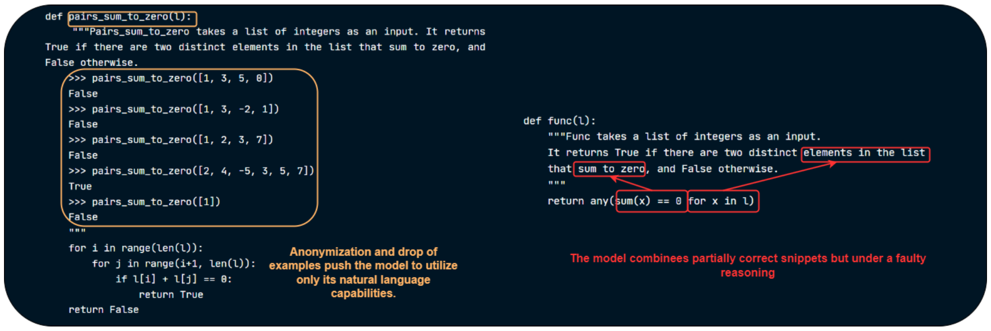
(b) *Example drop* in Codex (v1): The model exhibits signs of task comprehension, by comparing pairs of differences with the threshold variable. The presence of examples is crucial however, since only then the model proceeds to correctly check each item combination instead of a sequential check.


Dropped words do not contribute to missing context since it can be recovered from other parts of the function.

```
from typing import List

def string_xor(a: str, b: str) -> str:
    """Input are two strings a and b consisting only of 1s and 0s.
    Perform binary XOR on these inputs and return result also as a string.
    >>> string_xor('010', '110')
    '100'
    """
    return ''.join([str(int(a, 2) ^ int(b, 2)) for a, b in zip(a, b)])
```

```
from typing import List

def string_xor(a: str, b: str) -> str:
    """Input are a and b of 1s and 0s.
    Perform XOR on these inputs and return result.
    >>> string_xor('010', '110')
    '100'
    """
    return ''.join([str(int(a) ^ int(b))])
```

(c) *Drop All* in Bloom (175B): After the removal of the keywords, the context of the task remains intact: The *two strings* keyword can be assumed by observing the function arguments, and the *binary / string* keywords by the examples and return type signature of the function. Nevertheless, the model fails to generate a correct solution.


Anonymization and drop of examples push the model to utilize only its natural language capabilities.
The model combinees partially correct snippets but under a faulty reasoning

```
def pairs_sum_to_zero(l):
    """Pairs_sum_to_zero takes a list of integers as an input. It returns
True if there are two distinct elements in the list that sum to zero, and
False otherwise.
    >>> pairs_sum_to_zero([1, 3, 5, 0])
    False
    >>> pairs_sum_to_zero([1, 3, -2, 1])
    False
    >>> pairs_sum_to_zero([1, 2, 3, 7])
    False
    >>> pairs_sum_to_zero([2, 4, -5, 3, 5, 7])
    True
    >>> pairs_sum_to_zero([1])
    False
    """
    for i in range(len(l)):
        for j in range(i+1, len(l)):
            if l[i] + l[j] == 0:
                return True
    return False
```

```
def func(l):
    """Func takes a list of integers as an input.
    It returns True if there are two distinct elements in the list
    that sum to zero, and False otherwise.
    """
    return any(sum(x) == 0 for x in l)
```

(d) *Anonymize + Drop Examples* in Incoder 6B: Using only natural language understanding of the problem description, the model creates partially informed subparts that are not combined correctly to solve the final task, signifying that hints from the function name / examples are used in the original solution.

References

Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *CoRR*, abs/1905.13319, 2019. URL `http://arxiv.org/abs/1905.13319`.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL `https://arxiv.org/abs/2108.07732`.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow. March 2021. doi: 10.5281/zenodo.5297715. URL `https://doi.org/10.5281/zenodo.5297715`. If you use this software, please cite it using these metadata.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 33: 1877–1901, 2020. URL `https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

E. Caballero. Description2code dataset, 8 2016. 2016. URL `https://github.com/ethancaballero/description2code`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022a.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022b.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. pp. 1536–1547, November 2020. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139`.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Max Glockner, Vered Shwartz, and Yoav Goldberg. Breaking NLI systems with sentences that require simple lexical inferences. pp. 650–655, July 2018. doi: 10.18653/v1/P18-2103. URL `https://aclanthology.org/P18-2103`.

Maarten Grootendorst. Keybert: Minimal keyword extraction with bert. 2020. doi: 10.5281/zenodo.4461265. URL `https://doi.org/10.5281/zenodo.4461265`.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021a.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021b.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. 2020. URL `https://openreview.net/forum?id=rygoURNYvS`.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. 32, 2019. URL `https://proceedings.neurips.cc/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf`.

Shane Legg. *Machine super intelligence*. PhD thesis, Università della Svizzera italiana, 2008.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. 2022a. doi: 10.48550/ARXIV.2206.14858. URL `https://arxiv.org/abs/2206.14858`.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. 2022. doi: 10.48550/ARXIV.2203.07814. URL `https://arxiv.org/abs/2203.07814`.

Paul Pu Liang, Chiyu Wu, Louis-Philippe Morency, and Ruslan Salakhutdinov. Towards understanding and mitigating social biases in language models. 2021. doi: 10.48550/ARXIV.2106.13219. URL `https://arxiv.org/abs/2106.13219`.

Bill Yuchen Lin, Seyeon Lee, Rahul Khanna, and Xiang Ren. Birds have four legs?! NumerSense: Probing Numerical Commonsense Knowledge of Pre-Trained Language Models. pp. 6862–6868, November 2020. doi: 10.18653/v1/2020.emnlp-main.557. URL `https://aclanthology.org/2020.emnlp-main.557`.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. pp. 158–167, Jul 2017. doi: 10.18653/v1/P17-1015. URL `https://aclanthology.org/P17-1015`.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for GPT-3? pp. 100–114, May 2022. doi: 10.18653/v1/2022.deelio-1.10. URL `https://aclanthology.org/2022.deelio-1.10`.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

M. Mirzayanov. Codeforces: Results of 2020. 2020.

Margaret Mitchell, Giada Pistilli, Yacine Jernite, Ezinwanne Ozoani, Marissa Gerchick, Nazneen Rajani, Sasha Luccioni, Irene Solaiman, Maraim Masoud, Somaieh Nikpoor, Carlos Muñoz Ferrandis, Stas Bekman, Christopher Akiki, Danish Contractor, David Lansky, Angelina McMillan-Major, Tristan Thrush, Suzana Ilić, Gérard Dupont, Shayne Longpre, Manan Dey, Stella Biderman, Douwe Kiela, Emi Baylora, Teven Le Scao, Aaron Gokaslan, Julien Launay, and

Niklas Muennighoff. The world's largest open multilingual language model: Bloom. URL `https://bigscience.huggingface.co/blog/bloom`.

John X. Morris, Eli Lifland, Jin Yong Yoo, and Yanjun Qi. Textattack: A framework for adversarial attacks in natural language processing. *CoRR*, abs/2005.05909, 2020. URL `https://arxiv.org/abs/2005.05909`.

Spyridon Mouselinos, Henryk Michalewski, and Mateusz Malinowski. Measuring clevrness: Black-box testing of visual reasoning models. *ICLR: International Conference on Learning Representations*, 2022.

