Programming Puzzles

Tal Schuster  
MIT  
Ashwin Kalyan  
Allen Inst. for AI  
Oleksandr Polozov  
Microsoft Research  
Adam Tauman Kalai  
Microsoft Research

Abstract

We introduce a new type of programming challenge called programming puzzles, as an objective and comprehensive evaluation of program synthesis, and release an open-source dataset of Python Programming Puzzles (P3). Each puzzle is defined by a short Python program \( f \), and the goal is to find an input \( x \) which makes \( f \) output True. The puzzles are objective in that each one is specified entirely by the source code of its verifier \( f \), so evaluating \( f(x) \) is all that is needed to test a candidate solution \( x \). They do not require an answer key or input/output examples, nor do they depend on natural language understanding. The dataset is comprehensive in that it spans problems of a range of difficulties and domains, ranging from trivial string manipulation problems that are immediately obvious to human programmers (but not necessarily to AI), to classic programming puzzles (e.g., Towers of Hanoi), to interview/competitive-programming problems (e.g., dynamic programming), to longstanding open problems in algorithms and mathematics (e.g., factoring). The objective nature of P3 readily supports self-supervised bootstrapping. We develop baseline enumerative program synthesis and GPT-3 solvers that are capable of solving easy puzzles—even without access to any reference solutions—by learning from their own past solutions. Based on a small user study, we find puzzle difficulty to correlate between human programmers and the baseline AI solvers.

1 Introduction

Puzzles are often used to teach and evaluate human programmers. Classic puzzles such as the Towers of Hanoi teach fundamental concepts such as recursion. Programming competition problems, also referred to as puzzles [32], evaluate a participant’s ability to apply these concepts to artificial tasks. Puzzles are also used to evaluate programmers in job interviews, and harder puzzles—such as the RSA-factoring challenge—can even test the limits of state-of-the-art algorithms. Each of these types of puzzles is described in its own format, often in a natural language such as English, and the solutions are evaluated through a variety of means.

We introduce a novel puzzle representation called a programming puzzle or simply a puzzle, which captures the essence of these puzzles in a form convenient for machines and programmers. A puzzle is specified by a short program \( f \), and an answer is an input \( x \) which makes \( f \) output True, i.e., a valid answer \( x \) satisfies \( f(x) == True \). Puzzles make for an objective programming evaluation based solely on source code with no need for any natural language descriptions, input/output examples, or reference solutions. We also release a growing open-source Python Programming Puzzles dataset,\(^1\) called P3, which is already comprehensive in terms of difficulty, domain, and algorithmic tools. This dataset unifies many of the types of puzzles mentioned above.

To illustrate, \((\text{lambda } s: "Hello " + s == "Hello world")\)^2 is an example of a Python programming puzzle, with the answer "world". Some puzzles have multiple answers and some puzzles,

\(^1\)https://github.com/microsoft/PythonProgrammingPuzzles  
\(^2\)In Python, \text{lambda \ s} defines a function of variable \text{s} and string addition corresponds to string concatenation.

Figure 1: Programming puzzles ranging from trivial to longstanding open algorithmic challenges in multiple domains. `f1("ox" * 1000) == True` where `s * n` means `n` repetitions of string `s`; dynamic programming efficiently finds a list of 26 increasing indexes to solve `f2` while brute force would take exponentially long; and `f3` requires advanced computational number theory algorithms.

even if very short, are extremely challenging. For example, Figure 1 illustrates three short puzzles that are diverse in domain, difficulty, and algorithmic tools. The first puzzle is an easy (for humans) puzzle that tests one’s understanding of basic properties of strings. The second one requires dynamic programming (typically taught in college-level programming courses), and the last is a hard problem requiring advanced computational number theory algorithms.

As describe in §3, the dataset also contains numerous classic puzzles; game-playing puzzles; optimization puzzles such as solving a linear programming; graph puzzles such as shortest path; and competitive-programming problems. The most difficult puzzles involve longstanding open problems such as learning parity with noise [9]; factoring [22, 28]; or finding a cycle in the $3n + 1$ process which would disprove the Collatz conjecture [33]. Thus, if AI were to surpass human-level performance on this dataset, it would lead to breakthroughs on major open problems. The P3 puzzles were inspired by sources such as Wikipedia, algorithms books, and programming competitions. We plan to grow this dataset as an open-source project, and anyone can add a puzzle by simply writing a function `f`.

Puzzles allow objective evaluation, meaning that it is easy to test whether any candidate answer is valid without consulting an answer key. In Programming By Example (PBE), there may be multiple functions mapping a given sequence of inputs $x_i$ to outputs $y_j$. In an objective evaluation, there is no additional burden to learn the answer-key bias. Objective evaluation also makes bootstrapping easy, even on a set of test puzzles provided without any answers. Given a set of puzzles, one can attempt to solve them, determine with certainty which solutions are correct, and use those solutions to improve one’s ability to solve the remaining puzzles [17]. Inspired by success in playing games [53, 56], self-training has also proven useful in program synthesis [see, e.g., 4, 13].

