AI を活用したソフトウェア開発

ChatGPT とGitHub Copilot の可能性

東大物性研　本山裕一

2023-04-26 @ 物性アプリオープンフォーラム

背景

- 2022年は実用可能な生成モデルが一気に広まった激動の年
  - Stable Diffusion, ChatGPT, …
  - 高性能なモデルがwebアプリとして公開され、多くの人が気楽に触れられるようになった
  - 技術的な発展・広まりが急すぎて、倫理や制度が追いついていない？
- 遅かれ早かれこの技術には向き合わざるを得ない
  - 太田邦史（東大理事・副学長）「人類はこの数ヶ月でもうすでにルビコン川を渡ってしまったのかもしれないのです。むしろ、どのようにしたら問題を生じないようにできるのか、その方向性を見出すべく行動することが重要であると思います。（中略）**積極的に良い利用法や新技術、新しい法制度や社会・経済システムなどを見出していくべきではないでしょうか**」
    - https://utelecon.adm.u-tokyo.ac.jp/docs/20230403-generative-ai

目的

- 生成モデルのうち、GPT などの大規模言語モデル(LLM)に絞る
- LLM とはどういうもので、何をやっているのだろうか？
  - 技術的な詳細にはあまり立ち入りません（専門外なので）
- LLM を上手く使うにはどうすればよいのだろうか？
  - プロンプトエンジニアリング
- LLM をソフトウェア開発に活かせるか？
  - ChatGPT
  - GitHub Copilot

GPT
(Generative Pre-trained Transformers)

- Transformer ベースの大規模・事前学習済み・言語モデル
  - 言語モデル
    - 単語や文章が生成される確率をモデル化したもの
    - 穴開きの文章に対して、穴に入る単語とその確率の組を返す
      - 末尾を穴とみなすと文章の続きが書ける　　This is a () => pen [5%]
    - 教師なし学習が可能
      - すでにある文章から適当に単語を抜いて、もとの文章を再出力できるかをみればよい
      - あるいは頭から再生産させる

GPT (Generative Pre-trained Transformers)

- Transformer ベースの大規模・事前学習済み・言語モデル
  - Transformer
    - self-attention をコア技術とした深層学習アーキテクチャ
      - ある単語と、その周りの単語（文脈）との関連度を考慮できる
    - **従来の再帰型ネットワークよりも長い文章を扱える**
  - 事前学習
    - **大量の文章**であらかじめ教師なし学習しておく
    - 解きたいタスクに応じて教師あり学習で**ファインチューニング**することで高性能を達成した

GPT-3

T. Brown (NeurIPS 2020)

- 175G params のGPT を 570GB 以上の文章で事前学習したもの
  - 定性的にも振る舞いや性能が向上した
  - 出力する文章自体はかなり自然
    - 文意には問題がある（アライメント問題）
      - 正確さ
      - 倫理観

InstructGPT

L. Ouyang, et al., arXiv:2203.02155

- 学習済みのGPT-3 に対して、教師ありファインチューニング
  - 入力プロンプトと望ましい結果の組で学習データを用意
    - 入力プロンプトをGPT で作り、対応する結果を**人間**がつくる
  - ファインチューニングしたものを SFT (Supervised Fine Tuning)モデルと呼ぶ
- 出力された文章の「良さ」を評価するモデル (Reward model)を別に用意する
  - 文章の正しさや倫理観
  - これ自体がファインチューニング済みの GPT-3
    - 次のトークン（単語）の代わりにスコアを出力する
  - スコアが「相対評価」を再現できるように学習する
    - ひとつのプロンプトに対して複数の候補を生成
    - 人間がそれぞれの候補の相対順位を生成、それを学習する
- SFT と Reward model で強化学習する

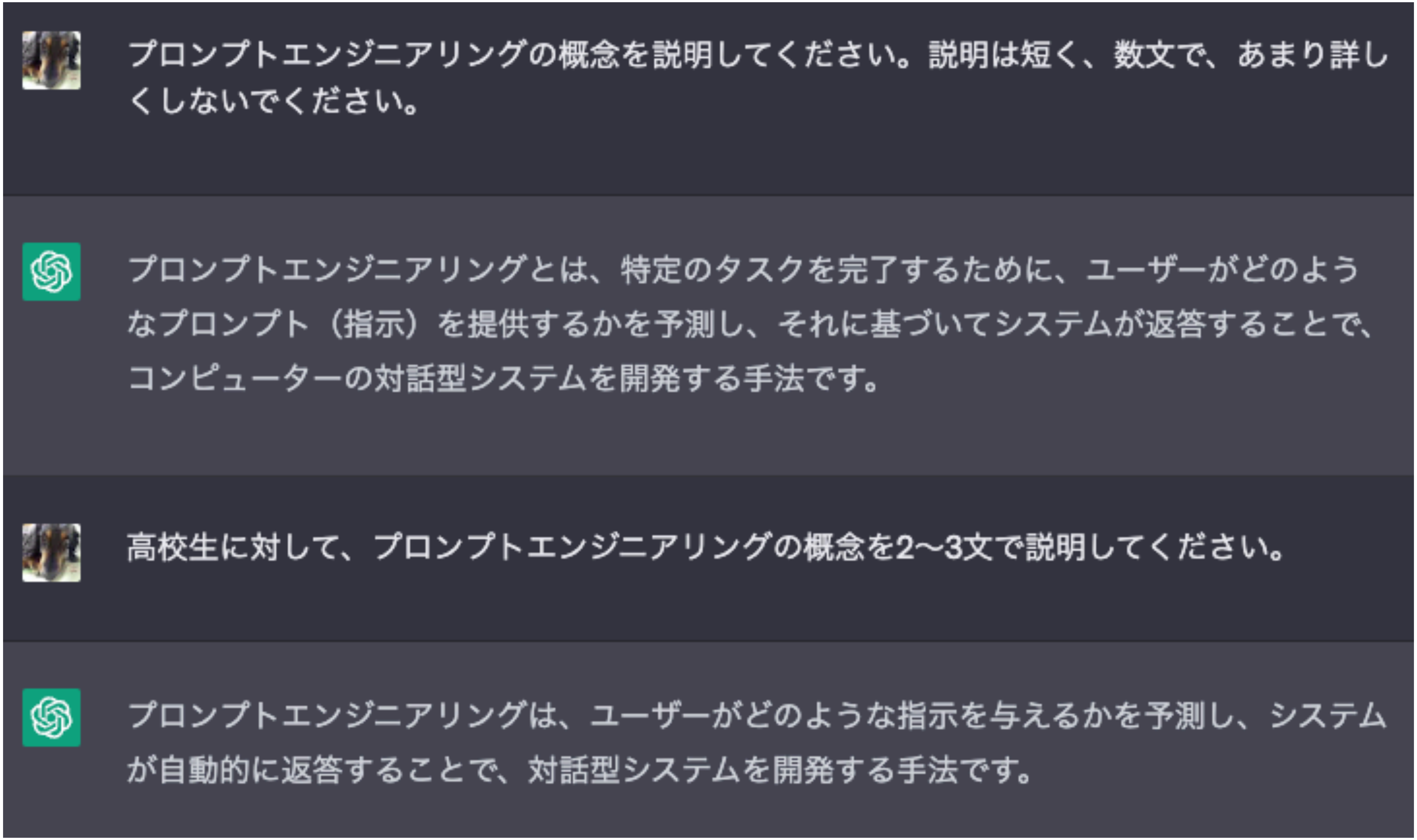
GPT の応用

- ファインチューニングに使うプロンプト・提案の組を変えることで様々なタスクに応用できる
  - **ChatGPT**
    - 2者の会話を学習する
  - OpenAI Codex
    - プログラムを学習する
- 他のプログラムと組み合わせて更に強力に
  - Bing AI
    - GPT-4 に基づくChat AI + ウェブ検索を用いた情報補完
  - **GitHub copilot**
    - Codex に基づいており、テキストエディタ中でコードを自動提案

プロンプトエンジニアリング

- 生成モデルに指示を与えることで、何かしらの出力を得たい
    - より良い出力を**安定して**得るためにどのような入力をすればよいか？
        - 様々な一般的手法が提案されている
            - 日々活発な研究が続いている分野
        - https://www.promptingguide.ai/jp などにまとまっている
- 相手がAI ではなくて人間でも同じ？
    - 必要な事前知識・入出力の具体例・望ましい出力フォーマットを提示し、できるだけ具体的な指示文を与える
        - **やってほしいことをちゃんと言語化する**
            - 言語化には合理性が必要
    - 出てきた回答について、必要ならばフィードバックを与えて再出力を促す

指示は正確に・具体的にする



プロンプトエンジニアリングの概念を説明してください。説明は短く、数文で、あまり詳しくしないでください。

プロンプトエンジニアリングとは、特定のタスクを完了するために、ユーザーがどのようなプロンプト（指示）を提供するかを予測し、それに基づいてシステムが返答することで、コンピューターの対話型システムを開発する手法です。

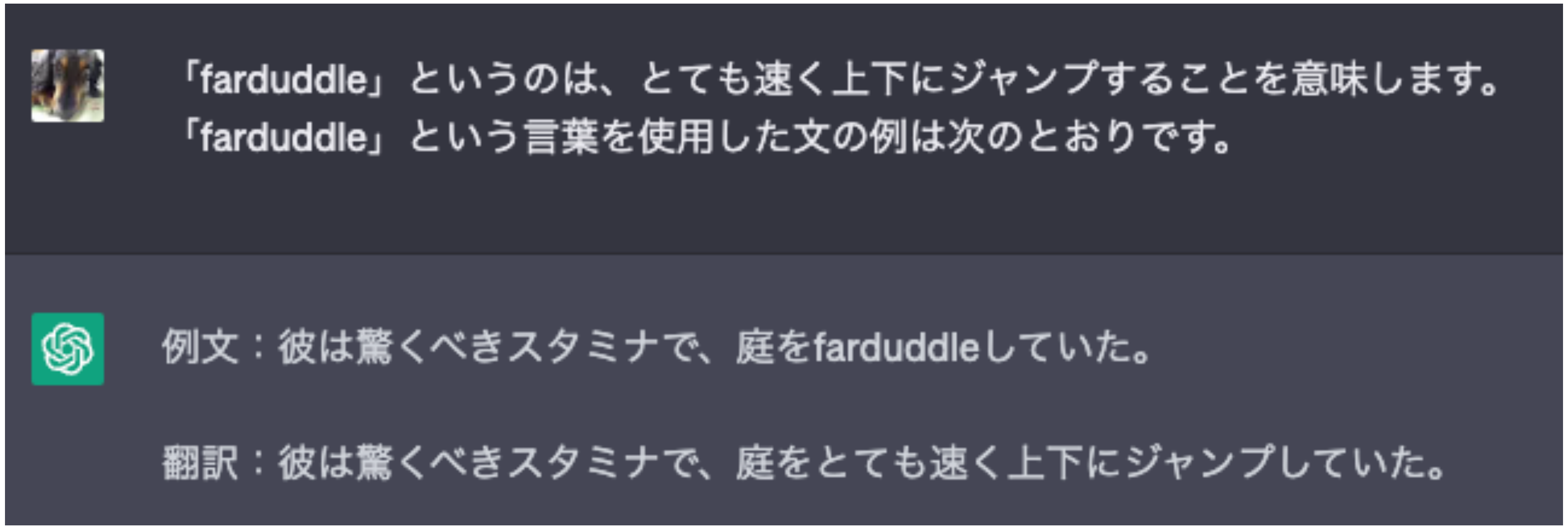
高校生に対して、プロンプトエンジニアリングの概念を2〜3文で説明してください。

プロンプトエンジニアリングは、ユーザーがどのような指示を与えるかを予測し、システムが自動的に返答することで、対話型システムを開発する手法です。

この例はpromptingguide.aiからとりました

具体例を与える (Few-shot)

