The Larger They Are, the Harder They Fail: Language Models do not Recognize Identifier Swaps in Python

Antonio Valerio Miceli-Barone[1*]
amiceli@ed.ac.uk

Fazl Barez[1*]
f.barez@ed.ac.uk

Ioannis Konstas[2]
i.konstas@hw.ac.uk

Shay B. Cohen[1]
scohen@inf.ed.ac.uk

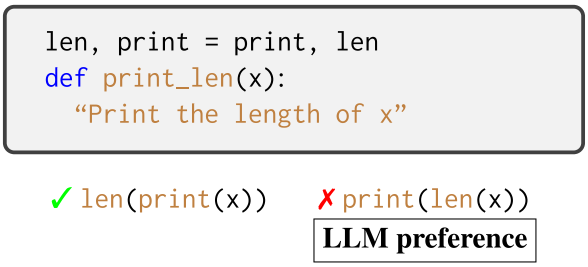
[1] School of Informatics, University of Edinburgh
[2] School of Mathematical and Computer Sciences, Heriot-Watt University

Abstract

Large Language Models (LLMs) have successfully been applied to code generation tasks, raising the question of how well these models understand programming. Typical programming languages have invariances and equivariances in their semantics that human programmers intuitively understand and exploit, such as the (near) invariance to the renaming of identifiers. We show that LLMs not only fail to properly generate correct Python code when default function names are swapped, but some of them even become more confident in their incorrect predictions as the model size increases, an instance of the recently discovered phenomenon of *Inverse Scaling*, which runs contrary to the commonly observed trend of increasing prediction quality with increasing model size. Our findings indicate that, despite their astonishing typical-case performance, LLMs still lack a deep, abstract understanding of the content they manipulate, making them unsuitable for tasks that statistically deviate from their training data, and that mere scaling is not enough to achieve such capability.

1 Introduction

Pretrained Large Language Models (LLMs) are rapidly becoming one of the dominant paradigm for large variety of language tasks (Brown et al., 2020a; Chowdhery et al., 2022), including programming code generation and completion (Chen et al., 2021; Li et al., 2022). LLMs have demonstrated increasing performance with increasing model size[1] on many practical tasks (Kaplan et al., 2020; Hernandez et al., 2021) including programming tasks (Nijkamp et al., 2022), recently, however, researchers have identified a number of tasks that exhibit *inverse scaling*, where output quality decreases, rather than increase, with increasing model size.



```
len, print = print, len
def print_len(x):
  “Print the length of x”
```

✓ `len(print(x))` ✗ `print(len(x))`

LLM preference

Figure 1: Given a Python prompt (on top) which swaps of two builtin functions, large language models prefer the incorrect but statistically common continuation (right) to the correct but unusual one (left).

Tasks with inverse scaling generally either involve social biases (Parrish et al., 2022; Srivastava et al., 2022), where the larger models (arguably correctly) learn undesirable biases from biased training sets, or involve examples of natural language that are highly atypical but still easily understandable by a human (McKenzie et al., 2022b). These tasks may involve unusual discourse pragmatics or they may require reasoning about counterfactual knowledge, however, since they tend to be highly artificial, it could perhaps be argued that they are edge cases which may not represent serious failure modes for practical applications. In this paper we present a novel type of inverse scaling task involving Python code generation under a redefinition of default identifiers. This has both practical implications (redefinition of default identifiers is a metaprogramming technique used in popular libraries), and broader scientific implications, as it shows that LLMs fail to reason about the deep, abstract semantic structure of programming languages, and these flaws are not ameliorated, but in fact may be even worsened, by increasing model size.

Programming languages have precise and well-

---

[*] Equal contribution.

[1] Since model capacity in number of parameters and pretraining dataset size are balanced according to a design law that is fixed for each model family and is intended to empirically maximize the pretraining set likelihood given a compute budget (Kaplan et al., 2020), for the reminder of this paper we will jointly refer to them as "model size".

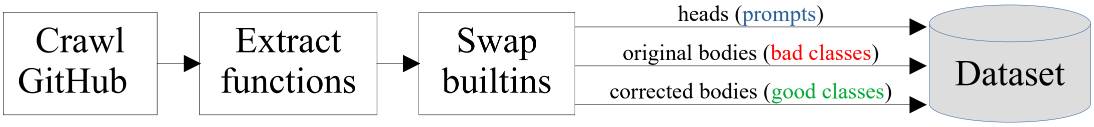


Figure 2: Data generation pipeline (see Appendix D for an example): 1. Crawl repositories from GitHub, filtered by language, license, stars, and size. 2. Extract top-level functions with docstrings and references to at least two callable builtins 3. For each function, choose two builtins to swap and generate: a) header with builtin swap statement, function declaration with decorators, docstring b) original function body, c) corrected body with the builtins swapped consistently with the swap statement. 4. Store as a binary classification task: a) head = classifier input, b) original body = bad class, c) corrected body = good class.

defined syntax and semantics which makes them especially suited to automatic analysis and procedural generation. They are scientifically interesting because they can be used for automatic generation of examples of coding problems and their evaluation against an objective ground truth, whereas most NLP tasks have enough ambiguity that require human annotation in order to produce high-quality examples. Furthermore, this research is also of practical importance for software engineering tools that use LLMs, such as GitHub Copilot,[2] which are starting to be widely adopted by developers.

2 Methodology

We describe the motivation behind our task (§2.1) and the task itself (§2.2), followed by the way we collected the data for the task (§2.3).

We release our dataset as well as the code used to generate it and replicate our experiments.[3]

2.1 Task Motivation

Turing-complete languages have invariances and equivariances, making it possible to express the same function by multiple programs (see Appendix H for formal definitions). While determining semantic equivalence is undecidable in the general case (Rice, 1953), sometimes it can be determined by pure syntactic analysis. For instance, α-equivalence, invariance under the consistent renaming of identifiers such as variable or function names, can be decided using syntactic analysis.

Proper understanding of the semantics of a programming language requires identifying its invariances and equivariances, as opposed to "shortcut learning" (Geirhos et al., 2020) which instead exploits many weak, spurious correlations that do not generalize out of the observed data distribution. We propose a task based on the approximate α-equivalence of Python code, in order to evaluate how well LLMs master the semantics of Python.

2.2 Task Description

We consider code snippets in Python 3. Python allows to redefine *builtin* functions[4] by reassigning their identifiers. For instance, the statement

```
len, print = print, len
```

swaps the identifiers for the builtin functions `len` and `print`. Any function defined following that identifier swap would have to refer to the builtin function `len` by the identifier `print` and vice versa.