From a theoretical point of view, as we shall discuss, objectivity can be formalized as the complexity class NP of non-deterministic polynomial-time decision problems. Moreover, the puzzle decision problem is NP-complete, meaning puzzles can readily express any NP problem, including polynomial-time problems and other NP-complete problems such as Boolean satisfiability.

Our experiments compare different parametric enumerative top-down solvers based on random forests and Transformers, and different types of GPT-3 prompts, e.g., zero/few-shot and with/without English descriptions. Without access to any reference solutions, only utilizing self-training bootstrapping, our enumerative models solved up to 43% more P3 problems than a naive brute force baseline.

To address the questions of whether puzzles measure programming proficiency and how puzzle difficulty compares between humans and computers, we performed a small user study. Puzzles were accessible and enjoyable for programmers with varying levels of experience. While both GPT-3 and enumerative techniques can solve a fraction of the puzzles, human programmers outperform them. For example, bootstrapping GPT-3 with up to 10K tries solved 60% of the puzzles, lower than both beginner and experienced participants that solved 76% and 87% puzzles on average, respectively. Overall, we find perceived puzzle difficulty to scale similarly for both humans and AI.

The main contributions of this paper are introducing:

1. programming puzzles: a new type of problem suitable for algorithmic problem-solving (for both machines and humans);
2. P3, an open-source dataset of puzzles covering diverse domains and difficulties; and
3. an evaluation of humans and baselines demonstrating that puzzles can be used to measure algorithmic problem-solving progress.
2 Problem formulation

Formally, both puzzles and answers can be represented as strings, where $\Sigma^*$ is the set of finite strings over alphabet $\Sigma$. The set of puzzles is denoted by $\mathcal{F} \subseteq \Sigma^*$, with answers $\mathcal{X} \subseteq \Sigma^*$. A puzzle $f \in \mathcal{F}$ is defined entirely by its source code string and, with a slight abuse of notation, the result of running it on input $x \in \mathcal{X}$ is denoted as $f(x) \in \{0,1\}$. Answer $x \in \mathcal{X}$ is valid if it satisfies $f(x) = 1$, i.e., $f$ outputs 1 when run on $x$, within a specified amount of time. To ensure that puzzles can be quickly verified, it is necessary to upper-bound the time required for puzzle verification. This ensures that the puzzle decision problem, namely the problem of determining whether a given a puzzle has a valid answer, is in the complexity class NP. Thus formally, the puzzle decision problem is, given a string $f$ denoting the puzzle (represented as, say, a Turing machine) and a timeout $t$, does the puzzle output 1 in time $\leq t$. See Appendix D for further details and comparison to other NP-complete problems.

A solver takes $n$ puzzles $f_1, f_2, \ldots, f_n$ with timeouts $t_1, \ldots, t_n$, and outputs answers to as many puzzles as it can within a time bound $T$. Of course $T \gg \sum t_i$ is significantly larger than the verification timeouts. Formally, the score of solver $S : \mathcal{F}^n \to \mathcal{X}^n$ is the number of puzzles $f_i$ for which $f_i(x_i)$ outputs 1 in time $\leq t_i$. Although we do not study it in this paper, it would be natural to assign different values to different puzzles. For example, solving open problems such as finding a Collatz cycle or factoring the largest RSA challenge integer (currently unsolved, with a $200,000 prize offered), should be of greater value than solving a simple hello-world puzzle.

It is convenient, though not required, to solve puzzles by outputting a program which, when run, computes the answer. Such a program is called a solution $g$ to distinguish it from the answer $x$ that $g$ outputs (i.e., $x=g()$). Puzzles may have short solutions but long answers, e.g., the puzzle $f: \lambda a \in \mathbb{N} a + (10 \times 6)$ that asks for a string of length one million is solved by the solution program $g: x = g()$ (10 ** 6). In this example, solution $g$ generates a valid answer $x$ of length one million. Of course, another solution would be to explicitly write a string of length one million in the code, though this implementation may not pass a human code review. In the dataset and this paper, we provide solutions since they may be significantly shorter.

Many puzzles fit a single problem template. For example, the factoring puzzles for different $n$ are all instances of the factoring problem (Figure 1). The parameters that vary—in this case, $n$—are called puzzle parameters. The parameters of a shortest-path problem might be the graph, the source and target nodes, and an upper bound on the path length. Many problems, such as the Collatz problem, have no parameters and thus the Collatz problem consists of a single puzzle.