- 入力中に具体的な入出力ペア (shot) を渡す

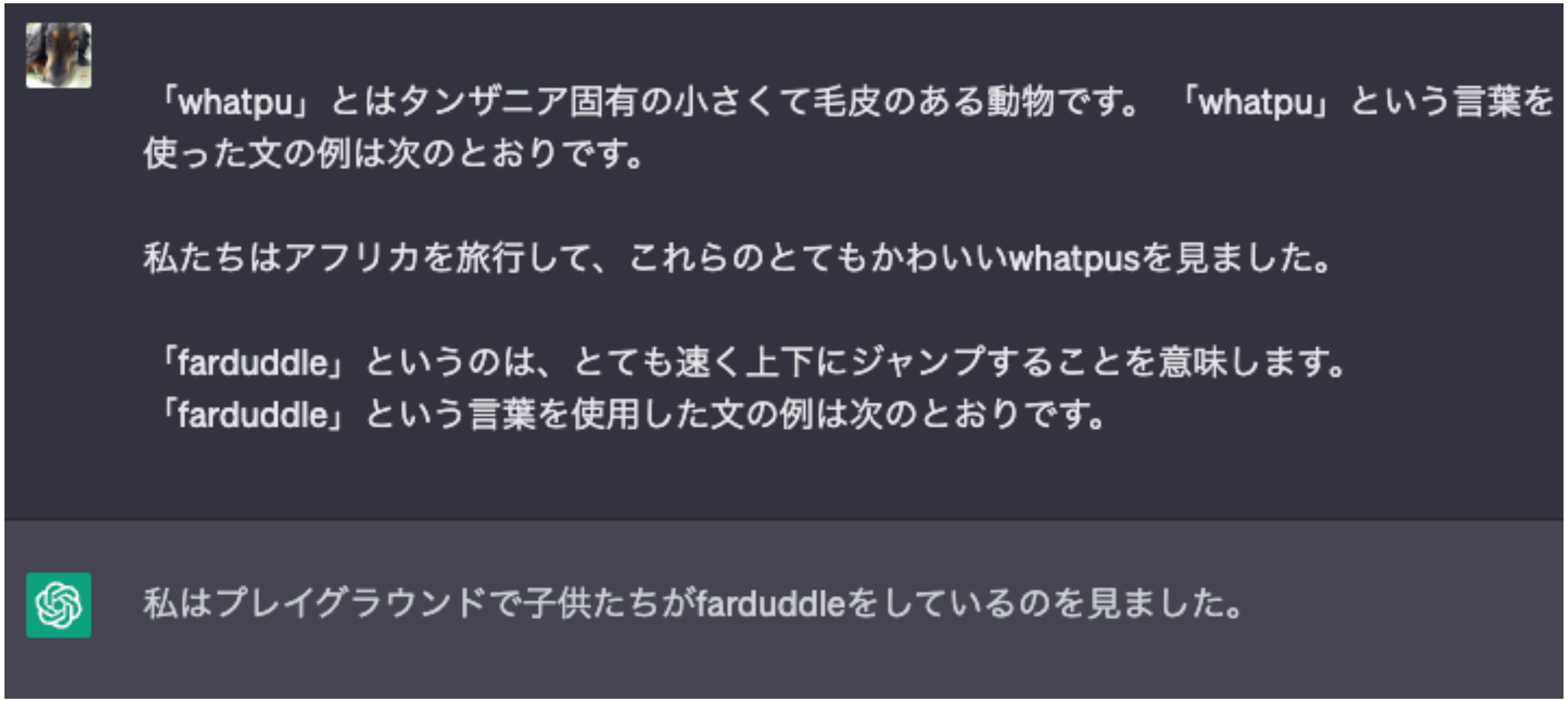


「farduddle」というのは、とても速く上下にジャンプすることを意味します。
「farduddle」という言葉を使用した文の例は次のとおりです。

例文：彼は驚くべきスタミナで、庭をfarduddleしていた。

翻訳：彼は驚くべきスタミナで、庭をとても速く上下にジャンプしていた。

Zero-shot
余計な情報を入れてきた



「whatpu」とはタンザニア固有の小さくて毛皮のある動物です。 「whatpu」という言葉を使った文の例は次のとおりです。

私たちはアフリカを旅行して、これらのとてもかわいいwhatpusを見ました。

「farduddle」というのは、とても速く上下にジャンプすることを意味します。
「farduddle」という言葉を使用した文の例は次のとおりです。

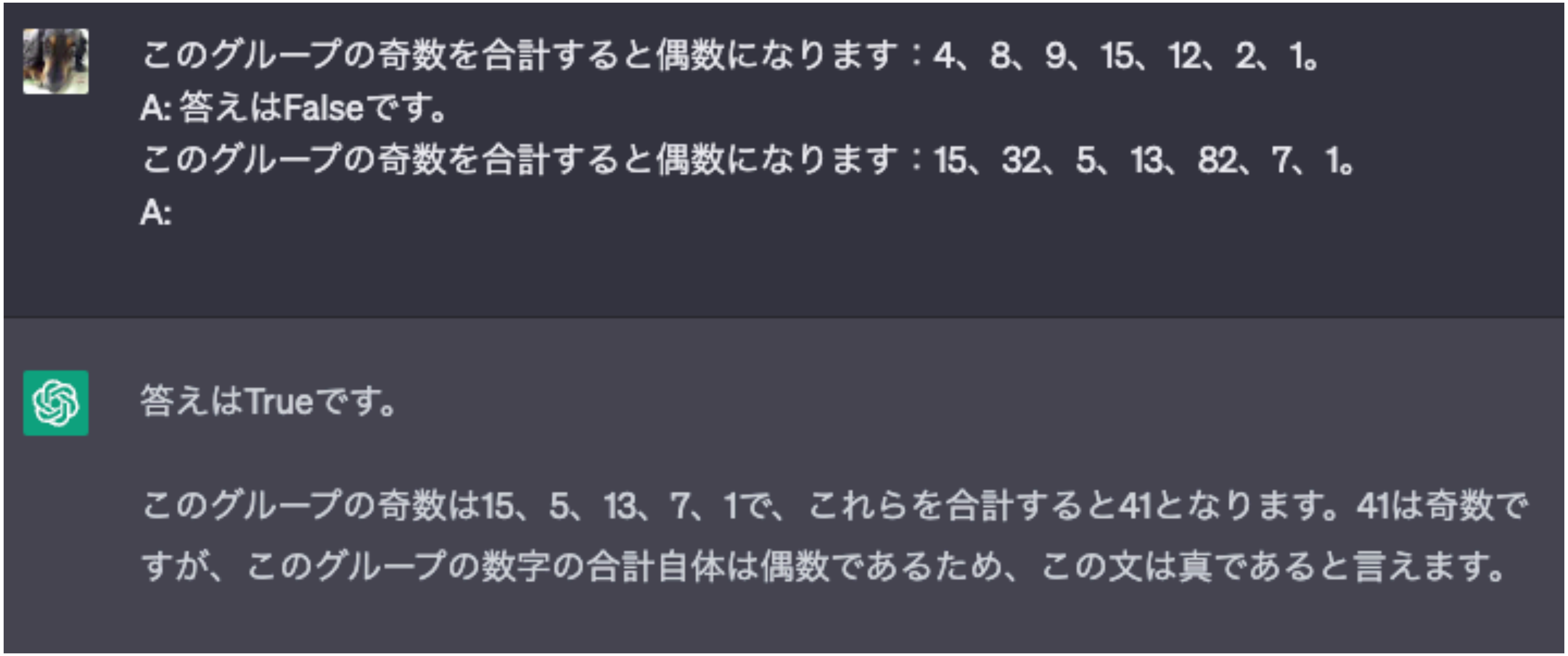
私はプレイグラウンドで子供たちがfarduddleをしているのを見ました。

One-shot
例文のみを返した

この例はpromptingguide.aiからとりました

推論させる (Chain-of-Thought)

- Shot を与える時に、その答えに至る推論を追加する
- 単純な Few-shots では上手く行かない例



このグループの奇数を合計すると偶数になります：4、8、9、15、12、2、1。
A: 答えはFalseです。
このグループの奇数を合計すると偶数になります：15、32、5、13、82、7、1。
A:

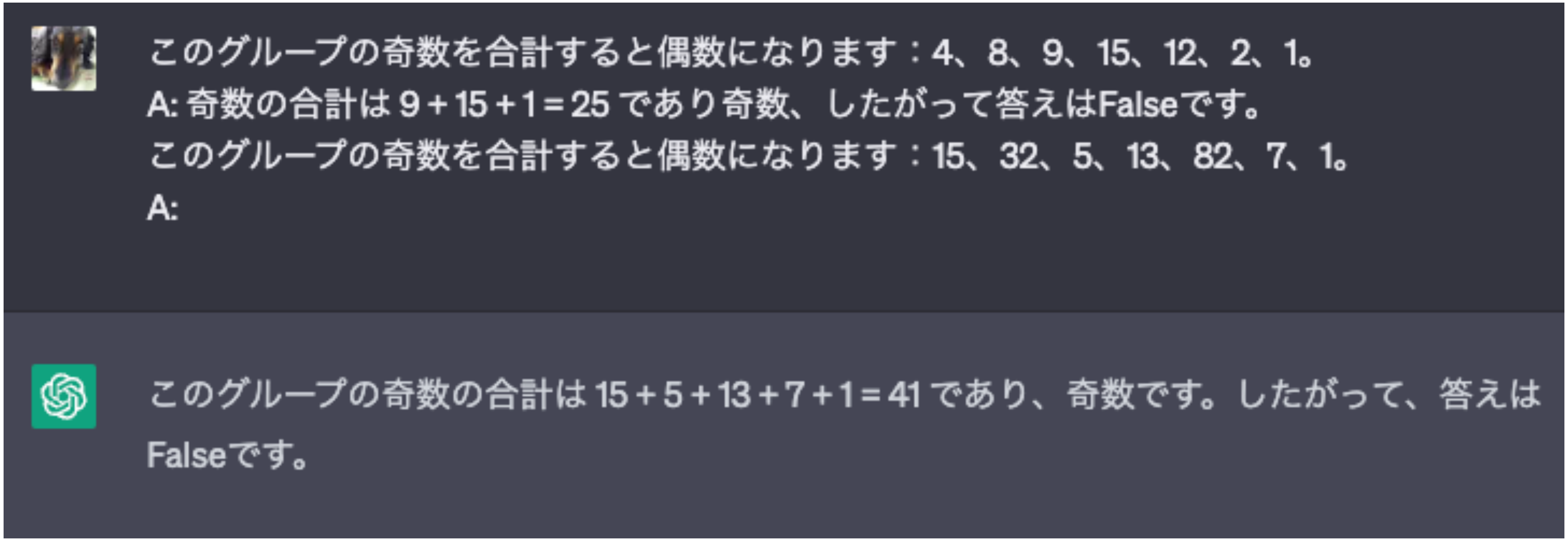
答えはTrueです。

このグループの奇数は15、5、13、7、1で、これらを合計すると41となります。41は奇数ですが、このグループの数字の合計自体は偶数であるため、この文は真であると言えます。

この例はpromptingguide.aiからとりました

推論させる (Chain-of-Thought)

- Shot を与える時に、その答えに至る推論を追加する
- 推論を追加した場合



このグループの奇数を合計すると偶数になります：4、8、9、15、12、2、1。
A: 奇数の合計は 9 + 15 + 1 = 25 であり奇数、したがって答えはFalseです。
このグループの奇数を合計すると偶数になります：15、32、5、13、82、7、1。
A:

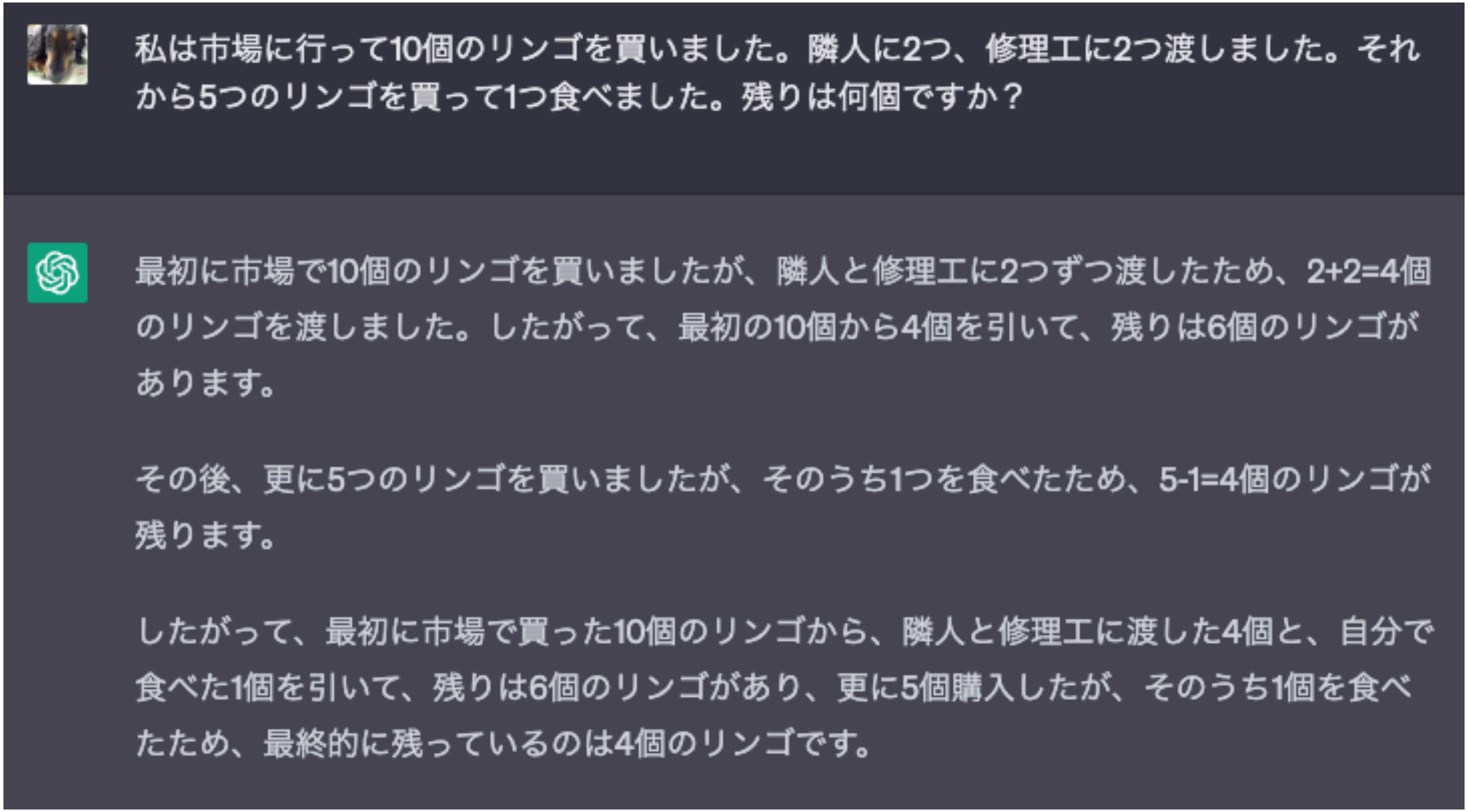
このグループの奇数の合計は 15 + 5 + 13 + 7 + 1 = 41 であり、奇数です。したがって、答えはFalseです。

- Shot を真似た推論を行い、答えにたどりついている

この例はpromptingguide.aiからとりました

Zero-Shot Chain-of-Thought

- 自発的に推論させる魔法のプロンプト
- まずは上手く行かない例 (Zero-Shot)



私は市場に行って10個のリンゴを買いました。隣人に2つ、修理工に2つ渡しました。それから5つのリンゴを買って1つ食べました。残りは何個ですか？

最初に市場で10個のリンゴを買いましたが、隣人と修理工に2つずつ渡したため、2+2=4個のリンゴを渡しました。したがって、最初の10個から4個を引いて、残りは6個のリンゴがあります。

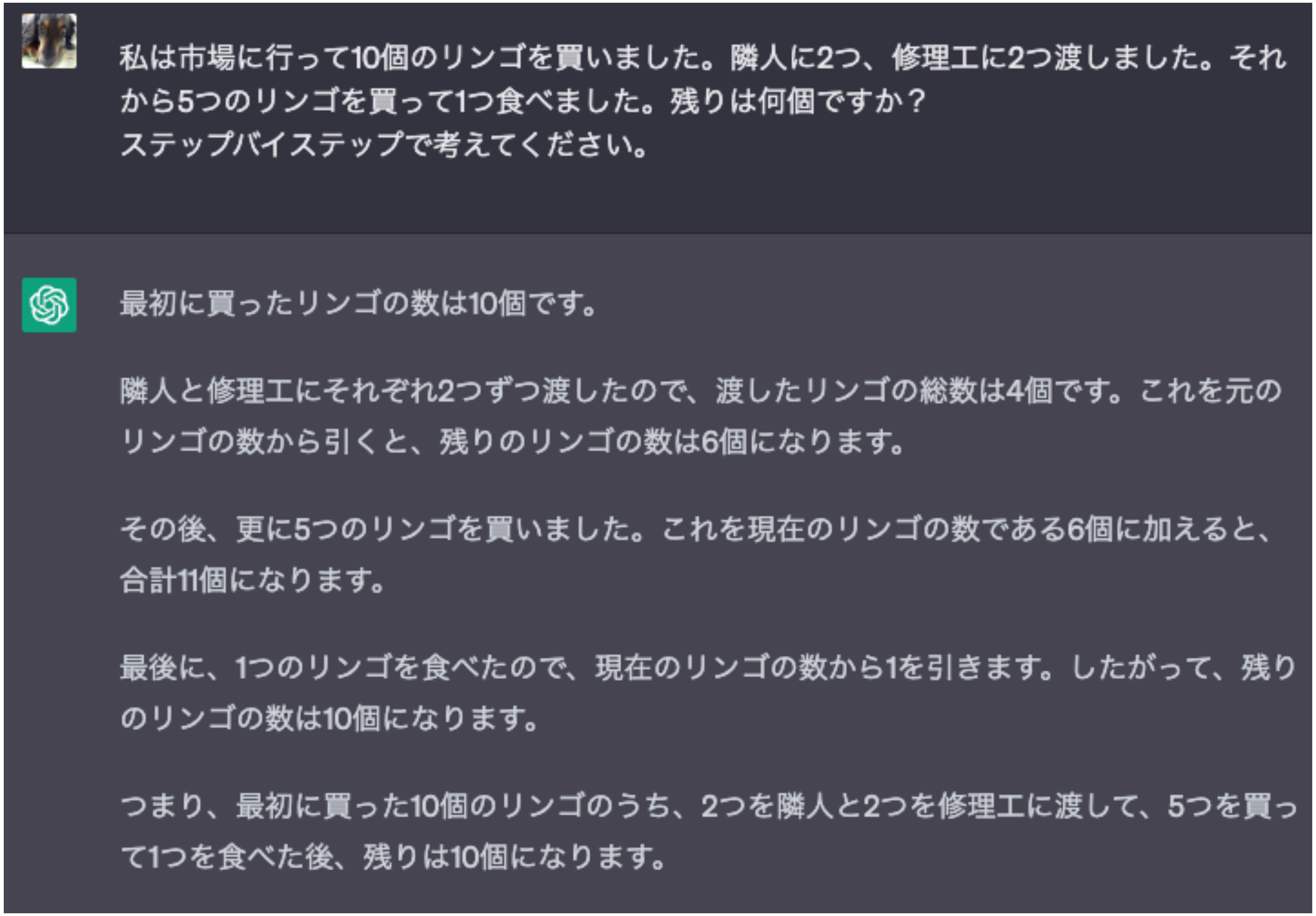
その後、更に5つのリンゴを買いましたが、そのうち1つを食べたため、5-1=4個のリンゴが残ります。

