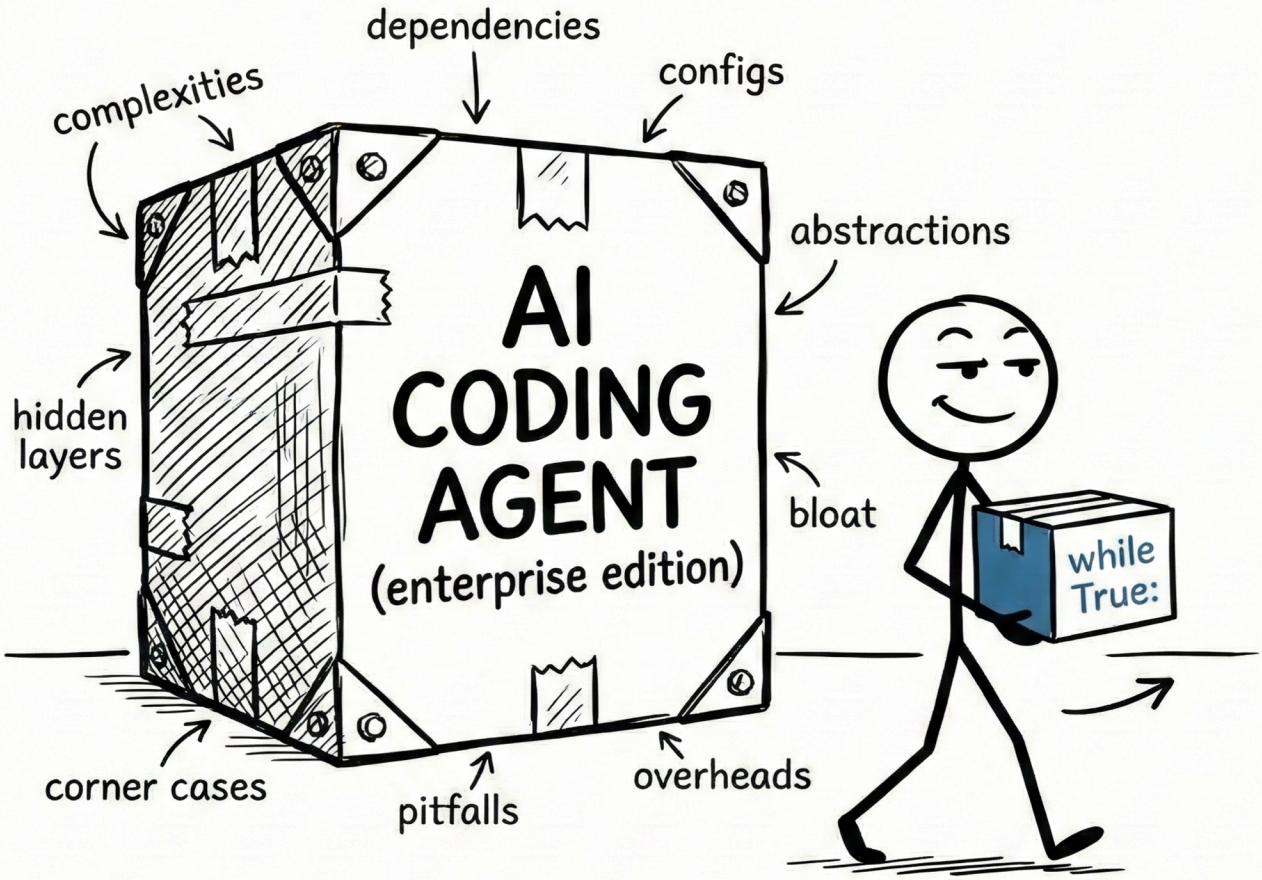
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



Owen Ou

日本語版

コーディングエージェントの作り方 (Build Your Own Coding Agent)

魔法なしで学ぶ Pure Python による AI エージェント開発ガイド

Owen Ou と TranslateAI

本書はこちらで販売中です
https://leanpub.com/build-your-own-coding-agent-ja

この版は 2026-02-18 に発行されました。

Twitter でシェアしませんか？

本書に関するコメントをTwitterでシェアして Owen Ou と TranslateAI を応援してください！

本書をシェアする際のお薦めツイートはこちらです：

HTTP リクエストとシェルだけで、一からコーディングエージェントを作ってみました。「AI の魔法」の正体は、結局 while ループと API 呼び出しだけでした。