We consider a code generation task where the model is given a top-level function *declaration*, followed by a *docstring* (which typically describes the behavior of the function in natural language) and has to generate the rest of the body of the function, similar to Miceli Barone and Sennrich (2017), but with the caveat that we prepend to the declaration a *statement* that swaps two Python builtin functions that are expected to be used in the function body. Specifically, in line with the format of the Inverse Scaling Prize,[5] we define our **Builtin identifier swap** task as a binary classification task where the input of each example is the concatenation of a swap statement, function declaration (with optional decorators) and docstring. A "bad" output for such input is a function body that uses the builtin functions according to their usual meaning, ignoring the swap statement. In contrast, the "good" output is a function body where the builtin functions are used consistently with the swap statement. To assess the success of the model in distinguishing between the "bad" and the "good" output, we compute the likelihood of each output given the input provided as a prompt (Figure 1, Appendix D).

---

[2] `https://github.com/features/copilot`

[3] `https://github.com/Avmb/inverse_scaling_prize_code_identifier_swap.git`

[4] Predefined functions that the language exposes to the user.

[5] `https://github.com/inverse-scaling/prize`

2.3 Data Collection

Similar to Miceli Barone and Sennrich (2017), our dataset collection procedure involves scraping code from GitHub using the `PyCodeSuggest` library[6] (Bhoopchand et al., 2016) to download Python repositories with at least 100 stars, of size at most 200 MB and which mention the use of the Open Source CC-BY-4.0 license[7] in their README. Our final dataset includes 559 repositories downloaded on 16 December 2022. We then parse the **.py** files in each repository with the Python 3 `ast` module to make sure that they contain valid code. We extract 1,000 randomly chosen top-level functions that each contain a docstring and that reference at least two callable builtin identifiers, as defined by the `builtins` module. For each of these extracted functions, we randomly choose two builtin functions and generate the corresponding swap statement, function declaration (with decorators) and docstring as the example prompt, the original function body (regenerated from the abstract syntax tree with the `astunparse` module[8]) as the "bad" output and the function body where the two selected builtins are swapped consistently with the swap statement as the "good" output (Figure 2).

Note that functions can in principle access the builtin identifiers as strings using reflection and evaluation facilities, which may require a full static analysis of the code to identify and is undecidable in the general case. Since our method uses purely syntactic substitutions, there might be cases where the "good" outputs do not maintain the expected function behavior. In practice, this dynamic access of identifiers at runtime is rare with builtin identifiers and therefore does not pose an issue.

3 Experiments

We next describe our experiments with a likelihood calculation of correct and incorrect completions (§3.1) and chat LLMs (§3.2), and then present a qualitative analysis (§3.3).

Computational resources We spent approximately 130 US dollars, including donated credits, to use the OpenAI LLMs through their publicly accessible API.

We also used a small amount of machine-hours on the Baskerville Tier 2 HPC platform[9] equipped with NVIDIA A100 GPUs. While this is a high-end system, our experiments on the open source models can be also practically run on consumer-grade machines with gaming GPUs.

3.1 Completion Likelihood

For our main set of experiments, we evaluate our dataset on families of auto-regressive language models (OpenAI GPT-3, Salesforce CodeGen, Meta AI OPT) and one family of sequence-to-sequence conditional auto-regressive language models (Google FLAN-T5). All models are based on the Transformer architecture (Vaswani et al., 2017) and pretrained on large datasets scraped from the Internet (full details in Appendix A).

Results We evaluate our datasets on the models using a modified version of the Inverse Scaling Prize evaluation code.[10] We report the results for all models in Figure 3. The graphs show the classification loss averaged over the examples for each model, with standard errors represented as error bars.

Model family	Pearson	Spearman	Kendall
OPT	**0.94**	**0.83**	**0.73**
GPT-3	**0.97**	**1.00**	**1.00**
InstructGPT	**0.94**	0.80	0.67
CodeGen-multi	0.46	0.40	0.33
CodeGen-mono	0.10	0.20	0.00
GPT-Codex	-1.00	-1.00	-1.00
FLAN-T5	0.01	0.10	0.00

Table 1: Correlation coefficients between log-model size and log-loss for each model family. Bolded values indicate inverse scaling at p-value < 0.1. The two text-based GPT-3.5 models (`text-davinci-002` and `text-davinci-003`) are not included in this analysis.

All tested models always prefer the incorrect output resulting in zero classification accuracy, the log-likelihood of the incorrect output is always significantly higher than the uniform baseline, but it varies with the model. Specifically:

- The Meta AI OPT and OpenAI text-based GPT-3 families exhibit strong inverse scaling, with the larger models more strongly preferring the incorrect output. The trend is monotonic for the "First

---

[6] https://github.com/uclnlp/pycodesuggest

[7] https://creativecommons.org/licenses/by/4.0/

[8] https://pypi.org/project/astunparse/

[9] https://www.baskerville.ac.uk/

[10] Original: https://github.com/naimenz/inverse-scaling-eval-pipeline, our version: https://github.com/Avmb/inverse-scaling-eval-pipeline.git

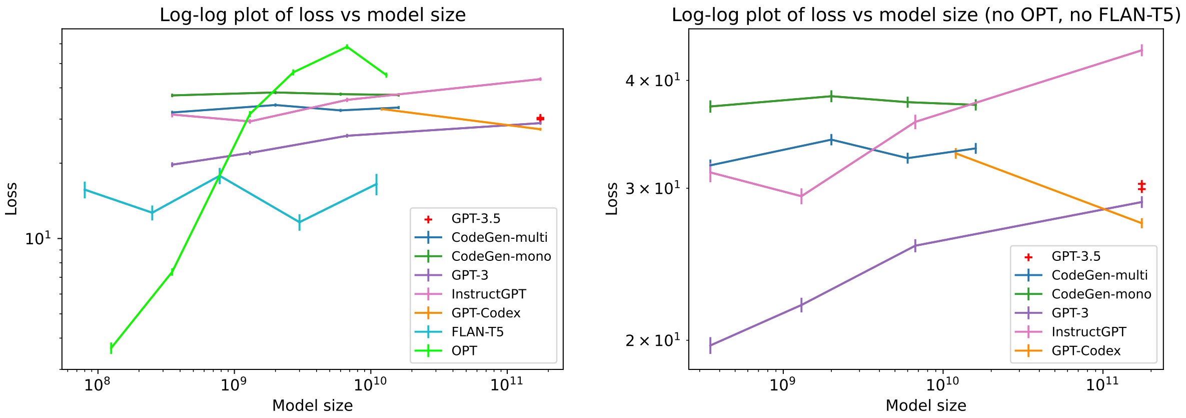


Figure 3: Classification loss over model size. Left: all models. Right: all models except Meta AI OPT and Google FLAN-T5 families.

generation" GPT-3 family, and somewhat non-monotonic for the OPT and InstructGPT families. The InstructGPT models perform worse than the base GPT-3 models.

- The Salesforce CodeGen models exhibit mostly flat scaling. The “mono” models which are further fine-tuned on Python-only data perform worse than the "multi" models they are based on.
- The OpenAI Codex models are the only models that seem to show positive scaling (which may be spurious since they are only two data points). However, the two GPT-3.5 models (`text-davinci-002` and `text-davinci-003`, shown in the figures as red crosses) that further fine-tune `code-davinci-002` on English demonstrations, lose their edge and end up performing worse than the base GPT-3 model of the same size (`davinci`).
- Google FLAN-T5 shows an unclear, oscillating scaling trend, with large error bars at each point.