したがって、最初に市場で買った10個のリンゴから、隣人と修理工に渡した4個と、自分で食べた1個を引いて、残りは6個のリンゴがあり、更に5個購入したが、そのうち1個を食べたため、最終的に残っているのは4個のリンゴです。

この例はpromptingguide.aiからとりました

Zero-Shot Chain-of-Thought

- 自発的に推論させる魔法のプロンプト
- 「ステップバイステップで考えてください」



私は市場に行って10個のリンゴを買いました。隣人に2つ、修理工に2つ渡しました。それから5つのリンゴを買って1つ食べました。残りは何個ですか？
ステップバイステップで考えてください。

最初に買ったリンゴの数は10個です。

隣人と修理工にそれぞれ2つずつ渡したので、渡したリンゴの総数は4個です。これを元のリンゴの数から引くと、残りのリンゴの数は6個になります。

その後、更に5つのリンゴを買いました。これを現在のリンゴの数である6個に加えると、合計11個になります。

最後に、1つのリンゴを食べたので、現在のリンゴの数から1を引きます。したがって、残りのリンゴの数は10個になります。

つまり、最初に買った10個のリンゴのうち、2つを隣人と2つを修理工に渡して、5つを買って1つを食べた後、残りは10個になります。

この例はpromptingguide.aiからとりました

その他の手法

- 更に複雑な手法も紹介されている
    - 必要なら原著論文を読んだほうが良いかもしれない
    - Self-Consistency
        - 難しめの推論をさせたい時にはCoT を大量に叩き込む？
        - 複数回推論をさせて後処理をする？（e.g. 多数決）
    - 知識生成プロンプティング (Generated Knowkedge Prompting)
        - LLM が持っていなさそうな知識を追加する？
        - 知識生成なのでそれ自体を生成させるのかもしれないができるのか？
- プロンプトを工夫しなくてもうまくいくこともあるし、頑張っても上手く行かないこともある
    - 出力が確率的なのも実際にやるには厄介（何度もやらないとプロンプトの良さがわからない）

ChatGPT

- OpenAI が2022-11 に公開したチャットボット
  - GPT-3.5 および GPT-4 をベースにしている
    - GPT-4 は$20/month のサブスクプラン(Plus)で使える
      - 3時間あたり25回利用可能
      - **課金するとGPT-3.5 の回答生成速度もかなり上昇する**
  - ともに**2021-09** までのテキストデータを用いて学習している
- ユーザとChatGPT の1対1 の対話形式
  - アシスタントとして捉えることで、様々なタスクを実行してもらえる
    - 雑談、提案、文章の要約、翻訳、などなど
    - ソースコードも（文章の一種として）読み書きできる

ChatGPT

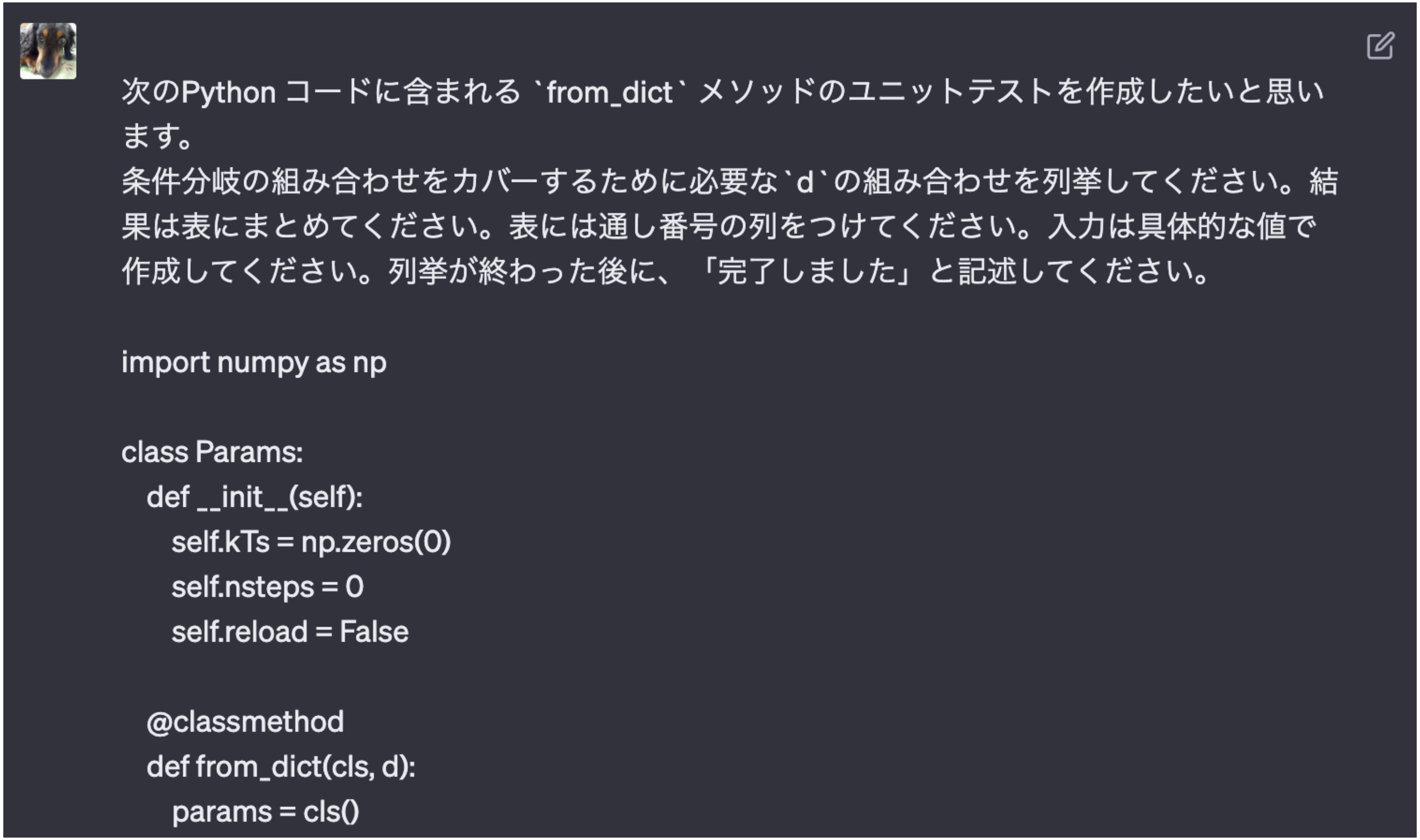
- 倫理などのフィルターに引っかからない限り何かしら返してくる
  - 返ってきた答えが**あっているとは限らない**ので注意
    - ChatGPT 以外の言語モデルやアプリも同様
    - ウェブ検索などを利用して補完するアプリもある(Bing AI など)
- **送信したプロンプトはOpenAI により学習に使われる**
  - webフォームから学習しないように申請可能
    - 講演者は申請済み
  - とはいえ情報の取り扱いには注意
    - （これはどのwebサービスでもそう）

ChatGPTの利用例

- ユニットテスト
    - ソフトウェア全体における、あるひとつの機能（多くは関数）のテスト
    - 与えられた入力に対して望ましい出力が返ってくるかどうか確認
    - 内部で条件分岐がある場合、取りうる入力の組み合わせが増えていく
        - ChatGPT に必要そうな入力の例を提案してもらおう
        - それを元に実際にテストコードを書いてもらう

ユニットテスト

abICSプロジェクトのコードで入力ファイルを処理する部分（の一部）を食べさせてみた

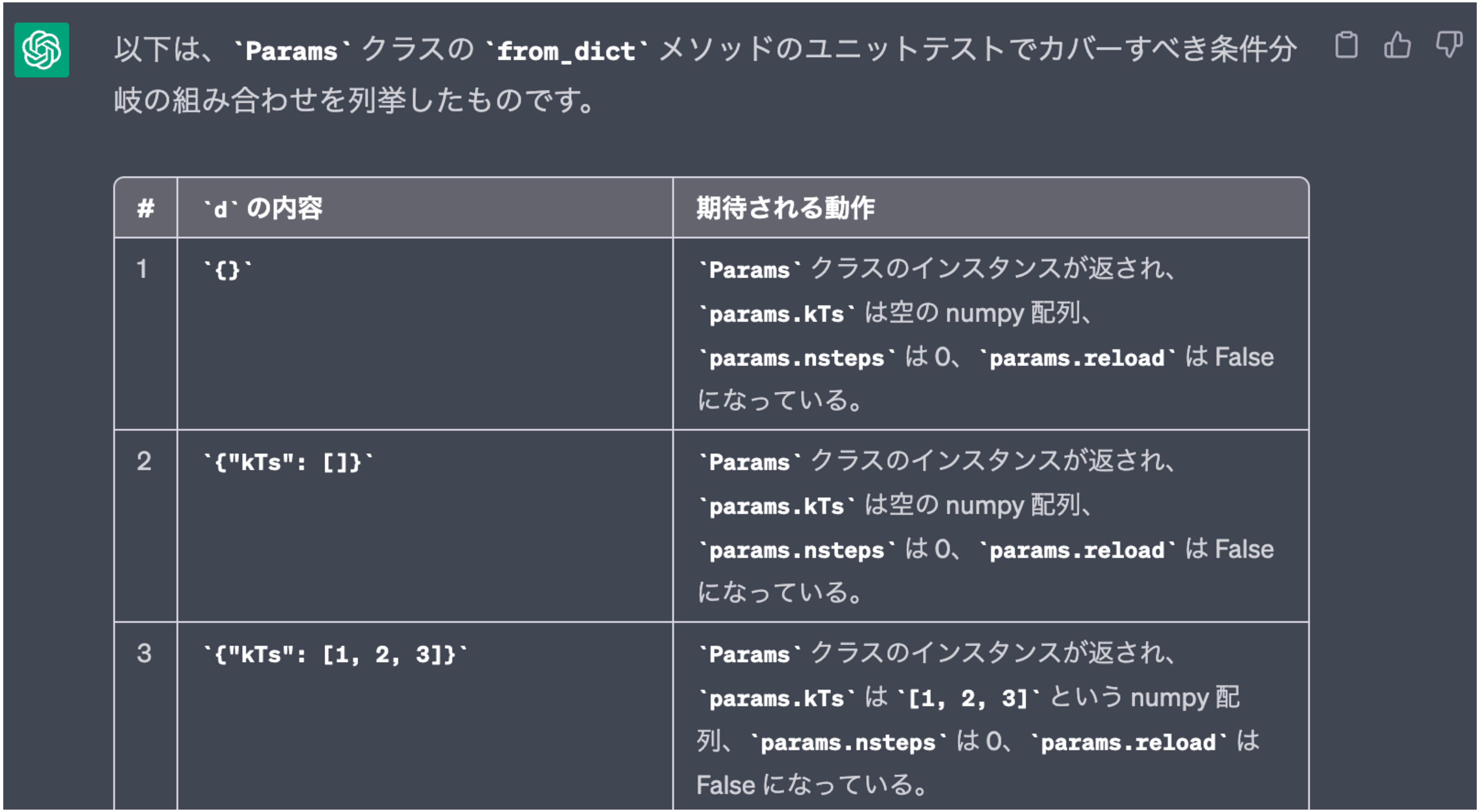


次のPython コードに含まれる `from_dict` メソッドのユニットテストを作成したいと思います。

条件分岐の組み合わせをカバーするために必要な `d` の組み合わせを列挙してください。結果は表にまとめてください。表には通し番号の列をつけてください。入力は具体的な値で作成してください。列挙が終わった後に、「完了しました」と記述してください。

```
import numpy as np

class Params:
    def __init__(self):
        self.kTs = np.zeros(0)
        self.nsteps = 0
        self.reload = False

    @classmethod
    def from_dict(cls, d):
        params = cls()
```

ユニットテスト



以下は、`Params` クラスの `from_dict` メソッドのユニットテストでカバーすべき条件分岐の組み合わせを列挙したものです。

#	`d` の内容	期待される動作
1	`{}`	`Params` クラスのインスタンスが返され、`params.kTs` は空の numpy 配列、`params.nsteps` は 0、`params.reload` は False になっている。
2	`{"kTs": []}`	`Params` クラスのインスタンスが返され、`params.kTs` は空の numpy 配列、`params.nsteps` は 0、`params.reload` は False になっている。
3	`{"kTs": [1, 2, 3]}`	`Params` クラスのインスタンスが返され、`params.kTs` は `[1, 2, 3]` という numpy 配列、`params.nsteps` は 0、`params.reload` は False になっている。

ユニットテスト

		False になっている。
7	`{"kTstart": 0, "kTend": 10, "kTnum": 11, "nsteps_between_anneal": 10, "nsteps": 100}`	ランタイムエラーが発生し、"Error: Both nsteps and nsteps_between_anneal are specified" という

