Program Synthesis with Large Language Models

Jacob Austin* **Augustus Odena***

Maxwell Nye† **Maarten Bosma** **Henryk Michalewski** **David Dohan** **Ellen Jiang** **Carrie Cai**

Michael Terry **Quoc Le** **Charles Sutton**

Google Research
* denotes equal contribution
jaaustin@google.com, augustusodena@google.com

Abstract

This paper explores the limits of the current generation of large language models for program synthesis in general purpose programming languages. We evaluate a collection of such models (with between 244M and 137B parameters) on two new benchmarks, MBPP and MathQA-Python, in both the few-shot and fine-tuning regimes. Our benchmarks are designed to measure the ability of these models to synthesize short Python programs from natural language descriptions. The Mostly Basic Programming Problems (MBPP) dataset contains 974 programming tasks, designed to be solvable by entry-level programmers. The MathQA-Python dataset, a Python version of the MathQA benchmark, contains 23914 problems that evaluate the ability of the models to synthesize code from more complex text. On both datasets, we find that synthesis performance scales log-linearly with model size. Our largest models, even without finetuning on a code dataset, can synthesize solutions to 59.6% of the problems from MBPP using few-shot learning with a well-designed prompt. Fine-tuning on a held-out portion of the dataset improves performance by about 10 percentage points across most model sizes. On the MathQA-Python dataset, the largest fine-tuned model achieves 83.8% accuracy. Going further, we study the model's ability to engage in dialog about code, incorporating human feedback to improve its solutions. We find that natural language feedback from a human halves the error rate compared to the model's initial prediction. Additionally, we conduct an error analysis to shed light on where these models fall short and what types of programs are most difficult to generate. Finally, we explore the semantic grounding of these models by fine-tuning them to predict the results of program execution. We find that even our best models are generally unable to predict the output of a program given a specific input.

2개의 벤치마크로 프로그래밍 능력 측정

MBPP
- 974개의 프로그래밍 과제
- 사람으로 치면 입문자 수준

MathQA-Python
- MathQA의 파이썬 버전
- 23914개의 문제
- 좀더 복잡한 텍스트로부터 코드를 합성하는 능력 평가

1 Introduction

오래된 재기, 부활

Program synthesis is a longstanding goal of artificial intelligence research [Manna and Waldinger, 1971, Waldinger et al., 1969, Summers, 1977, Shaw et al., 1975, Pnueli and Rosner, 1989, Manna and Waldinger, 1975], dating as far back as the 1940s and 50s [Copeland, 2012, Backus et al., 1957]. There has been a recent resurgence of interest in techniques (both symbolic and 'neuro-symbolic') for synthesizing programs [Balog et al., 2017, Devlin et al., 2017, Ellis et al., 2018, 2020, Odena et al., 2020], but these techniques have largely been applied to restricted domain-specific languages (DSLs) [Gulwani, 2011] or to languages that are more fully featured but that nevertheless are designed specifically with synthesis in mind [Odena and Sutton, 2020]. Modern general-purpose languages like Python or C++ have mostly been out-of-reach as targets. This is unfortunate, because it materially restricts the set of possible downstream applications. Synthesis methods that target problems across domains in general purpose languages have the potential to enable new tools that benefit the workflows of both novice and expert programmers.

Two emerging themes from the research literature point to a possible new approach for this problem (for a more detailed review, see Section 8). First, large language models have shown impressive new abilities to generate natural language

†Max is affiliated with MIT, but did this work while interning at Google Research.

prompt

```
Write a python function to check if a given number is one less than twice its reverse. Your code should satisfy these tests:

assert check(70) == False
assert check(23) == False
assert check(73) == True
```

model

```
def check(n) :
    if n == 2*int(str(n)[::-1])-1 :
        return True
    else :
        return False
```

prompt

```
Write a function to find the smallest missing element in a sorted array. Your code should satisfy these tests:

assert smallest_missing([0, 1, 2, 3, 4, 5, 6], 0, 6) == 7
assert smallest_missing([0, 1, 2, 6, 9, 11, 15], 0, 6) == 3
assert smallest_missing([1, 2, 3, 4, 6, 9, 11, 15], 0, 7) == 0
```

model

```
def smallest_missing(arr, n, m):
    smallest = min(n, m)
    for i in range(n, m + 1):
        if arr[i] <= smallest:
            smallest += 1
    return smallest
```

prompt

```
Write a Python function to sort the given array by using merge sort. Your code should satisfy these tests:

assert merge_sort([3, 4, 2, 6, 5, 7, 1, 9]) == [1, 2, 3, 4, 5, 6, 7, 9]
assert merge_sort([7, 25, 45, 78, 11, 33, 19]) == [7, 11, 19, 25, 33, 45, 78]
assert merge_sort([3, 1, 4, 9, 8]) == [1, 3, 4, 8, 9]
```

model

```
def merge_sort(arr):
  if len(arr) < 2:
    return arr;
  mid = len(arr) // 2;
  left = arr[:mid];
  right = arr[mid:];
  left = merge_sort(left);
  right = merge_sort(right);
  merged = [];
  i = j = 0;
  while i < len(left) and j < len(right):
    if left[i] < right[j]:
      merged.append(left[i]);
      i += 1;
    else:
      merged.append(right[j]);
      j += 1;
  merged.extend(left[i:]);
  merged.extend(right[j:]);
  return merged;
```

Figure 1: Example programs synthesized (few-shot) by our largest model. The prompt is shown in purple, and the model's response in blue. The prompt also typically contains several few-shot examples in the same format, which are not shown here.

text [Brown et al., 2020, Raffel et al., 2019] and to solve a rapidly expanding set of modeling and reasoning tasks [Devlin et al., 2019, big-bench collaboration, 2021]. Second, over the past decade, machine learning approaches have been applied to source code text to yield a variety of new tools to support software engineering [Allamanis et al., 2018a]. This work has included pre-trained deep models such as CuBERT [Kanade et al., 2020], CodeBERT [Feng et al., 2020], PyMT5 [Clement et al., 2020], code2vec [Alon et al., 2019], and other T5 models trained on code [Mastropaolo et al., 2021].

Combining these two themes raises the question of whether large language models for natural language can be brought to bear to synthesize code in a general-purpose language. Such models emit code in 'token-space', and so it is not necessary to explicitly encode the grammar of the underlying language—they learn it from data. Furthermore, these models can be trained on large quantities of code, so they can learn about how various libraries interact with each other and what idiomatic, human-readable code looks like. Finally, large language models allow us to consider a more flexible type of program specification: in contrast to classical work on program synthesis that specifies the program using logical constraints or input-output examples [Gulwani et al., 2017], a program can be specified by a short natural language description, possibly combined with a few (e.g., 2 or 3) input-output examples.

In this paper, we study how a collection of large Transformer language models performs when applied to the synthesis of short programs written in general purpose programming languages. Examples of problems and model output are shown in Figure 1 and Figure 2.

In particular, this paper makes the following contributions:

1. We introduce two datasets to test Python code synthesis. The first is a new dataset called Mostly Basic Programming Problems (MBPP). It contains 974 short Python functions designed to be solved by entry-level programmers, text descriptions of those programs, and test cases to check for functional correctness (Section 2.1). This dataset consists of a large set of crowd-sourced questions and a smaller set of questions edited and hand-

2가지 데이터셋 소개함
1) MBPP
- 974개의 입문자 수준도 풀만한 수준
- 파이썬 함수, Text Description(Docstring)으로 구성
- Test 케이스는 함수가 올바른지 체크
- 많은 크라우드 소싱 문제와 좀더 작은 규모의 수기로 수정된 문제셋으로 구성

verified by the authors. The second is a Python synthesis dataset, containing 23914 problems, produced by rewriting the solutions to a subset of the MathQA dataset [Amini et al., 2019] into Python (Section 2.2). We call this dataset MathQA-Python. These two datasets exercise different points in the space of synthesis tasks: MBPP contains more usage of imperative control flow such as loops and conditionals, while MathQA-Python contains more complex natural language descriptions.

2) MathQA-Python
- 23914개의 문제, 좀더 복잡한 자연어 Description을 포함함
- MathQA 데이터셋 중 Python 부분의 솔루션을 다시 작성함

명령적인

MBPP는 Control Flow, Loop, 조건문 등을 다룸

MathQA-Python은 좀 더 복잡한 자연어 Description을 다룸

2. On both datasets, we show that a large language model performs surprisingly well at few-shot synthesis of Python programs from a prompt (Sections 4 and 7). Fine-tuning further on each of the datasets yields a further increase in synthesis performance. This is especially notable for MBPP because the fine-tuning set is extremely small (374 synthesis problems). We evaluate the model performance at scales ranging from 244M to 137B parameters, finding that performance continues to improve with increased model size. The largest models that we consider can synthesize solutions to 59.6% of the problems from MBPP using few-shot learning. For most model sizes, fine-tuning increases performance by about 10 percentage points. On the smaller, hand-verified MBPP dataset, we observe that the synthesis task is indeed easier: For the 100 problems that occur in both the original and edited datasets, few-shot model performance increases from 63% on the original dataset to 79% on the edited dataset. On the MathQA-Python dataset, the largest model achieves few-shot accuracy of 33.4% while fine-tuning it leads to a very high accuracy of 83.8%.

3. Going beyond single-step program synthesis, we study the model's ability to engage in dialog about code and improve its performance in response to natural-language feedback from humans (Section 5). We find that the model is able to incorporate short natural language hints to repair its outputs and clarify under-specified prompts, increasing few-shot performance from 30% without human feedback to 65% with four turns of dialog, yielding a 50% error reduction (Section 5.1).

4. We explore the semantic grounding of our models, investigating the extent to which these models can *execute* code given specific inputs (Section 6). We find that even our largest models are generally unable to predict the output of a program given a particular input, whether few-shot (Section 6.1) or with fine-tuning (Section 6.2). This suggests a large gap between what these models are doing and what we would consider "understanding."

5. We analyze sensitivity of performance to a variety of factors, including model size, number of examples in the prompt, the identity of examples in prompt, sampling technique, etc. Furthermore, we investigate two potential criticisms of synthesis from large language models: First, we find that solutions tend to generalize to held-out test cases, rather than simply parroting the answers in the prompt (Section 4.4), although there are occasional exceptions (Section 4.5). Second, we find that the overlap between the solutions in MBPP and the pre-training set is small, reducing the chance that our synthesis results are due to memorization (Section 4.8).

Our work is closely related to two recent efforts. The first is the APPS dataset [Hendrycks et al., 2021], which is a dataset of 10,000 problems from coding competitions. Hendrycks et al. [2021] evaluate large language models on this data, specifically finetuned GPT-2 [Radford et al., 2019] and GPT-Neo [Black et al., 2021], as well as few-shot prediction with GPT-3 [Brown et al., 2020]. Additionally, several datasets have been proposed to train and evaluate program synthesis methods based on data from programming competitions (Section 8.3). However, performance on these benchmarks has generally been poor. We conjecture that this is because programming competition problems are written in a style that obfuscates the underlying algorithms necessary to solve them. By contrast, our Mostly Basic Programming Problems dataset is designed to contain a more basic, literal description of the problems. We believe this shifts the focus more toward capabilities directly related to generating and understanding code.

MBPP는 좀더 기본적이고, 직역인 Description으로 작성됨

Secondly, and independently, Chen et al. [2021] have presented Codex, a Transformer LM on code following the GPT-3 architecture, evaluating its synthesis performance on a new benchmark of simple programming problems. The main differences are in the specifics of the pre-training data, and in the way that we investigate the model's performance. First, the training set for our models somewhat oversampled web pages that contain code, such as programming question and answer sites (see Section 3), but unlike Chen et al. [2021], the results reported in this paper do not include a further fine-tuning step on a large corpus of open-source code. Second, while the HumanEval benchmark introduced by Chen et al. [2021] is nominally similar to our MBPP, there are some differences: A small difference is in the type of prompts; while the HumanEval dataset generally contains I/O examples of the desired functions, their number and formatting is not consistent, in a way that mimics docstrings of professional software. In contrast, our dataset consistently contains three I/O examples, written as assert statements. We also evaluate our models on the MathQA dataset, which is completely different in character. Third, we report synthesis results for models of size up to 137B. We find that even our general LM, without code fine-tuning, has non-negligible performance on few shot synthesis, and we find that fine-tuning that model on a very small (374 items) set of examples is already enough to dramatically improve performance on synthesis tasks. Fourth, and perhaps most interestingly, we analyze the extent to which our LMs can be used as interactive tools, and present results showing that humans can interact with these models to significantly improve

HumanEval은 보통 원하는 함수의 입출력 예제를 포함하지만, Docstring 포맷을 이용함
반면, MBPP는 assert문으로 작성되며 3가지 선택지가 일관적임

their success rate. Finally, in keeping with our goal to explore and understand the performance of general-purpose language models on this task, we also explore whether these models can evaluate the code that they generate, and whether they are equally effective at generating code that solves traditional mathematical word problems.