Piotr Piekos, Henryk Michalewski, and Mateusz Malinowski. Measuring and improving bert's mathematical abilities by predicting the order of reasoning. *ACL: Association for Computational Linguistics*, 2021.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR*, abs/2105.12655, 2021. URL `https://arxiv.org/abs/2105.12655`.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Yasaman Razeghi, Robert L. Logan, Matt Gardner, and Sameer Singh. Impact of pretraining term frequencies on few-shot reasoning. 2022. doi: 10.48550/ARXIV.2202.07206. URL `https://arxiv.org/abs/2202.07206`.

Stuart Rose, Dave Engel, Nick Cramer, and Wendy Cowley. Automatic keyword extraction from individual documents. *Text Mining: Applications and Theory*, pp. 1 – 20, 03 2010. doi: 10.1002/9780470689646.ch1.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. 2019. URL `https://openreview.net/forum?id=H1gR5iR5FX`.

Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. *Advances in Neural Information Processing Systems*, 32, 2019.

Alon Talmor, Ori Yoran, Ronan Le Bras, Chandra Bhagavatula, Yoav Goldberg, Yejin Choi, and Jonathan Berant. Commonsenseqa 2.0: Exposing the limits of AI through gamification. *CoRR*, abs/2201.05320, 2022. URL `https://arxiv.org/abs/2201.05320`.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. Natural language processing with transformers. 2022a.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. Natural language processing with transformers. 2022b.

Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal adversarial triggers for attacking and analyzing NLP. pp. 2153–2162, November 2019. doi: 10.18653/v1/D19-1221. URL `https://aclanthology.org/D19-1221`.

Ben Wang and Aran Komatsuzaki. Gpt-j-6b: A 6 billion parameter autoregressive language model. May 2021.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Michihiro Yasunaga, Jure Leskovec, and Percy Liang. Lm-critic: Language models for unsupervised grammatical error correction. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

Honghua Zhang, Liunian Harold Li, Tao Meng, Kai-Wei Chang, and Guy Van den Broeck. On the paradox of learning to reason from data. *arXiv preprint arXiv:2205.11502*, 2022.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving few-shot performance of language models. pp. 12697–12706, 2021. URL `http://proceedings.mlr.press/v139/zhao21c.html`.

A Appendix

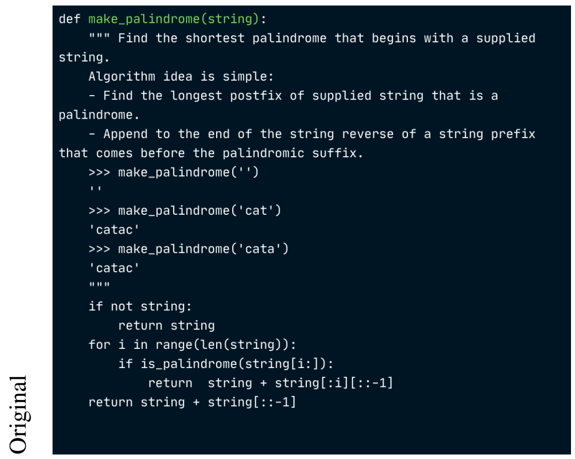
A.1 URLs of Code-Generation Models

Model Name	Link
KeyBert (Grootendorst, 2020)	https://github.com/MaartenGr/KeyBERT
Codeparrot (Tunstall et al., 2022b)	https://huggingface.co/codeparrot/codeparrot
Incoder (Fried et al., 2022)	https://github.com/dpfried/incoder
Codex-V2 (Chen et al., 2021)	https://beta.openai.com/

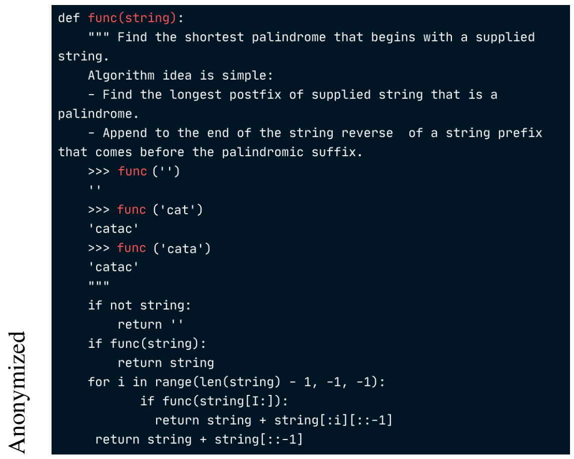
Table 7: URLs to Models. Table shows the URLs of models used in our investigations.

A.2 Qualitative Examples

We present examples of code generation failures caused by our framework across different models and scenarios. On each pair, the left image represents the original, unmodified challenge alongside the correctly generated solution. The right image contains the modified version of the challenge and the incorrect completion. Note that items that are the targets of our transformations are marked in green in the original challenge and red on the modified one. In the case of anonymization, the items are replaced by func, and in any other case are just removed.

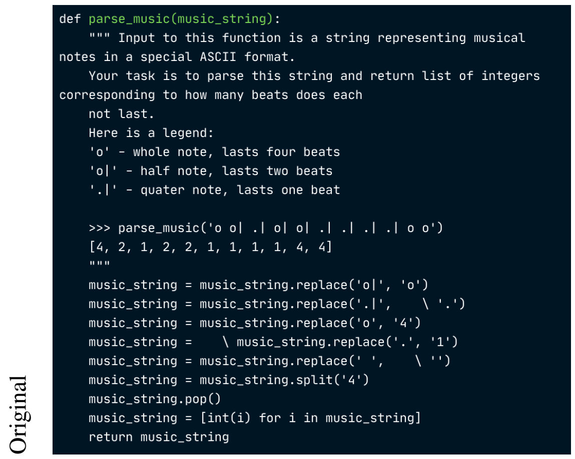


```
def make_palindrome(string):
    """ Find the shortest palindrome that begins with a supplied
string.
    Algorithm idea is simple:
    - Find the longest postfix of supplied string that is a
palindrome.
    - Append to the end of the string reverse of a string prefix
that comes before the palindromic suffix.
    >>> make_palindrome('')
    ''
    >>> make_palindrome('cat')
    'catac'
    >>> make_palindrome('cata')
    'catac'
    """
    if not string:
        return string
    for i in range(len(string)):
        if is_palindrome(string[i:]):
            return  string + string[:i][::-1]
    return string + string[::-1]
```

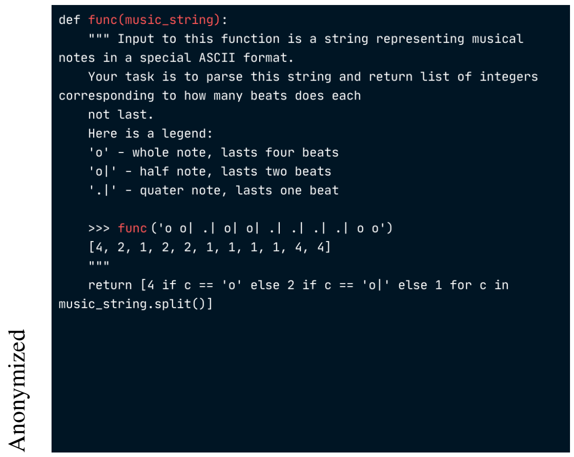


```
def func(string):
    """ Find the shortest palindrome that begins with a supplied
string.
    Algorithm idea is simple:
    - Find the longest postfix of supplied string that is a
palindrome.
    - Append to the end of the string reverse  of a string prefix
that comes before the palindromic suffix.
    >>> func ('')
    ''
    >>> func ('cat')
    'catac'
    >>> func ('cata')
    'catac'
    """
    if not string:
        return ''
    if func(string):
        return string
    for i in range(len(string) - 1, -1, -1):
           if func(string[I:]):
             return string + string[:i][::-1]
     return string + string[::-1]
```

Figure 4: Instance of anonymization effect on Codex-V2.