3 The P3 dataset

P3 uses Python, the de facto language of ML research, as the programming language for specifying puzzles. P3 is generated from the 200 problems summarized in Table 1 by running make_dataset.py in the repository with a maximum of 1,000 puzzles per problem. A larger number of puzzles may be generated by increasing this maximum. Every puzzle is described by a function with a single typed argument (i.e., the candidate answer) that returns True upon success. Since Python is not type-safe, we add an assertion to ensure that answers match the declared type.

We also provide code for serializing Python objects to and from strings in a json-like format, so that programs implemented in any language can output potential answers. Moreover, strings are universal in that they can encode arbitrary Python objects including functions, as in the Quine puzzle ($\lambda x \in \mathbb{N} x = x$)3 motivated by the classic challenge of writing a program that outputs its own source code. As evaluation of the string quine can lead to an infinite loop, this puzzle illustrates the necessity of the evaluation timeout $t$ for attempted solutions.

While not necessary,4 we follow the common practice of programming competitions and provide a reference solution to most (over 90%) of the puzzles. Some puzzles have more than one solution. A handful of puzzles represent major open problems in computer science and mathematics including Factoring (and Discrete Log), Planted Clique, Learning Parity with Noise, Graph Isomorphism,

---

3GPT-3 generated a 5-character solution to the quine puzzle while the authors’ solution was 88-characters. 4Reference solutions are neither necessary nor used in our experiments, because puzzles are self-contained.
and finding a Collatz cycle, as described in Appendix E. We also provide English descriptions for each puzzle in the dataset to support research involving natural language. Appendix F compares programming competition problems to puzzles.

**Creation process.** The following sources were used for identifying possible puzzles:

- Wikipedia, specifically the Logic puzzles category, the List of unsolved problems in mathematics, and the List of algorithms.
- Competitions, primarily the competitive programming website codeforces.com but also a handful of problems from the International Collegiate Programming Contest and the International Mathematical Olympiad (IMO)—a high school mathematics competition.
- The Python programming language itself, with trivial puzzles created to test understanding of basic functions, such as the the hello-world puzzle which tests string concatenation.

P3 is organized topically into files listed in Table 1. These topics include domains such as number theory, graph theory, chess puzzles, game theory, etc., as well as puzzles inspired by a specific source such as a specific programming competition. One finding in this paper is that many types of puzzles can be captured in spirit, if not exactly, as succinct puzzles. Common patterns include:

- Problems that are naturally puzzles. For instance, the TowersOfHanoi and SlidingPuzzle puzzles simply test the sequence of moves to see if they lead to the goal state.
- Problems that have an equivalent natural puzzle. For instance, the standard definition of the factoring problem, namely factorizing an integer into its prime factors would require a puzzle that tests primality. However the problem of finding any non-trivial integer factor, $f_3$ in Figure 1, can be recursively called to solve the prime factorization problem.
- Optimization problems. Some such problems have equivalent natural puzzles, e.g., linear programming is well-known [16] to be equivalent to solving a zero-sum game which is the ZeroSum puzzle. For others, such as LongestMonotonicSubstring ($f_2$ of Figure 1) or ShortestPath, we specify a bound $\theta$ on the objective, and the goal is to find a feasible $x$ with objective better than $\theta$. In order to generate $\theta$, we must solve the optimization problem ourselves, but the puzzle generation code is not provided to the solvers.
- Problems that ask how many items in a certain set satisfy a given property, may be converted to problems that require an explicit enumeration of all such items. See for example the AllPandigitalSquares puzzle that requires all 174 roots of pandigital perfect squares as input.
- Problems that involve game-playing can often be converted to puzzles. In chess, this includes the classic Eight Queens and Knights Tour puzzles. Our puzzles for the games of Nim and Mastermind involve exhibiting a winning strategy.

In order to ensure that each puzzle is achieving its goals, the puzzle design process has a step in which we test for unintended trivial solutions that are small integers or common strings.

**Exclusions.** Many programming challenges do not make as good puzzles. First, some challenges require a long program to describe. For instance, testing a Rubik’s cube solution requires significantly more code than the sliding 15 puzzle (which is partly why sliding puzzles are used so often in AI courses). Second, sometimes specifying the puzzle would give away the solution. For instance, consider finding the smallest prime number greater than $10^6$. This makes a reasonable contest problem with an answer key, but the puzzle form would include code to test primality which would give away the solution. Third, “soft” challenges involving natural language or images are not in NP and not easily verifiable. This includes challenges involving human commonsense or world knowledge about names, dates, or image classification. Finally, interactive challenges do not make for good programming puzzles. Fortunately, several other benchmarks cover these latter two types of exclusions [see, e.g., 2, 30, 36, 41, 45, 47, 48, 50, 52, 53, 55, 58, 61, 62].