2 Datasets

We construct two new datasets: one entirely new and the other modified from an existing benchmark. The first, Mostly Basic Programming Problems (MBPP), is an entirely new crowd-sourced programming dataset. The second is derived from the MathQA dataset [Amini et al., 2019] but casts the problem solutions as short Python programs.

2.1 Mostly Basic Programming Problems

The Mostly Basic Programming Problems dataset contains 974 short Python programs constructed by crowd-sourcing to an internal pool of crowdworkers who have basic knowledge of Python. We asked crowd-sourcing participants to write a short problem statement, a single self-contained Python function solving the problem specified, and three test cases that check for semantic correctness of the function. Participants also provided a ground-truth solution that passes all three test cases. Participants were instructed to write descriptions concrete enough that a human would be able to translate them into code without clarifications. They were also instructed to write code that is self-contained (that is, it runs by itself) and that does not print any results to the console. Use of internet references was allowed.

The problems range from simple numeric manipulations or tasks that require basic usage of standard library functions to tasks that require nontrivial external knowledge, such as the definition of particular notable integer sequences. Figure 1 shows an example problem statement with the associated test cases and a sample from our largest model prompted with that problem statement. To further characterize the contents of the dataset, we randomly sampled 100 of the questions and assigned one or more descriptive tags to each question. Of these questions, 58% were mathematical in nature (e.g., calculating the volume of a sphere), 43% involve list processing, 19% require string processing, 9% involve integer sequences, and 2% center around the use of other data structures. We did not impose any restrictions on the number of lines of code in the reference solution. The average, median, and maximum number of lines of code are 6.8, 5, and 50 respectively. The natural language descriptions are typically short, usually one sentence each.

While inspecting the dataset, we observed that some questions used uncommon function signatures (such as passing in a list and its length as two separate arguments to a function), lacked detail, were somewhat ambiguous (e.g., "Write a python function to count the number of squares in a rectangle."), or performed unexpected operations in a function that were paired with the provided tests (e.g., casting a float to an int before returning it, with the test performing integer comparisons). Given this, we manually inspected, edited, and pruned a subset of the questions, yielding 426 hand-verified questions, which we refer to as the edited dataset. For each question in the edited dataset, we ensured it had a standard Python function signature, that it was unambiguous to a human, and that its test cases accurately reflected the text description. We conduct most experiments on the full dataset, but analyze the effect of the curation of the edited dataset in Section 4.9.

MBPP는 세부설명 부족, 모호함, 자주 안쓰는 함수 시그니처가 있음

이를 사람이 직접 수정한 426개의 수정 버전이 있음

In the experiments described later in the paper, we hold out 10 problems for few-shot prompting, another 500 as our test dataset (which is used to evaluate both few-shot inference and fine-tuned models), 374 problems for fine-tuning, and the rest for validation. For evaluations involving the edited dataset, we perform comparisons with 100 problems that appear in both the original and edited dataset, using the same held out 10 problems for few-shot prompting and 374 problems for fine-tuning. We have programmatically checked that all reference code passes all tests under Python 3.6, and we have open-sourced all of the problems.[1]

2.2 MathQA-Python

MathQA 데이터셋은
- 수학적 단어 문제 및 그에 대한 MCQA 문제
- 올바른 답을 생성하는 도메인 특화 언어 프로그램

으로 구성됨

Compared to the short natural language descriptions in MBPP, our second dataset is representative of a different kind of program synthesis task. The MathQA dataset [Amini et al., 2019] is a dataset where each data point consists of a mathematical word problem, multiple-choice answers for that problem, and a program in a domain-specific language that produces the correct answer. To evaluate whether pre-training on source code is useful for this task, we translate this dataset into a Python program synthesis dataset by translating the ground-truth programs from the domain-specific language given in the paper to Python code. We refer to the converted dataset as MathQA-Python. Compared to MBPP

도메인 특화 프로그램을 파이썬으로 변환함

[1] https://github.com/google-research/google-research/tree/master/mbpp

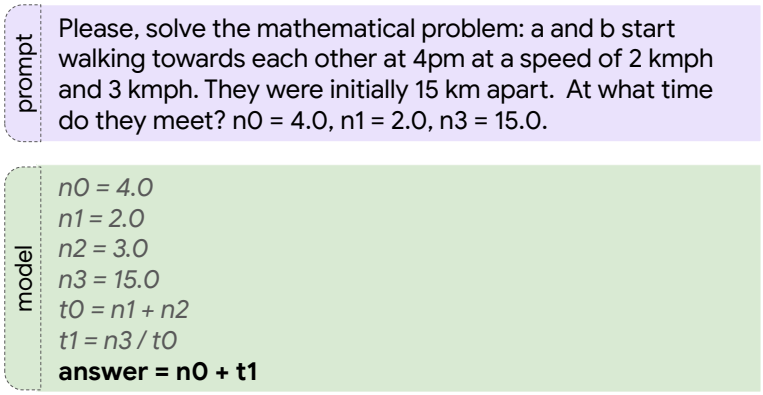

Figure 2: An example MathQA prompt along with a Python solution emitted by our largest model. Everything in purple is given as a prompt (along with some few-shot examples not shown). The equivalent DSL code is: `add(n1,n2)|divide(n3,#0)|add(n0,#1)`

MBPP는 루프, 조건문 등을 주로 포함,
MathQA-Python은 주로 직선적인 코드이지만 좀더 복잡한 자연어 Description으로 작성

which contains more usage of imperative control flow such as loops and conditionals, MathQA-Python contains mostly straight-line code, but more complex natural language descriptions. An example from this dataset is shown in Figure 2. Both the Python code and DSL code are used for fine-tuning and few-shot experiments. For the few-shot experiments, in the prompt we provide four examples of MathQA problems with their Python (or DSL) solutions. The model is tasked with returning Python or DSL code that computes the ground truth answer. We execute the sampled code to check for semantic correctness. This method of checking correctness forced us to filter the MathQA dataset to keep only those problems for which the code evaluates to the declared numerical answer, resulting in us removing 45% of problems. After this filtration we are left with 23914 problems, of which we use 19209 for training, 2822 for validation and 1883 for testing. The translation between DSL and Python is straightforward and we supply code that can be used to perform it.[2]

3 Model and Methods

The models we use in this paper are dense left-to-right decoder-only Transformer language models [Vaswani et al., 2017] trained on a combination of web documents, dialog data, and Wikipedia. Our experiments were conducted using models with non-embedding-parameter-counts ranging from 244 million to 137 billion. The pre-training dataset for the model contains 2.97B documents, which were tokenized into 2.81T BPE tokens with a vocabulary of 32K tokens using the SentencePiece library [Kudo and Richardson, 2018]. This data included web sites with both computer code and text, such as question and answer sites and tutorials, but source code files themselves were not specifically included, except where code appeared in other web sites. These web sites with code and text comprised about 13.8M documents containing 18.7B BPE tokens out of the pre-training data.

We test synthesis capabilities for both MBPP and MathQA-Python under two regimes: First, we use few-shot prompting as in Brown et al. [2020]. We hold out several example problems for the prompt and concatenate them, resulting in a longer version of the prompt seen in Figure 1 (or Figure 2 in the case of MathQA-Python). We then feed this prompt to the pre-trained model for completion. Second, we fine-tune the model on a training set. For MBPP, the training set is quite small (374 examples), so we fine-tune with a small learning rate (3e-5 for the largest model) for only 100 steps. For MathQA-Python, we fine-tune for longer. We generated the execution results using roughly analogous methods; see Section 6 for more details.

For all synthesis experiments, we measure functional correctness of the sampled code rather than some proxy for code quality like token accuracy or BLEU (see Section 4.7 for more about this). For the MBPP synthesis experiments, we check whether the code passes a set of test cases when executed (see Figure 1 for example test cases). For each problem in the test dataset, we use temperature sampling (with temperature 0.5) to generate 80 samples of code and then execute the code contained in the samples against tests for semantic correctness. The MathQA synthesis experiments are analogous.

[2] https://github.com/google/trax/blob/master/trax/examples/MathQA_Python_generation_notebook.ipynb

For the MBPP execution experiments, we check whether the model produces exactly the same results as executing the code. We use greedy decoding (temperature set to 0.0) to generate a single approximate most likely generation, and compare this to the string generated by executing the code.

4 MBPP Synthesis Results

Our primary results on MBPP are shown in Figure 3 and Figure 4. We show absolute performance and scaling behavior with model size for both few-shot (in the sense of Brown et al. [2020]) and fine-tuning across nearly three orders of magnitude. We find that samples from our models are able to solve a large fraction of the problems in the dataset, in the sense that the sampled code passes the three given test cases, and that synthesis performance scales approximately log-linearly with model size.

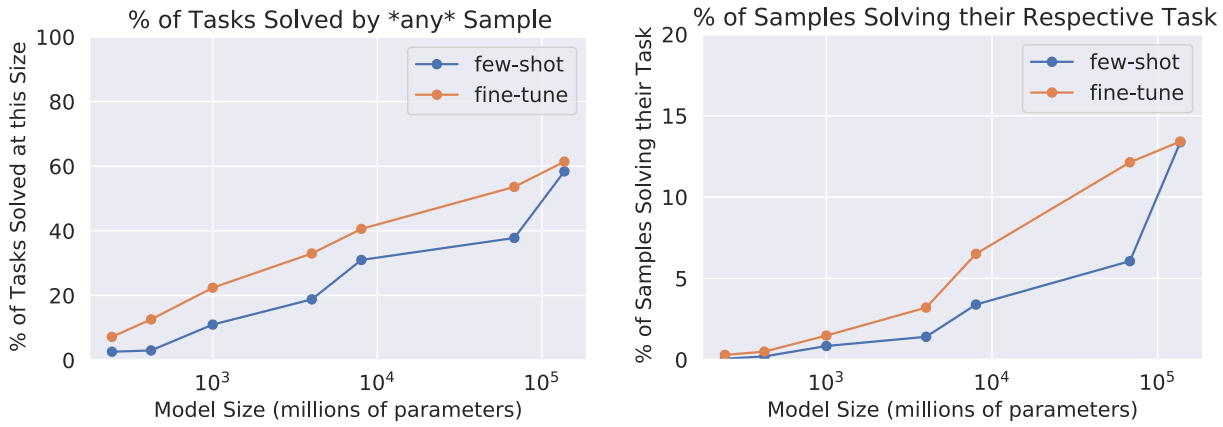

Figure 3: Performance vs model size, measured in two ways. (Left) Fraction of programs solved by *any sample* as model size is increased. This metric improves predictably as model size is increased, and fine-tuning gives a roughly constant improvement over few-shot prompting. The slope of the line shows no signs of decreasing for our largest models, which suggests that further performance gains can be had by making the model larger. (Right) Total fraction of sampled programs that solve a task, as model size is increased.

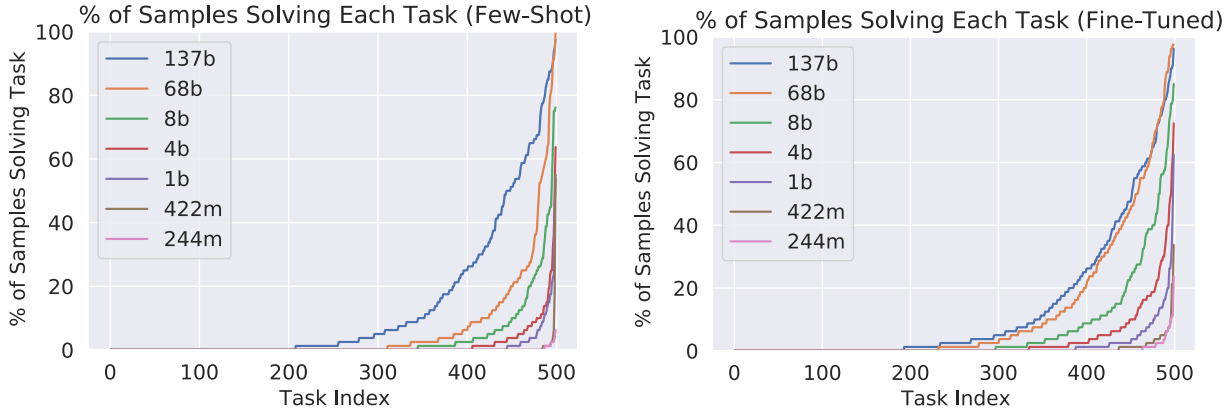

Figure 4: Fraction of samples solving each task. The x-axis represents the index of a particular task, sorted by the model performance. The y-axis represents the fraction of samples from the model that solved the task. In both cases, the curve is pushed "up and to the left" and the area under the curve increases as parameters are added to the model. This means that more tasks were solved by *any* sample, but also that bigger models can more reliably solve the "easier" problems. (Left) Results for few-shot prompting. (Right) Results for fine-tuned models. The gaps between models are more uniform for the fine-tuned results than for the few-shot results (which are noisy).