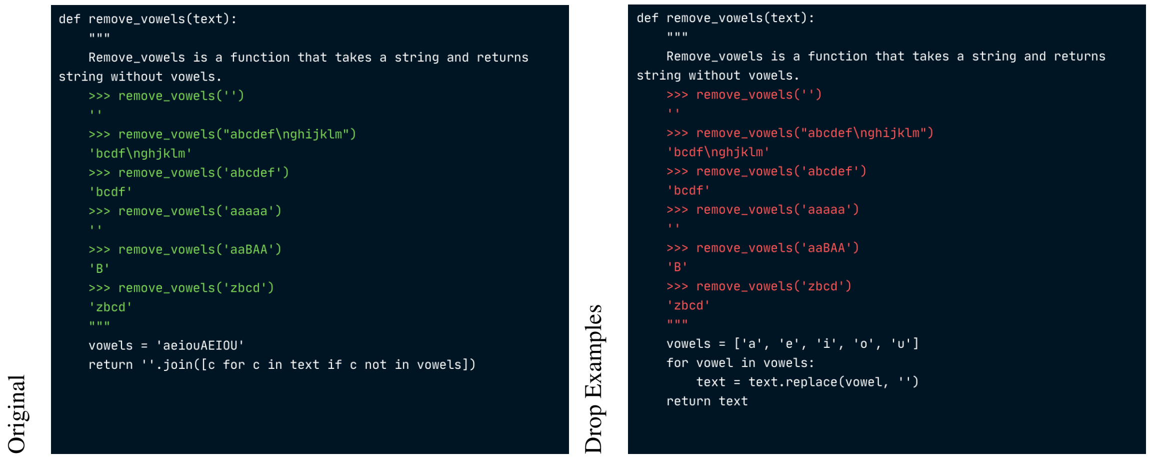


```
def parse_music(music_string):
    """ Input to this function is a string representing musical
notes in a special ASCII format.
    Your task is to parse this string and return list of integers
corresponding to how many beats does each
    not last.
    Here is a legend:
    'o' - whole note, lasts four beats
    'o|' - half note, lasts two beats
    '.|' - quater note, lasts one beat

    >>> parse_music('o o| .| o| o| .| .| .| .| o o')
    [4, 2, 1, 2, 2, 1, 1, 1, 1, 4, 4]
    """
    music_string = music_string.replace('o|', 'o')
    music_string = music_string.replace('.|',    \ '.')
    music_string = music_string.replace('o', '4')
    music_string =    \ music_string.replace('.', '1')
    music_string = music_string.replace(' ',    \ '')
    music_string = music_string.split('4')
    music_string.pop()
    music_string = [int(i) for i in music_string]
    return music_string
```



```
def func(music_string):
    """ Input to this function is a string representing musical
notes in a special ASCII format.
    Your task is to parse this string and return list of integers
corresponding to how many beats does each
    not last.
    Here is a legend:
    'o' - whole note, lasts four beats
    'o|' - half note, lasts two beats
    '.|' - quater note, lasts one beat

    >>> func('o o| .| o| o| .| .| .| .| o o')
    [4, 2, 1, 2, 2, 1, 1, 1, 1, 4, 4]
    """
    return [4 if c == 'o' else 2 if c == 'o|' else 1 for c in
music_string.split()]
```

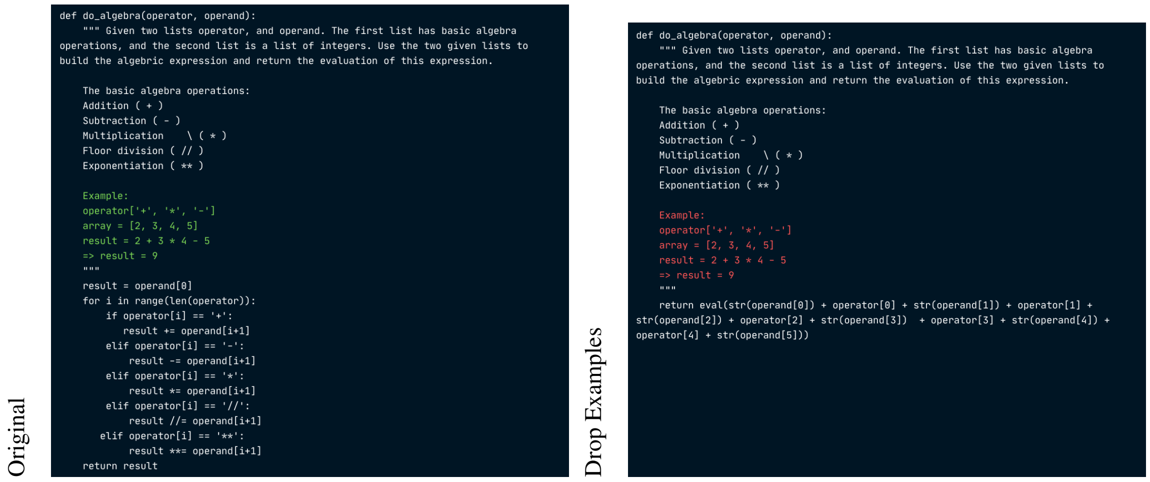
Figure 5: Instance of anonymization on Incoder-6B



```
def remove_vowels(text):
    """
    Remove_vowels is a function that takes a string and returns
string without vowels.
    >>> remove_vowels('')
    ''
    >>> remove_vowels("abcdef\nghijklm")
    'bcdf\nghjklm'
    >>> remove_vowels('abcdef')
    'bcdf'
    >>> remove_vowels('aaaaa')
    ''
    >>> remove_vowels('aaBAA')
    'B'
    >>> remove_vowels('zbcd')
    'zbcd'
    """
    vowels = 'aeiouAEIOU'
    return ''.join([c for c in text if c not in vowels])
```

```
def remove_vowels(text):
    """
    Remove_vowels is a function that takes a string and returns
string without vowels.
    >>> remove_vowels('')
    ''
    >>> remove_vowels("abcdef\nghijklm")
    'bcdf\nghjklm'
    >>> remove_vowels('abcdef')
    'bcdf'
    >>> remove_vowels('aaaaa')
    ''
    >>> remove_vowels('aaBAA')
    'B'
    >>> remove_vowels('zbcd')
    'zbcd'
    """
    vowels = ['a', 'e', 'i', 'o', 'u']
    for vowel in vowels:
        text = text.replace(vowel, '')
    return text
```

