

AI-050

補足資料：Module04

- 自然言語プロンプトを使用してコードを記述する
- AI モデルを使用して単体テストを構築し、複雑なコードを理解する
- 既存のコードのコメントとドキュメントを生成する

私はプログラマです。私の作業を、あなたはどのように支援できますか？ できることを教えてください。



もちろんです。私はあなたのプログラミング作業を様々な方法で支援できます。以下にいくつかの例を挙げます：

1. **コードのデバッグ**：プログラムにエラーがあった場合、その原因を特定し、修正するための提案を行うことができます。
2. **コードの最適化**：あなたのコードを分析し、パフォーマンスを向上させるための提案を行うことができます。たとえば、無駄なループを削除する、より効率的なデータ構造を提案するなどです。
3. **新しい技術の学習**：新しいプログラミング言語やフレームワークを学ぶ際に、その基本的な概念や使用方法を教えることができます。
4. **コードレビュー**：あなたのコードをレビューし、可読性やメンテナンス性を向上させるための提案を行うことができます。
5. **自動化**：定型的なタスクを自動化するスクリプトを作成するお手伝いをすることができます。
6. **ドキュメンテーション**：あなたのコードのドキュメンテーションを作成するお手伝いをします。
7. **プロジェクト管理**：タスクの管理やスケジューリング、進行状況のトラッキングなどを支援することができます。

以上のような事柄に対して支援を行うことが可能です。具体的に何かお困りのことがあれば、お気軽にお申し付けください。

あなたは、どのようなプログラミング言語、スクリプト言語、フレームワーク、IaCなどに対応していますか？



私は多くのプログラミング言語、スクリプト言語、フレームワーク、IaCに対応しています。具体的には以下のようなものを含まます：

1. プログラミング言語: Python, Java, C++, C#, JavaScript, TypeScript, Go, Rust, Ruby, Swiftなど。
2. スクリプト言語: Bash, PowerShell, Perl, PHPなど。
3. フレームワーク: Django, Flask, Spring, Angular, React, Vue.js, Ruby on Rails, Laravelなど。
4. IaC(Infrastructure as Code)ツール: Terraform, Ansible, Chef, Puppet, AWS CloudFormationなど。

また、これらの技術に関する質問に答えるだけでなく、コードのデバッグ、コードレビュー、コード最適化の提案などの支援も提供できます。それぞれの言語やツールに関する具体的な知識は常にアップデートされており、新たな技術にも対応できるようになっています。

- Codex
 - code-cushman-001 / 002
 - code-davinch-001 / 002
- GPT-3.5
 - gpt-35-turbo
 - gpt-35-turbo-16k
 - gpt-35-turbo-instruct
- GPT-4
 - gpt-4
 - gpt-4-32k

Completions API
(Completionsプレイグラウンド) で利用可

- Codex
 - code-cushman-001 / 002
 - code-davinci-001 / 002

- GPT-3.5

- gpt-35-turbo
- gpt-35-turbo-16k
- gpt-35-turbo-instruct

- GPT-4

- gpt-4
- gpt-4-32k

これらはより新しい
Chat Completions API
(Chatプレイグラウンド) で
利用できるのでオススメ

- システムメッセージの設定例

システム メッセージ ⓘ

あなたはPythonのプログラマです。

本資料ではPythonを例としますが、生成AIモデル自体はC#、Java、JavaScriptなど、非常に多くのプログラミング言語のコードを生成できます。

Pythonで二分探索を行うメソッドを記述してください。第一引数にint型の配列、第二引数に探索する数値、戻り値は探索する数値が見つかった配列の位置としてください。|



Pythonで二分探索を行うメソッドを以下のように記述できます。このメソッドは、指定された整数配列（第一引数）と探索する数値（第二引数）を受け取り、その数値が配列内に存在する場合、その位置（インデックス）を返します。見つからない場合は、`-1`を返します。

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid # 探索した数値の位置を返す
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1 # 見つからなかった場合

# 使用例
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 5

print(binary_search(arr, target)) # 出力: 4
```

このメソッドでは、まず配列をソートされた状態であると仮定しています。二分探索はソートされた配列に対してのみ有効です。`left`と`right`変数を使って探索範囲を管理し、探索する数値が中央値`mid`と一致するかどうかを確認します。数値が中央値より大きい場合は、探索範囲を右側に絞り、小さい場合は左側に絞ります。このプロセスを数値を見つけるか、または探索範囲が無くなる（`left > right`）まで繰り返します。

Pythonで、引数で与えられた時刻と現在時刻との差を計算し、その差が分単位なら「N分前」、時間単位なら「N時間前」、日単位なら「N日前」といったように文字列化するメソッドを書いてください。



Pythonで指定された時刻と現在時刻との差を計算し、適切な形式で文字列化するメソッドを以下に示します。このメソッドはdatetimeモジュールを使用していますので、まずはこのモジュールをインポートします。

