
Python Programming Puzzles (P3)

This repo contains a dataset of Python programming puzzles which can be used to teach and evaluate an AI's programming proficiency. We present code generated by OpenAI's codex neural network

solving many of these puzzles. We hope this dataset will **grow rapidly**, and it is already diverse in terms of problem difficulty, domain, and algorithmic tools needed to solve the problems. Please propose a new puzzle or browse newly proposed puzzles or contribute through pull requests.

To learn more about how well AI systems such as GPT-3 can solve these problems, read our two papers:

Programming Puzzles. Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, Adam Tauman Kalai. In *Proceedings of the Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track* (NeurIPS), 2021.

```
1 @inproceedings{
2   schuster2021programming,
3   title={Programming Puzzles},
4   author={Tal Schuster and Ashwin Kalyan and Alex Polozov and Adam Tauman
5     Kalai},
6   booktitle={Thirty-fifth Conference on Neural Information Processing
7     Systems Datasets and Benchmarks Track},
8   year={2021},
9   url={https://arxiv.org/abs/2106.05784}
```

To reproduce the results in that paper, see the solvers folder.

NEW self-teaching: In our second paper, we have Language Models (LMs) **generate their own puzzles** and, together with the Python interpreter, improve their own puzzle solving capability. Following our paper (arXiv, 2022), there have been several papers where an LM improves itself by checking its own solutions. However, our approach is potentially more powerful because we have the LM generate its own problems, not only its own solutions.

Language Models Can Teach Themselves to Program Better. Patrick Haluptzok, Matthew Bowers, Adam Tauman Kalai. In *Proceedings of the Eleventh International Conference on Learning Representations* (ICLR), 2023.

```
1 @inproceedings{
2   haluptzok2022selfteach,
3   title={Language Models Can Teach Themselves to Program Better},
4   author={Patrick Haluptzok, Matthew Bowers, Adam Tauman Kalai},
5   booktitle={Eleventh International Conference on Learning
6     Representations (ICLR)},
7   year={2023},
```

```
7 url=https://arxiv.org/abs/2207.14502}
8 }
```

To reproduce the results in that paper, see the ICLR2023 folder.

If you just want to dive right into solving a few puzzles, try the intro notebook at Binder that shows which puzzles the AI baselines solved and which they did not, so you can see how your programming compares.

What is a Python programming puzzle?

Each puzzle takes the form of a Python function that takes an answer as an argument. The answer is an input which makes the function return `True`. This is called *satisfying* the puzzle, and that is why the puzzles are all named `sat`.

```
1 def sat(s: str):
2     return "Hello " + s == "Hello world"
```

The answer to the above puzzle is the string `"world"` because `sat("world")` returns `True`. The puzzles range from trivial problems like this, to classic puzzles, to programming competition problems, all the way through open problems in algorithms and mathematics.

The classic Towers of Hanoi puzzle can be written as follows:

```
1 def sat(moves: List[List[int]]):
2     """
3     Eight disks of sizes 1-8 are stacked on three towers, with each
4     tower having disks in order of largest to
5     smallest. Move [i, j] corresponds to taking the smallest disk off
6     tower i and putting it on tower j, and it
7     is legal as long as the towers remain in sorted order. Find a
8     sequence of moves that moves all the disks
9     from the first to last towers.
10    """
11    rods = ([8, 7, 6, 5, 4, 3, 2, 1], [], [])
12    for [i, j] in moves:
13        rods[j].append(rods[i].pop())
14        assert rods[j][-1] == min(rods[j]), "larger disk on top of
15        smaller disk"
16    return rods[0] == rods[1] == []
```

The shortest answer is a list of 255 moves, so instead we ask for the AI to generate *code* that outputs an answer. In this case, the codex API generated the following code:

```
1 def sol():
2     # taken from https://www.geeksforgeeks.org/c-program-for-tower-of-
3     hanoi/
```

```

3     moves = []
4     def hanoi(n, source, temp, dest):
5         if n > 0:
6             hanoi(n - 1, source, dest, temp)
7             moves.append([source, dest])
8             hanoi(n - 1, temp, source, dest)
9     hanoi(8, 0, 1, 2)
10    return moves