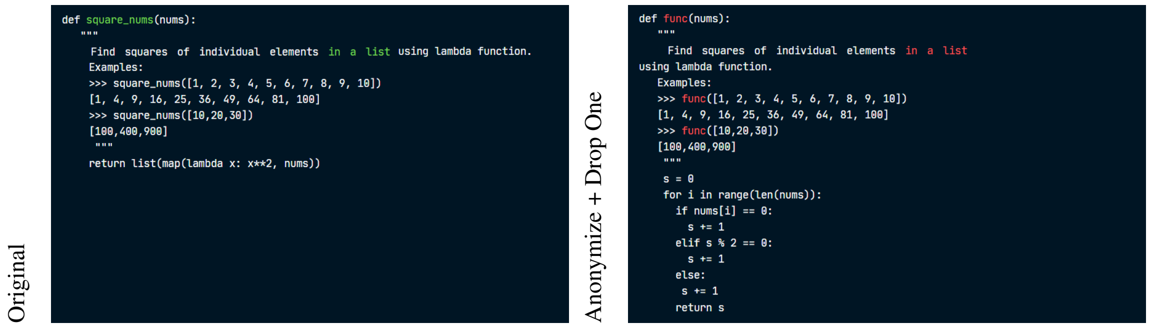
Figure 6: Instance of dropping the prompt examples on Codex-V2



```
def do_algebra(operator, operand):
    """ Given two lists operator, and operand. The first list has basic algebra
operations, and the second list is a list of integers. Use the two given lists to
build the algebric expression and return the evaluation of this expression.

    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication    \ ( * )
    Floor division ( // )
    Exponentiation ( ** )

    Example:
    operator['+', '*', '-']
    array = [2, 3, 4, 5]
    result = 2 + 3 * 4 - 5
    => result = 9
    """
    result = operand[0]
    for i in range(len(operator)):
        if operator[i] == '+':
           result += operand[i+1]
        elif operator[i] == '-':
            result -= operand[i+1]
        elif operator[i] == '*':
            result *= operand[i+1]
        elif operator[i] == '//':
            result //= operand[i+1]
       elif operator[i] == '**':
            result **= operand[i+1]
    return result
```

```
def do_algebra(operator, operand):
    """ Given two lists operator, and operand. The first list has basic algebra
operations, and the second list is a list of integers. Use the two given lists to
build the algebric expression and return the evaluation of this expression.

    The basic algebra operations:
    Addition ( + )
    Subtraction ( - )
    Multiplication    \ ( * )
    Floor division ( // )
    Exponentiation ( ** )

    Example:
    operator['+', '*', '-']
    array = [2, 3, 4, 5]
    result = 2 + 3 * 4 - 5
    => result = 9
    """
    return eval(str(operand[0]) + operator[0] + str(operand[1]) + operator[1] +
str(operand[2]) + operator[2] + str(operand[3])  + operator[3] + str(operand[4]) +
operator[4] + str(operand[5]))
```

Figure 7: Instance of dropping the prompt examples on CodeParrot-1.7B

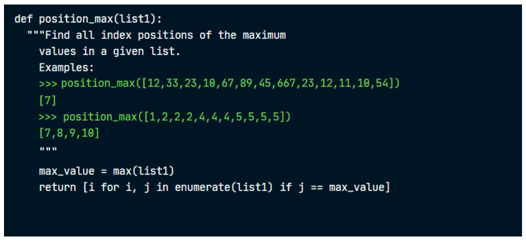


```
def square_nums(nums):
   """
    Find  squares  of  individual  elements  in  a  list  using lambda function.
    Examples:
    >>> square_nums([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
    >>> square_nums([10,20,30])
    [100,400,900]
     """
    return list(map(lambda x: x**2, nums))
```

```
def func(nums):
   """
     Find  squares  of  individual  elements  in  a  list
using lambda function.
   Examples:
   >>> func([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
   [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
   >>> func([10,20,30])
   [100,400,900]
    """
    s = 0
    for i in range(len(nums)):
      if nums[i] == 0:
        s += 1
      elif s % 2 == 0:
        s += 1
      else:
       s += 1
      return s
```

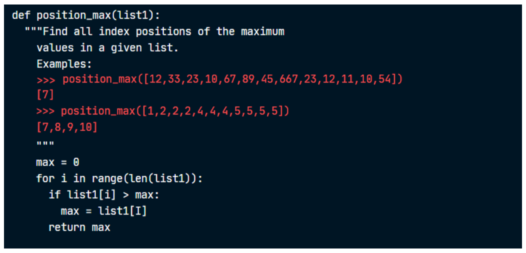
Figure 8: Instance of dropping the prompt examples on Bloom-1.7B

Original



```
def position_max(list1):
  """Find all index positions of the maximum
    values in a given list.
    Examples:
    >>> position_max([12,33,23,10,67,89,45,667,23,12,11,10,54])
    [7]
    >>> position_max([1,2,2,2,4,4,4,5,5,5,5])
    [7,8,9,10]
    """
    max_value = max(list1)
    return [i for i, j in enumerate(list1) if j == max_value]
```

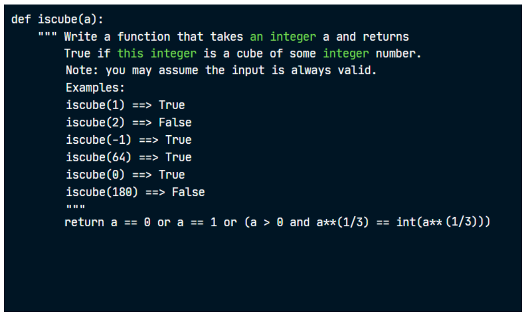
Drop Examples



```
def position_max(list1):
  """Find all index positions of the maximum
    values in a given list.
    Examples:
    >>> position_max([12,33,23,10,67,89,45,667,23,12,11,10,54])
    [7]
    >>> position_max([1,2,2,2,4,4,4,5,5,5,5])
    [7,8,9,10]
    """
    max = 0
    for i in range(len(list1)):
      if list1[i] > max:
        max = list1[I]
      return max
```

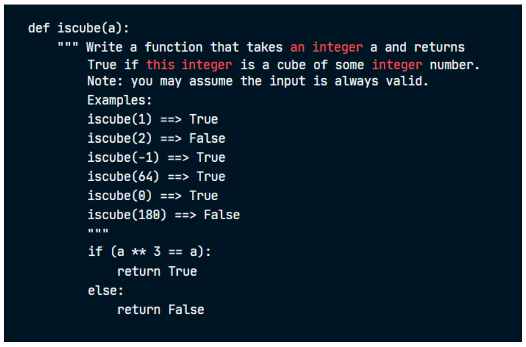
Figure 9: Instance of dropping the prompt examples on CodeParrot-110M

Original



```
def iscube(a):
    """ Write a function that takes an integer a and returns
        True if this integer is a cube of some integer number.
        Note: you may assume the input is always valid.
        Examples:
        iscube(1) ==> True
        iscube(2) ==> False
        iscube(-1) ==> True
        iscube(64) ==> True
        iscube(0) ==> True
        iscube(180) ==> False
        """
        return a == 0 or a == 1 or (a > 0 and a**(1/3) == int(a** (1/3)))
```

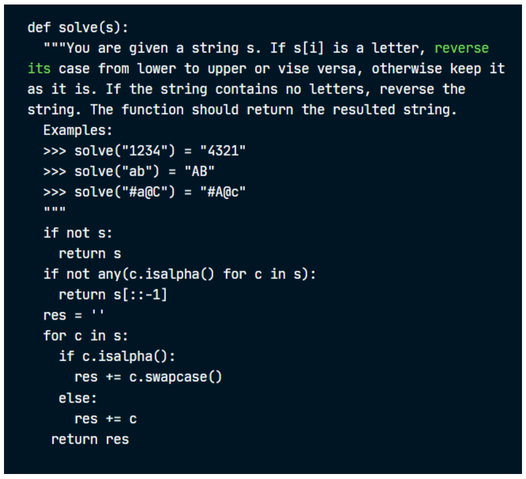
Drop All



```
def iscube(a):
    """ Write a function that takes an integer a and returns
        True if this integer is a cube of some integer number.
        Note: you may assume the input is always valid.
        Examples:
        iscube(1) ==> True
        iscube(2) ==> False
        iscube(-1) ==> True
        iscube(64) ==> True
        iscube(0) ==> True
        iscube(180) ==> False
        """
        if (a ** 3 == a):
            return True
        else:
            return False
```