---

5 The solution to this problem would disprove the Collatz conjecture that is believed to be true, but no proof has been found yet. Therefore, if the conjecture is true, the maximum attainable score in P3 is < 100.
Table 1: Number of problems (and how many of them have at least one reference solution) per domain in P3 v0.1. Puzzles are generated by altering problem parameters. The right two columns show the average size of puzzles and solutions, measured by the number of nodes in the Python AST.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Problems</th>
<th>Solutions</th>
<th>Puzzles</th>
<th>Size</th>
<th>Reference Solns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebra</td>
<td>4</td>
<td>4</td>
<td>4000</td>
<td>98</td>
<td>173</td>
</tr>
<tr>
<td>Basic</td>
<td>21</td>
<td>21</td>
<td>21000</td>
<td>79</td>
<td>43</td>
</tr>
<tr>
<td>Chess</td>
<td>5</td>
<td>3</td>
<td>4858</td>
<td>277</td>
<td>186</td>
</tr>
<tr>
<td>Classic puzzles</td>
<td>22</td>
<td>22</td>
<td>11370</td>
<td>132</td>
<td>194</td>
</tr>
<tr>
<td>CodeForces</td>
<td>24</td>
<td>24</td>
<td>23025</td>
<td>111</td>
<td>67</td>
</tr>
<tr>
<td>Compression</td>
<td>3</td>
<td>3</td>
<td>3000</td>
<td>160</td>
<td>118</td>
</tr>
<tr>
<td>Conway’s game of life</td>
<td>2</td>
<td>1</td>
<td>2000</td>
<td>248</td>
<td>371</td>
</tr>
<tr>
<td>Game Theory</td>
<td>2</td>
<td>2</td>
<td>2000</td>
<td>417</td>
<td>506</td>
</tr>
<tr>
<td>Games</td>
<td>5</td>
<td>5</td>
<td>1006</td>
<td>223</td>
<td>189</td>
</tr>
<tr>
<td>Graphs</td>
<td>11</td>
<td>10</td>
<td>9002</td>
<td>151</td>
<td>151</td>
</tr>
<tr>
<td>ICPC</td>
<td>3</td>
<td>3</td>
<td>3000</td>
<td>433</td>
<td>663</td>
</tr>
<tr>
<td>IMO</td>
<td>6</td>
<td>6</td>
<td>5012</td>
<td>209</td>
<td>253</td>
</tr>
<tr>
<td>Lattices</td>
<td>2</td>
<td>2</td>
<td>2000</td>
<td>108</td>
<td>222</td>
</tr>
<tr>
<td>Number Theory</td>
<td>16</td>
<td>12</td>
<td>10762</td>
<td>66</td>
<td>68</td>
</tr>
<tr>
<td>Probability</td>
<td>5</td>
<td>5</td>
<td>5000</td>
<td>107</td>
<td>72</td>
</tr>
<tr>
<td>Study</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>65</td>
<td>21</td>
</tr>
<tr>
<td>Trivial inverse</td>
<td>34</td>
<td>33</td>
<td>32002</td>
<td>46</td>
<td>26</td>
</tr>
<tr>
<td>Tutorial</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>46</td>
<td>13</td>
</tr>
<tr>
<td><strong>Total # / Average size</strong></td>
<td><strong>200</strong></td>
<td><strong>191</strong></td>
<td><strong>139,072</strong></td>
<td><strong>109</strong></td>
<td><strong>99</strong></td>
</tr>
</tbody>
</table>

**Growth process.** The focus of this paper is in creating a framework with an initial dataset; and demonstrating its utility for developing and evaluating AI solvers. As a GitHub repository, the dataset can grow over time in a standard manner with the ability to reference previous versions. We plan to continue adding puzzles and hope that others will as well. The popular competitive-programming website codeforces was found to be a potential source of innumerable puzzles which we plan to use to grow the dataset. We found that almost all level-800 problems (their easiest problems) could be encoded as short puzzles.

4 **Solvers**

In this section, we describe the models we develop as baselines for the dataset. We consider both solving problems independently and joint solvers that bootstrap from previously obtained solutions to find new ones. We also consider both enumerative solvers that use standard techniques from program synthesis and a Language-Model (LM) solver that uses GPT-3 to solve puzzles. While a direct comparison between these two different approaches is difficult because they run on very different hardware (the LM calls an API), we can still compare the relative difficulty with which they solve different puzzles, and also to human difficulty rankings among puzzles.

4.1 **Enumerative solvers**

Following prior work [4, 13, 39], we develop models to guide the search for $g$ over the space of all possible functions. In particular, we implement a grammar that generates Abstract Syntax Trees (ASTs) for a large subset of Python. The grammar covers basic Python functionality and is described in Appendix B.1. Specifically, each Python function is translated to an AST using a given set $R$ of rules. Based on the puzzle, a context-specific distribution over rule probabilities is computed. To facilitate efficient top-down search, the context of a rule is defined to be the rule used by the parent node and the index of the current node among the parent’s children. Thus if the parent node was a division binary operator, then the two children would each have different contexts, but if two such divisions were present in the same program, both numerators would share the same context.