```
from datetime import datetime, timedelta

def time_difference(past_time_str):
    # 現在時刻を取得
    now = datetime.now()

    # 引数で与えられた時刻をdatetimeオブジェクトに変換
    # 引数のフォーマット例: '2023-01-30 15:30:00'
    past_time = datetime.strptime(past_time_str, '%Y-%m-%d %H:%M:%S')

    # 差を計算
    delta = now - past_time

    # 差を日、時間、分で表現
    days = delta.days
    hours, remainder = divmod(delta.seconds, 3600)
    minutes = remainder // 60

    # 差を文字列で表現
    if days > 0:
        return f"{days}日前"
    elif hours > 0:
        return f"{hours}時間前"
    elif minutes > 0:
        return f"{minutes}分前"
    else:
        return "たった今"

# 使用例
# 過去の時刻を文字列で指定
past_time_str = '2023-04-01 12:30:00'
print(time_difference(past_time_str))
```

このメソッドでは、最初に現在の時刻を取得し、次に引数で与えられた過去の時刻文字列をdatetimeオブジェクトに変換します。その後、現在時刻と過去の時刻との差（delta）を計算し、この差を日、時間、分に分けて適切な文字列を返します。

注意点として、このメソッドは引数の時刻文字列のフォーマットが'%Y-%m-%d %H:%M:%S'であることを前提としています。異なるフォーマットの時刻を扱いたい場合は、strptimeメソッドのフォーマット指定子を適宜変更する必要があります。また、delta.daysが負の値になる場合（未来の時刻が指定された場合）は考慮していないので、そのようなケースを扱いたい場合は追加の条件分岐が必要になります。

このコードをC#で書き直してもらえますか？

もちろんです。Pythonで書かれた時刻の差を計算し、その差を「N分前」、「N時間前」、「N日前」といった形で文字列化するメソッドをC#に書き直してみましょう。

C#では、`DateTime`クラスを用いて日時を扱います。また、日時の差を扱うには`TimeSpan`クラスを使用します。以下に、指定された時刻と現在時刻との差を計算し、適切な文字列で返すC#のメソッドを示します。

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // 使用例
        // 過去の時刻を文字列で指定（フォーマットは "yyyy-MM-dd HH:mm:ss"）
        string pastTimeStr = "2023-04-01 12:30:00";
        Console.WriteLine(TimeDifference(pastTimeStr));
    }

    static string TimeDifference(string pastTimeStr)
    {
        // 現在時刻を取得
        DateTime now = DateTime.Now;

        // 引数で与えられた時刻をDateTimeオブジェクトに変換
        // 注意: DateTime.ParseExactはフォーマットが一致しない場合例外を投げるので適切に処理する必要がある
        DateTime pastTime = DateTime.ParseExact(pastTimeStr, "yyyy-MM-dd HH:mm:ss", null);

        // 差を計算
        TimeSpan delta = now - pastTime;

        // 差を日、時間、分で表現し、文字列で返す
        if (delta.Days > 0)
        {
            return $"{delta.Days}日前";
        }
        else if (delta.Hours > 0)
        {
            return $"{delta.Hours}時間前";
        }
        else if (delta.Minutes > 0)
        {
            return $"{delta.Minutes}分前";
        }
        else
        {
            return "たった今";
        }
    }
}
```

このC#のコードでは、`DateTime.Now`で現在の時刻を取得し、`DateTime.ParseExact`メソッドで引数で与えられた時刻文字列を`DateTime`オブジェクトに変換しています。その後、現在時刻と過去の時刻との差（`TimeSpan`オブジェクト）を計算し、その差を日、時間、分に分けて適切な文字列を返しています。

引数の時刻文字列のフォーマットは"`yyyy-MM-dd HH:mm:ss`"としていますが、異なるフォーマットを扱いたい場合は、`DateTime.ParseExact`メソッドの第二引数で指定するフォーマット文字列を変更してください。また、`DateTime.ParseExact`はフォーマットが一致しない場合に例外を投げるので、実際のアプリケーションでは適切な例外処理を行う必要があります。