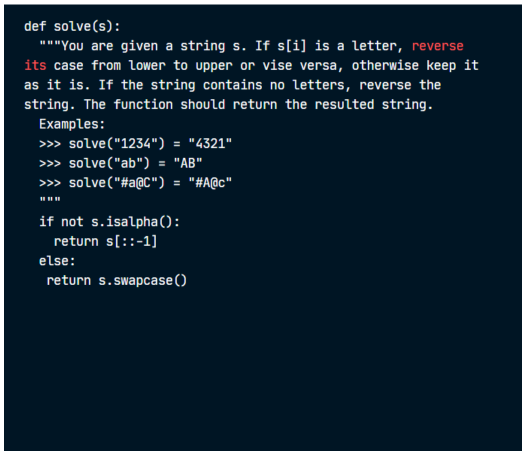
Figure 10: Instance of dropping the prompt examples on Bloom-175B

Original



```
def solve(s):
  """You are given a string s. If s[i] is a letter, reverse
its case from lower to upper or vise versa, otherwise keep it
as it is. If the string contains no letters, reverse the
string. The function should return the resulted string.
  Examples:
  >>> solve("1234") = "4321"
  >>> solve("ab") = "AB"
  >>> solve("#a@C") = "#A@c"
  """
  if not s:
    return s
  if not any(c.isalpha() for c in s):
    return s[::-1]
  res = ''
  for c in s:
    if c.isalpha():
      res += c.swapcase()
    else:
      res += c
   return res
```

Drop All



```
def solve(s):
  """You are given a string s. If s[i] is a letter, reverse
its case from lower to upper or vise versa, otherwise keep it
as it is. If the string contains no letters, reverse the
string. The function should return the resulted string.
  Examples:
  >>> solve("1234") = "4321"
  >>> solve("ab") = "AB"
  >>> solve("#a@C") = "#A@c"
  """
  if not s.isalpha():
    return s[::-1]
  else:
   return s.swapcase()
```

Figure 11: Instance of dropping the prompt examples on Incoder-1.6B

A.3 Quantative Results

We present our full results table, including the CodeParrot(110M) and Codex(v1) results. Note here that experiments involving the large version of the Bloom Model were done once in the case of pass@100 metric due to restrictions with the API request limits.

		Human Eval		MBPP	
Model	Method of Attack	Pass@1 (T=0.2)	Pass@100 (T=0.8)	Pass@1 (T=0.2)	Pass@100 (T=0.8)
Codeparrot (110M) (Tunstall et al., 2022b)	None	3.8	12.7	5.1	26.2
	Drop One	3.3 (±0.1)	9.7 (±0.3)	4.1 (±0.1)	16.3 (±0.5)
	Drop All	3.1 (±0.1)	7.2 (±0.5)	3.9 (±0.1)	15.7 (±0.7)
	Drop Ex	3.8 (±0.0)	9.9 (±0.2)	5.0 (±0.0)	18.4 (±0.3)
	Anon	3.4 (±0.1)	8.7 (±0.2)	4.4 (±0.1)	16.1 (±0.5)
	Anon+Drop One	3.0 (±0.1)	7.5 (±0.5)	4.0 (±0.1)	13.6 (±1.1)
	Anon+Drop All	1.9 (±0.2)	6.9 (±0.5)	3.9 (±0.2)	12.0 (±1.5)
	Anon+Drop Ex	3.4 (±0.1)	8.7 (±0.3)	4.3 (±0.2)	16.1 (±0.8)
Codeparrot (1.5B) (Tunstall et al., 2022b)	None	4.1	17.8	6.1	31.2
	Drop One	3.9 (±0.1)	13.2 (±0.4)	4.2 (±0.2)	26.8 (±0.8)
	Drop All	3.6 (±0.3)	11.1 (±0.6)	3.9 (±0.2)	21.7 (±1.1)
	Drop Ex	3.7 (±0.0)	14.3 (±0.2)	5.3 (±0.0)	27.5 (±0.7)
	Anon	3.8 (±0.1)	12.5 (±0.2)	4.7 (±0.2)	23.2 (±0.9)
	Anon+Drop One	3.3 (±0.2)	9.5 (±0.7)	3.9 (±0.1)	20.2 (±1.5)
	Anon+Drop All	2.1 (±0.3)	8.9 (±1.1)	3.9 (±0.2)	17.9 (±1.8)
	Anon+Drop Ex	3.7 (±0.2)	11.8 (±0.9)	4.6 (±0.1)	22.8 (±0.9)
Bloom (1.7B) (Tunstall et al., 2022b)	None	4.3	14.6	6.6	37.2
	Drop One	3.0 (±0.2)	12.2 (±0.6)	2.7 (±0.3)	27.6 (±1.2)
	Drop All	2.4 (±0.3)	9.8 (±0.9)	2.6 (±0.3)	24.2 (±1.8)
	Drop Ex	3.6 (±0.1)	12.8 (±0.5)	3.1 (±0.2)	29.0 (±0.9)
	Anon	3.6 (±0.1)	11.6 (±0.5)	3.1 (±0.1)	27.5 (±1.1)
	Anon+Drop One	2.4 (±0.3)	9.1 (±1.1)	2.4 (±0.5)	25.3 (±1.8)
	Anon+Drop All	1.8 (±0.5)	8.5(±1.3)	2.0 (±0.6)	23.1 (±2.3)
	Anon+Drop Ex	3.4 (±0.2)	11.6 (±0.6)	3.0 (±0.3)	26.7 (±1.3)
Incoder (1.6B) (Fried et al., 2022)	None	11.3	24.2	14.6	56.7
	Drop One	10.5 (±0.1)	22.3 (±0.9)	11.5(±0.4)	45.4 (±1.1)
	Drop All	9.7 (±0.3)	17.6 (±1.2)	12.8 (±0.6)	42.1 (±1.9)
	Drop Ex	11.3 (±0.2)	22.2 (±1.5)	14.4(±0.3)	43.8 (±0.7)
	Anon	9.1 (±0.1)	21.8 (±0.8)	11.3 (±0.5)	45.2 (±0.8)
	Anon+Drop One	7.4 (±0.7)	21.5 (±1.8)	10.5 (±0.6)	44.9 (±2.4)
	Anon+Drop All	6.3 (±0.9)	17.5 (±2.2)	8.0 (±0.8)	41.3(±2.5)
	Anon+Drop Ex	8.7 (±0.5)	21.3 (±1.6)	11.2 (±0.5)	43.5(±1.0)

Table 8: First part of results on Human Eval (Chen et al., 2021) and MBPP (Shin et al., 2019) datasets, for four tested models.

