Programming Puzzles

Github.com/Microsoft/PythonProgrammingPuzzles

Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, Adam Tauman Kalai

Puzzles

Goal: A unified problem representation for teaching AI models to code and objectively evaluating their progress.

What is a programming puzzle?

A **function** `f(y,x)` in any programming language (e.g., Python) that **returns a Boolean** value.

The challenge is to **find the input `y` that satisfies** the function within time limit *t* (i.e., makes it return **True**).

A **solution** `g(x)` is the source code to generate `y`.

Example:

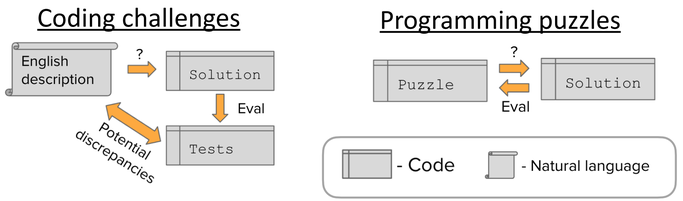
```
def f(y: str, x="Hello world"):
    return "Hello " + y == x

def g(x):   # solution.
    y = x.split(" ")[-1]
    return y  # answer.

assert f(g())
```

A **solver** takes *n* puzzles and predicts solutions.

Why are puzzles important?

Models are getting better at code generation. We need an **objective evaluation** of coding proficiency to measure and spur progress.

Pure code evaluation



Puzzles focus on the algorithmic challenge; not mixing with world knowledge or English, and no hidden test cases.

The P3 dataset

A large (and growing) collection of Python puzzles.

Available in a Github repository/ json files with reference human-written and AI solutions.

Currently has 397 problems:

Comprehensive in domain

Algebra; Basic Python programming; Chess; Classic puzzles; Programming challenges; Compression; Conway's game of life; Games; Graphs; ICPC; IMO; Number Theory; Probability; ...

Comprehensive in required algorithmic tools

Recursion; Linear programming; Dynamic programming; Convex optimization; Sorting; Graph search; Programming language specific operations (such as string manipulations); ...

Comprehensive in difficulty

From trivial puzzles to major open, prize offered, algorithmic and math problems.

Evaluation

Straight-forward self-validation (simply executing the code).

Zero/few-shot: given a puzzle, try to solve it with as few tries as possible.

Test time bootstrapping: given a collection of puzzles, try to solve as many as possible with limited tries per puzzle.

Self-validation allows the solver to identify valid solutions, learn from them, and **improve the solution search for harder puzzles**.

Example of Python puzzles

Three puzzles with varying difficulties and domains:

```
# Find a string that when reversed and concatenated with "world" gives "Hello world"
def f1(y: str):
    return y[::-1] + "world" == "Hello world"
```

Comments for presentation only

```
# Tower of Hanoi, often teaches recursion. Move [i, j] means move top disk on tower i to j, with 1 ≤ i,j ≤ 3
def f2(moves: List[List[int]], num_disks=8):
    state = [1] * num_disks  # All disks start at tower 1.
    for [i, j] in moves:
        assert state.index(i) <= (state + [1, 2, 3]).index(j), "bigger disk on top"
        state[state.index(i)] = j  # Move smallest disk from tower i to tower j.
    return state == [3] * num_disks  # All disks must end on tower 3.
```

```
# Find a non-trivial integer factor d of a large number n
def f3(d: int, n=100433627766186892221372630609062766858404681029709092356097):
    return 1 < d < n and n % d == 0
```

Valid solution to `f2` generated by Codex-cushman (Med. prompt):

```
def g6(num_disks=8):
    # Algorithm is equivalent to moving all disks.
    # From https://en.wikipedia.org/wiki/Tower_of_Hanoi#Adv[...]
    def hanoi(n, p, q, r):
        if n > 0:
            hanoi(n - 1, p, r, q)
            moves.append([p, r])
            hanoi(n - 1, q, p, r)
    moves = []
    hanoi(num_disks, 1, 2, 3)
    assert f6(moves, num_disks)
    return moves
```

Experiments

Enumerative solvers

Iterate over ASTs by their *learned* likelihood.

Uniform: unparameterized; iterating from shortest to longest.

Random forest: puzzles are encoded as bag-of-rules.

Transformer: pretrained RoBERTa to encode puzzles and rules.

Language model solvers

Using **GPT-3** or **Codex**, generate solutions as strings. Prompts:

Short: zero-shot.

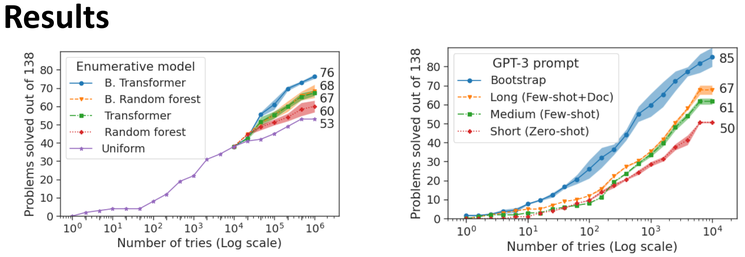
```
def f(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2

assert True == f(...
```

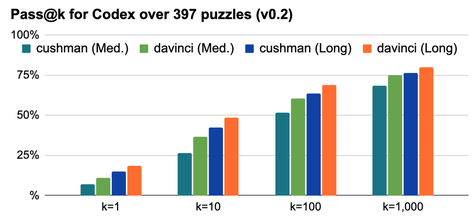
Medium: five-shot (using 5 tutorial puzzles).

Long: five-shot + English descriptions to the puzzles.

Bootstrap: starts from zero-shot and adds examples to the prompt as they are found.

Results





Significant room for improvement for autocompletion (small k)

User study

21 participants (10 beginners; 11 experienced) were given up to 6 minutes per puzzle to solve 30 puzzles.

Very positive feedback.

Positive correlation in perceived difficulty between humans and AI.

Puzzles evaluate the algorithmic proficiency of AI models and allow comparisons against human coders.

Codex-davinci results