We measure performance as a function of parameter count in two different ways: the fraction of problems that are solved by *any* sample from the model and the fraction of samples that solve their respective problem. The fraction-of-problems metric is a natural notion of correctness, because if this model were to be used in practice, we could automatically filter out model samples that do not pass the test cases. The fraction-of-samples metric, by contrast, gives a sense of the overall reliability of the model. We find that performance according to the fraction-of-problems metric is quite predictable while performance according to the fraction-of-samples metric is less so.

We observe limitations on the types of problems these models can solve (some are simply unsolvable) and many solved problems tend to have only 1 or 2 (out of 80) samples which solve the task. We examine this and other limitations in later sections. We also find that our results are not strongly sensitive to the number of examples (asserts) shown to the model, but do depend strongly on the specific examples provided as prompts.

4.1 Synthesis Performance Improves as Model Size Increases

We measure synthesis performance at various model sizes, from 244 million parameters up to 137 billion. As explained above, performance is measured in two different ways: First we measure—for each problem independently—whether that problem was solved by *any* of the samples drawn from the model for that problem. Performance on this metric scales predictably with model size: the fraction of tasks solved is linear in the logarithm of the model size. The largest model can solve roughly 60 percent of the problems it sees given 80 samples. For this metric, fine-tuning seems to give a roughly constant boost in performance across model sizes. See Figure 3 (left) for more details. Second, we measure – across all samples generated for all problems – the fraction of samples solving their respective task. This corresponds to the area under the curve in Figure 4 and is depicted in Figure 3 (right). Performance on this metric improves as model size increases, but it scales up less predictably than does performance on the first metric. For this metric, fine-tuning tends to improve performance, but the relationship between fine-tuned performance and few-shot performance is much more variable as a function of model size than for the other metric.

Additionally, we analyze the types of errors made by the models: Figure 5 shows the breakdown of error types as a function of model size for the few-shot experiments. We define runtime errors as any errors (other than syntax or type errors) that cause the program not to produce a result. For most model sizes, runtime errors are more common than syntax errors; even the smallest models can write syntactically correct Python code around 80% of the time. However, type errors and other syntax errors do represent the majority of samples drawn from the smallest model. As model size increases, the frequencies of both run-time and syntactic errors decrease dramatically. For the largest (137B) model, over 63% of all failures are due to failing the test assertions, as opposed to run-time or syntactic errors.

4.2 Synthesis Performance is Insensitive to Number of Test Cases in Prompt

The example prompt in Figure 1 shows all three of the test assertions that the model output will be checked against. We measured whether including less than 3 of the assertions in the prompt would result in a serious drop in performance. Interestingly, it did not: the model with 3 asserts in the prompt solved only 3 extra problems compared to the model with 1 assert only. This suggests that the model is mostly not using those test cases to reason about semantics. More specifically, it also suggests that, even though we prompt the model with all three test asserts, the model is in general not "overfitting" to test-cases (though we explore exceptions in Section 4.5).

# of Prompt Examples	% of Problems Solved	% of Samples Solving Task
0	43.2%	10.23%
1	55.2%	15.30%
2	**59.0%**	15.14%
3	58.4%	**16.77%**

Table 1: Few-shot performance of the 137B parameter model as a function of number of prompting examples. The prompts for row zero only provide the function name. The bold text in the left column shows 59.0 instead of 59.6 because there is a small amount of run-to-run variance in the number of problems solved.

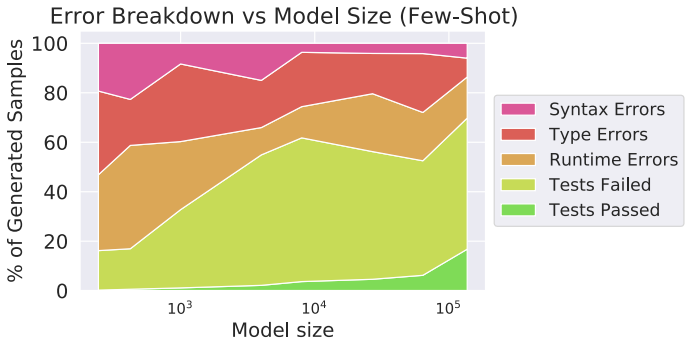

Figure 5: Breakdown of error type as a function of model size. The figure shows the breakdown of error type across all samples across all test tasks. 'Runtime errors' are defined as any errors (other than syntax or type errors) that cause the program not to produce a result. All error types decrease in frequency as model size increases.

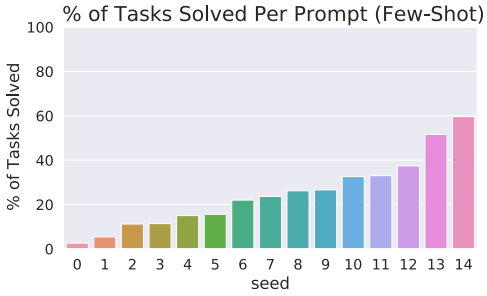

Figure 6: Performance as a function of which prompt examples are chosen, as measured by fraction of tasks solved by at least one sample. The seed label corresponds to the random seed used to choose which held-out examples are shown as prompts. Seeds are ordered by the fraction of tasks solved by that seed.

4.3 Performance is Sensitive to Prompt Examples

While model performance is not strongly sensitive to the number of test cases included in the prompt, few-shot performance is quite sensitive to the particular examples given in the prompt. We measure this sensitivity in Figure 6, where each seed corresponds to a particular, distinct choice of prompting examples. We find that while one set of examples (seed 14) is able to solve 60% of tasks, many other examples solve far fewer.

The large influence of these prompt examples is also noticeable qualitatively: failed synthesis attempts often include references to e.g. a data structure that was instantiated in one of those examples in the prompt. These results suggest that methods such as prompt-tuning [Li and Liang, 2021, Lester et al., 2021] could yield substantial performance improvements in this domain.

One failure mode for the poorly performing prompts is that they lead to long, repetitive samples. Often, very long prompts produce many samples that do not fit with the 512 token context window (even with a context window of 1024 tokens, this failure mode is still pronounced). Qualitatively, we notice that short prompts with compact examples that make use of external libraries lead to the best synthesis performance.

We also find that the set of problems solved with one prompt seed is not always a strict subset or superset of another: for example, seed 13 solves 19 problems (39, 62, 100, 168, 188, 206, 209, 233, 254, 315, 340, 365, 368, 382, 400, 434, 471, 474, 497) which are not solved by seed 14. Ensembling these prompts by counting a problem as solved if it is solved by any of the seeds boosts the percentage of problems solved from 59.6% to 66.4%.

4.4 Solutions Typically Generalize to Held-Out Test Cases

Consider task 11 from MBPP, which asks the model to *"Write a python function to remove first and last occurrence of a given character from the string."*. All of the solutions emitted by our best model pass all three test cases, but the test cases do not fully test the function's semantics (as shown in Figure 7).

None of the test cases use strings which contain more than than two of the specified character. Upon inspection, we realized that all of the sampled solutions would simply delete all occurrences of the specified character. To estimate how widespread this phenomenon was, we sampled 50 of the 500 test programs and wrote 'adversarial' tests cases for them. On those 50 problems, 33 had solutions solving all of the normal test cases, and 29 had solutions solving all of the normal test cases and all of the 'challenge test cases', for solution rates of 66% and 58% respectively. Thus, we can roughly estimate that something like 12% $((66-58)/66)$ of what we are counting as solutions (e.g. in Section 4.1) would fail to satisfy adversarially generated test cases. This is a nontrivial fraction, but it also means that almost 90% of solutions will generalize in the sense measured here.

asserts

```
# Normal test cases
assert remove_Occ("hello","l") == "heo"
assert remove_Occ("abcda","a") == "bcd"
assert remove_Occ("PHP","P") == "H"

# Challenge test cases
assert remove_Occ("hellolloll", "l") == "helollol"
assert remove_Occ("", "l") == ""
```

Figure 7: Test cases for Task 11. The normal test cases incorrectly allow a program that deletes all occurrences of the given character, rather than only the first and last. The challenge test cases exercise this corner case.

4.5 Programs Sometimes Overfit to Assert Statements

Very occasionally, the model will produce a solution that passes the test cases trivially by reading the assert statements and trying to hard-code an if-expression that passes them. For example, one of the problems asks for a function that checks if a given integer is a Woodall number (that is, a number belonging to the sequence $1, 7, 23, 63, 159, 383, 895, ...$). This problem includes three asserts (see Figure 8), only one of which specifies a number that is actually a Woodall number: 383. The model simply emits a program that returns True if the input is 383 and False otherwise, which is not correct.

단순히

모델이 단순히 assert문을 읽고,
통과하는 함수를 하드코딩하는 경우가 있음

prompt

```
You are an expert Python programmer, and here is your
task: Write a function to check if the given number is
woodall or not. Your code should pass these tests:

assert is_woodall(383) == True
assert is_woodall(254) == False
assert is_woodall(200) == False
```

model

```
def is_woodall(n):
  result = False
  if n == 383:
    return = True
  return result
```

Figure 8: In rare cases, the model generates a program which trivially passes the test asserts but does not solve the problem. This program does not correctly check if the given input is a Woodall number, it simply returns true if the input is 383.

This is interesting and perhaps somewhat alarming, though it also highlights that the model does have some abstract notion of the relationship between the test cases and the generated program. We can infer from the results in Section 4.2 and 4.4 that this "overfitting" to the test cases is not a widespread problem.

4.6 Sampling Strategy is Critical for Good Performance

Since tests or input-output examples can be machine checked, it is standard [Devlin et al., 2017] for synthesis algorithms to generate and evaluate many samples (often even enumerating and checking all possible programs). We investigate the scaling performance of our largest model with the number of samples evaluated across different sampling strategies: temperature sampling with varying temperatures and beam search. Figure 9 shows the number of tasks solved by the 137B model as the number of samples increases. We find that lower temperatures (more greedy decoding) perform better with only a single evaluation allowed, but higher temperature, less greedy strategies begin to solve more tasks within a budget of 10 samples. We also find that beam search performs extremely poorly, worse than any of the temperature

settings considered – empirically we found this was due to beam search often producing results that looped or repeated lines of code endlessly.

4.7 Synthesis Performance Correlates Poorly with BLEU Score

As noted by Hendrycks et al. [2021] and Chen et al. [2021], we find that BLEU score between generated samples and reference programs does not correlate well with synthesis performance. Figure 10 shows two curves: the fraction of samples which solve a given task and the average BLEU score of samples compared to the reference program. We find little correlation between the two. This can be explained by the fact that semantically identical programs can potentially have very low n-gram overlap; for example, because of identifier renaming.

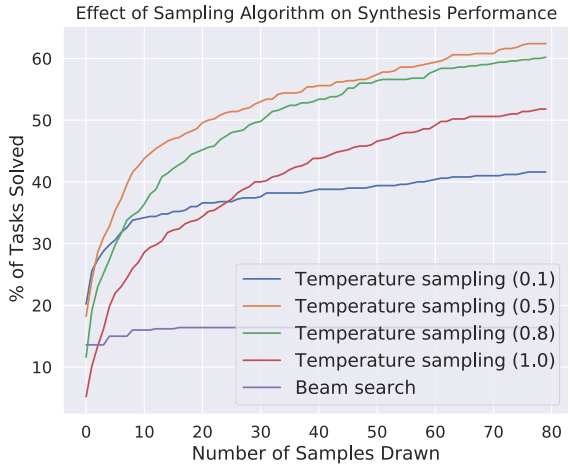

Figure 9: Higher temperatures achieve better scaling with more samples, but perform worse with a smaller budget.

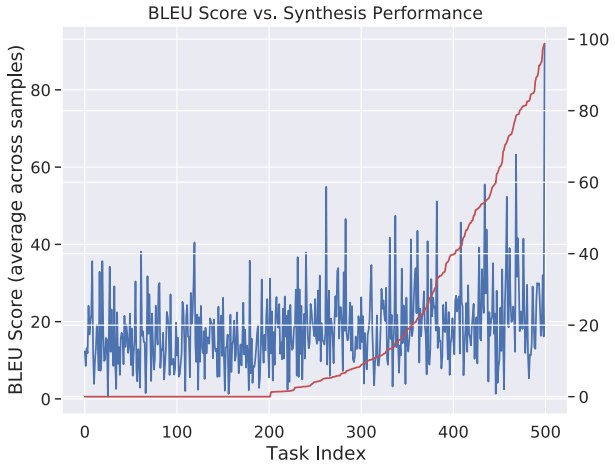

Figure 10: Comparison of BLEU score and synthesis performance for the 137B parameter model. No strong correlation is observed.