Model	Method of Attack	Human Eval Pass@1 (T=0.2)	Human Eval Pass@100 (T=0.8)	MBPP Pass@1 (T=0.2)	MBPP Pass@100 (T=0.8)
Incoder (6B) (Fried et al., 2022)	None	15.2	47.0	19.4	65.1
	Drop One	12.1 (±0.3)	35.3 (±1.2)	18.9 (±0.5)	52.6 (±1.1)
	Drop All	10.2 (±0.5)	28.2 (±1.4)	15.6 (±0.5)	47.0 (±1.9)
	Drop Ex	12.7 (±0.3)	29.5 (±0.9)	17.4 (±0.3)	50.3 (±0.7)
	Anon	11.6 (±0.2)	32.9 (±0.9)	14.8 (±0.6)	50.7 (±0.8)
	Anon+Drop One	8.1 (±0.7)	30.6 (±1.7)	13.5 (±0.7)	46.7 (±2.4)
	Anon+Drop All	7.5 (±1.3)	25.2 (±2.3)	11.2 (±1.1)	38.9 (±2.5)
	Anon+Drop Ex	11.2 (±0.4)	28.1 (±1.1)	14.5 (±0.5)	50.2 (±1.0)
Codex (v1) (Chen et al., 2021)	None	39	82.9	51.7	83.4
	Drop One	29.2 (±0.2)	78 (±1.3)	48.3 (±0.4)	78.7 (±1.0)
	Drop All	30 (±0.4)	67.2 (±1.7)	33.9 (±0.8)	67.3 (±1.9)
	Drop Ex	32.9 (±0.1)	73.7 (±1.1)	42.1 (±0.2)	70.1 (±0.9)
	Anon	35.3 (±0.1)	81.7 (±1.2)	50.8 (±0.2)	81.5 (±1.2)
	Anon+Drop One	23.7 (±0.5)	67.0 (±2.3)	44.1 (±0.7)	67.7 (±2.6)
	Anon+Drop All	19.5 (±0.9)	62.1 (±2.7)	40.7 (±1.4)	61.4 (±3.1)
	Anon+Drop Ex	27.4 (±0.3)	65.2 (±1.6)	36.7 (±0.3)	67.7 (±1.5)
Codex (v2) (Chen et al., 2021)	None	49.4	91.4	60.1	86.3
	Drop One	36.0 (±0.1)	86.2 (±0.8)	56.0 (±0.3)	79.2 (±1.1)
	Drop All	37.1 (±0.3)	73.7 (±1.3)	52.1 (±0.6)	69.5 (±1.8)
	Drop Ex	41.4 (±0.1)	81.0 (±1.1)	48.8 (±0.3)	70.7 (±0.9)
	Anon	44.5 (±0.2)	90.4 (±1.1)	57.9 (±0.3)	81.7 (±1.0)
	Anon+Drop One	29.8 (±0.7)	74.4 (±2.1)	51.2 (±1.1)	69.5 (±2.3)
	Anon+Drop All	24.2 (±0.8)	68.7 (±2.8)	47.2 (±1.3)	63.8 (±3.0)
	Anon+Drop Ex	34.1 (±0.3)	72.5 (±1.1)	42.6 (±0.4)	70.5 (±1.3)
Bloom (176B) (Tunstall et al., 2022b)	None	16.4	57.2	20.8	62.4
	Drop One	12.8 (±0.3)	48.6	15.8 (±0.3)	51.4
	Drop All	11.5 (±0.6)	40.2	14.2 (±0.5)	44.4
	Drop Ex	15.2 (±0.2)	43.3	15.8 (±0.2)	50.1
	Anon	14.0 (±0.3)	48.3	15.1 (±0.1)	51.2
	Anon+Drop One	12.8 (±0.4)	41.9	13.6 (±0.7)	46.8
	Anon+Drop All	10.3 (±0.8)	36.8	12.6 (±1.1)	38.4
	Anon+Drop Ex	14.0 (±0.3)	39.8	14.3 (±0.3)	47.8

Table 9: Second part of results on Human Eval (Chen et al., 2021) and MBPP (Shin et al., 2019) datasets, for four tested models.

A.4 Notes on Semantic Preservation

Maintaining semantics in a programming challenge is a non-trivial task. From a pure-language perspective, any modified text should be lexically correct, syntactically sound, and semantically similar to the original. Furthermore, any transformations should respect the formatting rules of the programming language in which the challenge is written. Thus, character-level attacks or substitution by similar words is outside of our scope. Augmenting language-reserved words or variables can also lead to runtime errors. For example, in Python3, changing ‘print‘ to ‘pint‘ or changing a ‘for‘ to a still eligible python keyword ‘or‘ can lead to syntactic errors. Another example would be swapping the word ‘large‘ with a synonym, such as ‘tall,‘ in a docstring that refers to sorting numbers from the largest to the smallest. Demanding a model or a human to understand a ’tall’ number is far-fetched.

With our introduction of Blocks of Influence, the task of semantic preservation becomes much easier. Starting from the *Name Block*, swapping the name of any function with a token such as ‘func‘ is a safe option that causes no harm to the overall understanding. In the case of the *Example Block*, the same applies to removing one or more examples. From the perspective of a code-solving participant, the examples do not provide any direct guidance. Their purpose is to act as ‘soft checks,‘ confirming or dismissing some initial ideas before forming the correct solution. In all proposed methods involving these two blocks, enough information is available in the *Description Block*and the respective unaffected other block.

Finally, as far as the *Description Block* is concerned, our method removes keywords that are assumed to provide hints, making the tested model to exploit shortcuts to a correct generation. Our framework utilizes context-aware filtering (described in Section 5.1-Framework) so that each removed keyword can be deducted from the rest of the challenge. This can be seen in Figure 3c. Semantically, the resulting docstrings are understandable from a human standpoint after some manual introspection. Small linguistic inconsistencies can be found, mostly in the case of *Drop All* augmentation.

However, in order to quantify the possible reduction of understanding these changes cause, we employ the following test, inspired by the work of Yasunaga et al. (2021): We collect a random sample of 200 coding challenges from the HumanEval and MBPP. Each sample is transformed according to the modes presented in Table10. For the unmodified and each transformed version of the challenge, we then calculate its log probability score using a pretrained language model, preferably trained on code.
The main idea is that the model will act as a soft critic, assessing the ability of each challenge to be understood under the assumption of small changes. Poorly understood prompts will be assigned a log probability score far lower than the unmodified one. Note that we omit results for the *Drop Examples* method. In this case, the log probabilities will change rapidly since many words are removed, which violates the method’s assumption of comparing lightly changed prompts. We calculate log probability similarity through the following formula:

$$LogProbSim = 100 - \frac{LogProbAfter - LogProbBefore}{LogProbBefore}$$

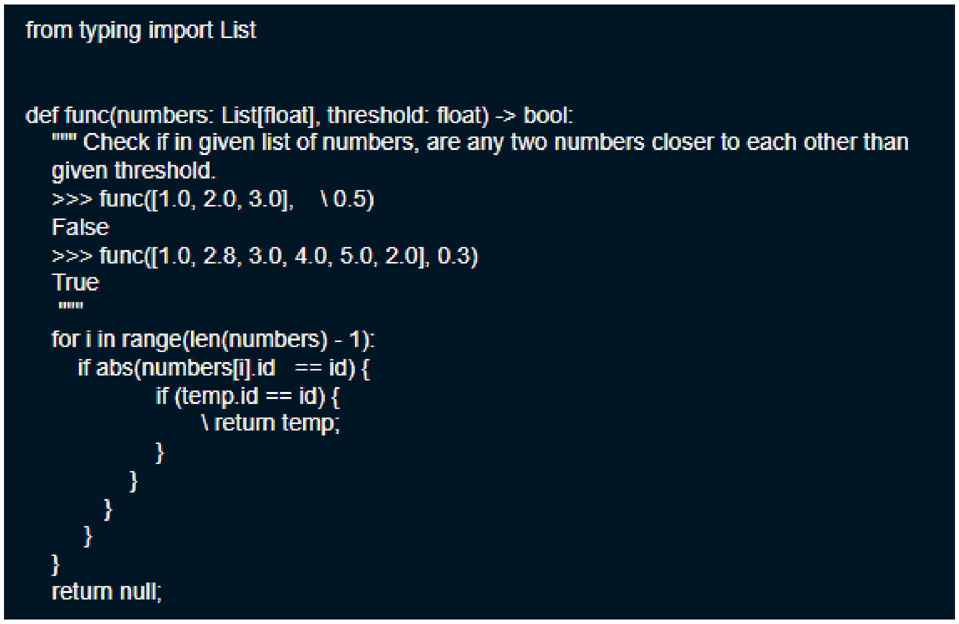
We choose Codex-v2 as our language model critic, and present our results below:

Mode	LogProb Similarity (%)
Original	100.0 (± 0.0)
Anonymization	98.5 (± 1.2)
Drop One	97.3 (± 1.5)
Drop All	95.3 (± 1.9)
Anonymization + Drop One	95.8 (± 1.4)
Anonymization + Drop All	94.6 (± 2.3)

Table 10: Log Probability Similarity scores for different methods of our framework.