本書のハッシュタグは #buildyourowncodingagent です。

本書に関するコメントを検索する場合は、次のリンクをクリックして下さい。Twitter のハッシュタグを使って検索できます。

#buildyourowncodingagent

妻へ、息子へ、両親へ、そして祖父母へ —— 魔法ではなく、努力あるのみということを教えてくれた皆さんへ。私が築き上げるもの全ては、皆さんのお陰です。

Contents

はじめに

対象読者

あなたは AI の誇大宣伝に懐疑的なソフトウェアエンジニアです。

デモを見てきました。フレームワークも試してきました。LangChain アプリが的外れな応答を繰り返し、本番環境のデータベースを削除してしまうのも目にしてきました。そして考えます：*「もっと良い方法があるはずだ」* と。

実際にあります。この本は、AI エージェントが実行されるときに実際に何が起きているのかを理解したい開発者のために書かれています。マーケティング用の図解ではありません。「推論エンジン」という抽象的な概念でもありません。実際の HTTP リクエストです。実際の`while` ループです。

Python が読めて、ウェブアプリや CLI ツールを作ったことがあれば、必要な知識は十分です。

作成するもの

Nanocode は、ターミナルで動作するコーディングエージェントです。この本を読み終える頃には、以下のことができるようになります：

- コードベース内のファイルの読み書き
- シェルコマンドの実行
- ピュア Python を使用したコード検索
- セッション間でのコンテキストの保持
- 危険な操作の前に許可を求める
- ドキュメントや回答をウェブで検索する

これを`requests`、`subprocess`、`pytest` だけを使って一から構築します。LangChain は使いません。ベクトルデータベースも使いません。「オーケストレーションフレームワーク」も使いません。`print()` でデバッグできる Python だけです。

唯一の例外：第 11 章でウェブ検索用に`ddgs` を追加します—軽量な依存関係を 1 つだけ使用します。

テストアプローチ

本書では「テストと並行」アプローチを採用しています。各機能について:

1. コンセプトを紹介します—なぜこれが必要なのか
2. まずテストを示し、成功の形を理解します
3. テストをパスするコードを実装します
4. pytest で検証します

これにより、印刷物での完全な red-green-refactor のセレモニーなしに**テスト駆動開発**の考え方を学べます。また、重要な問題も解決します:LLM を実際に呼び出して LLM 駆動のアプリケーションをテストすることはできません。API 呼び出しは遅く、高価で、非決定的だからです。

第 3 章では、`FakeBrain` パターンを導入します—予測可能なレスポンスを返すテストダブルです。これにより、1 回の API 呼び出しも行わず、1 セントも使わずに、テストスイート全体を実行できます。

コード例

本書は「コードファースト」アプローチに従います。各章は前の章の内容を基に構築され、すべてのコード例は動作するファイルから抽出されています。

コードの入手方法:

- **GitHub**: GitHub からクローンまたはダウンロード[1]
- **Leanpub**: 購入時のダウンロード可能なリソースに完全なソースコードが含まれています

コードは章ごとに整理されています(`ch01/`、`ch02/`など)。各フォルダには以下が含まれます:

- `nanocode.py` — その章の完全な実行可能なエージェント
- `test_nanocode.py` — API 呼び出しなしでコードが動作することを検証するテスト

任意の章のフォルダをコピーしてそこから始めることができます。`pytest` を実行して、コードが期待される動作と一致することを確認できます。

[1] https://github.com/owenthereal/build-your-own-coding-agent

本書で使用される表記規則

本書全体を通して、3 種類の注釈が登場します：

> **開発のヒント：**このプロジェクトを超えて適用できるアーキテクチャの知恵です。これらは自分の仕事で再利用できるパターンです。

警告：セキュリティまたは安全性のリスクです。これらに注意を払ってください—無視するとファイルが削除されたり API キーが漏洩したりする可能性があります。

補足：深掘りと余談です。有用な背景情報ですが、初回の読書では本筋を失わずにスキップできます。

コードの解説は一貫したパターンに従います：まずコンテキスト（なぜこのコードが必要か）、次にコード自体、そして重要な部分の行ごとの説明です。

それでは、コードを書き始めましょう。

第 I 部：頭脳

これらの最初の章では、エージェントの骨格を構築していきます：生の HTTP を介して Claude と対話する`while` ループです。第 4 章までには、Adapter パターンを通じて複数の LLM プロバイダーをサポートするようになり、「AI エージェント」が単なるループと頭脳、そしてメッセージのリストに過ぎないことを理解することになります。