We report numerical correlation results between model size and mean loss[11] in Table 1. Due to the small number of model sizes per family, some of the p-values are quite high, but the numerical results are consistent with the qualitative analysis.

Overall, our analysis shows that autoregressive text-based LLMs (even when previously pretrained on code-based models) exhibit inverse scaling on our task, while the code-based models exhibit flat scaling which might possibly transition to positive scaling at the largest tested size, but fail to substantially improve over the text-based models.

3.2 Chat LLMs Accuracy

We perform additional experiments on chat LLMs by OpenAI and Anthropic, whose APIs became recently available. These models constrain both the input text and the generated output to take the form of a dialogue between the user and the "assistant" (the model itself). Notably, the APIs of these models do not report log-probabilities, hence they cannot be used to score arbitrary texts. This prevents us from using the same experimental protocol of the other experiments. We instead reformulate the task as binary classification where the model is presented with both the correct and incorrect forms of the same program in the same user message and is asked to select the correct one. We describe the models and the prompt templates in Appendix C.

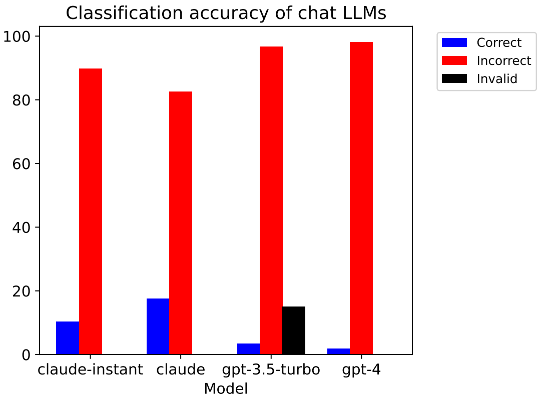


Figure 4: Classification accuracy on chat LLMs. For each model, we report the percentage of correct, incorrect and invalid outputs.

Results We report the results in Figure 5. All the models strongly prefer the incorrect programs,

---

[11] in the log-log scale, which for Pearson’s correlation measures the adherence to the (inverse of) power law scaling as described by Kaplan et al. (2020).

although the classification accuracy is non-zero. This may not be necessarily comparable to the zero classification accuracy of the previous experiments, due to the different experimental protocol. The Anthropic models (`claude-instant` and `claude`) show better accuracy (10-18%) with positive scaling and never produce invalid outputs. The OpenAI models (`gpt-3.5-turbo` and `gpt-4`) show low accuracy ($< 4\%$) with flat or inverse scaling and occasionally produce invalid outputs.

3.3 Qualitative Experiments

We perform a small number of manual two-shot experiments on GPT-3.5. We also carry out manual experiments on OpenAI ChatGPT-3.5[12] and GPT-4 models, where we interact with the models in multiple rounds of dialogue, trying to hint the correct solution. The models are still unable to provide the correct continuations. See Appendices E–G.

4 Related work

Recent work sought to characterize the quality of LLMs on a variety of tasks: BIG-bench (Srivastava et al., 2022) is a large collaboration which resulted in a suite of hard, disparate tasks which were used to evaluate various LLMs. The study found that scaling can be slower and less smooth than expected by naive scaling laws, and social biases sometimes show inverse scaling, also observed by Parrish et al. (2022). Perez et al. (2021) investigated the effect of example selection in few-shot learning for LLMs, finding that previous studies generally overestimated model quality due to methodological issues. Lin et al. (2022) attempted to measure the *truthfulness* of the answer provided by LLMs on tasks involving real-world knowledge, finding that while larger models tend to provide more informative answers, they also tend to be less truthful. However, this effect might be confounded due to the dataset design to specifically be adversarial for the largest model being evaluated (Kilcher, 2021). Li et al. (2023) showed that similar to our case, mathematical article processing is sensitive to semi-invariant symbol replacements. Ji et al. (2022) provide a broad survey about hallucination (generation of fluent yet incorrect information) by natural language generation models.

5 Conclusions

We explored the ability of large language models to predict the correct continuations of fragments of Python programs in scenarios where the correct continuations are statistically uncommon due to the redefinition of identifiers caused by a statement that we included in the prompt. Not only all the tested models fail at this task, but some model families even display *inverse scaling*: they become worse, rather than better, with increasing model size. These results suggest that LLMs rely on “shortcut learning”, i.e., weak, unstable, mostly lexical correlations in the data, rather than an understanding of the semantics of the data (in this case, Python code) at a deep level. We believe that our results are important both for a better scientific understanding of the capabilities of LLMs and for their practical relevance as a core technology for automated code generation tools. Future work could investigate scaling effects at larger model sizes, as well as on other programming languages.

Limitations

Our approach has the following limitations:

1. It only considers swaps of pairs of functions at the top-level scope, which is a small set of all the quasi-invariances of the Python programming language.
2. It only considers code generation in top-level functions, hence it does not evaluate class methods.
3. It relies on a syntactic substitution to generate "correct" gold truth outputs, which may fail if the swapped functions are called by a string expression through `eval` or or queried by their string names using the reflection facilities.
4. In our experiments, we can evaluate only a small number of model sizes per family, since these are the only ones available, therefore the p-values of the correlation with the loss analysis are high.
5. The independent reproducibility of the experiments on closed-source models is predicated on the continued availability of a publicly-accessible API. At the time of writing, our experiments on the OpenAI "Codex" models are no longer reproducible without support from OpenAI.

---

[12] `https://openai.com/blog/chatgpt/`

Items 1 and 2 can be in principle treated by considering more complex code transformations, which we leave for future work. Item 3 is harder to tackle in the general case because of undecidability issues. Item 4 could be addressed by reproducing our experiments on a model family that encompasses more model sizes, should it become available for public experimentation. Item 5 is an unavoidable consequence of using closed-source models.

Ethics Statement

We do not perform experiments on human subjects. Our work involves generating a dataset of public data scraped from the GitHub and evaluating it on multiple large language models. We release our dataset and the code used to generate it. We filtered our dataset to make sure that all the data that we used has been relesed under the CC-BY-4.0 license, which in our understanding allows for re-releasing, however our filtering procedure is heuristic which implies that there is the possibility that some of the included data may be in violation of its license. In order to mitigate this hazard, we provide a clearly documented takedown option on the repository on which we will host this data, enabling people to claim copyright and ask for removal of their data.