Pythonでint型の配列の数値の合計を返すメソッドを書いてください

Pythonで整数型の配列（リスト）内の数値の合計を返すメソッドは、以下のように簡単に書くことができます。このメソッドは`sum`関数を使用しています。`sum`関数は、イテラブルなオブジェクト（この場合はリスト）の全ての要素の合計を計算します。

```
def sum_of_integers(arr):  
    return sum(arr)  
  
# 使用例  
arr = [1, 2, 3, 4, 5]  
print(sum_of_integers(arr)) # 出力: 15
```

このメソッド`sum_of_integers`は、引数として整数のリスト`arr`を受け取り、その要素の合計を返します。`sum`関数はPythonに組み込まれている関数であり、非常に便利に使えます。この関数はリストの要素が数値である限り、要素の合計を計算してくれます。

このコードの単体テストを生成してください。テストケースは1件でOKです。

Pythonで単体テストを行うには、`unittest`モジュールを使用するのが一般的です。以下に、先ほど示した`sum_of_integers`関数のための単体テストケースを1件含むテストクラスを示します。

まず、テスト対象の`sum_of_integers`関数を含むファイルを`my_module.py`と仮定します。その関数は以下のとおりです。

```
# my_module.py

def sum_of_integers(arr):
    return sum(arr)
```

次に、この関数のための単体テストを以下のように書きます。

```
# test_my_module.py

import unittest
from my_module import sum_of_integers

class TestSumOfIntegers(unittest.TestCase):
    def test_sum_of_integers(self):
        arr = [1, 2, 3, 4, 5]
        result = sum_of_integers(arr)
        self.assertEqual(result, 15)

if __name__ == '__main__':
    unittest.main()
```

このテストクラス`TestSumOfIntegers`には、`test_sum_of_integers`というメソッドが含まれています。このメソッドは、`sum_of_integers`関数に整数のリスト`[1, 2, 3, 4, 5]`を渡し、その戻り値が期待される合計値`15`と一致するかを検証します。`assertEqual`メソッドを使用して、実際の結果と期待される結果を比較しています。

テストを実行するには、コマンドラインから以下のコマンドを入力します。

```
python -m unittest test_my_module.py
```

これにより、`unittest`モジュールが`test_my_module.py`スクリプトを読み込み、`TestSumOfIntegers`クラス内のテストメソッドを実行します。テストが成功すると、テスト結果として`ok`が表示されます。

以下のコードが何をするのか説明してくれますか？

```
for num in range (1, 101):  
    if num % ( 3*5 ) ==0:  
        print ("Fizz Buzz")  
    elif num % 3 == 0:  
        print ("Fizz ")  
    elif num % 5 ==0:  
        print ("Buzz")  
    else:  
        print(num)
```

このコードは、1から100までの数値（両端を含む）に対して、以下のルールに基づいて出力を行うプログラムです。この種類の問題は一般に「FizzBuzz」として知られています。

- 数値が3と5の両方で割り切れる場合（つまり、15で割り切れる場合）、"Fizz Buzz"と出力します。
- 数値が3で割り切れる場合（ただし、前の条件には該当しない）、"Fizz"と出力します。
- 数値が5で割り切れる場合（ただし、前の2つの条件には該当しない）、"Buzz"と出力します。
- 数値が上記のいずれの条件にも該当しない場合、その数値をそのまま出力します。

このプログラムは、プログラミングの練習問題としてよく使われます。主に条件分岐、ループ、および数値の操作に慣れることを目的としています。

具体的には、

- 3と5の両方で割り切れる数（15, 30, 45, 60, 75, 90）の場合には"Fizz Buzz"と出力されます。
- 3で割り切れるが5で割り切れない数（例：3, 6, 9, 12...）の場合には"Fizz"と出力されます。
- 5で割り切れるが3で割り切れない数（例：5, 10, 20, 25...）の場合には"Buzz"と出力されます。
- 上記のいずれにも該当しない数（例：1, 2, 4, 7...）はその数値がそのまま出力されます。

このプログラムを実行すると、1から100までの数値に対して上記のルールに従った出力がコンソールに表示されます。

以下のコードの各行にコメントを追加してもらえますか？

```
====
```

```
def calculateTotalPrice(item, quantity):
```

```
    if item == 'apple':
```

```
        return quantity * 0.5
```

```
    elif item == 'banana':
```

```
        return quantity * 0.75
```

```
    elif item == 'orange':
```

```
        return quantity * 0.6
```

```
    else:
```

```
        return 0
```

```
====
```