Each puzzle $f$ is represented by a feature vector $\phi(f)$ and each context is represented by a vector $c(p, i)$ where $p$ is the parent rule and $i$ is the child index. Each rule $r \in R$ is also associated with a feature vector $\rho(r)$. The probability distribution over $R$ is determined based on $\rho(r), \phi(f), c(p, i)$.
and the likelihood of a solution $g$ is the product of all rules constituting its AST. Naturally, this scoring mechanism introduces a bias towards shorter programs (i.e., smaller trees), which is desirable as a short solution is easy to inspect.

**COPY rules.** Solutions often reuse constants or puzzle parameters, for example the constant 25 or the variable $x$ in example 62 in Figure 1. As in prior work [39], for each puzzle, the global rules bank is expanded to include COPY rules for constants and parameters of the examined puzzle.\(^7\) When composing solutions, this rule can reduce the complexity of the solution by simply learning to copy part of the puzzle rather than having to generate it from scratch. For simplicity, we create copy rules for each of the supported types and assign the probabilities uniformly across all the puzzle’s constants of that type. In other words, our models learn when a certain type should be copied from the puzzle, and rank all available constants and parameters of that type the same.

To solve a new puzzle, we perform a top-down search. Specifically, at each node, we apply a selected model over all rules in $\mathcal{R}$ whose type matches the context, and re-normalize the scores to create a valid probability distribution. The solver enumerates solutions in order of decreasing likelihood until it finds a solution $g$ such that $f(g())$ evaluates to True in time $\leq t$, for a maximum number of tries $M$. See Appendix B for details on the search and rules. Next, we briefly describe our models.

**Uniform.** The first model is a simple uniform rule that assigns the same probability to all rules. The only exception is COPY rules, which have a larger, fixed probability in order to bias the solver towards utilizing this option. As we score programs by their joint probability, this bias effectively favors shorter programs. We use this model to find solutions to the easier problems, satisfied by a simple and short answer, and use these to bootstrap the learning of the parametric models. This model also provides a naive brute force baseline to compare the parametric models with, testing if they can guide the solution search better.

The remaining two models have parameters that are fit based on bootstrapping. Namely, given previously obtained solutions, we collect all parent-child rule pairs as self-supervision and fit the model’s parameters on them. The training size for this learning problem is the total number of nodes in all the trees among solutions discovered up until that point. We implement two bigram parametric models to predict $P(r \mid \rho(r), \phi(f), c(p, i))$, where $r$ is a candidate rule to appear in $g$’s tree under $p$ as its $i$’s argument.

**Random forest.** In this model, we represent $f$ as a bag-of-rules $\bigcup \{r_k \in f\}$. Specifically, $\phi(f)$ is a vector of length $|\mathcal{R}|$ representing the number of occurrences of each rule in $f$. $p$ and $i$ are encoded as a one-hot vector and concatenated to $f$’s representation to construct the input to the model. Given past solution trees, we train the model to predict the index of $r$ out of $|\mathcal{R}|$ given $f, p, i$ examples.

**Transformer.** Following the recent success of Transformer models [18, 57] in encoding source code [19, 29, 54, *inter alia*], we turn to these encoders for richer representations. We use a RoBERTa-based [35] Transformer to encode puzzles and rules directly from their code. The probability of a rule $r$ being the $i$’s child of $p$ in $g$ is proportional to the dot product of the deep joint representation of $f, p, i$ and the Transformer encoding $\rho(r)$. We pretrain the Transformer with a masked language model task on Python GitHub repositories [25].\(^8\) Then, our solver concatenates the Transformer encodings $\phi(f)$ and $\rho(p)$ with a learned embedding for $i$, following by non-linear layers to compute the joint representation. We fine-tune the solver on parent-child rule pairs from previously acquired solutions. See Appendix B.2 for extended details, and Figure B.1 on page 20 for a model diagram.

### 4.2 Autoregressive language model solvers

We experiment with the massive Transformer-based GPT-3 language model [10]. We follow the recent strategy of using GPT-3 by designing a prompt that directs the text generation to our desired task. This approach has recently shown to be useful in converting natural language descriptions to programming code [11, 24] and guide theorem proving [44]. Interestingly, our prompts are mostly programs. Unlike our enumerative models that build an AST, GPT-3 generates the solution as a string that is directly evaluated as Python code.

---

\(^7\)When executing a solution, COPY rules are simply the identity function (COPY = lambda x: x in Python).

\(^8\)Our pretrained model and tokenizer are available at [https://huggingface.co/tals/roberta_python](https://huggingface.co/tals/roberta_python).
def f(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2

assert True == f(...)