第 1 章：ゼロマジック・マニフェスト

過去 2 年間に AI アプリケーションの構築を試みた方なら、「フレームワーク疲れ」を感じたことがあるでしょう。

人気のライブラリをインストールします。`ReasoningEngine` をインポートします。`.run()` を呼び出します。「Hello World」の例では魔法のように動作します。しかし、実際の作業—たとえば Python ファイルのインポート文を残したまま特定の行を編集するような作業—を試みた途端、動作しなくなってしまいます。

そして、フレームワークを使用したがために、修正することができません。抽象クラス、ファクトリーパターン、そして「チェーン」の層を掘り進めながら、幻覚を引き起こしている 1 つのプロンプトを見つけようと奮闘することになります。

ここではそのようなアプローチは取りません。

この本は「マジック」への反逆です。私たちは「ゼロマジック」アプローチを取ります：純粋な Python で本番環境対応のコーディングエージェント「**Nanocode**」を構築します。LangChain も、AutoGPT も、Pydantic も使いません。

なぜでしょうか？自律エージェントは魔法ではないからです。それは単なる`while` ループに過ぎません。

エージェントとは実際には何か？

ベンチャーキャピタルのマーケティングを取り除けば、「エージェント」は単なる**サーモスタット**です。

サーモスタットは温度を読み取り（入力）、目標値と比較し（判断）、ヒーターをオンにします（アクション）。そして待機し、これを繰り返します。それだけです。AI エージェントも同じことを行います。ただし温度の代わりにテキストを使用するだけです。

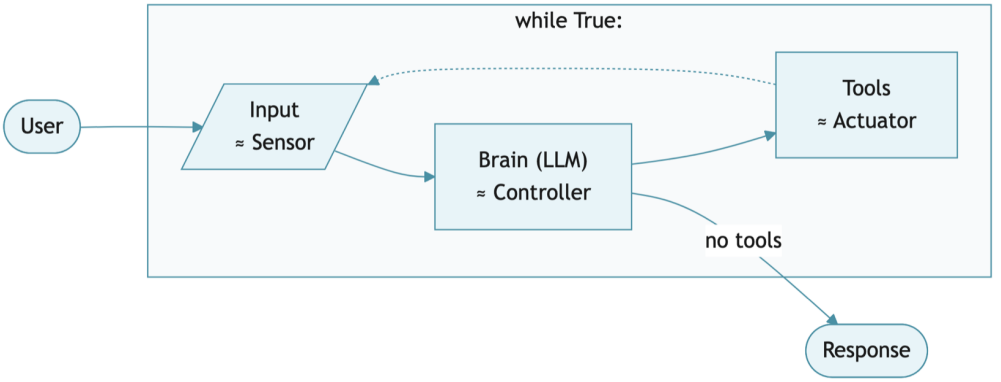


図 1. エージェントループ：ユーザー入力は、入力（センサー）が頭脳/LLM（コントローラー）に流れ、ツール（アクチュエーター）を起動し、頭脳が応答を出力するまで入力に戻る while ループを通過する

より具体的には、エージェントには 4 つの部分があります。**頭脳**は LLM—テキストを送信すると、テキストを返す、ステートレスな関数です。外部世界とやり取りするために、ファイルの読み取りやコマンドの実行などの**ツール**を呼び出します。これらすべてが、タスクが完了するまで繰り返し実行される**ループ**（`while True`）の中に存在し、**メモリ**—単なる Python のリスト—が会話履歴を蓄積していきます。（このリストはプログラムが終了すると消えます。永続的なストレージは第 6 章で追加します。）

`while` ループが書けるなら、エージェントは構築できます。

ゼロから構築することで、フレームワークユーザーには持ち得ないものを手に入れることができます：制御です。エージェントがループに陥ったとき、どの行のコードが原因なのかを正確に把握できます。API の請求額が高額になったとき、どこでトークンが漏れているのかを正確に見ることができます。