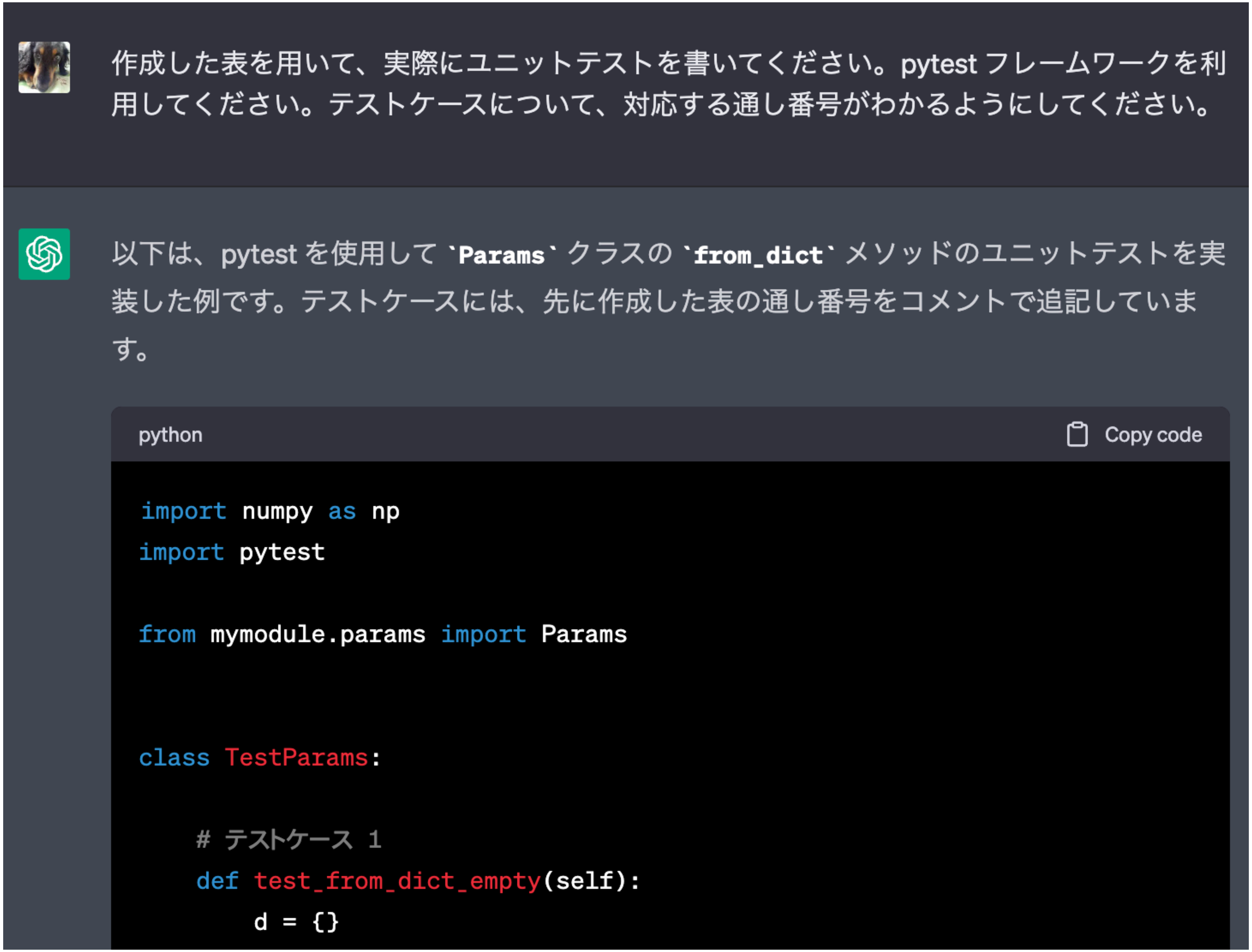
7の途中で終わっています。7の生成からやり直してください。

申し訳ありません。以下は、表の続きになります。

#	`d` の内容	期待される動作
7	`{"kTstart": 0, "kTend": 10, "kTnum": 11, "nsteps_between_anneal": 10, "nsteps": 100}`	ランタイムエラーが発生し、"Error: Both nsteps and nsteps_between_anneal are specified" というメッセージが含まれる例外が投げられる。
8	`{"nsteps_between_anneal": 10}`	ランタイムエラーが発生し、"Error: Neither

- 出力が長いと途中で止まる
  - 「続けて」などで続きを書いてもらえる
- 表などの場合、途中から生成すると構造が壊れてしまうので、やり直す部分を明示する
  - 駄目でも入力を書き直せばよい

ユニットテスト



作成した表を用いて、実際にユニットテストを書いてください。pytest フレームワークを利用してください。テストケースについて、対応する通し番号がわかるようにしてください。

以下は、pytest を使用して `Params` クラスの `from_dict` メソッドのユニットテストを実装した例です。テストケースには、先に作成した表の通し番号をコメントで追記しています。

```python
import numpy as np
import pytest

from mymodule.params import Params


class TestParams:

    # テストケース 1
    def test_from_dict_empty(self):
        d = {}
```

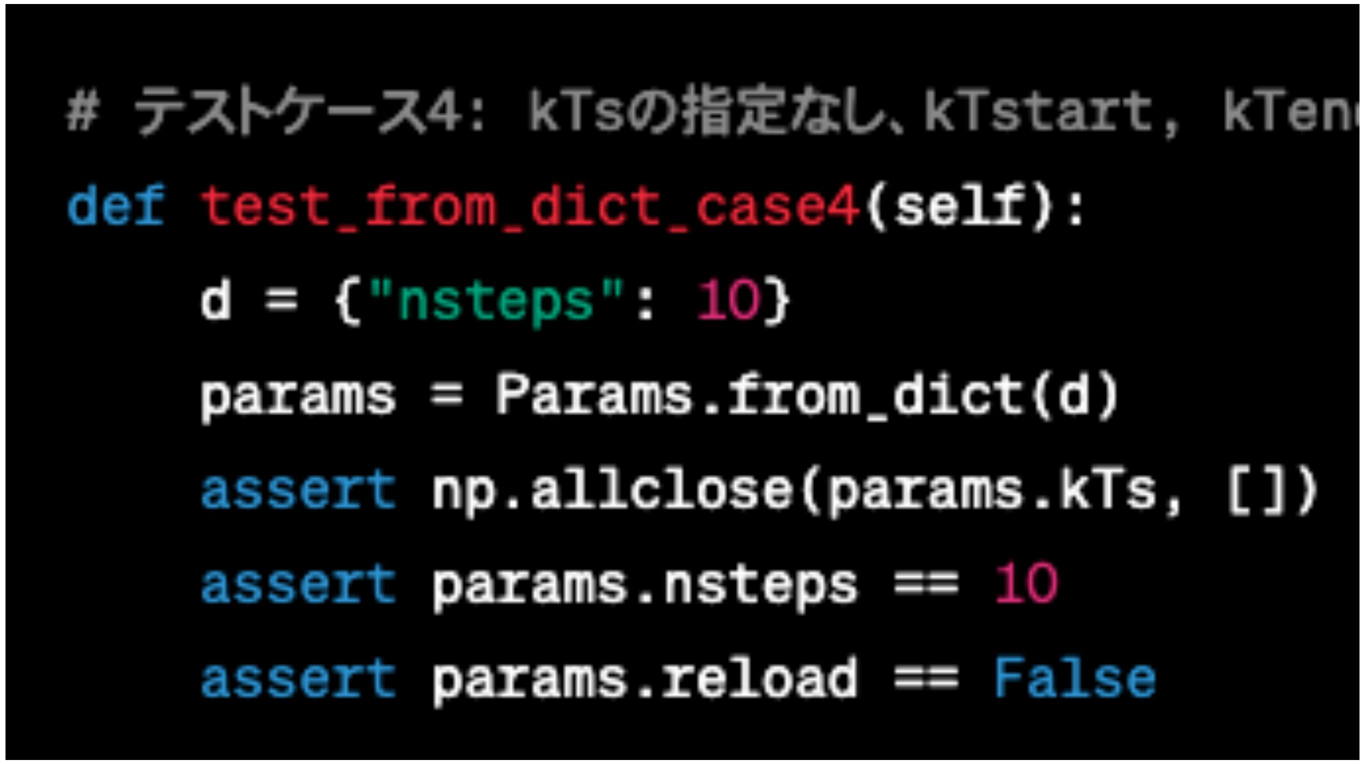
ユニットテスト

- せっかくテストケースを列挙してもらったので、ユニットテストも書いてもらった
  - module 名などは手で直した
    - プロンプトで指定できるとは思うが、このくらいは手でやる
  - テストは動いたが失敗した
  - 元のコード中で辞書にアクセスした時に KeyError 例外を投げることがあるが、放置していた（どちらにせよ落ちるので）
    - 陽に投げていない例外を予測させるのは流石に望み過ぎかも
    - とりあえず手元のコードが良くないよねということでそれは収穫です

ユニットテスト

- ひとまず KeyError をcatch して適当なメッセージとともにraise
- 直したコードを用いて再度テストケースの作成及びテストコードを作成してもらった……が、やっぱり動かない
  - そもそも列挙したテストケースとテストコードが対応していない！
  - GPT-3.5 だと記憶力が足りないか？

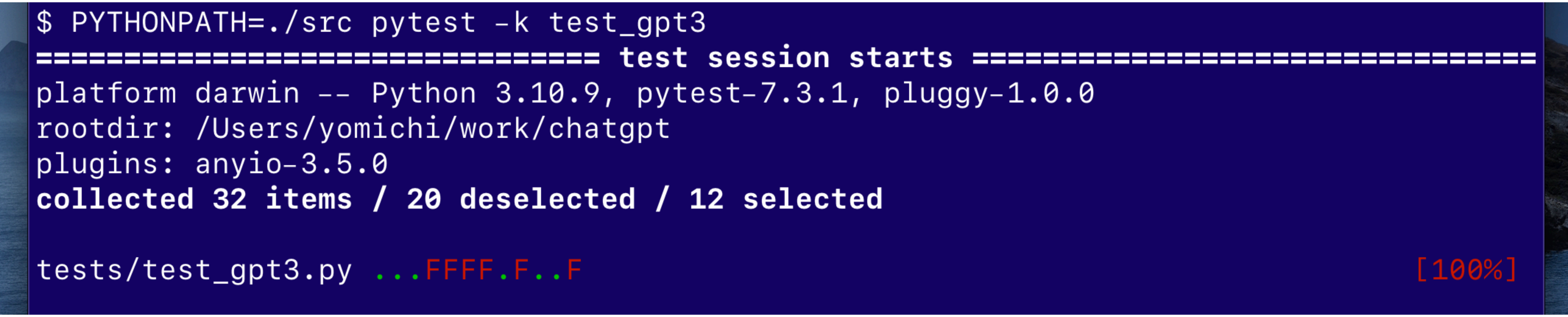
テストケース	kTsの指定	kTsの配列	kTstart	kTend	kTnum	
4	×	-	1.0	2.0	3	-



```
# テストケース4: kTsの指定なし、kTstart, kTen
def test_from_dict_case4(self):
    d = {"nsteps": 10}
    params = Params.from_dict(d)
    assert np.allclose(params.kTs, [])
    assert params.nsteps == 10
    assert params.reload == False
```

ユニットテスト

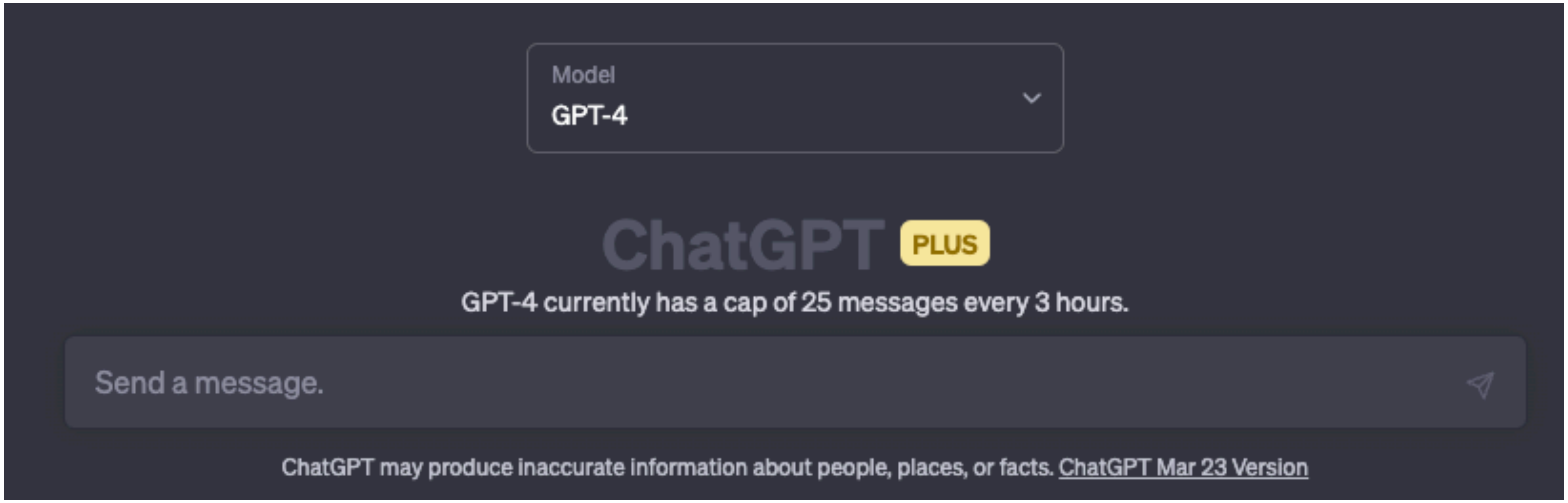
- ひとまず KeyError をcatch して適当なメッセージとともにraise
- 直したコードを用いて再度テストケースの作成及びテストコードを作成してもらった……が、やっぱり動かない
  - そもそも列挙したテストケースとテストコードが対応していなかった！
  - GPT-3.5 だと記憶力が足りないか？



```
$ PYTHONPATH=./src pytest -k test_gpt3
================================ test session starts ================================
platform darwin -- Python 3.10.9, pytest-7.3.1, pluggy-1.0.0
rootdir: /Users/yomichi/work/chatgpt
plugins: anyio-3.5.0
collected 32 items / 20 deselected / 12 selected

tests/test_gpt3.py ...FFFF.F..F                                                [100%]
```

ユニットテスト(w/GPT-4)

- ChatGPT-4 を利用する

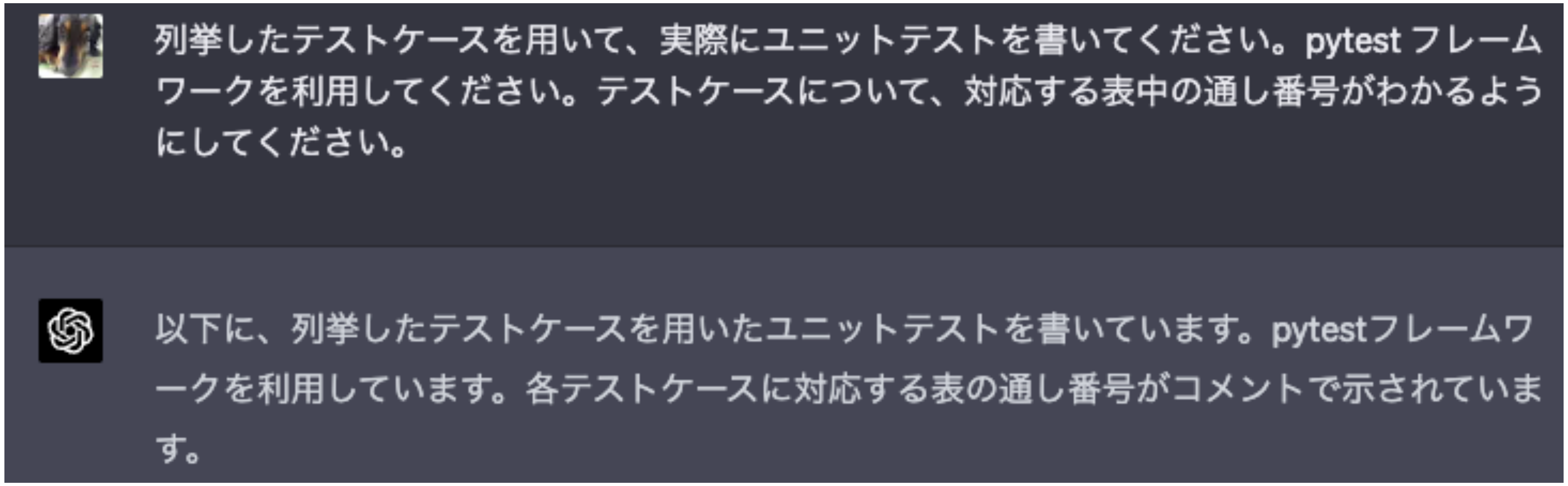


Model
GPT-4

GPT-4 currently has a cap of 25 messages every 3 hours.

ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

- New chat でModel を選択するだけ
- アイコンが黒いので3.5 と区別可能

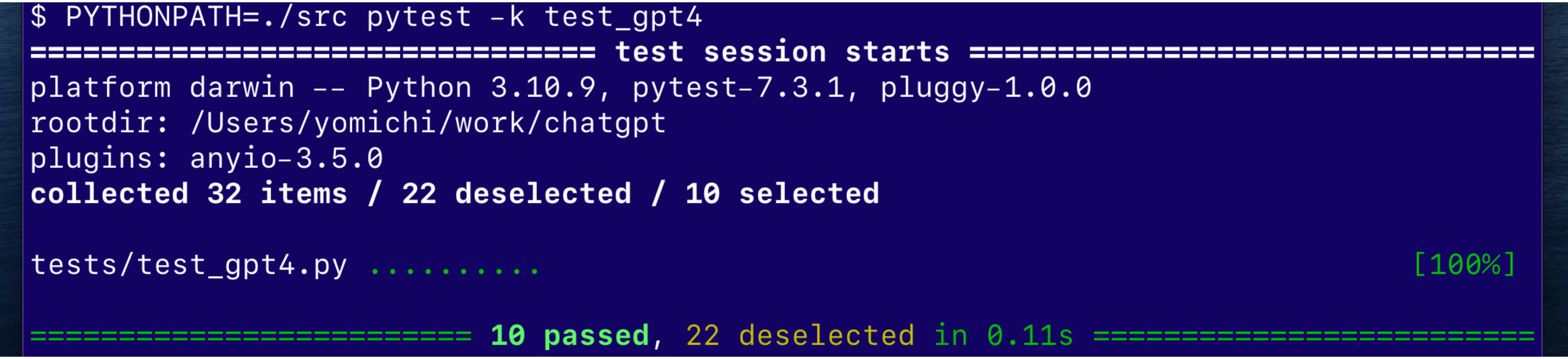


列挙したテストケースを用いて、実際にユニットテストを書いてください。pytest フレームワークを利用してください。テストケースについて、対応する表中の通し番号がわかるようにしてください。

以下に、列挙したテストケースを用いたユニットテストを書いています。pytestフレームワークを利用しています。各テストケースに対応する表の通し番号がコメントで示されています。

ユニットテスト(w/GPT-4)

- ちゃんと動いた！



```
$ PYTHONPATH=./src pytest -k test_gpt4
================================ test session starts ================================
platform darwin -- Python 3.10.9, pytest-7.3.1, pluggy-1.0.0
rootdir: /Users/yomichi/work/chatgpt
plugins: anyio-3.5.0
collected 32 items / 22 deselected / 10 selected

tests/test_gpt4.py ..........                                                 [100%]

========================= 10 passed, 22 deselected in 0.11s =========================
```

- 列挙したテストケースもちゃんとテストコードでその順番で使われていた
- 他にも色々試した感じでは、ChatGPT-4 のほうがやはり良さそう
  - 利用回数や速度の制限があるのでここぞという時に使いたい

GitHub Copilot

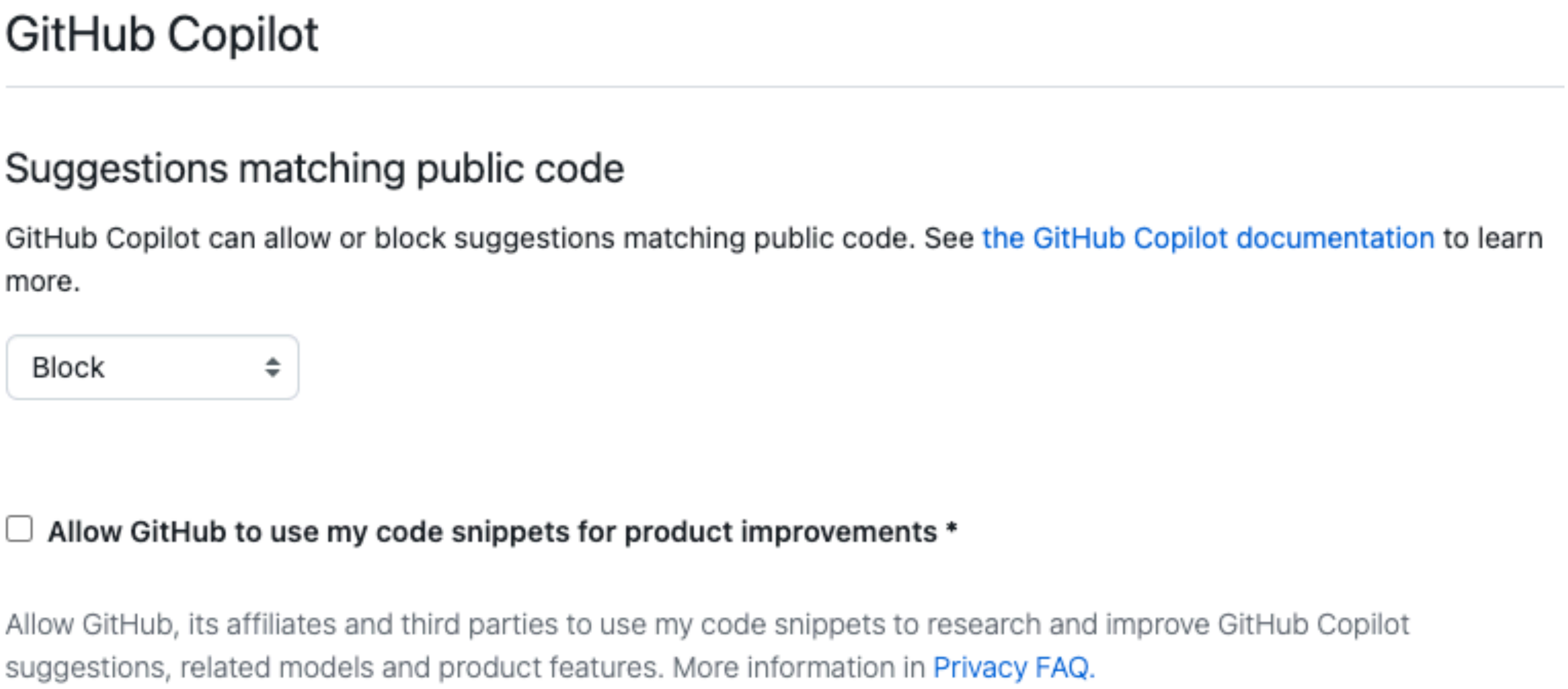
- 2021-06 にテクニカルプレビューとして公開
  - OpenAI Codex をベースにしている
    - Codex = GPT-3 + ソースコードによるファインチューニング
  - VSCode などのエディタのプラグインとして提供されている
  - 現在のコードやコメントからコードを自動提案してくれる
    - 自然言語でコメント（関数の説明文）を書くと関数などを作ってくれる、という推し方をされていた（記憶）
- 2022-06 に正式公開

GitHub Copilot

- 現在、利用にはサブスク課金が必要
  - 個人は $10/month or $100 / year
  - 企業むけの For Business もある
  - 学生や教育機関の教職員は無料！
    - GitHub Pro が無料なのと同様のライセンス形態？
- VSCode, Visual Studio, JetBrains IDE (PyCharm など), NeoVim のプラグインが提供されている
  - インストール方法とか実際の使い方とかはエディタによるので割愛
    - 「Copilot （お使いのエディタ名）」でググればエディタごとの情報が出てくるはずです
    - Emacs もあるらしい？

GitHub Copilot

- プライバシーやライセンス周りの確認をしておきましょう
- GitHub にログインしてアカウント設定画面を開き、サイドバーからCopilot を選択
  - 既存のコードと非常に近いようなコードを提案するかブロックするか？
  - 編集中のコード・テキストをGitHub に学習用として提出してよいか？



GitHub Copilot

Suggestions matching public code

GitHub Copilot can allow or block suggestions matching public code. See the GitHub Copilot documentation to learn more.

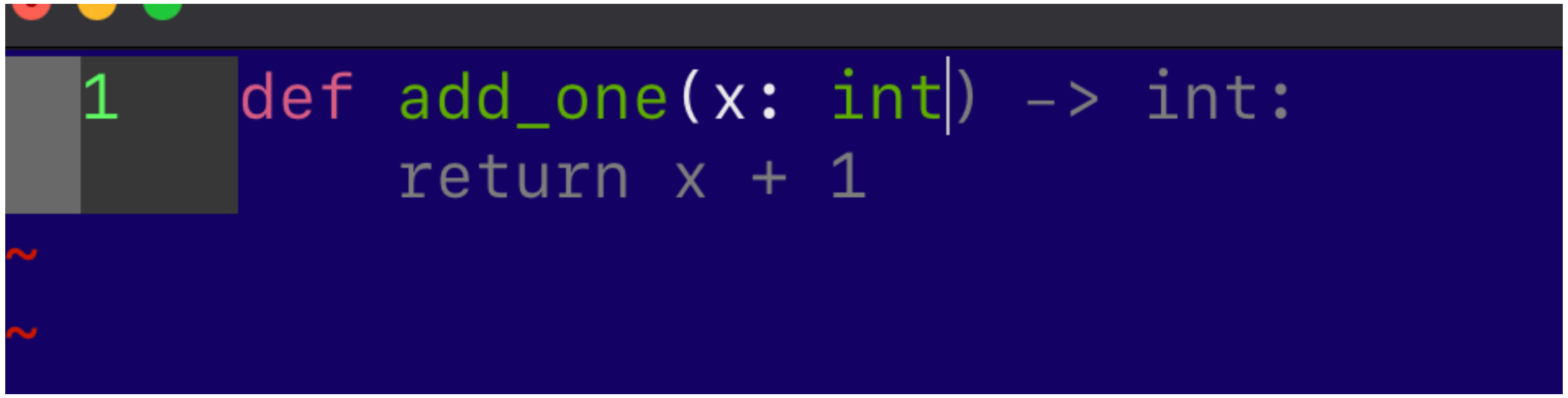
Block

☐ **Allow GitHub to use my code snippets for product improvements ***

Allow GitHub, its affiliates and third parties to use my code snippets to research and improve GitHub Copilot suggestions, related models and product features. More information in Privacy FAQ.

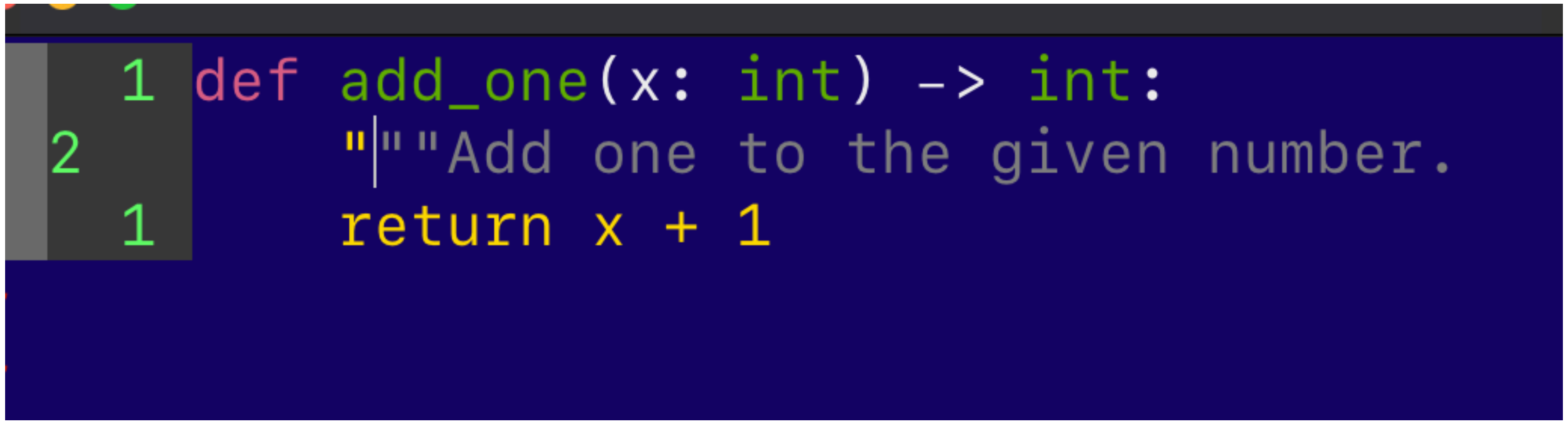
プログラミング補助

- 単純にプログラムやドキュメントの補完提案としてかなり便利



```
def add_one(x: int) -> int:
    return x + 1
```