```

This was not on its first try, but that is one of the advantages of puzzles—it is easy for the computer to check its answers so it can generate many answers until it finds one. For this puzzle, about 1 in 1,000 solutions were satisfactory. Clearly, codex has seen this problem before in other input formats—it even generated a url! (Upon closer inspection, the website exists and contains Python Tower-of-Hanoi code in a completely different format with different variable names.) On a harder, less-standard Hanoi puzzle variant that requires moving from particular start to end positions, codex didn't solve it on 10,000 attempts.

Next, consider a puzzle inspired by this easy competitive programming problem from codeforces.com website:

```

1  def sat(inds: List[int], string="Sssuubbstrissingg"):
2      """Find increasing indices to make the substring "substring""""
3      return inds == sorted(inds) and "".join(string[i] for i in inds) ==
        "substring"

```

Codex generated the code below, which when run gives the valid answer [1, 3, 5, 7, 8, 9, 10, 15, 16]. This satisfies this puzzle because it's an increasing list of indices which if you join the characters "Sssuubbstrissingg" in these indices you get "substring".

```

1  def sol(string="Sssuubbstrissingg"):
2      x = "substring"
3      pos = string.index(x[0])
4      inds = [pos]
5      while True:
6          x = x[1:]
7          if not x:
8              return inds
9          pos = string.find(x[0], pos+1)
10         if pos == -1:
11             return inds
12         inds.append(pos)

```

Again, there are multiple valid answers, and again this was out of many attempts (only 1 success in 10k).

A more challenging puzzle that requires dynamic programming is the longest increasing subsequence problem which we can also describe with strings:

```
1 def sat(x: List[int], length=20, s="Dynamic programming solves this
  classic job-interview puzzle!!!"):
2     """Find the indices of the longest substring with characters in
    sorted order"""
3     return all(s[x[i]] <= s[x[i + 1]] and x[i + 1] > x[i] for i in
        range(length - 1))
```

Codex didn't solve this one.

The dataset also has a number of open problems in computer science and mathematics. For example, Conway's 99-graph problem is an unsolved problem in graph theory (see also Five \$1,000 Problems (Update 2017))

```
1 def sat(edges: List[List[int]]):
2     """
3     Find an undirected graph with 99 vertices, in which each two
    adjacent vertices have exactly one common
4     neighbor, and in which each two non-adjacent vertices have exactly
    two common neighbors.
5     """
6     # first compute neighbors sets, N:
7     N = {i: {j for j in range(99) if j != i and ([i, j] in edges or [j,
    i] in edges)} for i in range(99)}
8     return all(len(N[i].intersection(N[j])) == (1 if j in N[i] else 2)
        for i in range(99) for j in range(i))
```

Why puzzles? One reason is that, if we can solve them better than human programmers, then we could make progress on some important algorithms problems. But until then, a second reason is that they can be valuable for training and evaluating AI systems. Many programming datasets have been proposed over the years, and several have problems of a similar nature (like programming competition problems). In puzzles, the spec is defined by code, while other datasets usually use a combination of English and a hidden test set of input-output pairs. English-based specs are notoriously ambiguous and test the system's understanding of English. And with input-output test cases, you would have to have solved a puzzle before you pose it, so what is the use there? Code-based specs have the advantage that they are unambiguous, there is no need to debug the AI-generated code or fears that it doesn't do what you want. If it solved the puzzle, then it succeeded by definition.

For more information on the motivation and how programming puzzles can help AI learn to program, see the paper:

Programming Puzzles, by Tal Schuster, Ashwin Kalyan, Alex Polozov, and Adam Tauman Kalai. 2021 (Link to be added shortly)

Click here to browse the puzzles and solutions

The problems in this repo are based on: * Wikipedia articles about algorithms, puzzles, and math problems. * The website codeforces.com, a popular website for programming competition problems * Olympiad problems from the International Collegiate Programming Contest and International Mathematical Olympiad. * (NEW!) Problems inspired by the human-eval dataset from the codex paper.