4.8 Pre-train / Test Overlap Seems to be Minimal

A common concern about results on large language models is that the models are likely to have seen something substantially similar to the test data in their very large training set, causing the test accuracy to overstate performance in practice [Brown et al., 2020]. Even though we created this dataset from scratch, it is still possible that this is an issue for some tasks for two reasons. First, some tasks are very simple (e.g. 'reverse a string') and so surely will be represented in the training data in some way. Second, crowd-sourcing participants may have made use of reference materials from the internet that could also have appeared in the pre-training dataset for our models.

To quantify this concern we investigated how many lines of code appear in both the training data for our models and the ground-truth programs for the Mostly Basic Programming Problems. We examined each document in the pre-training data (excluding non-English documents and conversational data) and counted the number of lines that overlap with the ground-truth program for each problem. We then found the document with the most matching lines and the number of matching lines in MBPP. We stripped whitespace at the beginning and end of the line. We excluded lines from this analysis which appear more than twice anywhere in MBPP, as these are likely to be common Python keywords such as `return` or `continue`.

Figure 11 contains a visualization of the results. Broadly speaking, there was not much overlap. Only 32 of 974 problems (3.3%) had more than half of their lines matched somewhere in the pre-training data and 91.5% had only one or two lines of code that overlapped with the training set.

4.9 Comparing Performance Between the Original and Edited Questions

As described in Section 2.1, we created a subset of the larger MBPP dataset consisting of questions that were manually inspected and edited for consistency. We then ran experiments on 100 questions that appear in both the original dataset

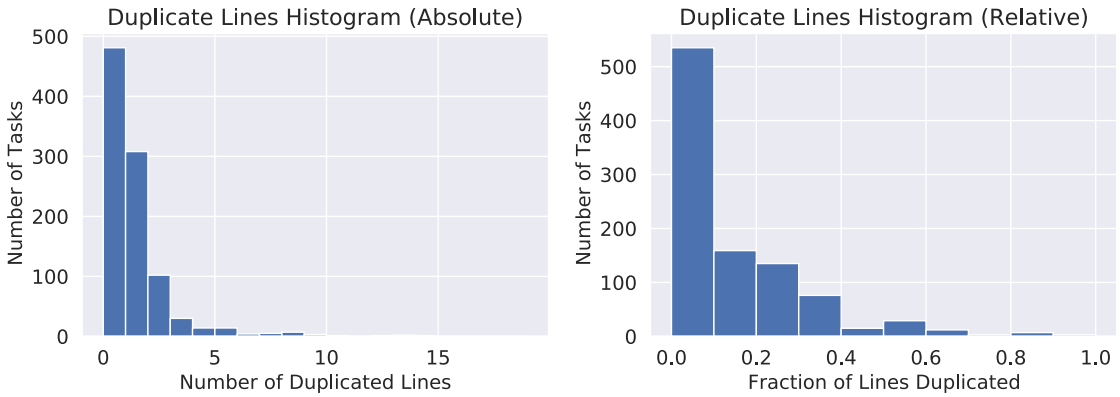

Figure 11: Number of lines of code that appear in both the pre-training data and in the python programming dataset. The left chart shows the absolute number of lines and the right chart shows the relative number of lines, as a percentage of the lines in the ground-truth program.

Model Size	Edited?	% of Problems Solved	% of Samples Solving Task
8B		35%	4.46%
8B	✓	**45%**	**7.36%**
68B		48%	8.02%
68B	✓	**61%**	**12.95%**
137B		63%	20.78%
137B	✓	**79%**	**31.85%**

Table 2: Performance comparison between original and manually edited dataset on 100 problems.

and this edited dataset. In this set of 100 questions, 56% of the questions' text was edited, 30% of the test cases were edited, and 71% included edits to either the questions or test cases. Using this dataset, we ran experiments using few-shot prompting for models with 8B, 68B, and 137B parameters.

Table 2 summarizes model performance on the original and edited dataset. As can be seen, model performance increases when using the edited dataset for each experiment. Table 3 shows that 16-19% of the problems can be solved using one dataset, but not the other, across model sizes. Within this same subset of problems, for 81-100% of the problems, the model is able to produce a correct solution using the edited version of the question, rather than with the original version (across model sizes tested). However, within this subset of questions, 12-31% had no differences in either the question text or test cases for the three model sizes, indicating general variability in model performance.

We manually examined each of the 38 problems for which model responses (on the sanitized and unsanitized data) were not both right or both wrong. In these 38 problems, 15 included edits to the problem text, but not the test cases, 7 problems included edits to the test cases but not the problem text, 7 included edits to both the problem text and test cases, and 9 problems had no edits to either the problem text or test cases.

For the 15 problems whose problem text was edited, but had no changes to the test cases, 11/15 included more detail in the problem text (e.g., specifying that a list should be flattened and summed, where the "flatten" detail was previously omitted). 4/15 of the edits included details related to the function's signature (e.g., specifying that a list of lists should be returned), 2/15 removed requirements (such as the requirement to use a regex in the solution code), and 2/15 rewrote the problem text. For the seven problems with edits to both the problem text and test cases, 5/7 included more details and 2/7 added details related to the function's signature.

For the 7 problems with differences in the test cases, but no differences in the problem text, 3/7 edited test cases modified the problem's function signature (e.g., changing it to return a list rather than a string representation of a list), 2/7 problems attempted to perform comparisons between floating point numbers directly (rather than testing whether the numbers were approximately equal), one set of test cases tested floating point equality for a function that returned integers, and one problem added an additional test case. For the seven questions with edits to both the problem text and test cases, 3/7 changed the function signature of the test, 2/7 created a more robust test (comparing sets rather than lists,

Model size	Problems solved in exactly one dataset	Of problems solved in exactly one dataset: Solved in edited dataset	Solved with no edits to text or test cases
8B	16%	81%	31%
68B	19%	84%	16%
137B	16%	100%	12%

Table 3: Statistics of problems that could be solved in only one of the edited versus the original MBPP datasets. When a problem can be solved in one dataset and not the other, it is more likely to be solved in the edited dataset compared to its original formulation.

when order was not important for a function returning a set of unique values), 1/7 corrected floating point comparison issues, 1/7 fixed an error in a test case, and 1/7 added a test case.

In general, these observations suggest the importance of specificity and details in the natural language request sent to the model, with more details seeming to lead to a higher likelihood of the model being able to produce correct code (as might be expected). Having a function signature that matches conventions also seems to be important (which is also expected).

4.10 Qualitative Analysis of Error Modes

To deepen our understanding of model behavior and error modes, we conducted a qualitative error mode analysis by examining hand-verified problems for which most samples were incorrect, culminating in several themes (Table 4).

Problems with multiple constraints or sub-problems: First, difficult problems (as measured by model performance) often had multiple constraints or multiple implicit sub-steps. For example, the question *"Write a function to find the maximum difference between the number of 0s and number of 1s in any sub-string of the given binary string"* involves not only counting 0s and 1s, but also finding substrings. Likewise, *"Write a function to find the longest palindromic subsequence in the given string"* requires both finding palindromic subsequences and determining the longest one. In contrast, easy problems tended to be shorter and more atomic (e.g. "Write a python function to find the sum of an array."). In multiple-constraint problems, the model often generated a partial solution that addressed only a sub-component of the problem. For instance, in the digits example above, one model solution correctly counted 0s and 1s but did not do so over all substrings. In the palindrome problem, the model merely recorded indices of mirror-imaged letters, but did not use those indices to find palindromic subsequence and did not write logic to find the longest one. This suggests that the model may struggle more with complex, multi-part problems that combine many atomic operations.

Problems with more-common siblings: Relatedly, some low-performing problems appeared to have variants that are more common, resulting in the model solving a more common version of the problem. For example, given the problem *"Write a python function to find the largest number that can be formed with the given list of digits."*, the model found the largest number among the list of digits, rather than the largest number that can be formed from them. A similar error occurred when a complex problem asked for the *"maximum difference"* but the model computed the *"maximum"* instead. Given the plethora of problems on the internet that involve finding the largest number from among a list, this model behavior is perhaps not surprising. However, given that the model may latch onto keywords found in ubiquitous programming problems, this does pose a unique challenge for the long tail of problems that may be closely related to (or have keywords in common with) typical programming problems. We might consider these types of errors "linguistic off-by-one" errors, where a small change in words might lead to a large semantic difference.

Miscellaneous errors: Other miscellaneous error patterns included difficulty solving advanced math problems (e.g. *"Write a function to find the nth hexagonal number"*), producing incomplete skeleton code rather than the code itself, or a failure to apply common-sense reasoning (e.g. *"convert a list to a tuple"* led the model to convert each item of the list into a tuple).

Table 4: Qualitative analysis of highest- and lowest-performing problems

	Theme	Examples

Highest-performing problems	Single operations	`Write a function to remove all whitespaces from a string.`
		
		`Write a python function to find the maximum of two numbers.`

	Common "coding interview" type questions	`Write a function to merge multiple sorted inputs into a single sorted iterator`

Lowest-performing problems	Problems demanding multiple constraints or multiple sub-problems	`Write a function to find the maximum difference between the number of 0s and number of 1s in any sub-string of the given binary string`
		(Sub-problems: count 0s and 1s, find difference, find max across all sub-strings)
		
		`Write a function to find the longest palindromic subsequence in the given string`
		(Sub-problems: keep track of mirror-imaged letters, find palindromes, find longest one)

	Problems that have a more-common sibling with similar keywords	`Write a python function to find the largest number that can be formed with the given list of digits.`
		(Model solves more-common problem: finds the largest number among the list of digits)
		
		`Write a python function to reverse only the vowels of a given string.`
		(Model solves more-common problem: finds all vowels in the string)

	Specialized math problems	`Write a function to find eulerian number a(n, m).`

5 Human-Model Collaboration Results

While large language models are impressive program synthesizers in some respects, they are far from being able to reliably solve difficult engineering problems without intervention. This raises the question of whether these models can be useful as interactive tools that can refine or correct their predictions in response to user feedback. We are specifically curious about two possible forms of collaboration:

- Whether a human and model together are able to solve tasks that are challenging for either alone.
- Whether human feedback can help a model refine its outputs, especially in the presence of initially ambiguous or under-specified requests.

In this section, we conduct preliminary experiments to measure the extent to which these forms of collaboration are possible. For concurrent work that addresses these topics, also see Jiang et al. [2021].

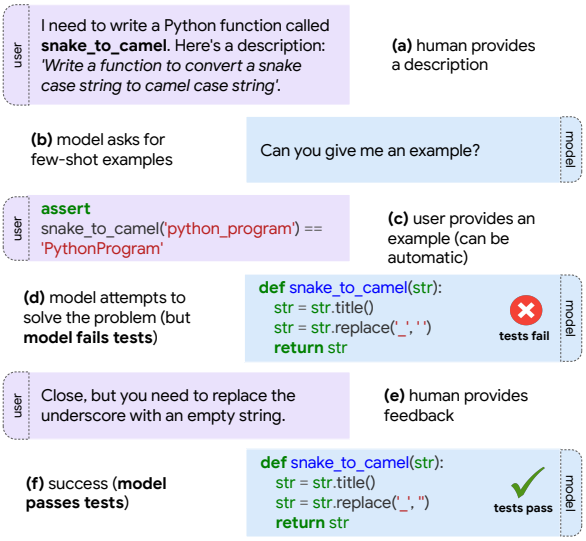

Figure 12: An overview of the "flow" of the human-model collaboration experiments. The human gives a description of the desired program and then guides the model toward the correct solution via dialog.

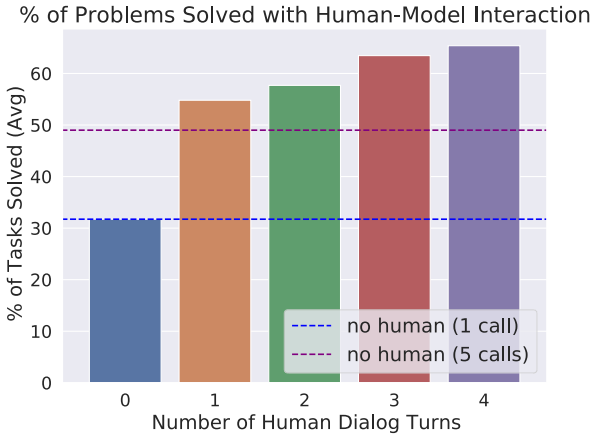

Figure 13: Percent of problems solved as the number of human dialog interventions increases. With 4 interventions, the solve rate increases from 30% to over 65%. Except for the purple horizontal baseline (which corresponds to 5 samples from the model), all pass-rates in this figure were computed using a single sample from the model.

5.1 Human Feedback Improves Synthesis Performance

We select 50 problems from the edited MBPP dataset (see Section 4.9) and assign human participants to collaborate with the model to solve these tasks using only natural language dialog. The model is prompted as in the experiments in Section 4, but with few-shot examples and priming asserts given as dialog turns: for instance *"I need to write a function called [function name]. Here's a description: [docstring]."*. The model sees several examples of collaboration in this few-shot prompt, after which the dialog with the participant begins. Participants were instructed to provide one-sentence hints or corrections to the model that would guide the model toward finding the correct solution. The hints were allowed to contain references to Python identifiers, but not large snippets of code, and participants were limited to a maximum of four hints. The full set of instructions given to participants can be found in Appendix A.2.