もちろんです。以下に、指定されたコードの各行に対してコメントを追加しました。

```
# calculateTotalPrice関数を定義します。この関数は、アイテム名と数量を引数として受け取ります。
def calculateTotalPrice(item, quantity):
    # アイテムがリンゴ('apple')の場合
    if item == 'apple':
        return quantity * 0.5 # リンゴの単価(0.5)と数量を掛け合わせた値を返します。
    # アイテムがバナナ('banana')の場合
    elif item == 'banana':
        return quantity * 0.75 # バナナの単価(0.75)と数量を掛け合わせた値を返します。
    # アイテムがオレンジ('orange')の場合
    elif item == 'orange':
        return quantity * 0.6 # オレンジの単価(0.6)と数量を掛け合わせた値を返します。
    # 上記のいずれのアイテムでもない場合
    else:
        return 0 # 0を返します。これは、未定義のアイテムには価値がないことを意味します。
```

この関数は、特定のアイテム（リンゴ、バナナ、オレンジ）の合計価格を計算するために使用されます。アイテムが指定されていれば、そのアイテムの単価に数量を掛けた値を返し、指定されたアイテムがこれらの中に入らない場合は0を返します。

以下のコードをリファクタリングしてください。

====

```
def calculateTotalPrice(item, quantity):
```

```
    if item == 'apple':
```

```
        return quantity * 0.5
```

```
    elif item == 'banana':
```

```
        return quantity * 0.75
```

```
    elif item == 'orange':
```

```
        return quantity * 0.6
```

```
    else:
```

```
        return 0
```

====

リファクタリングとはソフトウェアの挙動を変えず、その内部構造を整理することです。その効果としてコードが整理され、可読性、保守性の向上、バグの削減などがあります。

以下のコードは、指定されたアイテムと数量に基づいて合計価格を計算する`calculateTotalPrice`関数をリファクタリングしたものです。このリファクタリングでは、アイテムとその単価をマッピングする辞書を使用しています。これにより、関数をより簡潔にし、将来的にアイテムの価格を更新したり、新しいアイテムを追加したりする際の変更を容易にします。

```
def calculateTotalPrice(item, quantity):  
    # アイテムとその単価をマッピングする辞書  
    prices = {  
        'apple': 0.5,  
        'banana': 0.75,  
        'orange': 0.6  
    }  
  
    # 辞書からアイテムの単価を取得し、数量と掛け合わせる  
    # アイテムが辞書に存在しない場合は0を返す  
    return prices.get(item, 0) * quantity  
  
# 使用例  
total_price = calculateTotalPrice('apple', 3)  
print(total_price) # 出力: 1.5
```

このリファクタリングにより、将来的にアイテムを追加または価格を変更する場合、`prices`辞書の内容を更新するだけで済むようになります。また、`prices.get(item, 0)`により、指定されたアイテムが辞書に存在しない場合にはデフォルト値として0を返すようにしています。これにより、関数の最後に`else`ブロックを設ける必要がなくなり、コードがより簡潔になります。

以下のクエリの問題点を教えてください

====

```
input_data = "' OR '1'='1"
```

```
query = f"SELECT * FROM users WHERE username = '{input_data}'"
```

====

指定されたクエリには、SQLインジェクションという深刻なセキュリティ上の脆弱性があります。SQLインジェクション攻撃を受けると、攻撃者は悪意のあるSQLコードをデータベースに注入し、不正な操作を行うことができます。この攻撃はデータの盗み出し、データの変更、データベースの破壊など、多岐にわたる影響を及ぼす可能性があります。

問題の詳細を説明します：

```
input_data = '' OR '1'='1'
query = f"SELECT * FROM users WHERE username = '{input_data}'"
```

このコードでは、ユーザーからの入力（`input_data`）を直接SQLクエリに組み込んでいます。攻撃者はこの入力を利用して、常に真と評価される条件（`'1'='1'`）を注入できます。これにより、クエリが以下のように解釈されます：

```
SELECT * FROM users WHERE username = '' OR '1'='1'
```

ここで、`'1'='1'`は常に真であるため、この条件は常に満たされます。これにより、攻撃者はデータベース内の`users`テーブルからすべてのユーザー情報を取得できます。これは、プライバシーの侵害やデータの悪用につながる可能性があります。