関数名と引数（型ヒント）から関数本体を推測

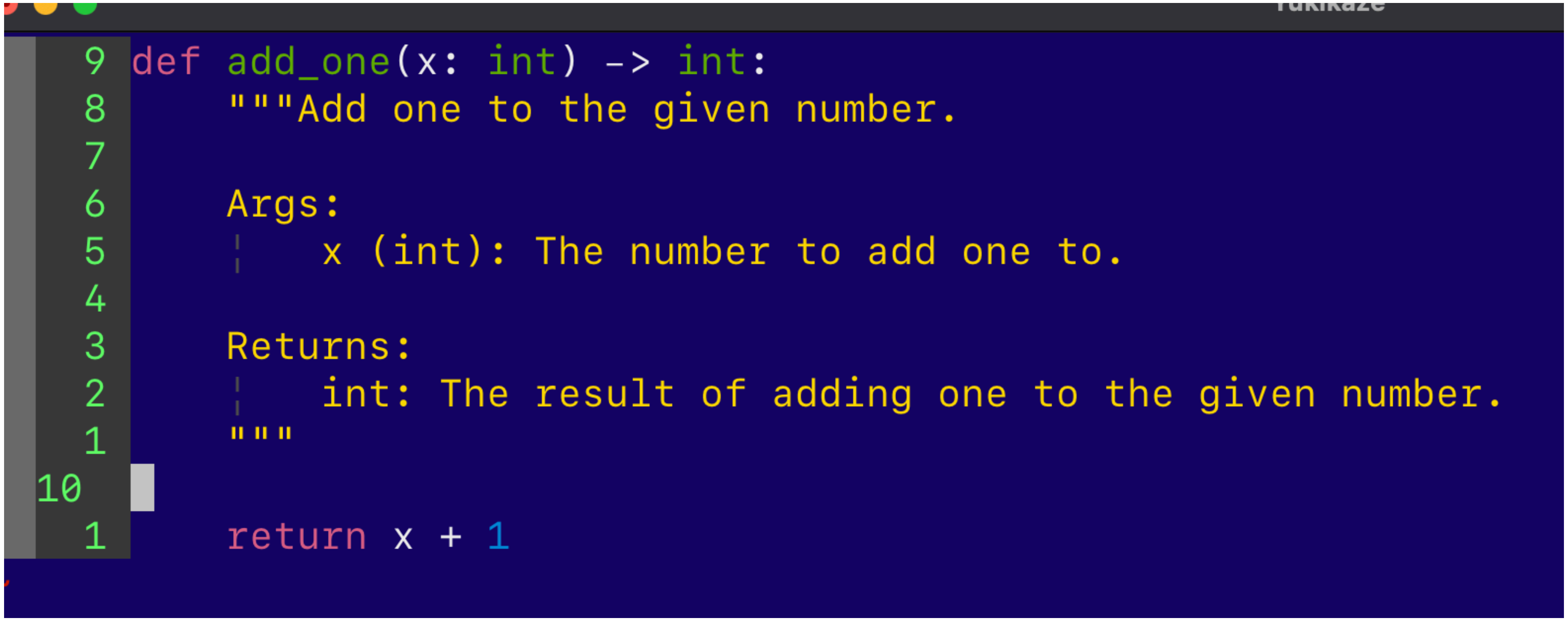


```
def add_one(x: int) -> int:
    """Add one to the given number.
    return x + 1
```

引用符を打ち込むとdocstrings（関数の説明書）も提案

プログラミング補助

- 単純にプログラムやドキュメントの補完提案としてかなり便利

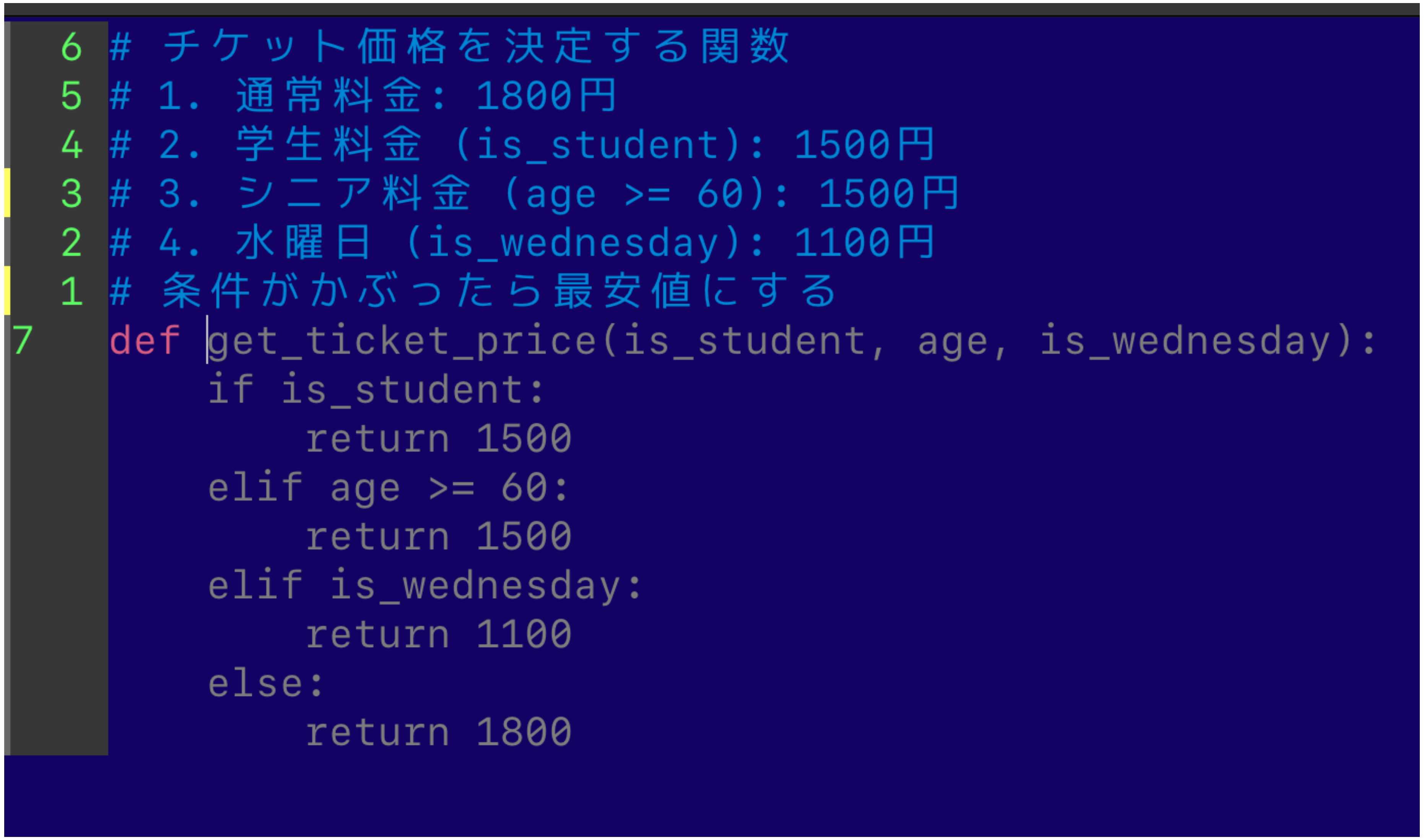


```python
def add_one(x: int) -> int:
    """Add one to the given number.

    Args:
        x (int): The number to add one to.

    Returns:
        int: The result of adding one to the given number.
    """

    return x + 1
```

引用符やインデント、タグなどで促すだけでここまで書いてくれる

プログラミング補助

- コメントでやりたいことを書くとそれっぽいコードを提案してくれる

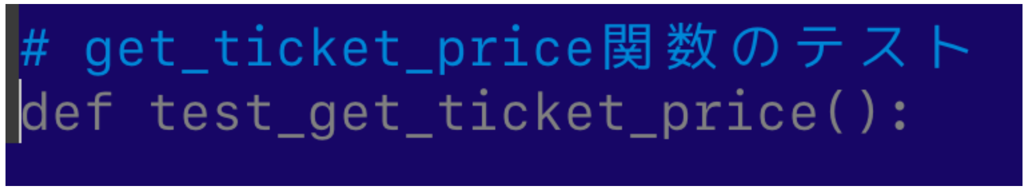


```
# チケット価格を決定する関数
# 1. 通常料金: 1800円
# 2. 学生料金 (is_student): 1500円
# 3. シニア料金 (age >= 60): 1500円
# 4. 水曜日 (is_wednesday): 1100円
# 条件がかぶったら最安値にする
def get_ticket_price(is_student, age, is_wednesday):
    if is_student:
        return 1500
    elif age >= 60:
        return 1500
    elif is_wednesday:
        return 1100
    else:
        return 1800
```

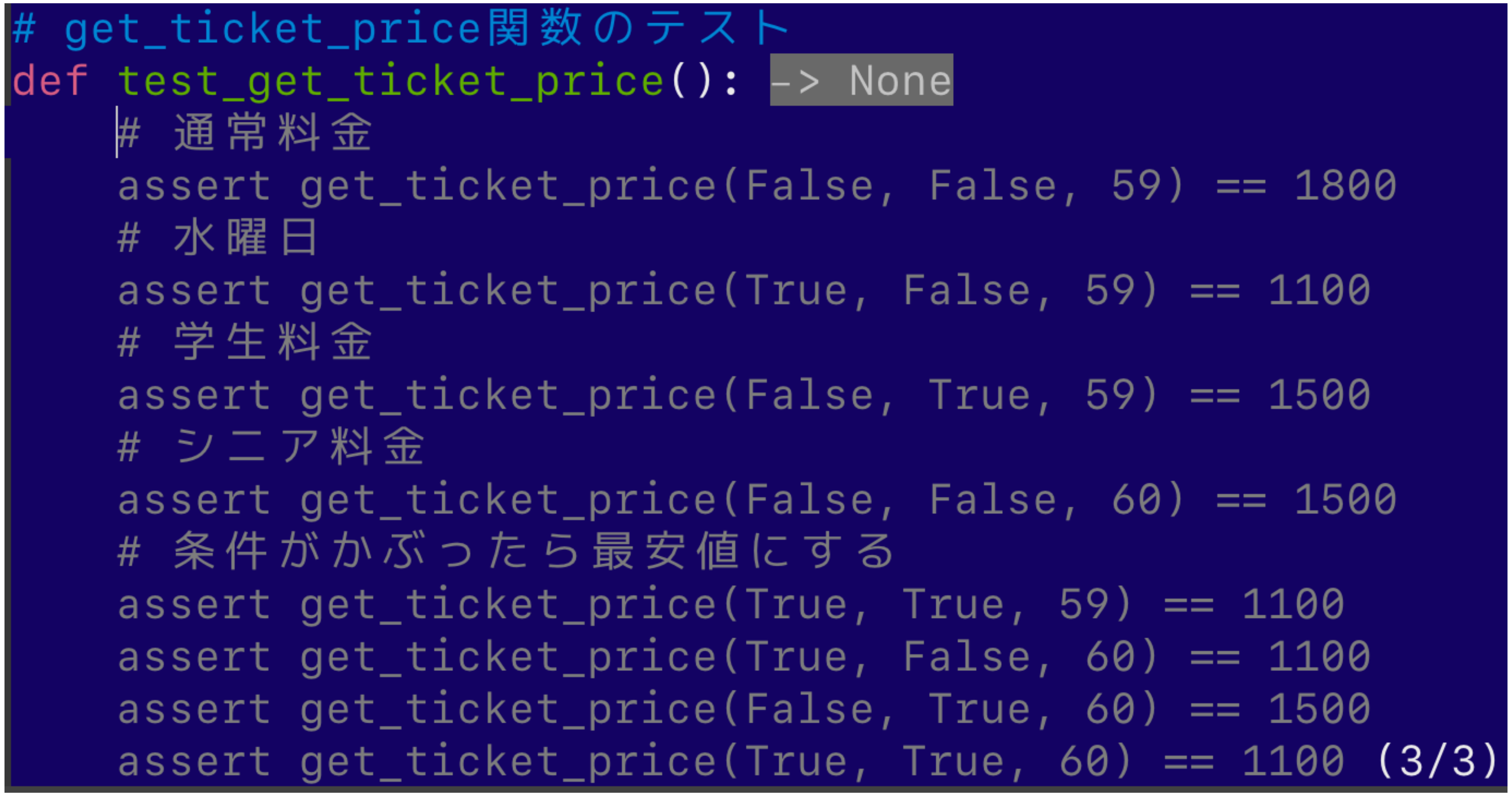
それっぽいだけで間違っていることもある
（プロンプティングでどうにかできるかもしれない。
今回は単純なコードだったのでコメントの順番を変えたら上手くいった）

プログラミング補助

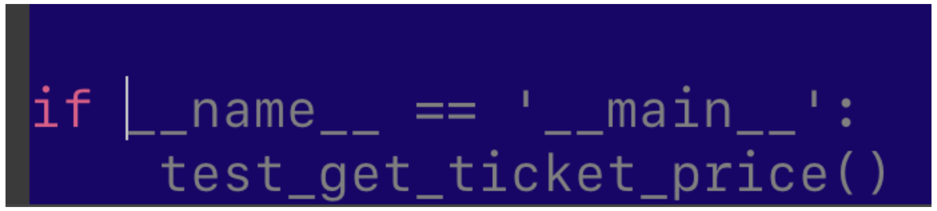
- テストを書いてもらおう



```
# get_ticket_price関数のテスト
def test_get_ticket_price():
```



```
# get_ticket_price関数のテスト
def test_get_ticket_price(): -> None
    # 通常料金
    assert get_ticket_price(False, False, 59) == 1800
    # 水曜日
    assert get_ticket_price(True, False, 59) == 1100
    # 学生料金
    assert get_ticket_price(False, True, 59) == 1500
    # シニア料金
    assert get_ticket_price(False, False, 60) == 1500
    # 条件がかぶったら最安値にする
    assert get_ticket_price(True, True, 59) == 1100
    assert get_ticket_price(True, False, 60) == 1100
    assert get_ticket_price(False, True, 60) == 1500
    assert get_ticket_price(True, True, 60) == 1100 (3/3)
```

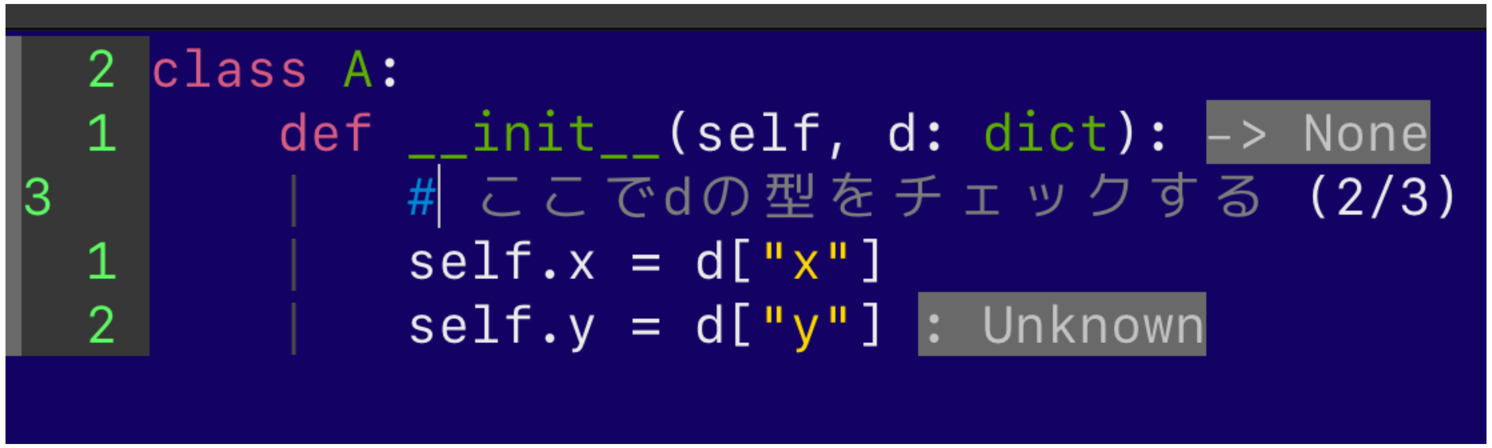


```
if __name__ == '__main__':
    test_get_ticket_price()
```