The results of this experiment (Figure 13) support the hypothesis that these models can improve or correct code based on human feedback. Counting all four dialog turns, the fraction of problems solved is increased from 30% to over 65%, and counting only one, from 30% to 55%. The purple horizontal line in Figure 13 corresponds to the fraction of problems solved when the model is allowed to use five samples instead of 1, so there is a sense in which a one-sentence human correction is worth more than a five-fold increase in the number of samples allowed. Furthermore, human feedback allows the model to solve 10 problems that it was totally unable to solve without human assistance. There are, however, diminishing marginal returns to human feedback, as might be expected.

Figure 14 shows two example interactions with the model which allowed it to solve previously unsolved problems. In the first, a human was able to point out mistakes the model had made as a result of an under-specified natural language prompt (mistakes a human was able to infer by looking closely at the assert statements). In the second example, the model predicts an overly complicated solution, which a human is able to tweak and correct over a number of follow-up turns.

5.2 Qualitative Analysis of Human-Model Dialogs

To gain a better understanding of how useful large models can be as assistive tools, we conducted a qualitative analysis of success and failure modes using the interactions collected for the above experiment, resulting in the following broad themes:

Humans are able to clarify under-specified prompts by examining test cases. Many problems do not precisely specify every detail of the semantics of the desired program. For example, one question in the original dataset asks

Figure 14: Two example human-model interactions. User text is purple and model text is blue. Left: an under-specified problem in which the user was able to point out corrections based on the example input. Right: a longer and more complex example in which the model makes small adjustments in response to feedback. Few-shot prompting examples are elided for compactness. Additional examples are shown in the appendix.

the user to *"write a function to count the most common words in a dictionary"*, but the test cases make clear that the function should only return the 4 most common words, and in descending order by frequency. The model, even when given these test cases, was unable to 'understand' those requirements. A human was able to tell the model to sort the output, reverse its order, and return only the top 4, which allowed the model to solve the problem. This interaction is shown in Figure 14 (Left).

Humans can often correct small context errors (often import and identifier errors). The model also frequently makes import or identifier errors, for example by forgetting to import a module it used in its code. Typically, a single dialog turn was enough for humans to help the model correct these errors (for example, by saying *"Great, but you never imported the re module."*). Humans also tended to help the model fix variable misuse errors (in which, for instance, an un-defined variable is referenced) in one turn. We also observed the model returning True instead of False, which a single dialog turn could correct.

The model can lose track of context or previously referenced code. We observed several cases where the model modified its code in an incorrect way in response to user feedback, but struggled to revert it or incorporate pieces of prior results. For instance, it rarely responded well to *"No, please go back to the previous response."* or *"Great, but you need to use the function signature from your first response."*. This problem became more pronounced as the number of dialog turns increased.

6 Program Execution Results

A common criticism of language models like the ones we use in this paper is that they learn statistical correlations between tokens without an underlying world-model or mechanism for systematic reasoning, and therefore do not understand the *meaning* of what they are describing [Bender and Koller, 2020]. On the other hand, recent work has provided evidence that, in some natural language contexts, pre-trained Transformers are able to implicitly construct approximate representations of the semantics of the worlds they describe in text [Li et al., 2021]. We would like to ask a related question for code: Do pre-trained language models *understand* the underlying semantic state of the code they are synthesizing? Computer programs are an especially promising domain for this kind of analysis, because unlike natural language, the semantics of a program can be defined precisely, by specifying the result of its execution.

In this section, we investigate to what extent our models have this understanding by asking them to predict the result of executing the ground-truth programs from MBPP on test inputs. We also study how this execution ability is related to synthesis performance.

We are specifically interested in the following questions:

- Can models execute Python functions, and how does execution performance depend on what information is in the prompt?
- How does fine-tuning on execution tasks impact the performance of execution?
- How does fine-tuning on execution tasks impact the performance on synthesis tasks?

In asking these questions, we are inspired in part by previous work that specifically trains deep architectures to learn how to execute programs [Zaremba and Sutskever, 2014, Bieber et al., 2020]. In contrast to that work, our goal is to use the learning-to-execute problem as a lens to understand the capabilities of large language models over source code, rather than to obtain the best model of program execution per se. To answer these questions, we conduct a series of experiments, focusing on our largest model (137B).

6.1 Few-Shot Execution Performance is Poor

In our first experiment, we evaluate the few-shot performance of our 137B model on code execution. For each task, the MBPP dataset contains ground truth source code, a natural language description, and three input-output examples. We task the model with predicting the output of the ground truth program if it is run on one of the given test case inputs. Since this performance might be sensitive to details of the prompt, we investigate how execution performance depends on that information. Specifically, we ablate the presence or absence of the ground truth code, natural language description, and test cases in the prompt. This results in seven different prompt configurations, as shown in Table 5.[3] The prompt templates for each prompt condition can be found in Listing 1 in the Appendix. We query models using a sampling temperature $T = 0.0$, which is equivalent to greedy decoding.

In our first set of experiments (Table 5, left), we evaluate correctness on a single test case. For prompt configurations requiring test cases, we use the two remaining test cases. Overall execution performance is relatively poor, with accuracy never exceeding 29% for any prompt type. Results indicate that including test cases in the prompt seems to help more than any other individual component. Including test cases *and* natural language descriptions in the prompt lead to the highest overall performance—higher than using the code itself. Because the code unambiguously describes the semantics, whereas test cases do not, this suggests that models are in some sense not really "reading" the source code and using it to execute. Models trained on general text corpora may be better at inducing patterns from as few as two input-output examples than they are at predicting the execution of code.

Evaluating on only one test case might provide a noisy overestimate of functional correctness. Therefore, our second set of experiments (Table 5, right) investigates whether the models can correctly infer the output for multiple test cases. For this experiment, we only judge a sample correct if it gives the correct output for both test cases. Accuracy for testing on two examples is lower than for one example. For the prompt configurations in Table 5 that do not include test cases, the prompt does not change between this experiment and the last one, so the drop in performance across these configurations must be due to the model failing to generalize across test-inputs when predicting the execution result.

[3] We note that the prompt types which do not contain source code should probably not be referred to as *execution* tasks; for example, the case where only input-output examples are included is equivalent to what has been dubbed "neural program induction". [Devlin et al., 2017]

Table 5: Execution results as information in the prompt is varied. Left: Testing on 1 held-out test case. Prompts with test cases contain 2 of them. Right: Testing simultaneously on 2 held-out test cases. Prompts with test cases contain a single one. Across multiple configurations, execution performance is greatest when the prompt contains test cases. Furthermore, fine-tuning increases accuracy for code execution, but this effect appears to be washed out by the presence of test cases in the prompt.

	2 prompt examples, 1 test example		1 prompt example, 2 test examples	
	Few-shot	Fine-tuned	Few-shot	Fine-tuned
code	16.4%	20.8%	8.6%	9.0%
code+NL desc.+examples	24.6%	23.2%	9.8%	8.4%
code+NL desc.	15.6%	20.6%	9.0%	8.2%
code+examples	**28.8%**	**27.4%**	11.6%	12.0%
NL desc.+examples	**28.6%**	**28.2%**	**12.8%**	**13.0%**
NL desc.	17.6%	18.8%	8.4%	8.6%
examples	27.2%	26.2%	10.2%	**13.0%**

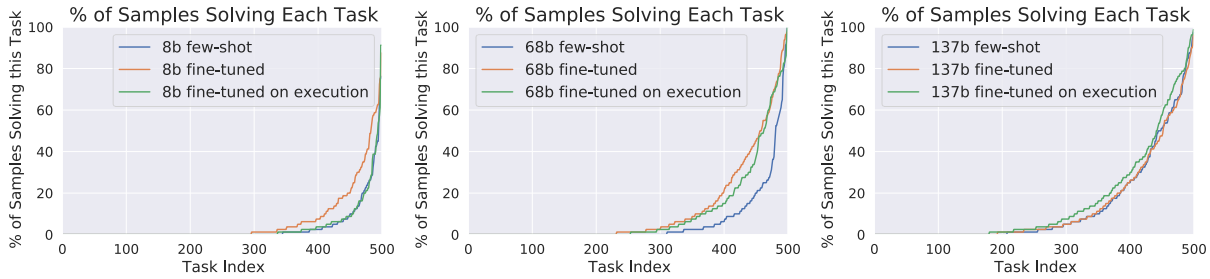

Figure 15: Synthesis performance of models fine-tuned on the execution task. While synthesis performance of the 8B model is not improved by fine-tuning on the execution task, the 137B model achieves slightly better synthesis performance when fine-tuned on execution, suggesting that larger models may be better able to transfer knowledge from execution training to synthesis evaluation.

6.2 Fine-tuning on Execution Slightly Improves Execution Performance

To test the effects of fine-tuning on execution performance, we construct a fine-tuning corpus for execution, built using the 374 training and 90 validation tasks used for synthesis fine-tuning (Section 2.1). For each task, we include an execution trial for each of the 7 prompt configurations from Section 6.1. We also vary the number of test cases in the prompt and test cases to test on (also as in Section 6.1). This gives a total of 14 related data-points for each task. Overall, this fine-tuning corpus consists of $14 \times 374 = 5236$ training data-points and $14 \times 90 = 1260$ validation data-points. We fine-tune for 100 steps using a batch size of 8192 tokens per batch.

Our fine-tuning results are shown in Table 5. Our results indicate that fine-tuning improves the performance of code execution, but that this improvement is not present when test cases are part of the prompt. In particular, there is a positive difference between fine-tuned and few-shot performance only for prompts which contain source code but do not contain test cases.

6.3 Fine-tuning on Execution has a Small Effect on Synthesis Performance

We also investigate how models fine-tuned on execution perform on the program synthesis task which is the main focus of this paper. We perform the few-shot program synthesis evaluation from Section 4 on the models fine-tuned on execution from Section 6.2 above. As in Section 4, we perform few-shot prompting with $k = 3$ example synthesis tasks in the prompt, and include all three example asserts for each task.

We perform this experiment using the 8B, 68B, and 137B models (Figure 15). For the 8B model, fine-tuning on execution prompts does not increase performance beyond the few-shot performance. Performance of the 137B model shows a small improvement when fine-tuned on the execution dataset, of about 2.3% more samples per problem solving

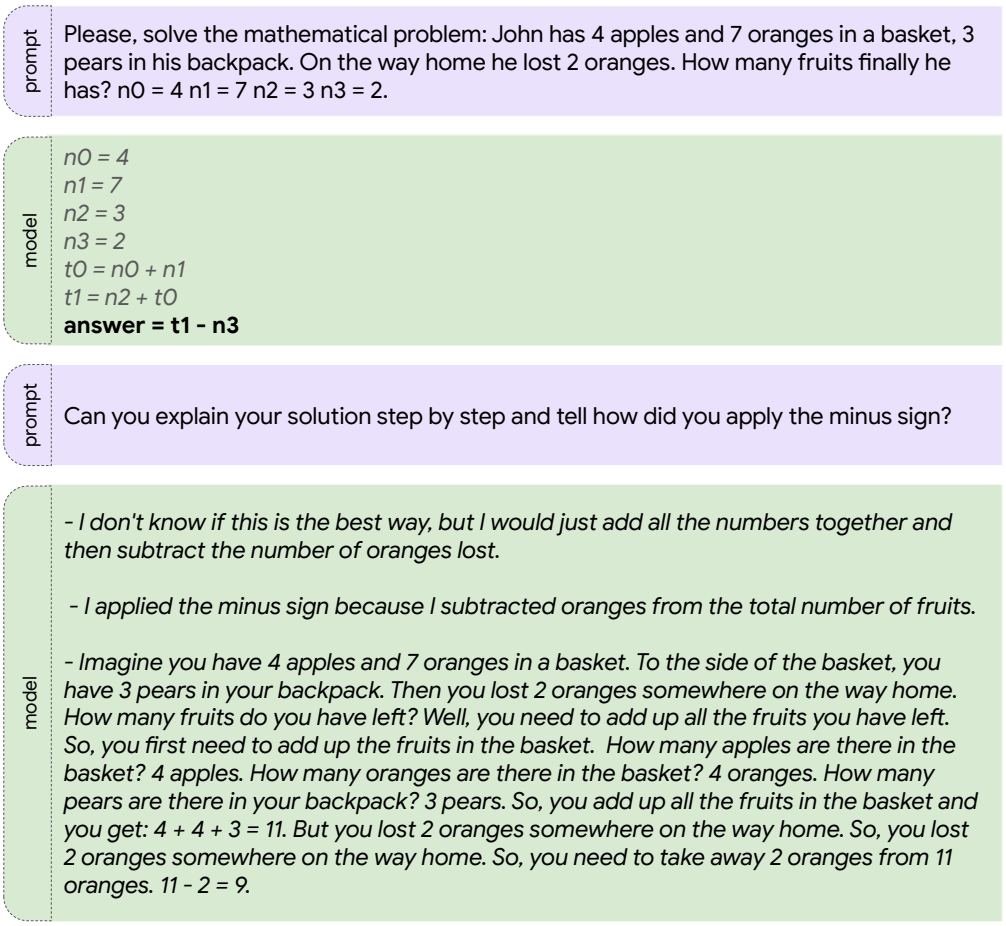

Figure 16: An example of a simple MathQA-style problem used as an additional test. We first verified that the model can produce a correct solution. Next, given a correct solution, the model was asked "Can you explain your solution step by step and tell how did you apply the minus sign?". The three responses at the bottom of the listing are independently sampled from the 137B model, fine-tuned for the Python MathQA task. The first two are correct, whereas the final response descends into mathematical rambling.

the task and 3.6% more tasks solved by any sample, compared to fine-tuning on the synthesis dataset). We suspect that training on more detailed execution data [Zaremba and Sutskever, 2014, Bieber et al., 2020] may further improve performance.