Notebooks

The notebooks subdirectory has some relevant notebooks. Intro.ipynb has a dozen puzzles indicating which ones the AI solved and did not Try the notebook at Binder and see how your programming compares to the AI baselines!

Demo.ipynb has the 30 problems completed by our users in a user study. Try the demo notebook and see how your programming compares to the AI baselines!

Hackathon

During a Microsoft hackathon July 27-29, 2020, several people completed 30 user study puzzles. We also had tons of fun making the puzzles in Hackathon_puzzles.ipynb. These are of a somewhat different flavor as they are more often [hacks](#) like

```
1 def sat(x):
2     return x > x
```

where the type of `x` is clearly non-standard. The creators of these puzzles include github users: Adam Tauman Kalai, Alec Helbling, Alexander Vorobev, Alexander Wei, Alexey Romanov, Keith Battauchi, Kodai Sudo, Maggie Hei, Mariia Mykhailova, Misha Khodak, Monil Mehta, Philip Rosenfield, Qida Ma, Raj Bhargava, Rishi Jaiswal, Saikiran Mullaguri, Tal Schuster, and Varsha Srinivasan. You can try out the notebook at (link to be added).

Highlights

- Numerous trivial puzzles like reversing a list, useful for learning to program
- Classic puzzles like:
 - Towers of Hanoi
 - Verbal Arithmetic (solve digit-substitutions like SEND + MORE = MONEY)
 - The Game of Life (e.g., finding oscillators of a given period, some **open**)

-
- Chess puzzles (e.g., knight's tour and n-queen problem variants)
 - Two-player games
 - Finding optimal strategies for Tic-Tac-Toe, Rock-Paper-Scissors, Mastermind (to add: connect four?)
 - Finding minimax strategies for zero-sum bimatrix games, which is equivalent to linear programming
 - Finding Nash equilibria of general-sum games (**open**, PPAD complete)
 - Math and programming competitions
 - International Mathematical Olympiad (IMO) problems
 - International Collegiate Programming Contest (ICPC) problems
 - Competitive programming problems from codeforces.com
 - Graph theory algorithmic puzzles
 - Shortest path
 - Planted clique (open)
 - Elementary algebra
 - Solving equations
 - Solving quadratic, cubic, and quartic equations
 - Number theory algorithmic puzzles:
 - Finding common divisors (e.g., using Euclid's algorithm)
 - Factoring numbers (easy for small factors, over \$100k in prizes have been awarded and **open** for large numbers)
 - Discrete log (again **open** in general, easy for some)
 - Lattices
 - Learning parity (typically solved using Gaussian elimination)
 - Learning parity with noise (**open**)
 - Compression
 - Compress a given string given the decompression algorithm (but not the compression algorithm), or decompress a given compressed string given only the compression algorithm
 - (to add: compute huffman tree)
 - Hard math problems
-

-
- Conway's 99-graph problem (**open**)
 - Finding a cycle in the Collatz process (**open**)

Train-test split

The file `split.json` contains a suggested train-test split. This split was hand-selected by the puzzle authors, who are familiar with all puzzles, so that: there is minimal overlap between related puzzles in the two splits. In particular, for pairs of related puzzles, either both were placed in the training set or the test set.

Contributing

This project welcomes contributions and suggestions. Use your creativity to help teach AI's to program! See our wiki on how to add a puzzle.

Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit <https://cla.opensource.microsoft.com>.

When you submit a pull request, a CLA bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., status check, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the Microsoft Open Source Code of Conduct. For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

See the datasheet for our dataset.

Trademarks

This project may contain trademarks or logos for projects, products, or services. Authorized use of Microsoft trademarks or logos is subject to and must follow Microsoft's Trademark & Brand Guidelines. Use of Microsoft trademarks or logos in modified versions of this project must not cause confusion or imply Microsoft sponsorship. Any use of third-party trademarks or logos are subject to those third-party's policies.