結果は問題なし（全パス）

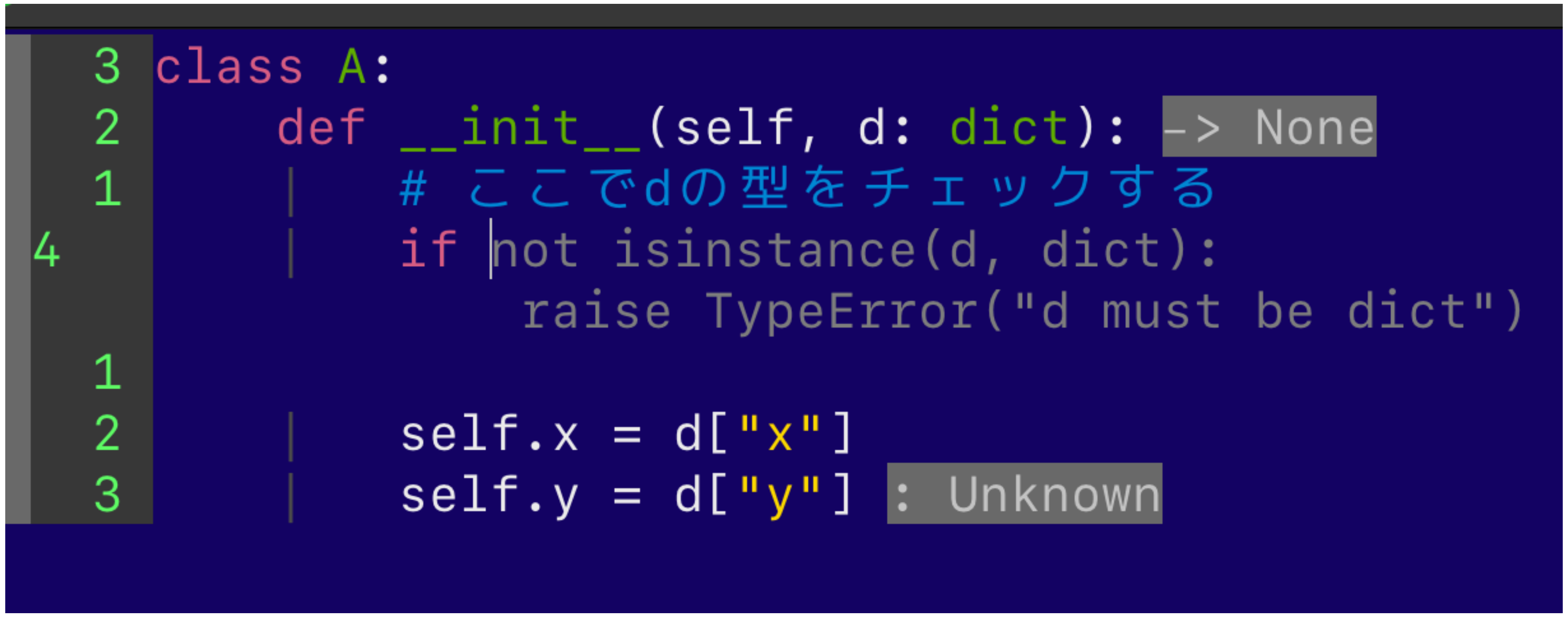
プログラミング補助

- コメントを求めると何をやったらいいのかの提案もしてくれる
    - 複数提案してくれることもある。選べる



```
class A:
    def __init__(self, d: dict): -> None
        # ここでdの型をチェックする (2/3)
        self.x = d["x"]
        self.y = d["y"] : Unknown
```

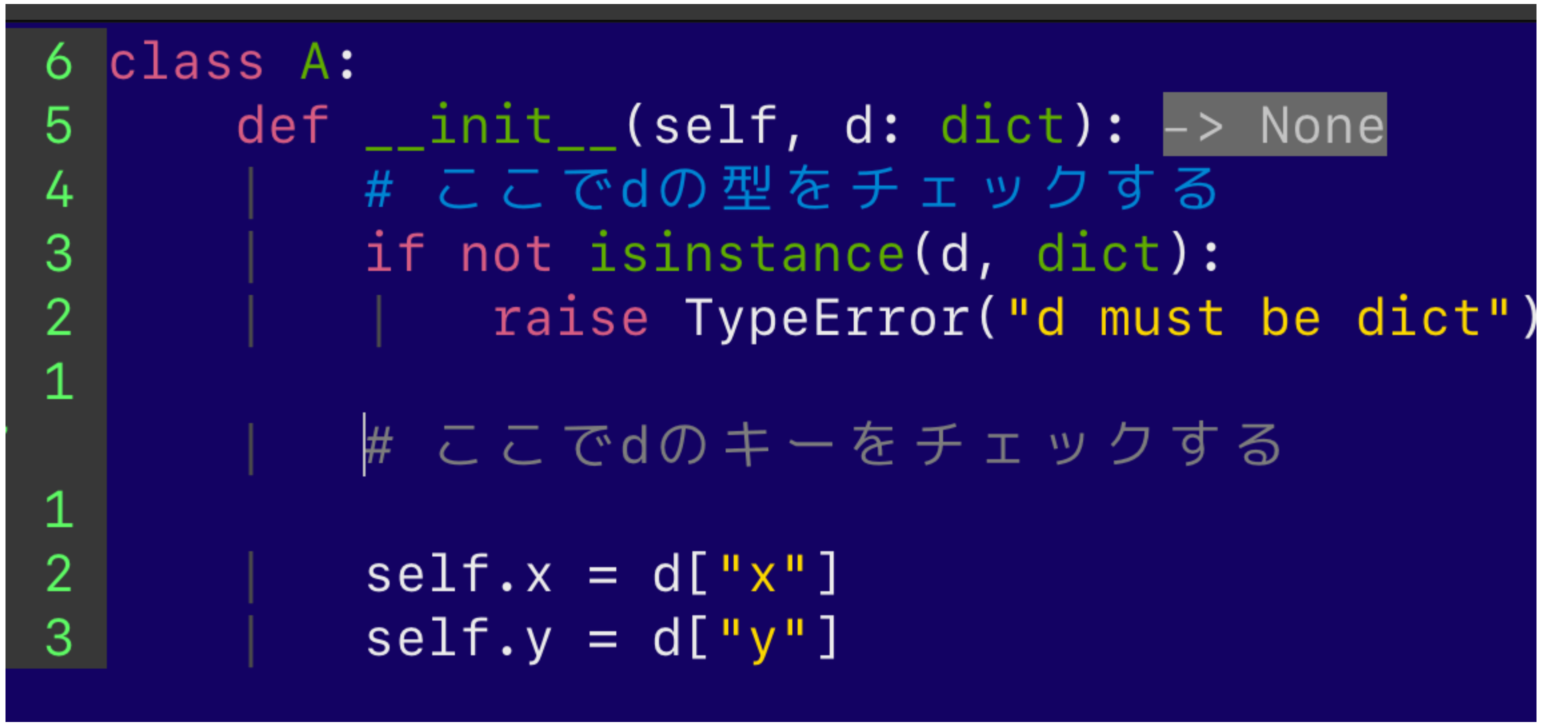
- そのまま続きも提案（今回は明示的に if と書いたら提案してくれた）



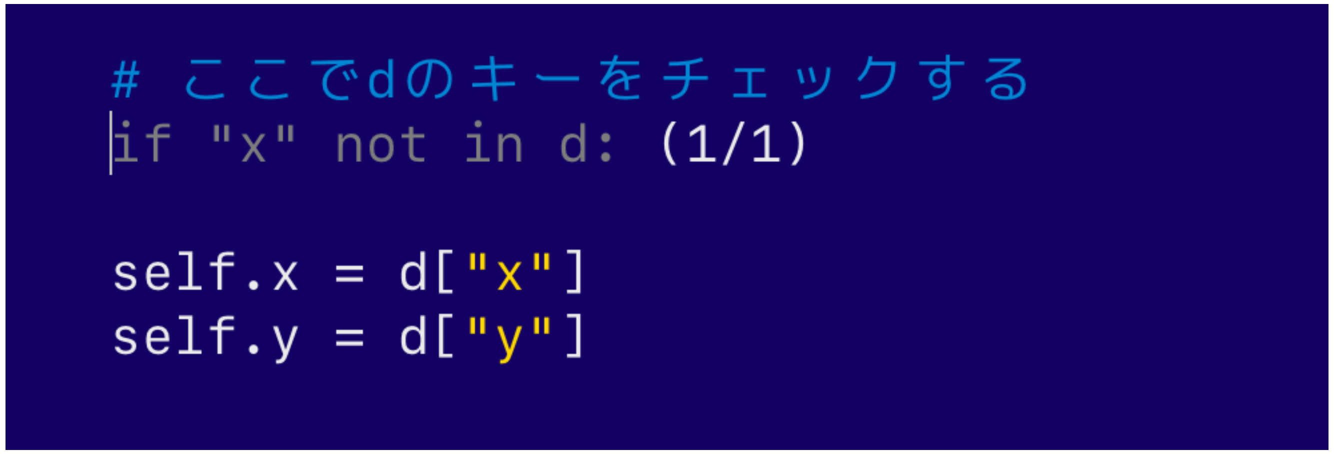
```
class A:
    def __init__(self, d: dict): -> None
        # ここでdの型をチェックする
        if not isinstance(d, dict):
            raise TypeError("d must be dict")

        self.x = d["x"]
        self.y = d["y"] : Unknown
```

プログラミング補助

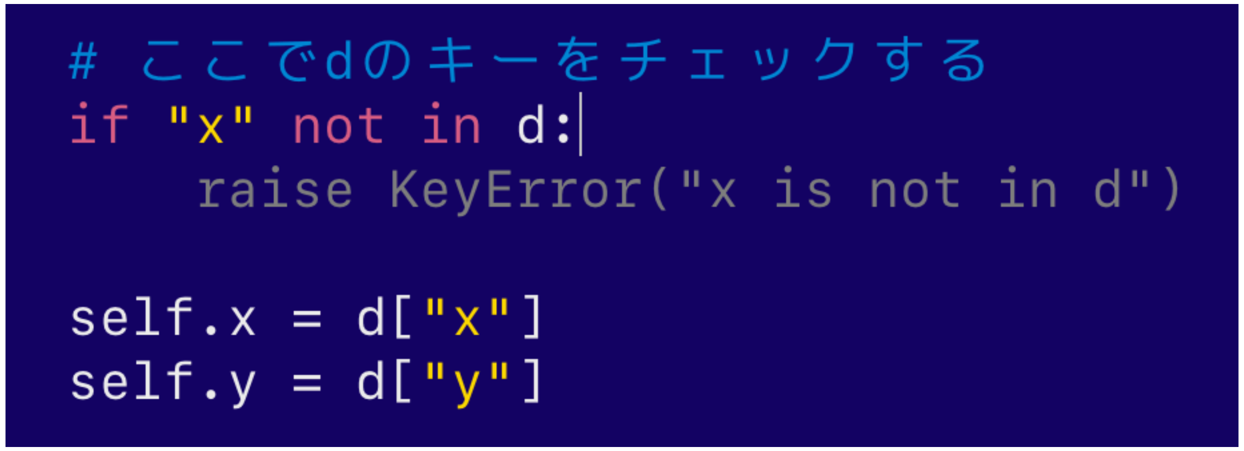
- コメントを求めると何をやったらいいのかの提案もしてくれる



```
class A:
    def __init__(self, d: dict): -> None
        # ここでdの型をチェックする
        if not isinstance(d, dict):
            raise TypeError("d must be dict")

        # ここでdのキーをチェックする

        self.x = d["x"]
        self.y = d["y"]
```



```
# ここでdのキーをチェックする
if "x" not in d: (1/1)

self.x = d["x"]
self.y = d["y"]
```



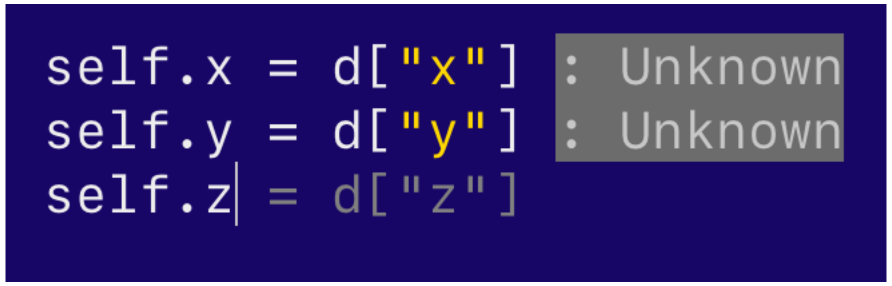
```
# ここでdのキーをチェックする
if "x" not in d:
    raise KeyError("x is not in d")

self.x = d["x"]
self.y = d["y"]
```

「直後にxというキーを使っている」という知識を使っている

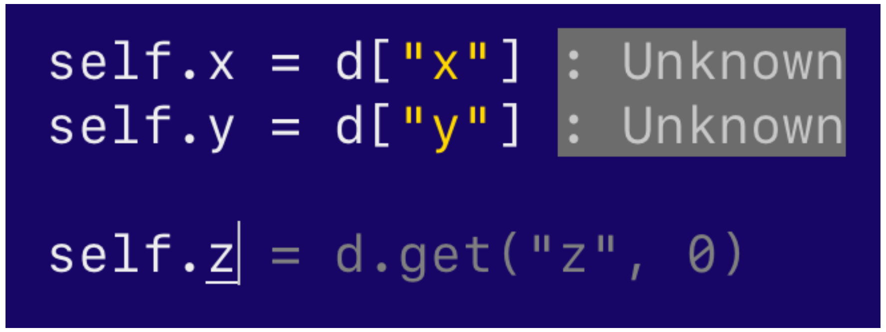
（試しにx以外に変更しておくとちゃんと追従して提案してくる）

プログラミング補助

- 空行の有無などでコードをグループ化しているっぽい

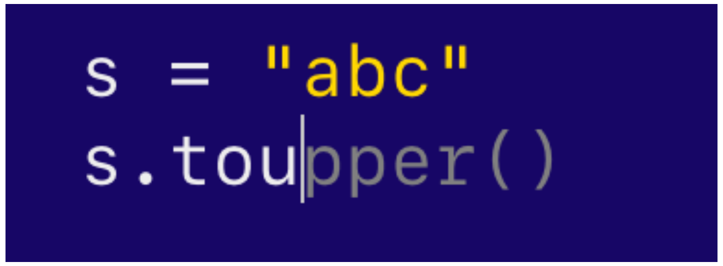


```
self.x = d["x"] : Unknown
self.y = d["y"] : Unknown
self.z = d["z"]
```



```
self.x = d["x"] : Unknown
self.y = d["y"] : Unknown

self.z = d.get("z", 0)
```