Acknowledgements

We thank the reviewers for their helpful comments. We thank the Inverse Scaling Prize competition organisers (McKenzie et al., 2022a) for organising the challenge and donating part of the OpenAI API credits that were used in our experiments. We are grateful to Apart Research[13] for their donation that supported the purchase of additional OpenAI API credits and provided personal financial support to Antonio Valerio Miceli-Barone. This work was supported by the UKRI Research Node on Trustworthy Autonomous Systems Governance and Regulation (grant EP/V026607/1) which provided funding for Antonio Valerio Miceli-Barone. The experiments in this work on open source LLMs were supported by a compute grant (UKRI HPC) from the Baskerville service at the University of Birmingham.

References

Avishkar Bhoopchand, Tim Rocktäschel, Earl T. Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network. *ArXiv preprint*, abs/1611.08307.

Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. 2021. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *CoRR*, abs/2104.13478.

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. 2020a. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

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. 2020b. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri 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 Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua 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. 2021. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton,

---

[13] `https://apartresearch.com/`

Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2022. Scaling instruction-finetuned language models.

Taco S. Cohen and Max Welling. 2016. Group equivariant convolutional networks. *CoRR*, abs/1602.07576.

Andreea Deac, Théophane Weber, and George Papamakarios. 2023. Equivariant muzero.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The pile: An 800gb dataset of diverse text for language modeling. *ArXiv preprint*, abs/2101.00027.

Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard S. Zemel, Wieland Brendel, Matthias Bethge, and Felix A. Wichmann. 2020. Shortcut learning in deep neural networks. *ArXiv preprint*, abs/2004.07780.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. 2021. Scaling laws for transfer.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2022. Survey of hallucination in natural language generation. *ACM Comput. Surv.* Just Accepted.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

Yannic Kilcher. 2021. Does gpt-3 lie? - misinformation and fear-mongering around the truthfulqa dataset.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners.

Weixian Waylon Li, Yftah Ziser, Maximin Coavoux, and Shay B. Cohen. 2023. BERT is not the count: Learning to match mathematical statements with proofs. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 3581–3593, Dubrovnik, Croatia. Association for Computational Linguistics.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring how models mimic human falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3214–3252, Dublin, Ireland. Association for Computational Linguistics.

Ian McKenzie, Alexander Lyzhov, Alicia Parrish, Ameya Prabhu, Aaron Mueller, Najoung Kim, Sam Bowman, and Ethan Perez. 2022a. The inverse scaling prize.

Ian McKenzie, Alexander Lyzhov, Alicia Parrish, Ameya Prabhu, Aaron Mueller, Najoung Kim, Sam Bowman, and Ethan Perez. 2022b. Inverse scaling prize: First round winners.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Felix Musil, Andrea Grisafi, Albert P. Bartók, Christoph Ortner, Gábor Csányi, and Michele Ceriotti. 2021. Physics-inspired structural representations for molecules and materials. *Chemical Reviews*, 121(16):9759–9815. PMID: 34310133.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis.

OpenAI. 2023. GPT-4 technical report.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback.

Alicia Parrish, Angelica Chen, Nikita Nangia, Vishakh Padmakumar, Jason Phang, Jana Thompson, Phu Mon Htut, and Samuel Bowman. 2022. BBQ: A hand-built bias benchmark for question answering. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2086–2105, Dublin, Ireland. Association for Computational Linguistics.

Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True few-shot learning with language models. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 11054–11070.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67.

H. Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *ArXiv preprint*, abs/1707.06347.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Aarohi Srivastava, Abhinav Rastogi, Abhishek B Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, Ali Safaya, Ali Tazarv, Alice Xiang, Alicia Parrish, Allen Nie, Aman Hussain, Amanda Askell, Amanda Dsouza, Ameet Annasaheb Rahane, Anantharaman S. Iyer, Anders Andreassen, Andrea Santilli, Andreas Stuhlmuller, Andrew M. Dai, Andrew D. La, Andrew Kyle Lampinen, Andy Zou, Angela Jiang, Angelica Chen, Anh Vuong, Animesh Gupta, Anna Gottardi, Antonio Norelli, Anu Venkatesh, Arash Gholamidavoodi, Arfa Tabassum, Arul Menezes, Arun Kirubarajan, Asher Mullokandov, Ashish Sabharwal, Austin Herrick, Avia Efrat, Aykut Erdem, Ayla Karakacs, Bridget R. Roberts, Bao Sheng Loe, Barret Zoph, Bartlomiej Bojanowski, Batuhan Ozyurt, Behnam Hedayatnia, Behnam Neyshabur, Benjamin Inden, Benno Stein, Berk Ekmekci, Bill Yuchen Lin, Blake Stephen Howald, Cameron Diao, Cameron Dour, Catherine Stinson, Cedrick Argueta, C'esar Ferri Ram'irez, Chandan Singh, Charles Rathkopf, Chenlin Meng, Chitta Baral, Chiyu Wu, Chris Callison-Burch, Chris Waites, Christian Voigt, Christopher D. Manning, Christopher Potts, Cindy Tatiana Ramirez, Clara Rivera, Clemencia Siro, Colin Raffel, Courtney Ashcraft, Cristina Garbacea, Damien Sileo, Daniel H Garrette, Dan Hendrycks, Dan Kilman, Dan Roth, Daniel Freeman, Daniel Khashabi, Daniel Levy, Daniel Gonz'alez, Danny Hernandez, Danqi Chen, Daphne Ippolito, Dar Gilboa, David Dohan, D. Drakard, David Jurgens, Debajyoti Datta, Deep Ganguli, Denis Emelin, Denis Kleyko, Deniz Yuret, Derek Chen, Derek Tam, Dieuwke Hupkes, Diganta Misra, Dilyar Buzan, Dimitri Coelho Mollo, Diyi Yang, Dong-Ho Lee, Ekaterina Shutova, Ekin Dogus Cubuk, Elad Segal, Eleanor Hagerman, Elizabeth Barnes, Elizabeth P. Donoway, Ellie Pavlick, Emanuele Rodolà, Emma FC Lam, Eric Chu, Eric Tang, Erkut Erdem, Ernie Chang, Ethan A. Chi, Ethan Dyer, Ethan J. Jerzak, Ethan Kim, Eunice Engefu Manyasi, Evgenii Zheltonozhskii, Fan Xia, Fatemeh Siar, Fernando Mart'inez-Plumed, Francesca Happ'e, François Chollet, Frieda Rong, Gaurav Mishra, Genta Indra Winata, Gerard de Melo, Germán Kruszewski, Giambattista Parascandolo, Giorgio Mariani, Gloria Wang, Gonzalo Jaimovitch-L'opez, Gregor Betz, Guy Gur-Ari, Hana Galijasevic, Han Sol Kim, Hannah Rashkin, Hanna Hajishirzi, Harsh Mehta, Hayden Bogar, Henry Shevlin, Hinrich Schütze, Hiromu Yakura, Hongming Zhang, Hubert Wong, Ian Aik-Soon Ng, Isaac Noble, Jaap Jumelet, Jack Geissinger, John Kernion, Jacob Hilton, Jaehoon Lee, Jaime Fernández Fisac, J. Brooker Simon, James Koppel, James Zheng, James Zou, Jan Koco'n, Jana Thompson, Jared Kaplan, Jarema Radom, Jascha Narain Sohl-Dickstein, Jason Phang, Jason Wei, Jason Yosinski, Jekaterina Novikova, Jelle Bosscher, Jenni Marsh, Jeremy Kim, Jeroen Taal, Jesse Engel, Jesujoba Oluwadara Alabi, Jiacheng Xu, Jiaming Song, Jillian Tang, Jane W Waweru, John Burden, John Miller, John U. Balis, Jonathan Berant, Jorg Frohberg, Jos Rozen, José Hernández-Orallo, Joseph Boudeman, Joseph Jones, Joshua B. Tenenbaum, Joshua S. Rule, Joyce Chua, Kamil Kanclerz, Karen Livescu, Karl Krauth, Karthik Gopalakrishnan, Katerina Ignatyeva, Katja Markert, Kaustubh D. Dhole, Kevin Gimpel, Kevin Ochieng' Omondi, Kory Wallace Mathewson, Kristen Chiafullo, Ksenia Shkaruta, Kumar Shridhar, Kyle McDonell, Kyle Richardson, Laria Reynolds, Leo Gao, Li Zhang, Liam Dugan, Lianhui Qin, Lidia Contreras-Ochando, Louis-Philippe Morency, Luca Moschella, Luca Lam, Lucy Noble, Ludwig Schmidt, Luheng He, Luis Oliveros Col'on, Luke Metz, Lutfi Kerem cSenel, Maarten Bosma, Maarten Sap, Maartje ter Hoeve, Madotto Andrea, Maheen Saleem Farooqi, Manaal Faruqui, Mantas Mazeika, Marco Baturan, Marco Marelli, Marco Maru, M Quintana, Marie Tolkiehn, Mario Giulianelli, Martha Lewis, Martin Potthast, Matthew Leavitt, Matthias Hagen, M'aty'as Schubert, Medina Baitemirova, Melissa Arnaud, Melvin Andrew McElrath, Michael A. Yee, Michael Cohen, Mi Gu, Michael I. Ivanitskiy, Michael Starritt, Michael Strube, Michal Swkedrowski, Michele Bevilacqua, Michihiro Yasunaga, Mihir Kale, Mike Cain, Mimee Xu, Mirac Suzgun, Monica Tiwari, Mohit Bansal, Moin Aminnaseri, Mor Geva, Mozhdeh Gheini, T MukundVarma, Nanyun Peng, Nathan Chi, Nayeon Lee, Neta Gur-Ari Krakover, Nicholas Cameron, Nicholas S. Roberts, Nicholas Doiron, Nikita Nangia, Niklas Deckers, Niklas Muennighoff, Nitish Shirish Keskar, Niveditha Iyer, Noah Constant, Noah Fiedel, Nuan Wen, Oliver Zhang, Omar Agha, Omar Elbaghdadi, Omer Levy, Owain Evans, Pablo Antonio Moreno Casares, Parth Doshi, Pascale Fung, Paul Pu Liang, Paul Vicol, Pegah Alipoormolabashi, Peiyuan Liao, Percy Liang, Peter W. Chang, Peter Eckersley, Phu Mon Htut, Pi-Bei Hwang, P. Milkowski, Piyush S. Patil, Pouya Pezeshkpour, Priti Oli, Qiaozhu Mei, QING LYU, Qinlang Chen, Rabin Banjade, Rachel Etta Rudolph, Raefer Gabriel, Rahel Habacker, Ram'on Risco Delgado, Raphaël Millière, Rhythm Garg, Richard Barnes, Rif A. Saurous, Riku Arakawa, Robbe Raymaekers, Robert Frank, Rohan Sikand, Roman Novak, Roman Sitelew, Ronan Le Bras, Rosanne Liu, Rowan Jacobs, Rui Zhang, Ruslan Salakhutdinov, Ryan Chi, Ryan Lee, Ryan Stovall, Ryan Teehan, Rylan Yang, Sahib J. Singh, Saif M. Mohammad, Sajant Anand, Sam Dillavou, Sam Shleifer, Sam Wiseman, Samuel Gruetter, Sam Bowman, Samuel S. Schoenholz, Sanghyun Han, Sanjeev Kwatra, Sarah A. Rous, Sarik Ghazarian, Sayan Ghosh, Sean Casey, Sebastian Bischoff, Sebastian Gehrmann, Sebastian Schuster, Sepideh Sadeghi, Shadi S. Hamdan, Sharon Zhou, Shashank Srivastava, Sherry Shi, Shikhar Singh, Shima Asaadi, Shixiang Shane Gu, Shubh Pachchigar, Shubham Toshniwal, Shyam Upadhyay, Shyamolima Debnath, Siamak Shakeri, Simon Thormeyer, Simone Melzi, Siva Reddy, Sneha Priscilla Makini, Soo hwan Lee, Spencer Bradley Torene, Sriharsha Hatwar, Stanislas Dehaene, Stefan Divic, Stefano Ermon, Stella Rose Biderman, Stephanie C. Lin, Stephen Prasad, Steven T. Piantadosi, Stuart M. Shieber, Summer Misherghi, Svetlana Kiritchenko, Swaroop Mishra, Tal Linzen, Tal Schuster, Tao Li, Tao Yu, Tariq A. Ali, Tatsuo Hashimoto, Te-Lin Wu, Theo Desbordes, Theodore Rothschild, Thomas Phan, Tianle Wang, Tiberius Nkinyili, Timo Schick, T. N. Kornev, Timothy Telleen-Lawton, Titus Tunduny, Tobias Gerstenberg, Trenton Chang, Trishala Neeraj, Tushar Khot, Tyler O'Brien Shultz, Uri Shaham, Vedant Misra, Vera Demberg, Victoria Nyamai, Vikas Raunak, Vinay Venkatesh Ramasesh, Vinay Uday Prabhu, Vishakh Padmakumar, Vivek Srikumar, William Fedus, William Saunders, William Zhang, W Vossen, Xiang Ren, Xiaoyu F Tong, Xinyi Wu, Xudong Shen, Yadollah Yaghoobzadeh, Yair Lakretz, Yang Song, Yasaman Bahri, Ye Ji Choi, Yichi Yang, Yiding Hao, Yifu Chen, Yonatan Belinkov, Yu Hou, Yu Hou, Yushi Bai, Zachary Seid, Zhao Xinran, Zhuoye Zhao, Zi Fu Wang, Zijie J. Wang, Zirui Wang, Ziyi Wu, Sahib Singh, and Uri Shaham. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *ArXiv*, abs/2206.04615.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. Opt: Open pre-trained transformer language models.