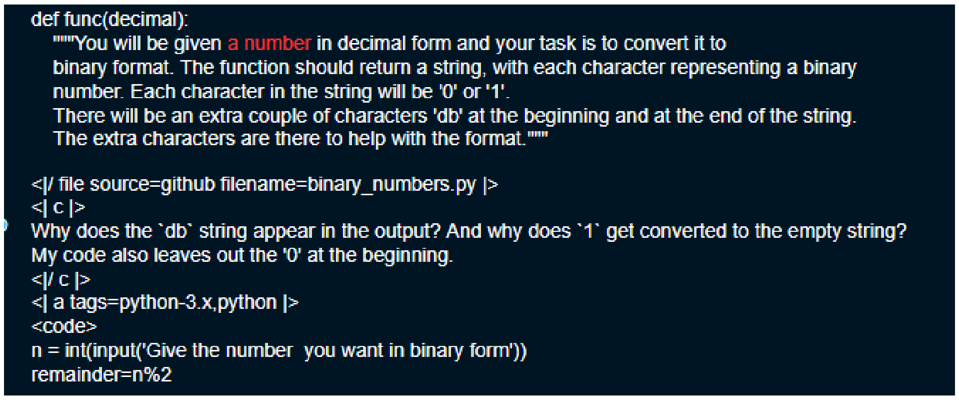
Our results show, that our transformations do not introduce any drastic changes to the challenge. Even in the most aggressive modification of *Anonymization + Drop All*, the critic assigns around 95% similarity between code challenges affected by it, versus their original form. We believe that this is a fair indicator that the tested models are observing semantically similar inputs during our experiments.

A.5 Few Interesting Examples



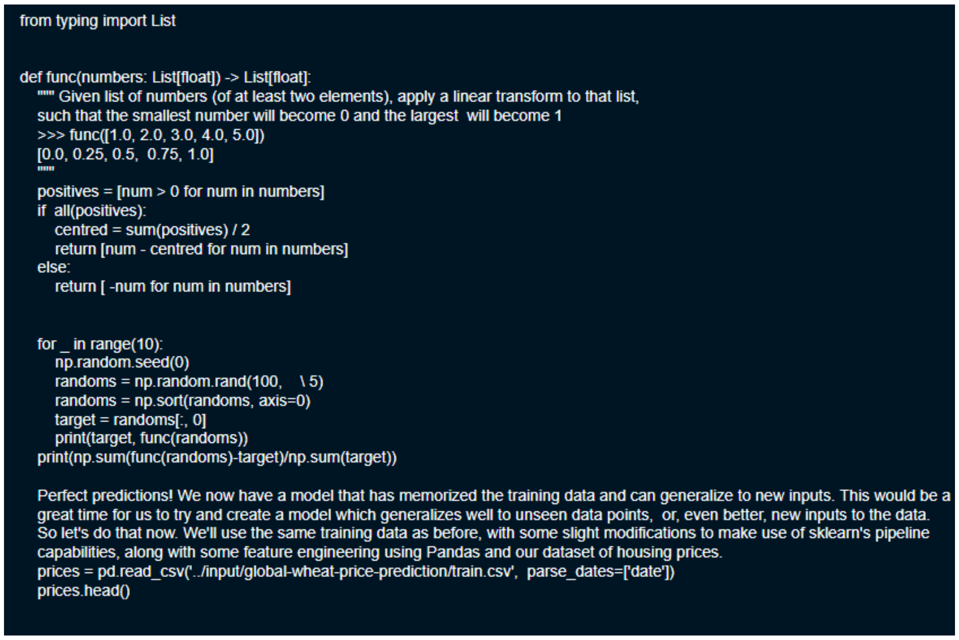
```
from typing import List


def func(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> func([1.0, 2.0, 3.0],    \ 0.5)
    False
    >>> func([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
     """
    for i in range(len(numbers) - 1):
        if abs(numbers[i].id   == id) {
                    if (temp.id == id) {
                        \ return temp;
                    }
                }
            }
        }
    }
    return null;
```

Figure 12: Bloom (175B) using Javascript instead of Python3 to complete a function with the *Anonymize* transformation.



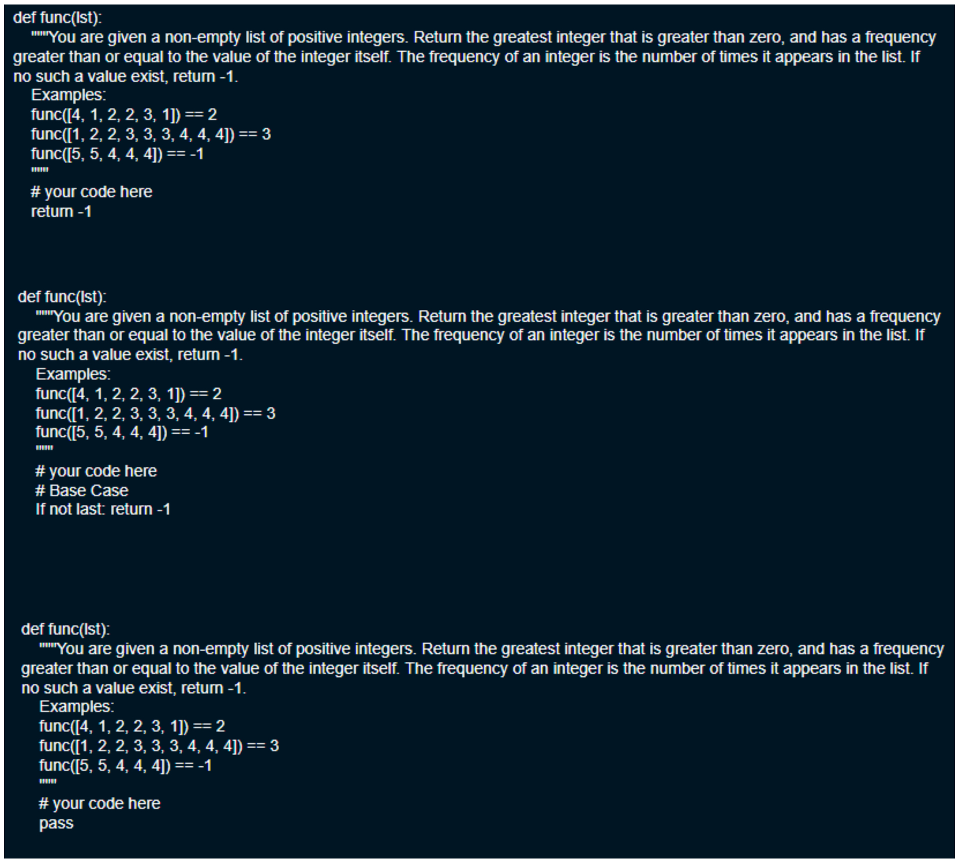
```
def func(decimal):
    """You will be given a number in decimal form and your task is to convert it to
    binary format. The function should return a string, with each character representing a binary
    number. Each character in the string will be '0' or '1'.
    There will be an extra couple of characters 'db' at the beginning and at the end of the string.
    The extra characters are there to help with the format."""

<|/ file source=github filename=binary_numbers.py |>
<| c |>
Why does the `db` string appear in the output? And why does `1` get converted to the empty string?
My code also leaves out the '0' at the beginning.
<|/ c |>
<| a tags=python-3.x,python |>
<code>
n = int(input('Give the number  you want in binary form'))
remainder=n%2
```