何を構築するのか

Nanocode はターミナルで動作する CLI ツールです。同僚のように会話することができます。ファイルを読み、コマンドを実行し、コードを編集します。

この本を終えるころには、Claude Sonnet 4.6（または DeepSeek、あるいは Ollama を介したローカルモデル）に接続し終えているでしょう。ファイルの読み書きやシェルコマンドの実行を行う手足となるツールと、コードベースを検索する目を与えます。そして、誤って`rm -rf /`を実行しないように安全装置を構築します。

プロジェクトのセットアップ

> **開発のヒント：** AI プロジェクトにとって最大の脅威は「依存関係の腐敗」です。AI ライブラリは急速に進化し、破壊的変更を加えます。`requests` と`subprocess` に限定することで、このエージェントは 5 年後でもおそらく動作するでしょう。

1. プロジェクトの初期化

```
mkdir nanocode
cd nanocode
git init
```

2. 仮想環境を作成する

AI ツールをグローバルにインストールしないでください。システムパッケージと競合します。

```
# Mac/Linux
python3 -m venv venv
source venv/bin/activate

# Windows
python -m venv venv
venv\Scripts\activate
```

3. 依存ライブラリのインストール

必要なライブラリは 3 つだけです：

- `requests` — LLM API と通信するため
- `python-dotenv` — `.env` ファイルから API キーを読み込むため
- `pytest` — API を呼び出さずにコードをテストするため

`requirements.txt` を作成します：

```
requests
python-dotenv
pytest
```

インストール:

```
pip install -r requirements.txt
```

4. キーを保護する

警告: API キーを GitHub にプッシュすると、ボットがそれを収集し、数分以内にあなたのアカウントの残高を搾取してしまいます。

`.gitignore` を作成します:

```
.env
__pycache__/
venv/
.DS_Store
.nanocode/
```

AgentStop 例外

イベントループを書く前に、クリーンな終了機構が必要です。コード全体に散らばった`break` 文よりも、例外の方が適切です。

コンテキスト: 例外はエラー処理だけでなく、制御フロー機構としても機能します。ユーザーが`/q` と入力した時、`AgentStop` を発生させます。メインループがこれをキャッチして、クリーンに終了します。

コード:

```python
# --- Exceptions ---

class AgentStop(Exception):
    """Raised when the agent should stop processing."""
    pass
```

これはnanocode.py の先頭に配置します。これはマーカー例外で、ロジックは含まず、単なる合図として機能します。

Agent クラス

ここで中心となる抽象化について説明します:Agent クラスです。状態とロジックを 1 つの場所にまとめることで、テストが容易になります。

背景:すべてのロジックをmain() に配置することも**できます**。しかし、その場合、テストのためにinput() とprint() をモック化する必要が出てきます。ロジックをAgent.handle_input() として抽出することで、直接テストすることが可能になります。

コード:

```python
class Agent:
    """A coding agent that processes user input."""

    def __init__(self):
        pass

    def handle_input(self, user_input):
        """Handle user input. Returns output string, raises AgentStop to
        ↪  quit."""
        if user_input.strip() == "/q":
            raise AgentStop()

        if not user_input.strip():
            return ""

        return f"You said: {user_input}\n(Agent not yet connected)"
```

ウォークスルー:

- **13-14 行目：**現時点では空のコンストラクタです。後の章で`brain` と`tools` を追加します。
- **18-19 行目：**終了コマンド`/q` をチェックします。特別な値を返す代わりに`AgentStop` を発生させます。
- **21-22 行目：**空の入力をスキップします。空文字列を返します（表示する出力なし）。
- **24 行目：**入力をそのままエコーバックします。これはプレースホルダーです—後で、これを Brain に送信します。

テストによる成功の定義

メインループを書く前に、テストが必要です。

`test_nanocode.py` を作成します：

```
import pytest
from nanocode import Agent, AgentStop


def test_handle_input_returns_string():
    """Verify handle_input returns a string for normal input."""
    agent = Agent()
    result = agent.handle_input("hello")
    assert isinstance(result, str)
    assert "hello" in result


def test_empty_input_returns_empty_string():
    """Verify empty/whitespace input returns empty string."""
    agent = Agent()
    assert agent.handle_input("") == ""
    assert agent.handle_input("   ") == ""
    assert agent.handle_input("\n") == ""


def test_quit_command_raises_agent_stop():
    """Verify /q raises AgentStop exception."""
    agent = Agent()
```

```
    with pytest.raises(AgentStop):
        agent.handle_input("/q")


def test_quit_command_with_whitespace():
    """Verify /q works with surrounding whitespace."""
    agent = Agent()
    with pytest.raises(AgentStop):
        agent.handle_input("  /q  ")
```