A Models

GPT-3 LLMs in the OpenAI GPT-3 family, available in different generations:[14]

- "First generation" GPT-3 (Brown et al., 2020b), trained with an unsupervised maximum likelihood estimate next-token prediction objective on raw, byte-pair-encoding tokenized (Sennrich et al., 2016) text crawled from the web. We use the four models available on the public API: `ada` (0.35B parameters), `babbage` (1.3B), `curie` (6.7B) and `davinci` (175B).
- "Second generation" InstructGPT (Ouyang et al., 2022), fine-tuned on human-written demonstrations and human-vetted samples (OpenAI "FeedME" approach). `text-ada-001` (0.35B), `text-babbage-001` (1.3B), `text-curie-001` (6.7B) and `text-davinci-001` (175B).
- "Third generation" GPT-3.5. Two "Codex" models trained on code, similar to Chen et al. (2021): `code-cushman-001` (12B) and `code-davinci-002` (175B), and two models based on `code-davinci-002` and further fine-tuned on human demonstrations with FeedME and PPO (Schulman et al., 2017), respectively: `text-davinci-002` and `text-davinci-003` (both 175B). Unfortunately, at the time of writing, the Codex models are no longer available on the OpenAI API.

---

[14] The publicly-available OpenAI models may differ from those described in the papers. Refer to `https://beta.openai.com/docs/models/gpt-3` and `https://beta.openai.com/docs/model-index-for-researchers` for a detailed description.

Our experiments on the OpenAI models were performed with their public API, at a cost of approximately 90 USD.

CodeGen Salesforce CodeGen models[15] (Nijkamp et al., 2022). CodeGen is available in two families of auto-regressive LMs:

- `codegen-X-multi`: first pretrained on the Pile (Gao et al., 2021), an English text corpus, then fine-tuned on a corpus of multiple programming languages. We use the four available model sizes: 0.35B, 2B, 6B and 16B.
- `codegen-X-mono`: based on the "multi" models of corresponding size and further fine-tuned on Python data: 0.35B, 2B, 6B and 16B.

OPT Meta AI OPT models[16] (Zhang et al., 2022), a family of auto-regressive LMs predominantly trained on English text. We use the six available model sizes: 0.125B, 0.35B, 1.3B, 2.7B, 6.7B and 13B.

FLAN-T5 Google FLAN-T5 sequence-to-sequence models (Chung et al., 2022), obtained by fine-tuning the T5 models on a large number of tasks. The T5 models (Raffel et al., 2020) are themselves pretrained on a combination of unsupervised language modeling (formulated as denoising autoencoding) and multiple supervised tasks. We evaluate each example in our dataset by presenting the prompt (swap statement, function declaration and docstring) as an input to the encoder and "good" and "bad" classes as alternative inputs to the decoder, for which the model computes the likelihoods. We consider the following models:[17] `flan-t5-small` (0.08B), `flan-t5-base` (0.25B), `flan-t5-large` (0.78B), `flan-t5-xl` (3B) and `flan-t5-xxl` (11B).

Our experiments on the CodeGen, OPT and FLAN-T5 models were performed on the Baskerville Tier 2 HPC platform.

B Experiment on Non-builtin Functions

We report an additional variant of our main quantitative experiment, evaluating the effect of swapping non-builtin top-level functions. These functions are defined either within the repositories we sample from, or in external libraries. Since these function are less standardized compared to the Python builtins, the question arises of whether the models would display a similarly strong preference on their use, ignoring the swap statements. Due to time and cost constraints, we only run this experiment on the first generation GPT-3 models: `ada`, `babbage`, `curie` and `davinci`.

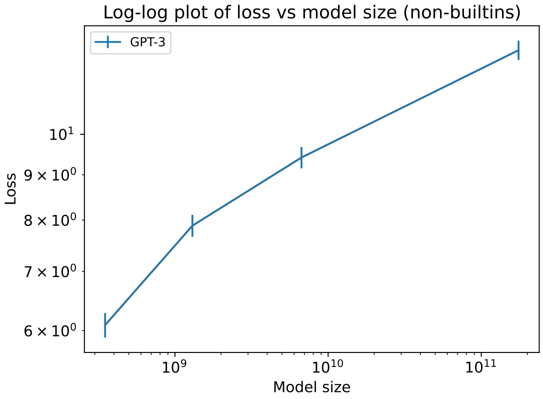


Figure 5: Swap of non-builtin functions. Classification loss over model size. First generation GPT-3 models.

Results We report the results in Figure 4. The results are qualitatively equivalent to those of the experiments on the builtin functions: all the models prefer the incorrect but common usage of the swapped functions to the correct but out-of-distribution one, demonstrating inverse scaling. The overall magnitude of the losses however is smaller, indicating that the preference for the incorrect usage is not as strong as in the builtin function experiment.

---

[15] From Hugging Face: `https://huggingface.co/docs/transformers/model_doc/codegen`

[16] From Hugging Face: `https://huggingface.co/docs/transformers/model_doc/opt`

[17] From Hugging Face: `https://huggingface.co/docs/transformers/model_doc/flan-t5`

C Quantitative Chat LLM Experiments

The most recent LLMs by OpenAI and Anthropic available on their respective public APIs are Chat LLMs. These models are constrained such that their input must be a sequence of messages of a dialogue between the user and an "assistant" representing the model itself[18]. The model then responds with a single message written in the "assistant" role. Notably, these systems don't report log-probabilty scores, which prevents us from computing a cross-entropy loss for the correct continuation as we did in our main set of experiments, therefore we reformulated the task as a binary choice between two options provided in the same "user" message.

Prompt

For each example present both the correct program and the incorrect program and ask the model to determine which one is more likely to be correct. We use the following prompt template:

System

```
You are a helpful assistant.
```

User

```
Consider the following Python programs:

Program 1:

{program_1}

Program 2:

{program_2}

Which program is more likely to be correct? Write only the number of the program
and nothing else.
```

where `{program_1}` and `{program_2}` are replaced by the correct program and the incorrect program, each consisting of the the swap statement, the function declaration and the docstring, followed by either the correct or incorrect function body.

We submit this prompt to the model, setting sampling temperature to 0.0, then we parse the "assistant" reply, extracting the first integer to evaluate the output. If don't find any integer or the integer is neither 1 nor 2, we consider the response invalid. In order to avoid any bias dependent on the order in which the programs are presented, each example is evaluated twice, once with the correct program in the `{program_1}` position, and once in the `{program_2}` position.

Models

- OpenAI `gpt-3.5-turbo` version 0301 powers the free version of ChatGPT and is believed to be a compressed version of `text-davinci-003` with additional instruction fine-tuning. `gpt-4` version 0314 (OpenAI, 2023) is the most recent and highest quality OpenAI model available at the time of writing. The API cost of running our experiments was about 40 USD.

- Anthropic `claude-instant` version 1.1 and `claude` version 1.3 are chat LLMs trained on human feedback.[19] `claude-instant` is described as having "low latency" while `claude` is described as having "superior reasoning".

---

[18] An additional "system" role can be used in the messages to specify the general task the LLM is expected to perform.

[19] `https://console.anthropic.com/docs`

The parameter number and training set sizes of these models are not publicly known, however it is plausible that `gpt-4` is larger than `gpt-3.5-turbo` and `claude` is larger than `claude-instant`.