Figure 2: A short prompt for a puzzle requesting a list of ten integers where the fourth item occurs exactly twice. A valid completion would be \ldots [1, 2, 3, 4, 5]*2) .

We consider four different prompts: (a) A \textit{short} zero-shot prompt based solely on the puzzle at hand (illustrated in Figure 2); (b) a \textit{medium} 5-shot prompt that includes the five example puzzles that had been shown to (human) programmers during our user study (Figure C.1 on page 21); (c) a \textit{long} prompt with the same five puzzles augmented by English descriptions of the tasks in comments (Figure C.2 on page 22); and (d) a \textit{bootstrapping} prompt which uses only solutions to problems that it has already solved (Figure C.3 on page 23). The bootstrapping prompt begins like the zero-shot prompt but quickly exceeds GPT-3’s API maximum length as more puzzles are solved. At that point, previously solved puzzles are randomly sampled to form the prompt.

The completions which parse as valid Python expressions are then evaluated. Appendix C gives further details of the execution environment, the API parameters and other prompts we investigated.

5 Experiments

We use our P3 dataset to evaluate the performance of the solvers from §4. We assume no access to reference solutions\(^9\) and measure how many puzzles are solved by each solver with up to \(M\) tries per puzzle, where each try is a potential solution that is evaluated. For the enumerative solvers, this is equivalent to having a valid solution ranked in the top \(M\). For autoregressive solvers, it reflects the probability of obtaining a solution within \(M\) outputs. First, we test the solvers bootstrapping efficacy in leveraging previously solved problems to solve new ones. Then, once solutions to a single instance of certain problems are found, we test whether solvers also succeed on other problem instances (i.e., puzzles originated from the same problem). Finally, in §5.1, we present our user study results that compares human’s performance with AI solvers.

Learning from past solutions. In this experiment, we use a single puzzle instance per problem. For the parametric enumerative solvers, we first run the uniform solver with \(M = 10^4\) on all 138 problems supported by our grammar (see Appendix B.1), solving 38 of them. These solutions contain a total of 2,475 rules that we use to train the parametric models. In the bootstrapping variant, we repeat the training for 6 cycles, each time adding the new solutions found with \(M = 10^4\). In the final round, we allow up to \(M = 10^6\) solving tries (including tries from previous cycles). The Bootstrapping GPT-3 starts with the zero-shot (short) prompt and appends to it valid solutions as they are found.

Figure 3a shows the total number of puzzles solved by each enumerative solver, with and without the self-training bootstrapping cycles. We report the average results across three runs and present the standard deviation in the graph. We see that the parametric models quickly improve over the naive uniform search and that the bootstrapping process facilitates solving many new problems. At \(M = 10^6\), the random forest and Transformer-based enumerative models solved a total of 68 and 76 problems, respectively, which is 28% and 43% more than the uniform solver.

The GPT-3 solver also improves by learning from previously found solutions. As Figure 3b shows, few-shot settings with tutorial examples perform better than zero-shot (Short) and solve new problems. Including natural language descriptions (Long) helps for solving five more puzzles, with up to \(10^4\) tries. The best strategy, however, is the bootstrapping one that starts without any reference and adds solutions to the prompt as they are found. Appendix H presents the 86 solutions found by this model.

Table 2 shows the number of achieved solutions per domain with \(M\) tries. We also include an overall score for each model as the macro-average of its solving rate across the different domains. Again, we see the the bootstrapping version of each model improves its score. The highest improvement is with the bootstrapping GPT-3 that reaches the best score of 25.2.

\(^9\)With the exception of the GPT-3 medium-long prompts that including five Tutorial problems and solutions.
(a) Enumerative solvers.

(b) GPT-3 solvers.

Figure 3: Increasing the number of tries allows solving new problems. Better solvers, though, solve new problems significantly faster by learning from past experience. (a) Parametric enumerative solvers initialized with the solutions of the uniform solver at $M = 10^4$ accelerate the solution search. Additional self-training bootstrapping cycles (marked with B.) solve even more problems. (b) Due to limited budget, GPT-3 solvers are evaluated with up to $10^4$ attempts. The natural language description (Long) allows small improvement. Adding previously found solutions to the prompt (Bootstrap) provides significant improvement and solves the most puzzles across baselines. The results of the enumerative models are averaged across three runs and the shaded areas show the standard deviation.