7 MathQA Results

We also evaluate our models on the MathQA and MathQA-Python datasets. The code in the MathQA dataset is different from MBPP, making less use of control flow and of the Python standard library, while the natural language is more complex. We experiment with both the domain-specific-language of the formulas in the original MathQA dataset, which we call MathQA-DSL, and the MathQA-Python dataset described in Section 2.2. As on the MBPP data (Section 4), we evaluate synthesis performance in both the few-shot prompting and the fine-tuning setting. We report accuracy in terms of functional correctness, that is, whether the program output by the model returns the correct answer to the word problems.

The results are summarized in Table 6. We find that the few-shot accuracy is 33.4% for the 137B model on the Python-formatted dataset. The fine-tuned models achieve very high accuracy: the best-performing model (137B on the DSL-formatted dataset) achieves 83.8% accuracy; see Table 6. Further, as with MBPP we can interpret the percentage

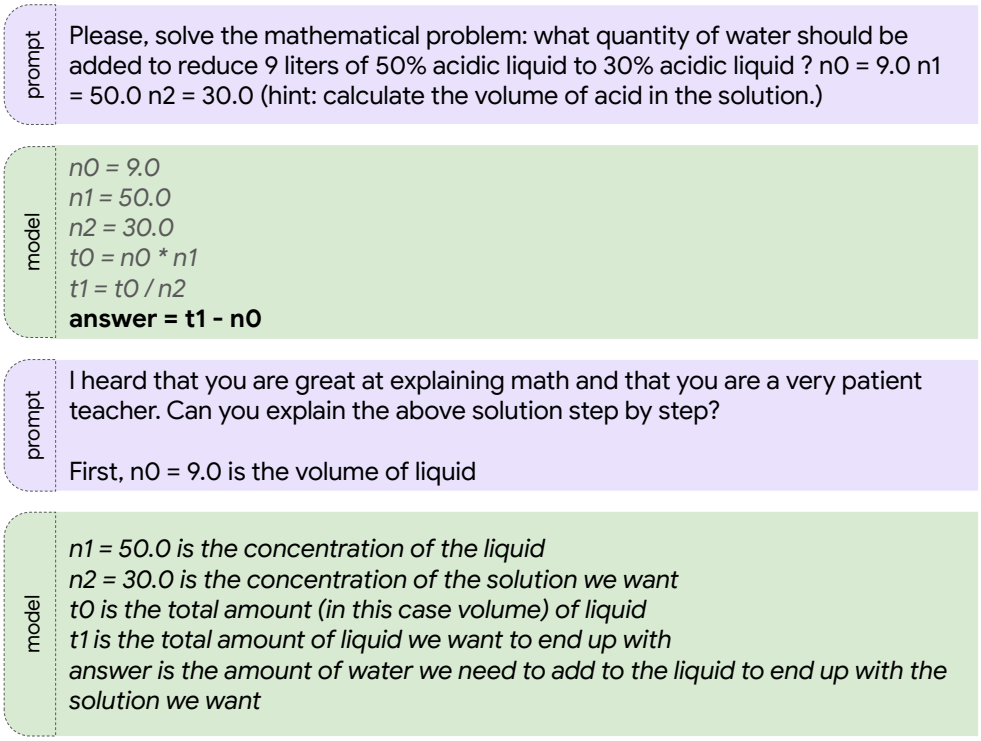

Figure 17: An example of a harder MathQA test problem. Without the parenthesized hint, it is solved by the 137B model in less than 10% of samples. With the hint, it is solved 40% of the time. Correct prompting can elicit a line-by-line explanation of the solution. The model answers are marked in color. Notice that the model only indirectly follows the hint and that the explanation with regard to $t0$ is incorrect: $t0$ *divided by* 100 is the volume of acid in the original solution (in litres). Explanations were obtained in a zero-shot mode and they contain various inaccuracies.

of samples solving each task as a measure of the model's confidence in its predictions. In Figure 18, we see that the finetuned models tend to have higher confidence, and the few-shot models much less so.

The few-shot models perform better on MathQA-Python compared to MathQA-DSL, which is expected because the MathQA DSL is unlikely to be similar to anything in the pre-training set. In contrast, the fine-tuned models achieve slightly higher accuracy on the DSL-formatted dataset compared to the Python-formatted dataset, indicating that the fine-tuning dataset we use has sufficiently many examples for the model to overcome its lack of familiarity with the DSL. This has promising implications for tasks like trying to teach a new programming language to a pre-trained model.

We also conducted an initial qualitative exploration of whether the model could respond to hints and explain its reasoning. Figure 16 shows an example for which the model is capable not only of solving MathQA-style problems, but also of carrying on a dialog about the proposed solution. Figure 17 shows how providing a hint to the model can in some cases increase the fraction of samples that solve the problem. Namely, without the hint ("calculate the volume of acid in the solution"), the 137B model fine-tuned on the Python code was able to solve the problem in fewer than 10% of samples. With the hint, the model samples correct answers 40% of the time. Moreover, we can elicit a line-by-line explanation of the solution with appropriate prompting (see blue section in Figure 17). Though we think these results are promising, we do not claim to have done a thorough evaluation of them here. They are presented more as a jumping-off-point for future work.

Table 6: MathQA accuracy for 8B, 68B and 137B models, measured by the percentage of tasks on the test set that are solved by any sample. Fine-tuning greatly increases performance for both the original DSL and the Python variant of the dataset. The gap between few-shot and fine-tuning performance is much larger for MathQA than for MBPP, but this is to be expected, because the fine-tuning dataset for the former is much larger.

	MathQA-DSL		MathQA-Python	
	Few-shot	Fine-tuned	Few-shot	Fine-tuned
8B	16.5%	79.0%	12.5%	74.7%
68B	**16.8%**	82.8%	22.3%	79.5%
137B	16.7%	**83.8%**	**33.4%**	**81.2%**

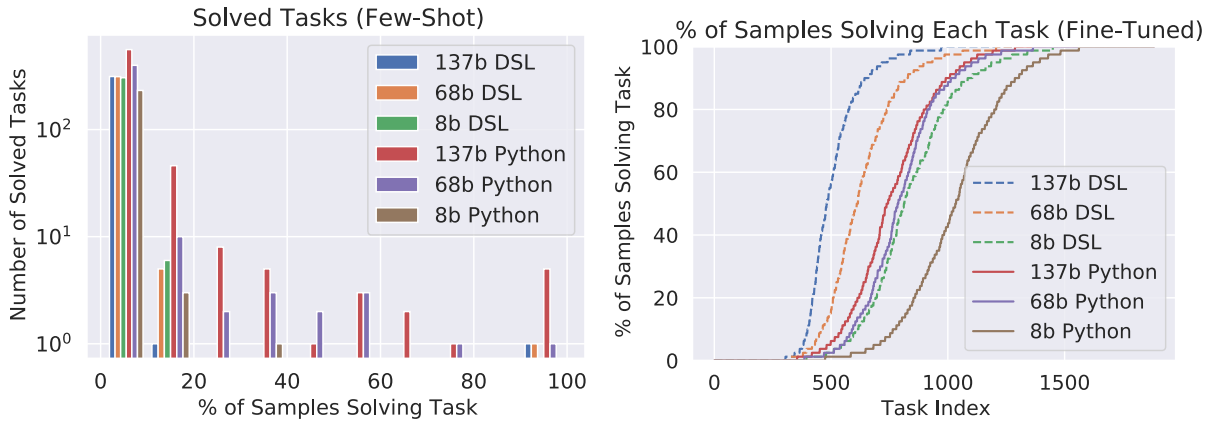

Figure 18: Fraction of samples solving each MathQA task represented as a histogram and a graph. In the case of the histogram each bucket shows the number of test tasks solved by the model (out of the total of all 1883 test tasks). The x-axis shows buckets $[1, 9]$, $[10, 19]$, $[20, 29]$, ... that refer to the percentage of samples solved by the model. In particular tall bars in the first bucket $[1, 9]$ mean that for majority of tasks only between 1 and 9 percent of samples were correct. In the figure on the right the x-axis represents the index of a particular task and the y-axis represents the fraction of samples for that task that actually solved the task. Though curves in this figure are substantially different then the ones in analogous Figure 4, the conclusion remains the same: the area under the curve increases as parameters are added to the model. This means that more tasks were solved by *any* sample, but also that bigger models can more reliably solve the "easier" problems.

8 Related Work

Our work is inspired by the long line of previous work on neural language models of natural language text [Mikolov et al., 2010, Sutskever et al., 2011, Józefowicz et al., 2016, Dai and Le, 2015, Peters et al., 2018, Howard and Ruder, 2018], especially recent large Transformer models [Radford et al., 2018, Brown et al., 2020].

8.1 Machine Learning for Program Synthesis

In the long history of program synthesis, methods have included deductive approaches, approaches based on enumerative and stochastic search, and constraint solving; for surveys, see Gulwani et al. [2017], Solar-Lezama [2018]. One important application of these methods has been in end-user programming, for example, to synthesize string manipulation programs in spreadsheets [Gulwani, 2011]. Many current systems rely on reducing the synthesis problem to a satisfiability problem, for example, Solar-Lezama et al. [2006] and Torlak and Bodik [2013].

Machine learning methods for program synthesis aim to learn cues from the problem description or from corpora of existing programs that help to write programs. Balog et al. [2017] use a neural network to predict properties, such as which functions will be called, of the target program from the input-output examples; these predictions can then be used

to guide a search over programs. Devlin et al. [2017] treated program synthesis as a sequence-to-sequence problem, mapping from the problem description to a description of the program in a spreadsheet domain. DreamCoder [Ellis et al., 2020] relaxes the requirement of defining a DSL, by learning a library that is useful for solving a training set of synthesis problems. Execution-guided synthesis methods execute the partial programs produced during synthesis, using the intermediate values to guide the search; learning methods for execution-guided synthesis include Zohar and Wolf [2018], Chen et al. [2019a], Ellis et al. [2019], Odena et al. [2020].

Many methods for program synthesis, both logic-based and learning-based, have been restricted to DSLs, but there have been some exceptions. For example, BAYOU generates API-heavy code in Java using a latent-variable probabilistic model [Murali et al., 2018]. Also, several different methods have been proposed for the problem of mapping a natural language description to code in general-purpose languages like Python [Ling et al., 2016, Yin and Neubig, 2017, Iyer et al., 2018].

Neural program induction methods are deep network architectures that aim to learn algorithms from input-output examples, by structuring the network in a way that corresponds to mathematical models of computation like Turing machines [Graves et al., 2014, Kaiser and Sutskever, 2016, Kurach et al., 2016, Graves et al., 2016]. This is a very different line of work from program synthesis, because program induction methods do not attempt to produce a program. Instead, they learn a neural network that maps directly from the input of the desired program to its output.

8.2 Machine Learning for Software Engineering

Over the past decade, a line of work has explored *machine learning for software engineering*, which applies machine learning methods to large corpora of source code, with the aim of using the models to develop tools for various tasks in software engineering. For an overview of machine learning methods applied to source code, see Allamanis et al. [2018a], or the more recent living literature review website [Allamanis, 2021].

Early work applied statistical n-gram models [Hindle et al., 2012, Allamanis and Sutton, 2013a] and neural networks [Maddison and Tarlow, 2014, Raychev et al., 2014] to code. Raychev et al. [2015] presented a method to predict program properties using a graph-structured conditional random field, which they applied to deobfuscate Javascript code by predicting names and a small set of types. Subsequent research over the following decade introduced deep learning methods for a variety of software engineering tasks.