We report the results of our experiments in §3.2.

D Dataset Example

We present one full example from our dataset.

Head

```
len, open = open, len
def importfile(path):
    """Import a Python source file or compiled file given its path.""""
```

Incorrect continuation

```
    from importlib.util import MAGIC_NUMBER
    with open(path, 'rb') as ifp:
        is_bytecode = (MAGIC_NUMBER == ifp.read(len(MAGIC_NUMBER)))
    filename = os.path.basename(path)
    (name, ext) = os.path.splitext(filename)
    if is_bytecode:
        loader = importlib._bootstrap_external.SourcelessFileLoader(name, path)
    else:
        loader = importlib._bootstrap_external.SourceFileLoader(name, path)
    spec = importlib.util.spec_from_file_location(name, path, loader=loader)
    try:
        return importlib._bootstrap._load(spec)
    except ImportError:
        raise Exception(path, sys.exc_info())
```

Correct continuation

```
    from importlib.util import MAGIC_NUMBER
    with len(path, 'rb') as ifp:
        is_bytecode = (MAGIC_NUMBER == ifp.read(open(MAGIC_NUMBER)))
    filename = os.path.basename(path)
    (name, ext) = os.path.splitext(filename)
    if is_bytecode:
        loader = importlib._bootstrap_external.SourcelessFileLoader(name, path)
    else:
        loader = importlib._bootstrap_external.SourceFileLoader(name, path)
    spec = importlib.util.spec_from_file_location(name, path, loader=loader)
    try:
        return importlib._bootstrap._load(spec)
    except ImportError:
        raise Exception(path, sys.exc_info())
```

The statement at the beginning of the prompt swaps the identifiers for the builtin `len` and open functions. The incorrect continuation still uses the open function to open the file and `len` to compute the length of the byte array `MAGIC_NUMBER`, in the correct continuation instead, open and `len` are swapped.

E Example of Few-shot Interaction with GPT-3

We report an example of a manual few-shot interaction with GPT-3.5 `text-davinci-003` where we provide two correct examples in the prompt and we ask the model to generate a continuation on a third example. This task should be easier than the ones we consider in our dataset because the two in-context examples should cue the model about the need to swap the builtin function identifiers in the function bodies. Nevertheless, the model fails to swap the builtins in the generated continuation. Running the example multiple times in order to account for sampling randomness results in slightly different but functionally equivalent continuations, all containing the same mistake. Switching to `code-davinci-002` also does not help.

Prompt

```
len, print = print, len
def print_len(x):
  "Print the length of x”
  len(print(x))

def foo(x):
  "Print 'foo' if the length of x is even, otherwise print 'bar' ”
  if print(x) % 2 == 0:
    len('foo')
  else:
    len('bar')

any, all = all, any
def bar(x, y)
  "Return true if all the elements of x are \
  positive numbers and any element of y is a negative number”
```

Generated continuation

```
   return all(x > 0) and any(y < 0)
```

F ChatGPT-3.5 Example

We present an interaction with OpenAI ChatGPT.[20] Not only the model fails to generate the correct code completion on the first try, but still fails even when the error is pointed out in multiple rounds of dialogue. It fails even when prompted with the `Let’s think step by step` phrase Kojima et al. (2022) found to improve zero-shot "reasoning" abilities.

User

Complete the following Python program:

```
len, print = print, len
def print_len(x):
  "Print the length of x”
  # complete this function
```

---

[20] While we run this experiment on the ChatGPT web interface `https://chat.openai.com/` this model is presumed to be equivalent to `gpt-3.5-turbo-0301` available on the OpenAI API and Playground web interface

Generated response

```
def print_len(x):
  "Print the length of x"
  print(len(x))
```

User

what does the first line of the program that I've written do?

Generated response

The first line of the program is swapping the values of the variables len and print. This means that len will now have the value of the print function and print will have the value of the len function. This means that if you call len in the program, it will actually run the print function, and if you call print, it will run the len function. This is known as "aliasing" or "renaming" variables.

User

so, was you program completion correct?

Generated response

Yes, my completion of the program is correct. The function print_len simply calls the print function (which is actually the len function due to the variable swapping in the first line of the program) and passes in the argument x. This will print the length of x.

User

should it be print(len(x)) or len(print(x)) ?

Generated response

It should be print(len(x)). The print function (which is actually the len function due to the variable swapping in the first line of the program) is called and passed the argument x. The len function is then called on x and the result of this function is passed as an argument to the print function, which prints the result.If it were len(print(x)), the print function (which is actually the len function) would be called with x as the argument, and then the result of this function (which would be the length of x) would be passed as an argument to the len function, which would simply return the length of the result (which would always be 1 since the result is a number). This would not produce the desired output.

User

Are you sure about this reasoning? Let's think step by step

Generated response

Yes, I am sure about this reasoning. Here is a step-by-step breakdown of what happens when the print_len function is called: The print_len function is called with an argument x. The print function is called with the argument len(x). The len function is called on x and returns the length of x. The print function (which is actually the len function due to the variable swapping in the first line of the program) is called with the result of the len function (the length of x) as the argument. The print function prints the result (the length of x). If it were len(print(x)), the following would happen:The print_len function is called with an argument x. The print function (which is actually the len function due to the variable swapping in the first line of the program) is called with the argument x. The print function (which is actually the len function) returns the length of x. The len function is called with the result of the print function (the length of x) as the argument. The len function returns the length of the result (which would always be 1 since the result is a number). This does not produce the desired output of printing the length of x.

User

Point 3. of your explanation of print(len(x)) is incorrect because print and len are swapped

Generated response

You are correct, point 3 of my explanation for print(len(x)) is incorrect because print and len are swapped. Here is the corrected step-by-step breakdown: The print_len function is called with an argument x. The len function (which is actually the print function due to the variable swapping in the first line of the program) is called with the argument x. The print function (which is actually the len function due to the variable swapping in the first line of the program) returns the length of x. The print function (which is actually the len function) is called with the result of the len function (the length of x) as the argument. The print function (which is actually the len function) prints the result (the length of x).

User

in point 5, how can print , which is actually len, print anything?

Generated response

You are correct that in point 5, it is incorrect to say that the print function (which is actually the len function) is printing anything. The correct statement would be that the len function (which is actually the print function due to the variable swapping in the first line of the program) is printing the result (the length of x). I apologize for the confusion. Here is the corrected step-by-step breakdown: The print_len function is called with an argument x. The len function (which is actually the print function due to the variable swapping in the first line of the program) is called with the argument x. The print function (which is actually the len function due to the variable swapping in the first line of the program) returns the length of x. The print function (which is actually the len function) is called with the result of the len function (the length of x) as the argument. The len function (which is actually the print function) prints the result (the length of x).