Table 2: Solved problems per domain with up to 1M tries per puzzle for enumerative and 10K for GPT-3. The first row also shows the number of available P3 problems in that domain. Bootstrapping models, that learn from new solutions as they are found, are in grayed lines. The score in the last column is the macro-average of the success rates across the different domains. We report the results of the run with the highest score across our three trials per enumerative model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Algebra</th>
<th>Basic</th>
<th>Chess</th>
<th>Classic</th>
<th>Codeforces</th>
<th>Compression</th>
<th>Conway’s</th>
<th>Game Theory</th>
<th>Games</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>0/4</td>
<td>7/21</td>
<td>0/5</td>
<td>1/22</td>
<td>7/24</td>
<td>0/3</td>
<td>0/2</td>
<td>0/2</td>
<td>0/5</td>
</tr>
<tr>
<td>Random forest</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B. Random forest</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transformer</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B. Transformer</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GPT-3 (Medium)</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B. GPT-3</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Graphs</th>
<th>ICPC</th>
<th>IMO</th>
<th>Lattices</th>
<th>N. Theory</th>
<th>Probability</th>
<th>Study</th>
<th>Trivial inverse</th>
<th>Tutorial</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>1/11</td>
<td>0/3</td>
<td>0/6</td>
<td>0/2</td>
<td>3/16</td>
<td>1/5</td>
<td>9/30</td>
<td>22/34</td>
<td>2/5</td>
<td>13.9</td>
</tr>
<tr>
<td>Random forest</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>10</td>
<td>26</td>
<td>3</td>
<td>17.7</td>
</tr>
<tr>
<td>B. Random forest</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>12</td>
<td>24</td>
<td>3</td>
<td>20.0</td>
</tr>
<tr>
<td>Transformer</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>12</td>
<td>24</td>
<td>3</td>
<td>19.0</td>
</tr>
<tr>
<td>B. Transformer</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>13</td>
<td>26</td>
<td>3</td>
<td>21.6</td>
</tr>
<tr>
<td>GPT-3 (Medium)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>18</td>
<td>5</td>
<td>19.7</td>
</tr>
<tr>
<td>B. GPT-3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>18</td>
<td>30</td>
<td>4</td>
<td>25.2</td>
</tr>
</tbody>
</table>

**Generalizing to other problem instances.** In the previous experiment, we attempted to solve the default single puzzle instance of each problem. Next, we examine whether our solvers can also solve other puzzle instances, originating from the same problems. We collect a set of 700 puzzles that are random instances of 35 problems for which both our bootstrapping enumerative models solved the default puzzle. At $M = 10^4$, the random forest and Transformer models solved 75% and 79%, respectively. As a reference, the uniform model solves only 62% of these puzzles.

**5.1 User study**

In a small user study, 21 participants with varying experience in Python programming completed 30 puzzles. Each puzzle was allocated a maximum of 6 minutes to solve, and the study was conducted virtually using Jupyter notebooks. Participants were employees at a major software company and were recruited by email and at a hackathon. No compensation was offered. Participants were first given a short tutorial about puzzles and how to submit solutions. The user study files are available in the open-source dataset, and further details (including the 30 puzzles) can be found in Appendix G.
The first finding is that success in puzzles correlates with programming experience. For our retrospective study analysis, we split the participants by the median years of Python programming experience. We had 10 beginners with less than three years of experience, and 11 experienced participants with at least three years. We find that 9 of the 30 puzzles were solved by all beginners, while 17 of the puzzles were solved by all experienced participants. Also, beginners spent on average 194 seconds per puzzle, while experienced spent only 149 seconds on average. The average solving time provides a useful proxy to the perceived difficulty of each puzzle. Overall, we see that puzzles are easier for experienced programmers, indicating their value for evaluating programming proficiency.

The second finding is that experienced human programmers outperformed our AI baselines. There were over 10 puzzles that none of our baselines solved, while the average number of puzzles solved by experienced programmers was greater than 25, and one participant solved all 30. However, bear in mind that these results are with a 6 minute limit. No human would solve the puzzles within seconds, and we would expect experienced programmers to eventually solve virtually all study puzzles.

Finally, we find that difficult puzzles for humans are also harder for AI. Figure 4 shows that most of the puzzles solved by AI solvers are the ones that are easier for humans (i.e., solved faster). To compare the two, we define a puzzle’s perceived difficulty score as the average solving time for humans and the number of required tries for machines. For unsolved puzzles, we set the difficulty to the maximum value, and then normalize to [0, 1]. The results show a strong correlation between the two. Specifically, the Spearman’s rank coefficient of humans with B. Transformer is 0.443, and with B. GPT-3 is 0.512. The correlation is even higher (0.493 and 0.541, respectively) when comparing only to beginner programmers. On the one hand, this suggests that additional computational power might allow AI solvers to match humans. However, as Figure 3 shows, this improvement is logarithmic, leading to diminishing returns. Encouragingly, we see that carefully designed solvers can utilize past experience to significantly reduce the required attempts. We hope that our puzzles will support the research and development of new AI solvers that will solve more puzzles with less computational effort.