Code completion has been a particular focus of interest [Raychev et al., 2016, Karampatsis et al., 2020, Svyatkovskiy et al., 2020, Kim et al., 2020]. Methods aim improving code readability by asking a model trained on a code corpus with good style to predict names of variables and methods in new code [Raychev et al., 2015, Allamanis et al., 2014, Alon et al., 2019]. Several methods have been proposed to do machine learning for type inference, for example, to add types to untyped code, such as when converting Javascript to Typescript [Hellendoorn et al., 2018, Pandi et al., 2020, Pradel et al., 2020, Wei et al., 2020]. Models trained over natural language and code have been applied within tools for improving comment quality and relevance [Louis et al., 2020, Panthaplackel et al., 2021]. Porting programs across languages has been treated as a learning problem similar to machine translation [Roziere et al., 2020, Nguyen et al., 2013, Karaivanov et al., 2014]. Program repair is the problem of automatically fixing bugs in programs, often based on a test suite [Le Goues et al., 2012, Long and Rinard, 2016]. Many learning methods have been proposed for program repair [Allamanis et al., 2018b, Tarlow et al., 2019, Hellendoorn et al., 2019, Dinella et al., 2019, Yasunaga and Liang, 2020, Chen et al., 2019b, Pradel and Sen, 2018].

Several pre-trained models for code have shown to be effective for transfer learning across software engineering tasks, including CuBERT [Kanade et al., 2020], CodeBERT [Feng et al., 2020], PyMT5 [Clement et al., 2020], code2vec [Alon et al., 2019], and other T5 models trained on code [Mastropaolo et al., 2021].

8.3 Benchmarks for Machine Learning over Source Code

Broadly, we identify three kinds of benchmark suites for machine learning over source code. First, *closed-domain* benchmarks for program synthesis ask systems to generate programs in a domain-specific language from a specification such as a logical formula or input-output examples. The most notable of these is the SyGuS competition [Alur et al., 2013], which includes tasks such as generating string transformations and bit-vector manipulations. Although the restriction to domain-specific languages is useful for building tractable systems, our benchmarks aim to evaluate program synthesis methods for general-purpose programming languages used by people to develop software.

Benchmarks for *machine learning for software engineering* are often assembled from corpora of open source projects, such as from Github. Benchmarks have been proposed for software engineering tasks including code completion

[Raychev et al., 2016, Allamanis and Sutton, 2013b], clone detection [Svajlenko et al., 2014], code search [Husain et al., 2019], predicting readable names to describe functions [Allamanis et al., 2016], and generating function text from docstrings [Iyer et al., 2018]. Multi-task benchmarks for these tasks have been collected into CodeXGlue [Lu et al., 2021]. Although these benchmarks are useful for evaluating ML support for a wide variety of important software engineering tasks, our goal is different: we seek to evaluate whether methods can learn to generate small, self-contained programs from a description of the task.

Finally, a third class of research benchmarks are collected from online programming competitions, such as CodeForces, TopCoder, and AtCoder. Such datasets include the Natural Program Synthesis (NAPS) dataset [Zavershynskyi et al., 2018], the Search-based Pseudocode to Code (SPoC) dataset [Kulal et al., 2019], the APPS dataset [Hendrycks et al., 2021], the PROGRES dataset [Alet et al., 2021], and the CodeNet dataset [Puri et al., 2021]. These datasets are similar in the source of programs, but differ in the kinds of natural language and code included in the dataset. Most notably, the SPoC dataset includes a pseudocode description which is a relatively literal line-by-line English transcription of each problem, while the APPS and CodeNet datasets include natural language descriptions of the program and test cases for each problem. The PROGRES dataset consists of problems built from sub-expressions of C++ CodeForces solutions, each specified by a large number of input-output examples. A different type of competition-like programming challenge is the programming puzzles dataset [Schuster et al., 2021], in which a problem is defined by a predicate that must be true of the desired program's output, for example, that a given path is indeed the shortest path between two nodes in a graph, or that a set of moves is a valid solution to a towers of Hanoi puzzle.

Although our benchmark tasks are similar in spirit to these programming competition datasets, they represent a different point in the design space, and one that we would suggest is complementary to previous work. Programming competition problems are often written so that the description includes a story which is engaging and makes identifying the algorithmic idea more challenging. In contrast, the natural language in Mostly Basic Programming Problems is a simpler description of the code's intended function. Therefore we hope both that this benchmark focuses more directly on the capabilities required to generate and understand code, and also that it a useful stepping stone to generating larger programs with more complex specifications.

9 Risks and Limitations

Chen et al. [2021] provide a detailed overview of risks and potential harms of large language models over code, discussing potential concerns that include over-reliance on generated outputs, misalignment, poisoning attacks [Schuster et al., 2020], and others. More broadly, Bender and Koller [2020] and Bender et al. [2021] discuss risks and potential harms of large language models for natural language. In this section, we limit our discussion to risks and limitations that are specific to our work.

The models we use in this paper have not been treated for safety, hence additional analysis of model outputs for potential harms is necessary before the use of the model in practice. For example, it is now increasingly understood that large language models can learn undesirable (e.g. biased) behavior from unlabeled training data, e.g., Bender and Koller [2020] and Bender et al. [2021], or can reveal training data, as well as sensitive information in the training data [Carlini et al., 2020]. It is possible that these risks are increased for an interactive use-case such as we described in Section 5.1. Further analysis of such risks and how to mitigate the risks for program synthesis are important directions for future work.

The energy cost and carbon footprint of the pre-training step for the models used in this paper are 451MWh and 26 tCO2e respectively. Because our fine-tuning datasets are relatively small in comparison, the estimated additional cost for the fine-tuning experiments in this paper is comparably very small.

Several limitations of our current model point toward interesting directions for future work:

1. Our benchmark programs are short and simple, and the programs solved by the model are the shortest and simplest among them. In other words, our benchmark has not yet captured the breadth and complexity of program synthesis.

2. Even when the model solves a task, it often does so with only one or two out of 80 samples. On the one hand, this is an acceptable limitation for downstream tasks, because we can machine-check the outputs against tests for semantic correctness. Additionally, if these capabilities are used in systems with a human in the loop, the sometimes incorrect output may be sufficient to support a user who can make the corrections necessary to put the generated code to use. On the other hand, this points toward a significant difference between the way the model is solving the problems and the way a human might. Possibly this can be fixed by further training the model to increase the probability of the outputs that pass the tests, but this seems more like a 'band-aid' than a deep fix.

3. The model cannot predict the outputs of programs on simple inputs (Section 6). This seems to us a prerequisite for claiming that the model 'understands' the programs it is synthesizing. Moreover, it seems like having a basic understanding of the semantics of code will be necessary for a wide variety of downstream tasks we might like such models to perform.

Some of the things we can do to address these limitations are clear. For instance, Figure 3 seems to suggest that simply using larger models will give nontrivial performance boosts. On the other hand, it is less clear how these models can be made more data efficient, or how (other than simply adding more relevant data) they can be made to better model the semantics of the code they emit. We hope that future work will address these and other issues.

10 Conclusion

We have conducted a large-scale study of how large language models perform at synthesis of short Python programs. Broadly speaking, we find that they perform surprisingly well, with the largest models synthesizing programs that pass all test cases for a majority of our benchmark problems. However, this good performance is predicated on being able to draw many samples from the model and machine-check them for correctness. From the perspective of downstream applications, this is perhaps acceptable. From the perspective of deciding whether these models "understand" computer programs in the same way that humans do, it is less so.

In that vein, we also tested whether these models could learn to execute existing programs on a given input. The results were poor, whether with few-shot prompting or when fine-tuning on other executions.[4] This suggests that — perhaps unsurprisingly — these models have not learned much about the semantics of programs simply by reading their text. This potentially has implications for thinking about grounding outside the program synthesis domain, and likely points toward future work on multi-modal models.

Finally, we tested whether these models could synthesize programs to solve simple mathematical word problems. Here we saw more success, especially when fine-tuning on a larger dataset. We briefly experimented with whether these models could give step-by-step explanations of their reasoning in this context, with promising but preliminary results.

Taken together, these results are exciting, but it is worth emphasizing that we are a long way from models that can synthesize complex applications without human supervision. The system we study here solves the problems it solves only given many tries, and the execution results in Section 6 suggest that there are important capabilities that these models still lack. In the near term, an important line of research is to find ways in which such systems can *augment* the capabilities of human programmers by acting collaboratively, perhaps by fixing errors or by helping with debugging. The dialog results in Section 5 and the MathQA results in Section 7 – where the model explains a partial solution – give a glimpse of what this might look like. In addition to increasing productivity for existing programmers, this could make programming much more widely accessible, empowering more people to interact with technology to meet their needs.

[4] This evaluation is perhaps slightly unfair, as we have not performed the obvious step of training the model on a much larger dataset of executions. This is an interesting direction for future work.

Author Contributions

Jacob Austin did the original experiments on MBPP, wrote much of the experimental code, did many of the MBPP experiments, and helped with paper writing. Augustus Odena wrote much of the experimental code, did many of the MBPP experiments, advised on the execution experiments, and did much of the writing. Max Nye wrote most of the code for the execution experiments, ran those experiments, wrote the execution portion of the paper, performed the error type analysis, and helped run some of the MBPP synthesis experiments. Maarten Bosma created the MBPP dataset, checked for duplication of MBPP data in the training dataset, and gave feedback on the paper. Henryk Michalewski wrote all of the code for the MathQA experiments, created MathQA-Python, ran the MathQA experiments, and wrote the MathQA section of the paper. David Dohan wrote and reviewed much of the code used to run the experiments and gave feedback on the paper. Ellen Jiang helped with early experiments, provided guidance, and performed qualitative analysis of model outputs. Carrie Cai provided guidance and qualitative analysis of model outputs. Michael Terry led the effort to sanitize the dataset and did qualitative analysis of the synthesis results. Quoc Le gave high-level scientific advice and gave feedback on the paper. Charles Sutton gave high-level scientific advice, fine-tuned the MBPP models, and did much of the writing.

Acknowledgements

We thank Daniel De Freitas Adiwardana for support and advice about the MBPP dataset.

References

Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A large-scale benchmark for few-shot program induction and synthesis. In Marina Meila and Tong Zhang, editors, *International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pages 175–186. PMLR, 18–24 Jul 2021. URL http://proceedings.mlr.press/v139/alet21a.html.

Miltiadis Allamanis. A survey of machine learning on source code. https://ml4code.github.io/, 2021. Accessed 2021-07-11.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference in Machine Learning (ICML)*, 2016.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), September 2018a.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*, February 2018b.

Miltos Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Working Conference on Mining Software Repositories (MSR)*, 2013a.

Miltos Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Working Conference on Mining Software Repositories (MSR)*, 2013b.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013. URL http://ieeexplore.ieee.org/document/6679385/.

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.

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM. doi: 10.1145/1455567.1455599. URL http://doi.acm.org/10.1145/1455567.1455599.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*, 2017. arXiv:1611.01989.

Emily M. Bender and Alexander Koller. Climbing towards NLU: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.463. URL https://aclanthology.org/2020.acl-main.463.

Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, pages 610–623, New York, NY, USA, March 2021. Association for Computing Machinery.

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8626–8637. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/62326dc7c4f7b849d6f013ba46489d6c-Paper.pdf.

big-bench collaboration. Beyond the imitation game: Measuring and extrapolating the capabilities of language models. *In preparation*, 2021. URL https://github.com/google/BIG-bench/.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large scale autoregressive language modeling with mesh-tensorflow, 2021. URL http://github.com/eleutherai/gpt-neo.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher 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. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. *arXiv preprint arXiv:2012.07805*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, Will Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, July 2021. URL http://arxiv.org/abs/2107.03374.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019a. URL https://openreview.net/forum?id=H1gfOiAqYm.

Zimin Chen, S J Kommrusch, M Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019b.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. PyMT5: Multi-mode translation of natural language and python code with transformers. 2020. URL http://arxiv.org/abs/2010.03150.

B.J. Copeland. *Alan Turing's Electronic Brain: The Struggle to Build the ACE, the World's Fastest Computer*. OUP Oxford, 2012. ISBN 9780199609154. URL https://books.google.com/books?id=YhQZnczOS7kC.

Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Neural Information Processing Systems*, 2015.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017. URL http://arxiv.org/abs/1703.07469.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, September 2019.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7805–7815. Curran Associates, Inc., 2018.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In *NeurIPS*, 2019.

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL https://arxiv.org/abs/2006.08381.

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. *Findings of the Association for Computational Linguistics: EMNLP 2020*, February 2020.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL http://doi.acm.org/10.1145/1926385.1926423.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 152–162, New York, New York, USA, 2018. ACM Press.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations (ICLR)*, September 2019.

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. May 2021. URL http://arxiv.org/abs/2105.09938.

Abram Hindle, Earl Barr, Zhendong Su, Prem Devanbu, and Mark Gable. On the "naturalness" of software. In *International Conference on Software Engineering (ICSE)*. 2012.

Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Association of Computational Linguistics (ACL)*, 2018.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. September 2019. URL http://arxiv.org/abs/1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. In *Empirical Methods in Natural Language Processing (EMNLP)*, August 2018.