この問題を防ぐためには、以下の対策を行う必要があります：

- プレースホルダーを使用したパラメータ化クエリの使用：SQLクエリにユーザー入力を含める際には、プレースホルダーとパラメータ化を使用して、入力を安全に処理します。
- 入力の検証とサニタイズ：ユーザーからの入力をデータベースに渡す前に、適切に検証（入力が期待される形式であることを確認）し、サニタイズ（有害な要素を取り除く）することが重要です。

これらの対策により、SQLインジェクション攻撃のリスクを大幅に減らすことができます。

このクエリに対して、SQLインジェクションの脆弱性を修正したクエリを記述してください。

SQLインジェクションの脆弱性を防ぐためには、パラメータ化されたクエリを使用することが最も効果的です。ただし、Pythonでの具体的な実装は使用しているデータベースAPIに依存します。ここでは、広く使用されているSQLite3とPyMySQL（MySQLのためのPythonライブラリ）を例にとって説明します。

SQLiteの例

```
import sqlite3

# データベースに接続
conn = sqlite3.connect('example.db')
c = conn.cursor()

# ユーザー入力
input_data = "' OR '1'='1"

# パラメータ化されたクエリ
query = "SELECT * FROM users WHERE username = ?"
c.execute(query, (input_data,))

# 結果の取得と使用
rows = c.fetchall()
for row in rows:
    print(row)

# 接続のクローズ
conn.close()
```

Pythonで1から100までを求めるコード片を書いてください

1から100までの合計を求めるPythonのコード例を以下に示します。

```
# 1から100までの合計を求める
total = 0
for i in range(1, 101):
    total += i

# 結果を出力
print(total)
```

このコードは、`range(1, 101)`を使って1から100までの数値のシーケンスを生成し、それらの数値を一つ一つ`total`に加算しています。`range`関数の第二引数は終端値であり、この値自体はシーケンスに含まれないため、`101`を指定することで1から100まで（101は含まない）の数値が生成されます。

このコードを実行すると、1から100までの合計値が出力されます。この場合、結果は5050となります。

このコードの計算をもっと効率的になるよう書き直せますか？

...

はい、1から100までの合計を求めるために、ガウスの公式を使用して計算をより効率的に行うことができます。この公式は、連続する自然数の合計を求めるための公式で、以下のように表されます。

$$\text{合計} = \frac{n \times (n + 1)}{2}$$

ここで (n) は最大の数です。この場合、(n = 100) なので、公式を使って計算します。

```
# ガウスの公式を使用して1から100までの合計を求める
n = 100
total = n * (n + 1) // 2

# 結果を出力
print(total)
```

このコードは、ループを使わずに直接計算を行うため、大幅に効率的です。ここで、整数の除算を行うために // 演算子を使用しています。これにより、浮動小数点数ではなく、整数の結果が得られます。

このコードを実行すると、以前と同様に1から100までの合計値5050が出力されますが、計算効率が大幅に向上しています。

Azure OpenAI を使用してコードを生成する利点は何ですか？

- a. 効率と生産性を向上させる
- b. バグと読みやすさの向上
- c. コーディングに費やされる時間の増加

Azure OpenAI を使用してコードを生成する利点は何ですか？

a. 効率と生産性を向上させる

b. バグと読みやすさの向上

c. コーディングに費やされる時間の増加

あるコードの続きを生成AIモデルに記述させる際、モデルにより多くの情報を提供すると、どうなりますか？

- a. より多くの情報を提供すると、モデルの正確性が低下します。
- b. より多くの情報を提供しても、モデルの正確性には影響しません。
- c. モデルに提供される情報が多いほど、応答の正確性が高くなります。

あるコードの続きを生成AIモデルに記述させる際、モデルにより多くの情報を提供すると、どうなりますか？

a. より多くの情報を提供すると、モデルの正確性が低下します。

b. より多くの情報を提供しても、モデルの正確性には影響しません。

c. モデルに提供される情報が多いほど、応答の正確性が高くなります。

以下のうち正しい記述はどれですか？

a. Azure OpenAI モデルは、コードが C# で記述されている場合にのみ、ある言語から別の言語にコードを翻訳できます。

b. ある言語で記述されたコードがあり、それを別の言語にする必要がある場合、Azure OpenAI を使用するといくつかの言語にそれを翻訳できます。

c. Azure OpenAI モデルでは、コードを Python から C# または Java にのみ翻訳できます。