Figure 13: Incoder (6B) disclosing the name of a file as well as some human-like questions when faced with a *Anonymize* + *Drop One* transformation.



```
from typing import List


def func(numbers: List[float]) -> List[float]:
    """ Given list of numbers (of at least two elements), apply a linear transform to that list,
    such that the smallest number will become 0 and the largest  will become 1
    >>> func([1.0, 2.0, 3.0, 4.0, 5.0])
    [0.0, 0.25, 0.5,  0.75, 1.0]
    """
    positives = [num > 0 for num in numbers]
    if  all(positives):
        centred = sum(positives) / 2
        return [num - centred for num in numbers]
    else:
        return [ -num for num in numbers]


    for _ in range(10):
        np.random.seed(0)
        randoms = np.random.rand(100,    \ 5)
        randoms = np.sort(randoms, axis=0)
        target = randoms[:, 0]
        print(target, func(randoms))
    print(np.sum(func(randoms)-target)/np.sum(target))

    Perfect predictions! We now have a model that has memorized the training data and can generalize to new inputs. This would be a
    great time for us to try and create a model which generalizes well to unseen data points,  or, even better, new inputs to the data.
    So let's do that now. We'll use the same training data as before, with some slight modifications to make use of sklearn's pipeline
    capabilities, along with some feature engineering using Pandas and our dataset of housing prices.
    prices = pd.read_csv('../input/global-wheat-price-prediction/train.csv',  parse_dates=['date'])
    prices.head()
```

Figure 14: Incoder (1.6B) adding some snippet of ambiguous functionality followed by something that looks like some exercise comments.



```
def func(lst):
    """You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a frequency
greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it appears in the list. If
no such a value exist, return -1.
    Examples:
    func([4, 1, 2, 2, 3, 1]) == 2
    func([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
    func([5, 5, 4, 4, 4]) == -1
    """
    # your code here
    return -1



def func(lst):
    """You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a frequency
greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it appears in the list. If
no such a value exist, return -1.
    Examples:
    func([4, 1, 2, 2, 3, 1]) == 2
    func([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
    func([5, 5, 4, 4, 4]) == -1
    """
    # your code here
    # Base Case
    If not last: return -1




def func(lst):
    """You are given a non-empty list of positive integers. Return the greatest integer that is greater than zero, and has a frequency
greater than or equal to the value of the integer itself. The frequency of an integer is the number of times it appears in the list. If
no such a value exist, return -1.
    Examples:
    func([4, 1, 2, 2, 3, 1]) == 2
    func([1, 2, 2, 3, 3, 3, 4, 4, 4]) == 3
    func([5, 5, 4, 4, 4]) == -1
    """
    # your code here
    pass
```

Figure 15: Three different instances of Codex (v1) completions to an anonymized problem.

A.6 Algorithms

Algorithm 1 Block of Influence Splitting

```
1: $cc$ : Code Challenge Instance
   # Locate function name, which is the next token after the last matched ”def”, and keep start and end index of it.
2: $name, start\_name\_index, end\_name\_index \leftarrow NameMatch(cc)$
   # Anything prior to the match, such as imports or helper functions is considered prefix.
3: $prefix \leftarrow cc[: start\_name\_index]$
   # Look for tokens such as (Example, example, >, >> and >>>).
4: **if** $ExampleMatch(cc[end\_name\_index :]) \neq None$ **then**
5:     $examples, start\_example\_index \leftarrow ExampleMatch(cc[end\_name\_index :])$
6: **else**
   #If no matches were found, look for uses of the function name in the challenge.
7:     $examples, start\_example\_index \leftarrow FunctionMatch(cc[end\_name\_index :])$
8: **end if**
   # The description should fall between the function name and the examples.
9: $description \leftarrow cc[end\_name\_index : start\_example\_index]$
   # Form the blocks and return.
10: $NameBlock \leftarrow prefix + name$
11: $DescriptionBlock \leftarrow description$
12: $ExampleBlock \leftarrow examples$
```

Algorithm 2 Keyword Identification

```
1: $KB$ : The KeyBert model
2: $nb$ : Name Block
3: $db$ : Description Block
4: $eb$ : Example Block
5: $kw :\leftarrow \varnothing$ Keywords
6: $fkw :\leftarrow \varnothing$ Filtered Keywords
   # Use the model to extract some initial unigram and bigram keywords.
7: $kw \leftarrow KB(db)$
   # Filter out keywords non-related to coding.
8: **for** $i \ in \ kw$ **do**
9:     **if** $cossim(i, [Python, Programming, Code]) > 0.7$ **then**
   # Look for similar word stems, or python language equivalents (e.g list → [], set → ()) in the name block and example block
10:         **if** $stem(i) \in [nb, eb] \ or \ equiv(i) \in [nb, eb]$ **then**
11:             $fkw \leftarrow i$
12:         **end if**
13:     **end if**
14: **end for**
15: **return**
```

Algorithm 3 Transformation and Execution

```
 1: $CM$ : The code generation model
 2: $cc$ : A coding challenge instance
 3: $nb$ : Name Block
 4: $fkw$ : Filtered Keywords
 5: $db$ : Description Block
 6: $eb$ : Example Block
 7: $org\_pa1$ : Original Pass@1 score
 8: $tra\_pa1$ : Transformed Pass@1 score
 9: $org\_pa100$ : Original Pass@100 score
10: $tra\_pa100$ : Transformed Pass@1 score
11: $mode$ : The transformation mode
    # Measure initial performance on the challenge
12: $org\_pa1, org\_pa100 \leftarrow CM(cc, T = 0.2), CM(cc, T = 0.8)$
13: if $mode = 0$ then # Anonymization
14:     $cc\_new \leftarrow swap(nb, "func") + db + eb$
15: else if $mode = 1$ then# Drop One
16:     $cc\_new \leftarrow nb + remove\_kw(db, choose\_single(fkw)) + eb$
17: else if $mode = 2$ then# Drop All
18:     $cc\_new \leftarrow nb + remove\_kw(db, fkw) + eb$
19: else if $mode = 3$ then# Drop Examples
20:     $cc\_new \leftarrow nb + db$
21: else if $mode = 4$ then# Anonymization + Drop One
22:     $cc\_new \leftarrow swap(nb, "func") + remove\_kw(db, choose\_single(fkw)) + eb$
23: else if $mode = 5$ then# Anonymization + Drop All
24:     $cc\_new \leftarrow swap(nb, "func") + remove\_kw(db, fkw) + eb$
25: else if $mode = 6$ then# Anonymization + Drop Examples
26:     $cc\_new \leftarrow swap(nb, "func") + db$
27: end if
28: $tra\_pa1, tra\_pa100 \leftarrow CM(cc\_new, T = 0.2), CM(cc\_new, T = 0.8)$
29: $dif\_1 \leftarrow \frac{tra\_pa1 - org\_pa1}{tra\_pa1}$
30: $dif\_100 \leftarrow \frac{tra\_pa100 - org\_pa100}{tra\_pa100}$
31: return dif_1, dif_100
```