Ellen Jiang, Edwin Toh, Alejandra Molina, Aaron Donsbach, Carrie Cai, and Michael Terry. Genline and genform: Two tools for interacting with generative language models in a code editor. *Adjunct Publication of the 34th Annual ACM Symposium on User Interface Software and Technology*, 2021.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning (ICML)*, 2020.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-Based statistical translation of programming languages. In *Onward*, 2014.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: Open-Vocabulary models for source code. In *International Conference on Software Engineering (ICSE)*, March 2020.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. March 2020. URL http://arxiv.org/abs/2003.13848.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Eduardo Blanco and Wei Lu, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pages 66–71. Association for Computational Linguistics, 2018. doi: 10.18653/v1/d18-2012. URL https://doi.org/10.18653/v1/d18-2012.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. SPoC: Search-based pseudocode to code. In *Advances in Neural Information Processing Systems*, 2019.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *International Conference on Learning Representations, (ICLR)*, 2016.

Claire Le Goues, Thanhvu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, January 2012.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. April 2021. URL http://arxiv.org/abs/2104.08691.

Belinda Z. Li, Maxwell Nye, and Jacob Andreas. Implicit representations of meaning in neural language models. *ArXiv*, abs/2106.00737, 2021.

Xiang Lisa Li and Percy Liang. Prefix-Tuning: Optimizing continuous prompts for generation. January 2021. URL http://arxiv.org/abs/2101.00190.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 298–312, New York, NY, USA, January 2016. Association for Computing Machinery.

Annie Louis, Santanu Kumar Dash, Earl T Barr, Michael D Ernst, and Charles Sutton. Where should I comment my code? A dataset and model for predicting locations that need comments. In *International Conference on Software Engineering (ICSE; NIER track)*, 2020.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. February 2021. URL http://arxiv.org/abs/2102.04664.

Chris J Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning (ICML)*, pages 649–657. 2014.

Zohar Manna and Richard Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6(2): 175–208, 1975.

Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3): 151–165, 1971.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of Text-To-Text transfer transformer to support Code-Related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347, May 2021.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations (ICLR)*, 2018.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), NIER*, 2013.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. *CoRR*, abs/2002.09030, 2020. URL https://arxiv.org/abs/2002.09030.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, and Charles Sutton. BUSTLE: bottom-up program-synthesis through learning-guided exploration. *CoRR*, abs/2007.14381, 2020. URL https://arxiv.org/abs/2007.14381.

Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. OptTyper: Probabilistic type inference by optimising logical and natural constraints. April 2020. URL http://arxiv.org/abs/2004.00348.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep Just-In-Time inconsistency detection between comments and source code. In *AAAI Conference on Artificial Intelligence*, 2021.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018.

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 179–190. ACM, 1989.

Michael Pradel and Koushik Sen. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):1–25, October 2018.

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 209–220, New York, NY, USA, November 2020. Association for Computing Machinery.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir 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. May 2021. URL http://arxiv.org/abs/2105.12655

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. https://blog.openai.com/language-unsupervised, 2018.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019. URL http://arxiv.org/abs/1910.10683.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *ACM PLDI*, 2014.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *ACM Symposium on Principles of Programming Languages (POPL)*, 2015.

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *OOPSLA*, 2016.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Adv. Neural Inf. Process. Syst.*, 33:20601–20611, 2020.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*, July 2020.

Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, and Adam Tauman Kalai. Programming puzzles. June 2021. URL http://arxiv.org/abs/2106.05784.

David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring LISP programs from examples. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 260–267, 1975. URL http://ijcai.org/Proceedings/75/Papers/037.pdf.

Armando Solar-Lezama. Introduction to program synthesis. https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htm, 2018. Accessed: 2018-09-17.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

Phillip D Summers. A methodology for LISP program construction from examples. *Journal of the ACM (JACM)*, 24(1): 161–175, 1977.

Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2011.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, September 2014.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. IntelliCode compose: Code generation using transformer. May 2020. URL http://arxiv.org/abs/2005.08025.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with Graph2Diff neural networks. November 2019. URL http://arxiv.org/abs/1911.01205.

Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, October 2013. Association for Computing Machinery.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

R.J. Waldinger, R.C.T. Lee, and SRI International. *PROW: A Step Toward Automatic Program Writing*. SRI International, 1969. URL https://books.google.com/books?id=3BITSQAACAAJ.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, 2020.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, May 2020.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Association for Computational Linguistics (ACL)*, 2017.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *ArXiv*, abs/1410.4615, 2014.

Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. NAPS: Natural program synthesis dataset. In *Workshop on Neural Abstract Machines & Program Induction (NAMPI)*, July 2018.

Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2094–2103. Curran Associates, Inc., 2018.

A Appendix

A.1 Instructions given to crowd workers and expert reviewers

Google Research is building a dataset for program synthesis in Python. This means we want to build Machine Learning algorithms which can automatically write small programs based on a natural language description. We are going to be using Google Colab to collect the data.

The description should be clear so that a human would be able to write a program without having to ask more questions. Please make sure that the description is using proper English grammar (uppercase the first word of the sentence, the instruction with a period). If unsure, you can copy and paste the description into Google Docs to use the grammar checker.

We ask you to put the code in a function. The cell code should not should not have any output (so don't use print). Instead the function should return the result. This way we can test the function automatically.

We ask you to write at least 3 assert statements to test your code (see colab linked below for examples). The test cell should not print anything which indicates that the tests passed. While it would be good to test edge cases this is not a requirement.

Imports can be in the global scope, but please import them every time you use them (so each cell should be able to run by itself). If you use a library, reimport it every time so that each solution can run by itself. Please do not define any global variables, but instead define them inside of the function.

Please use lowercase_with_underscores to define function names and try to give it a descriptive name if possible.

Please make sure that there are exactly 3 cells for each example (description, code and test cases). There is no limit on the number of lines of code.

Feel free to work together, but please make sure that your answers are different enough.

Figure 19: Instructions given to the crowd workers (edited slightly for clarity).

1. Well-defined, unambiguous question and test case:
Ensure the question is well-defined and unambiguous, given the question and a test case. If the question does not seem to be a good or useful question, flag it for removal.

2. No special conditions:
Remove any special conditions specified in the question (e.g., requirements to solve the problem using a regex, printing to the console, or using a lambda function).

3. Function signature looks "normal" (inputs and outputs):
Make sure the function signature is not unusual (e.g., one common case was to pass in a list and the length of that list).

4. Make sure the return values are well-specified:
Sometimes they return strings indicating success or failure; consider whether it could be changed to a standard Boolean value. If they use strings as enums, define these values in the natural language question.

5. Test cases are accurate:
Make sure the test cases contain no errors.

6. Float comparisons are handled correctly:
If the function returns floating point values, test using `math.isclose()`:
```
import math
math.isclose(a, b, rel_tol=0.001)
```

7. Questions asking for n elements of a list may not specify an expected order: disambiguate or adjust tests.
If a question asks for a subset of a list (e.g., the largest n numbers), but does not specify an order, add that specification to the question text.

8. Consider whether using sets (`set()`) in the asserts is the right way to test results

Figure 20: Instructions used to edit the problems.

A.2 Instructions for human-model collaboration experiments

Each user will be tasked with attempting 12 problems with at most 5 turns of dialog (including an initial automated turn). Each problem will be tackled by two people. After 5 turns the task is considered failed. If the model passes the test cases at any point, the task is considered solved. Instructions:

- Each human prompt is allowed to use one natural language sentence. You can use Python identifiers and expressions, but you can't use full statements, and it is encouraged to avoid lengthy Python expressions.
- For example, say "Close, but it needs to return i if count is equal to len(str)", rather than "You need to return i if count == len(str)".
- You can do some practice problems first to experiment with how best to interact with the model. After some experimentation, I'm giving it one problem as a prompt. You can try the practice problems as often as you want.
- Once you have finished the practice problems, navigate to the next cell and enter the problems you have been assigned into the input box. This will create the environment repeatedly in a for-loop. Alternatively, you can manually enter a problem ID.

Figure 21: Instructions for human-model collaboration experiments. Instructions have been lightly edited for publication.

A.3 Prompts for execution experiments

```
code: 'Consider the following Python function:\n\n{code}\n\nFill in the ??? below:\n\
   n{tests}'

code + NL + examples: 'Consider the following Python function:\n\n{code}\n\n' \
  + 'This function solves the task: "{description}"\n\n' \
  + 'The function satisfies the following examples:\n\n{prompt_asserts}\n\n' \
  + 'Fill in the ??? below:\n\n{tests}'),

code + NL : 'Consider the following Python function:\n\n{code}\n\n' \
  + 'This function solves the task: "{description}"\n\n' \
  + 'Fill in the ??? below:\n\n{tests}'),

code + examples: 'Consider the following Python function:\n\n{code}\n\n' \
  + 'The function satisfies the following examples:\n\n{prompt_asserts}\n\n' \
  + 'Fill in the ??? below:\n\n{tests}'

NL + examples: 'Consider a Python function called {function_name}.\n\n' \
  + 'This function solves the task: "{description}"\n\n' \
  + 'The function satisfies the following examples:\n\n{prompt_asserts}\n\n' \
  + 'Fill in the ??? below:\n\n{tests}'

NL : 'Consider a Python function called {function_name}.\n\n' \
  + 'This function solves the task: "{description}"\n\n' \
  + 'Fill in the ??? below:\n\n{tests}'

examples: 'Consider a Python function called {function_name}.\n\n' \
  + 'The function satisfies the following examples:\n\n{prompt_asserts}\n\n' \
  + 'Fill in the ??? below:\n\n{tests}'
```

Listing 1: Prompts for the execution experiments.

A.4 Additional example human-model interaction samples

```
user: I need to write a Python function called test_duplicate. Here's a description: '
    Write a function to find whether a given array of integers contains any duplicate
    element.'.

model: Can you give me some examples of how to use it?

user: Sure. Here is an example assert: 'assert test_duplicate(([1,2,3,4,5])) == False'
    .

model: def test_duplicate(array):
    for i in range(0,len(array)):
        for j in range(0,len(array)):
            if array[i] == array[j]:
                return True
    return False

info (not shown to model): This response failed some tests. Details:
    * Test 0 (False): test_duplicate(([1,2,3,4,5])) returned True and expected False.
    * Test 1 (False): test_duplicate(([1,2,3,4, 4])) returned True and expected True.
    * Test 2 (False): test_duplicate([1,1,2,2,3,3,4,4,5]) returned True and expected
    True.

user: Nice, but you should start the second for loop at i + 1.

model: def test_duplicate(array):
    for i in range(0,len(array)):
        for j in range(i + 1,len(array)):
            if array[i] == array[j]:
                return True
```

```
    return False
```

Listing 2: example interaction with model.

```
user: I need to write a Python function called count_Substrings. Here is a description
    : 'Write a python function to count the number of substrings with the sum of digits
     equal to their length.'.

model: Can you give me some examples of how to use it?

user: Sure. Here's an example assert: 'assert count_Substrings('112112') == 6'.

model: def count_Substrings(n):
    count = 0
    for i in range(len(n)):
        for j in range(i + 1, len(n)):
            if sum(int(c) for c in n[i:j]) == len(n[i:j]):
                count += 1
    return count

info (not shown to model): This response failed some tests. Details:
    * Test 0 (False): count_Substrings('112112') returned 6 and expected 6.
    * Test 1 (False): count_Substrings('111') returned 3 and expected 6.
    * Test 2 (False): count_Substrings('1101112') returned 9 and expected 12.

user: Very close. The upper bound on the inner for loop needs to be len(n) + 1.

model: def count_Substrings(n):
    count = 0
    for i in range(len(n)):
        for j in range(i + 1, len(n) + 1):
            if sum(int(c) for c in n[i:j]) == len(n[i:j]):
                count += 1
    return count
```

Listing 3: Another example interaction.

I need to write a Python function called sort_matrix. Here's a description: 'Write a function to sort a given matrix in ascending order according to the sum of its rows'.

Can you give me an example?

```
assert sort_matrix([[1, 2, 3], [2, 4, 5], [1, 1, 1]])==[[1, 1, 1], [1, 2, 3], [2, 4, 5]]
```

```
def sort_matrix(matrix):
  """
  Sort the matrix in ascending order according to the sum of its rows.
  """
  for i in range(len(matrix)):
    for j in range(len(matrix)):
      if matrix[i][j] > matrix[i+1][j]:
        temp = matrix[i][j]
        matrix[i][j] = matrix[i+1][j]
        matrix[i+1][j] = temp
  return matrix
```

tests fail

Hmm that's quite complicated. Could you use the sorted function?

I'm not familiar with that function.

tests fail

Oh it's a Python built-in function that sorts lists. You can use it with a key.

```
def sort_matrix(matrix):
  """
  Sort the matrix in ascending order according to the sum of its rows.
  """
  return sorted(matrix, key=sum)
```

tests pass

Hooray! 🎉🎉🎉

Figure 22: An extra dialog example.