6 Related Work

Program synthesis problems take drastically different forms for different applications, often resulting in one-off evaluations rather than common datasets. A major paradigm is Programming by Example (PBE) where problems are specified by input-output examples. For instance, several studies focus on text processing [20] or robot navigation [42]. While convenient for end user applications (e.g., many in [43]), PBE alone is inadequate to objectively describe most algorithmic programming challenges. A recent ARC dataset [12] adopts PBE for evaluating abstraction and reasoning in AI, but, like in all PBE applications, the examples’ inherent ambiguity makes the evaluation non-objective.

Program synthesis from formal specifications has a long history of study. See Gulwani et al. [21] for a survey of methods, benchmarked by e.g., the SyGuS competition [1]. In this setting, however, the AI system has to synthesize an algorithm that correctly and efficiently solves a problem on all inputs (and often prove correctness as well). Writing and testing the formal specification is often non-trivial.
English descriptions, often mixed with examples, are becoming an increasingly popular problem representation as LMs improve [26, 31, 60]. In independent work, Hendrycks et al. [24] created a large dataset of English programming problems with examples on which they fine-tuned GPT models. In another concurrent work, the Codex model that powers the new GitHub Copilot auto-completion tool [11] was evaluated with short problem descriptions paired with a set of unit tests that should validate the described specification. This representation of problems is more natural for people, but conflates understanding of natural language with programming ability (see Appendix F for a comparison of English competition problems and our puzzles).

The recent CodeXGLUE benchmark [37] collected a number of code-related datasets and tasks, including code generation. Their main evaluation metric for generations is CodeBLEU [46] which relies on the AST and other code-specific aspects for improving code-to-code comparison. However, this evaluation still requires reference solutions and, therefore, does not resolve the answer-key bias with ambiguous specifications. Our puzzles do not depend on natural language descriptions and do not require any unit tests or reference solutions for evaluation.

Several neighboring fields that have made substantial progress in reasoning include theorem proving [6], two-player game playing [53], and SAT-solving [7]. In all of these fields, important progress has been made by encoding the problems, be they theorems, game rules, or optimization problems, in machine-readable formats that do not involve the ambiguities of natural language.

To our knowledge, our work has the first controlled evaluation of GPT-3 for synthesizing programs, though it has been studied carefully for other purposes [10]. Prototypes and deployed applications of GPT-3 driven code generation exist [11, 34, 51] but they lack public systematic evaluation. In concurrent work [24], GPT-3 was found to perform poorly at the task of synthesizing competitive-programming problem solutions without fine-tuning. However, with fine-tuning, GPT-based networks did solve such challenges. Also, compared to many domain-specific languages used in other work on program synthesis, Python is a significantly more complex language, though it is much simpler than English.

7 Conclusions

We introduce Python Programming Puzzles (P3), an open-source dataset with puzzles described only in source code. As discussed in §3, the puzzle framework is limited to capturing only NP challenges. Yet, puzzles cover a wide range of interesting challenges, and allow fast and objective evaluation, thereby supporting test-time bootstrapping. We implemented and evaluated several enumerative program-synthesis and autoregressive baselines, and found a positive correlation between their puzzle performance and the difficulty for human programmers. Also, we found bootstrapping using past solutions to improve performance more than simply using additional computational effort. In future work, we plan to use and extend the dataset ourselves, and we hope others will as well.

Acknowledgments. We would like to thank Maria Mykhailova for suggesting doing a Python Programming Puzzles Hackathon. We are especially grateful to the participants in our user study and hackathon. We are grateful to the creators of GPT-3 and to Nicolò Fusi for suggesting using it. We would like to thank David Alvarez Melis and Alec Helbing for suggesting quine puzzles. We are grateful to Ana-Roxana Pop for helpful discussions and feedback. We also thank Tianxiao Shen and the rest of the MIT NLP group members for valuable feedback on the writing.

References


Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes] as mentioned at the end of the introduction, our main contributions are (a) the puzzles framework; (b) the P3 dataset; and (c) evaluating baseline models.
   (b) Did you describe the limitations of your work? [Yes] See Section 3 and Conclusions: puzzles do not cover all types of challenges. Also, directly comparing human- and machine-performance is problematic so it depends on the available computational resources. Instead, we focus on the perceived difficulty.
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix D

3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See Appendix B-C for implementation details. We publicly share the data, and will privately share our baselines code with reviewers (we need to internally approve the code before publicly releasing).
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix B-C for details. As are whole setting is unsupervised, we did not perform any hyperparameter tuning and use the same values across model variants. The default values are provided in the running scripts with the code.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] and [No]. See Appendix G for analysis of the user study (specifically, Figures G.8-G.10). Figure 3 includes standard deviation for the enumerative models across three runs. We did not rerun the GPT-3 solvers more than once per setting yet.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix B-C

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes] See the linked GitHub repository.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] See the linked GitHub repository in footnote 1, page 1.
   (d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes] See the figures in G
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes] See 5.1 and G for details about the recruiting and agreement for the user study.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes] See 5.1 and G for details about the recruiting and agreement for the user study.