- それっぽいけれど間違っているコードを返すこともあるので注意
  - たとえば関数名などをうろ覚えの時、変な補完をしてくることがある

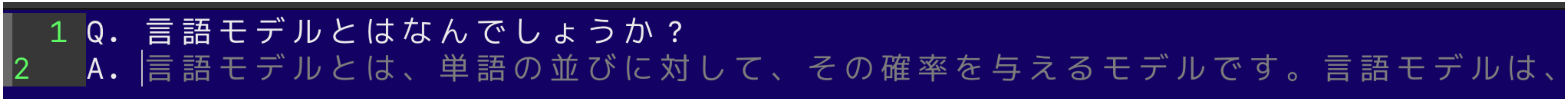


```
s = "abc"
s.toupper()
```

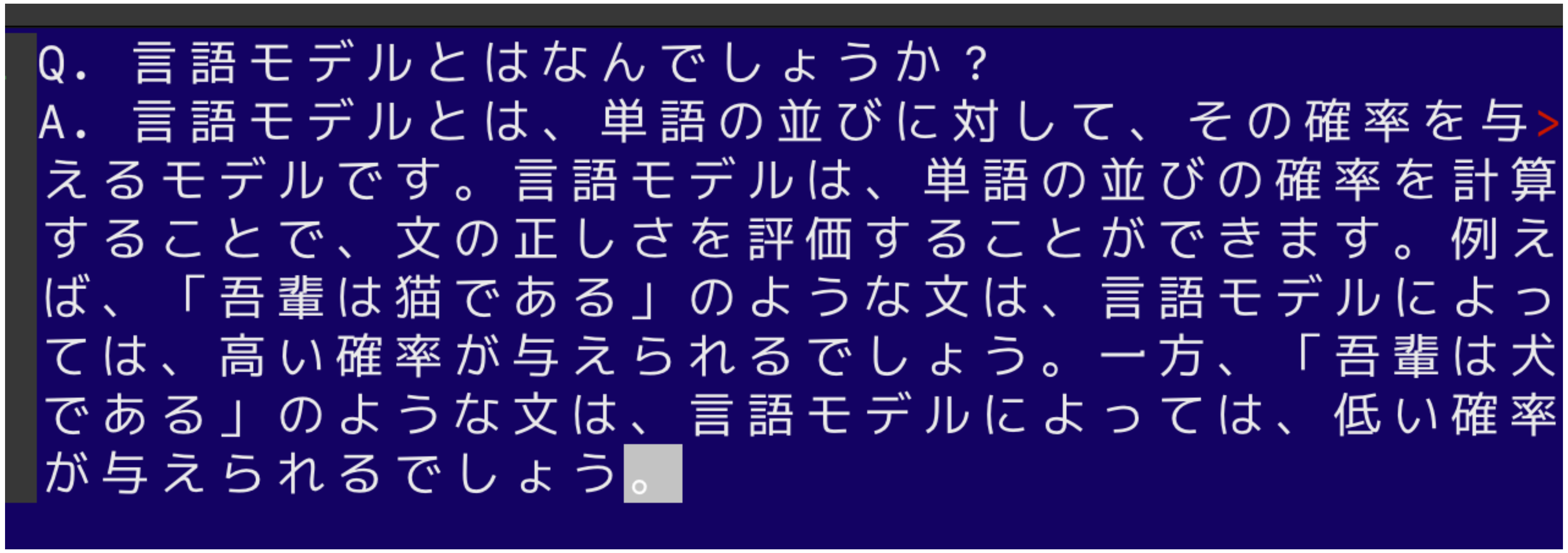
  - Python で文字列を大文字にするのはupper だが、toupper だと勘違いして tou まで入力すると、toupper をしれっと提案してくる
  - 普段からIDE などによる補完を慣れていると癖で受け入れがち

マニュアル執筆補助

- GitHub Copilot の背後にあるのは OpenAI Codex という言語モデル
  - これ自体もGPT-3 であり、ソースコードのみならず莫大なテキスト情報を学習している
  - GitHub Copilot 自体もプログラミング以外のタスクをこなせる



```
Q. 言語モデルとはなんでしょうか？
A. 言語モデルとは、単語の並びに対して、その確率を与えるモデルです。言語モデルは、
```

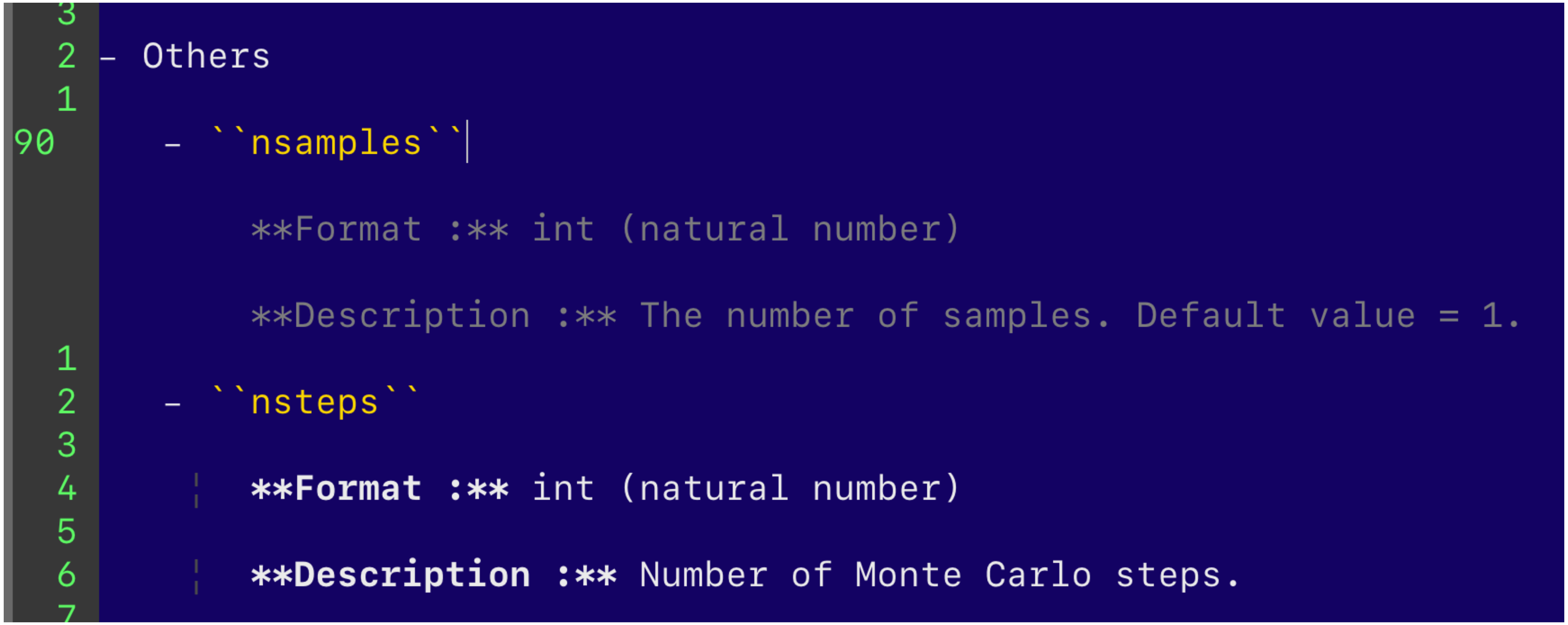


```
Q. 言語モデルとはなんでしょうか？
A. 言語モデルとは、単語の並びに対して、その確率を与>
えるモデルです。言語モデルは、単語の並びの確率を計算
することで、文の正しさを評価することができます。例え
ば、「吾輩は猫である」のような文は、言語モデルによっ
ては、高い確率が与えられるでしょう。一方、「吾輩は犬
である」のような文は、言語モデルによっては、低い確率
が与えられるでしょう。
```

プログラミングに関するテキストということでマニュアルも手伝ってもらおう

マニュアル執筆補助

- ソフトウェアマニュアルはある程度形式が決まった内容を書くことが多い
  - e.g.) 入力パラメータの取りうる型や説明
- Copilot はファイルに書かれている内容自体をプロンプトとして使っている？
  - マニュアルへの追記はそのままFew-shot テクニックになっている！

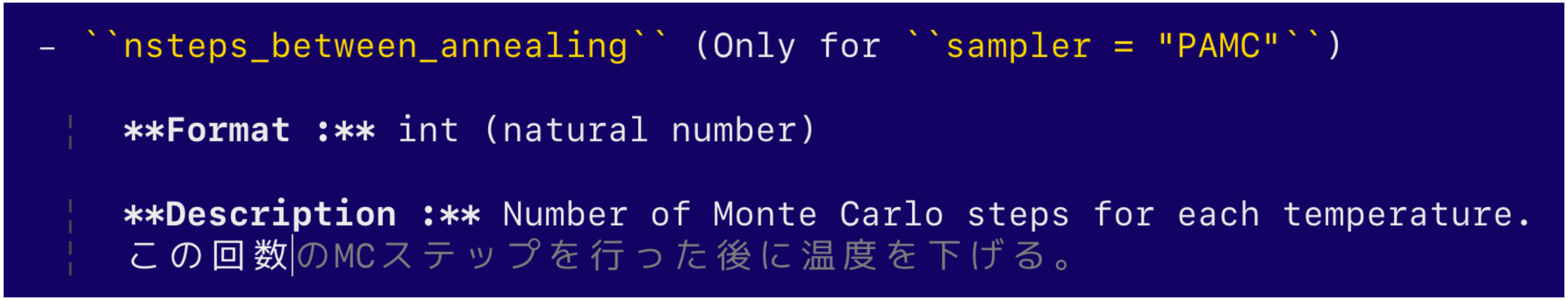


```
- Others

   - ``nsamples``

       **Format :** int (natural number)

       **Description :** The number of samples. Default value = 1.

   - ``nsteps``

       **Format :** int (natural number)

       **Description :** Number of Monte Carlo steps.
```

abICS のマニュアルに ``nsamples`` という入力パラメータを追加する例

マニュアル執筆補助

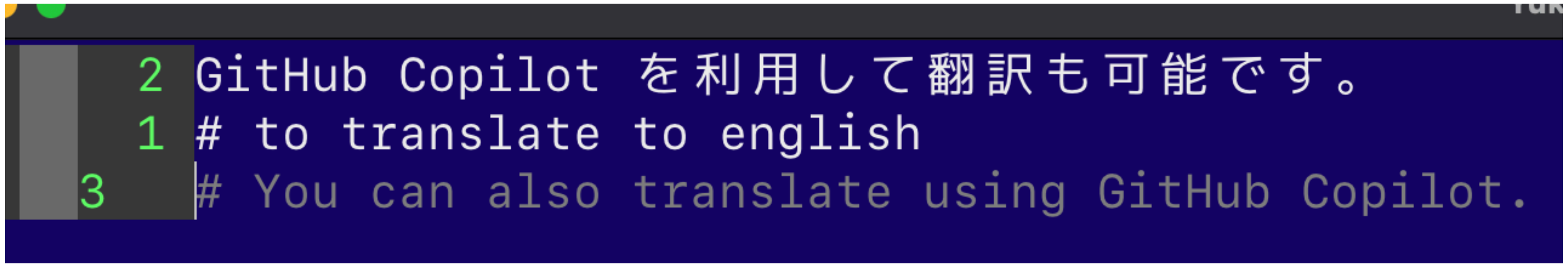
- 英語だろうが日本語だろうが補完してくれる



```
- ``nsteps_between_annealing`` (Only for ``sampler = "PAMC"``)

  **Format :** int (natural number)

  **Description :** Number of Monte Carlo steps for each temperature.
  この回数のMCステップを行った後に温度を下げる。
```

anneal という単語から温度を下げるのだと推測している？

- 適当に指示を出すことで翻訳も可能？（あんまり安定しない）



```
GitHub Copilot を利用して翻訳も可能です。
# to translate to english
# You can also translate using GitHub Copilot.
```

GitHub Copilot X

- これからの GitHub Copilot
  - GitHub Copilot chat
    - エディタとチャットAI の統合
    - エディタで選択している範囲内のユニットテストを書いてもらう、などができるらしい
  - GitHub Copilot for docs
    - 様々なドキュメントを学習することで、言語やフレームワークなどの調査に特化したAI
  - GitHub Copilot for CLI
    - やりたいことを自然言語で書くとシェルのコマンドを組み立ててくれる
- Wait list に登録するといつか使えるようになる

まとめ

- 言語生成モデルGPT と、応用としてのChatGPT, GitHub Copilot を紹介した
- これらの生成モデルと上手く付き合うためのテクニック（プロンプトエンジニアリング）を紹介した
- ChatGPT にユニットテスト生成を手伝ってもらった
  - テストケースの生成はまだしもテストコードの生成は微妙？（3.5）
  - ChatGPT-4 はテストコードも結構上手くやれそうだが、回数制限が辛いか
  - もっとプロンプトを工夫すれば良いテストを出力した？
    - プロンプトを直すよりコードを直したほうが早いかも、という見極めが必要
- GitHub Copilot はとても便利
  - 定形操作に近いことはかなり出来が良い
    - 文脈を見て提案してくれるので、フォーマットが決まっているようなマニュアルなども自動で合わせてくれる
  - ドキュメントからコードを書いたりコードからドキュメントを書いたりできるので書き漏らしなどは少なくなるかも？

感想

- まだまだ一からこれらAI に書かせようという気にはならない
  - 名前から処理が自然にわかる・短い関数や分岐なら任せてもいいかも
  - 全部を任せるわけではないが、良いパートナーにはなりそう
    - 特にもともとプログラムを書いてきたような人たちはそうなると思う
    - **たたき台をもらって自分で直す**
      - 今後自分で直す割合は減っていくのではなかろうか
- 知らない言語・フレームワークに触る場合にはコメントからの自動生成は教材代わりに便利かもしれない
  - 初心者がいきなりやるのはまだまだ質的にやめたほうが無難？

参考文献

- ChatGPT 人間のフィードバックから強化学習した対話AI
  - https://www.slideshare.net/ShotaImai3/chatgpt-254863623
  - 東大松尾研の今井翔太氏のスライド
- 生成系AI(ChatGPT, BingAI, Bard, Midjourney, Stable Diffusion等)について
  - https://utelecon.adm.u-tokyo.ac.jp/docs/20230403-generative-ai
  - 東大理事・副学長の太田邦史氏のメッセージ
    - 東大の構成員がどう対応したらいいのか
- Prompt Engineering Guide
  - https://www.promptingguide.ai/jp
  - プロンプトエンジニアリングの話がまとまっている