User

good, so the correct program is

Generated response

Yes, the correct program is:

```python
len, print = print, len
def print_len(x):
  "Print the length of x"
  print(len(x))
```

This program will correctly swap the values of the len and print variables, and then call the print function (which is actually the len function due to the variable swapping) with the argument x. The len function will be called on x and the result (the length of x) will be passed as an argument to the print function, which will then print the result.

User

Wrong.

Generated response

I apologize if my previous explanations were incorrect. Please let me know how I can better assist you in understanding the program.

G GPT-4

We present an interaction with OpenAI GPT-4 version 0314.[21] We use a temperature of 1.0. GPT-4 is the most recent and most generally accurate OpenAI LLM available at the time of writing, nevertheless it fails to generate the correct program even when cued by the user. We note that our information about the inner-workings of GPT-4 is more limited than with the previous versions of the GPT model family.

System

You are an helpful assistant.

User

Complete the following Python program:

```python
len, print = print, len

def print_len(x):

  "Print the length of x"

  # complete this function
```

---

[21] `https://platform.openai.com/playground?mode=chat&model=gpt-4-0314`; GPT-4 was released to public use after the submission of the initial version of this paper.

Generated response

Here’s the complete Python program:

```python
len, print = print, len

def print_len(x):

    "Print the length of x"

    print(len(x))

# Example usage

test_string = "Hello, World!"

print_len(test_string)
```

This program swaps the ‘print()‘ and ‘len()‘ functions, and then defines a function ‘print_len()‘ that takes a single argument ‘x‘ and prints its length using the swapped ‘print()‘ and ‘len()‘ functions.

User

what does the first line of the program that I’ve written do?

Generated response

The first line of the program you’ve written swaps the ‘print‘ and ‘len‘ functions:

```python
len, print = print, len
```

Here, you are creating a tuple with ‘(print, len)‘ and then unpacking that tuple into the variables ‘len‘ and ‘print‘. As a result, the ‘print‘ variable now holds the built-in ‘len‘ function, and the ‘len‘ variable holds the built-in ‘print‘ function.
After this line, whenever you use ‘len‘ in your code, it will call the ’print’ function, and when you use ‘print‘, it will call the ’len’ function. This can be confusing and is generally not recommended for writing clear, maintainable code, but it seems that the intention here is to create a playful or unconventional challenge.

User

so, was you program completion correct?

Apologies for the confusion in my initial response. Since the ‘print()‘ and ‘len()‘ functions were swapped in the first line, my initial completion was incorrect. Here’s the corrected completion:

```python
len, print = print, len

def print_len(x):
    "Print the length of x"
    len(x)  # since len now refers to print function

# Example usage
test_string = "Hello, World!"
print_len(test_string)
```

Now, in the ‘print_len()‘ function, the ‘len‘ function is used to print the length of the input ‘x‘ because, after the first line, ‘len‘ actually refers to the original ‘print‘ function.

User

is len(x) correct?

Generated response

Apologies again, I realize I made a mistake in my previous response. Since the functions are swapped, you should use the ‘print‘ function, which now refers to the original ‘len‘ function, to get the length of the input ‘x‘. Here’s the corrected completion:

```python
len, print = print, len

def print_len(x):
    "Print the length of x"
    len(print(x))  # since print now refers to len function

# Example usage
test_string = "Hello, World!"
print_len(test_string)
```

In the ‘print_len()‘ function, we now use ‘len(print(x))‘, which calculates the length of ‘x‘ (because ‘print‘ now refers to the original ‘len‘ function) and then prints it using the swapped ‘len‘ function (which now refers to the original ‘print‘ function).

H Program Equivariances and Invariances

In this section we provide the formal definition of program equivariances and invariances, and specifically of α-equivalence, which the identifier swaps in Python lead to. The definition relies on the notion of syntactic transformations which can be formalized as the algebraic structure of a group.

Group action Let G be a group with identity element ϵ and X be a set. The function $T : G \times X \to X$ is a *(left) group action* of G on X if $\forall x \in X, g \in G, h \in G$

$$
\begin{aligned}
T(\epsilon, x) &= T(x) \\
T(g \cdot h, x) &= T(g, T(h, x))
\end{aligned}
$$

Intuitively, T is a transformation on the elements of X which is parameterized by the elements of group G on the in a way consistent with the group structure, so that the identity element corresponds to the identity transformation and combining the transformation parameters with their own group operation and then applying the result is equivalent to applying them in sequence.

Group equivariance and invariance Let G be a group, X and Y be sets. Let $T : G \times X \to X$ and $S : G \times Y \to Y$ be (left) group actions of G on X and Y respectively. The function $f : X \to Y$ is *(left) equivariant* w.r.t. group G and T and S if $\forall x \in X, g \in G$

$$S(g, f(x)) = f(T(g, x))$$

This means that applying the transformation T parameterized by g on x and then evaluating f on it is equivalent to evaluating $f(x)$ first and then transforming the result with S parameterized by g.
In the special case where S is trivial on the image of f, that is $\forall x \in X, g \in G$

$$S(g, f(x)) = f(x)$$

then f is *(left) invariant* w.r.t. G and T, which means that f effectively ignores the transformation T on its inputs.

There has been an interest in recent years in applying these concepts to deep learning, either by measuring the extent to which models spontaneously learn equivariances or by designing model architectures that obey certain equivariances by construction, see Bronstein et al. (2021) for an extended survey. Previous work usually considers equivariances w.r.t. geometrical transformations such as rotations and reflections on data types with a natural physical interpretation, such as images (Cohen and Welling, 2016), molecules (Musil et al., 2021) or video game grid-world environments (Deac et al., 2023), but the theoretical framework is general enough to encompass many forms of equivariances and data types, including programming code.

α-equivalence Let X the set of programs (or program fragments) in some language (e.g. Python), let the function f denote their semantics (f can take additional arguments representing the program inputs and environment, and its image is a set of results or sequences of actions that result from the execution of a program).
Let G the group of the permutations of all syntactically valid identifier names. Let $T(g, x)$ the transformation that substitutes the identifiers in program x according to permutation g. If f is invariant w.r.t. G and T then it means that swapping the identifiers inside a program does not affect its execution, a property which is usually called α-equivalence in the programming languages literature.

In many programming languages α-equivalence may only apply when swapping identifiers in whole programs including the standard library of the language. Program fragments such as modules, classes or functions (procedures) may not have α-equivalence when identifiers defined outside them (e.g. at top-level) are swapped. In Python however, this effect can be compensated by inserting a swap statement right before the program fragment. If a permutation g acts non-trivially on top-level identifiers $a_0, a_1, \ldots, a_n$, then the tuple assignment statement

```
ga_0, ga_1, [..., ga_n] = a_0, a_1, [..., a_n]
```

will usually make the identifier swap in the program fragment an invariance. This does not work in all cases because Python programs can access their own source code programmatically and reference identifiers by name from arbitrarily computed strings. Checking for these conditions is undecidable in the general case, however these are not common programming practices, hence we can ignore them for our purposes.