テストを実行してください:

```
pytest test_nanocode.py -v
```

```
test_nanocode.py::test_handle_input_returns_string PASSED
test_nanocode.py::test_empty_input_returns_empty_string PASSED
test_nanocode.py::test_quit_command_raises_agent_stop PASSED
test_nanocode.py::test_quit_command_with_whitespace PASSED
```

すべて問題ありません。エージェントは基本的なケースを正しく処理しています。

補足:なぜ pytest なのでしょうか?pytest はtest_で始まる関数を見つけて実行します。定型コードも、クラスも必要ありません。テストコード自体は普通の Python です—特別な仕組みはありません。

メインループ

次は、エージェントをターミナルに接続する薄い I/O ラッパーです:

```
29  def main():
30      agent = Agent()
31      print("⚡ Nanocode v0.1 initialized.")
32      print("Type '/q' to quit.")
33
34      while True:
35          try:
36              user_input = input("\n❯ ")
37              output = agent.handle_input(user_input)
38              if output:
39                  print(output)
40
41          except (AgentStop, KeyboardInterrupt):
42              print("\nExiting...")
43              break
44
45
46  if __name__ == "__main__":
47      main()
```

ウォークスルー:

- **30-32 行目:**エージェントを作成し、起動メッセージを表示します。
- **36 行目:** `input()` はユーザーが何かを入力するまでブロックして待機します。
- **37-39 行目:** `handle_input()` を呼び出し、出力を表示します。
- **41-43 行目:** `/q` による`AgentStop` または Ctrl+C による`KeyboardInterrupt` をキャッチして、正常に終了します。

補足: Python の`input()` は一度に 1 行ずつ読み込みます。本書のプロンプトはすべて 1 行です。これによりコードがシンプルに保たれます—本番環境のエージェントでは readline や完全な TUI など、より高度な入力方式を使用します。

注目すべき点として:すべてのロジックは`Agent.handle_input()` に含まれています。`main()` は単なる I/O 用の接続コードです。これにより、標準入出力のモック化なしでエージェントのテストが可能になります。

実行してみましょう

```
python nanocode.py
```

表示される内容:

```
⚡ Nanocode v0.1 initialized.
Type '/q' to quit.

 hello
You said: hello
(Agent not yet connected)

 /q

Exiting...
```

これがシャーシです。次はエンジンです。

まとめ

以上がシャーシです:`Agent` クラス、`handle_input()` メソッド、そして`while True` ループです。まだ実用的な機能は備えていませんが、これから構築するものはすべてこの骨格に組み込まれていきます。テストによって、開発を進めながら既存の機能を壊さないようにすることができます。

第 2 章：生のリクエスト

ほとんどのチュートリアルでは `pip install anthropic` を実行するように指示します。しかし、私たちはそうしません。

SDK は真実を隠します。抽象化層を追加することで「Hello World」は簡単になりますが、「Error 400」のデバッグは悪夢となります。SDK を学ぶとき、あなたは SDK を学んでいるだけです。生の HTTP リクエストを学ぶとき、あなたはすべての SDK の下にあるプロトコルを学びます。

私たちは `requests` ライブラリだけを使って Claude にメッセージを送信します。Claude は Anthropic の主力 LLM で、コーディングタスクに関して最も優れたモデルの 1 つです。

API キーの取得

Claude と対話するには、API キーが必要です。これはあなたのクレジットカードのように機能する長い文字列です。

1. Anthropic Console にアクセスします。[1]
2. サインアップして支払い方法を追加します（最低 $5 のクレジット）。
3. 新しい API キーを作成し、`nanocode` という名前を付けます。
4. キーをコピーします（`sk-ant-...` で始まります）。

警告：このキーをパスワードのように扱ってください。このキーを持っている人は誰でもあなたのお金を使うことができます。

保管庫（.env）

このキーを安全に保管する場所が必要です。キーを直接コードに記述することは決してありません。

プロジェクトのルートディレクトリに `.env` という名前のファイルを作成します：

[1] https://console.anthropic.com/settings/keys

```
touch .env
```

開いてキーを貼り付けてください:

```
ANTHROPIC_API_KEY=sk-ant-api03-...
```

第 1 章で python-dotenv をインストールしましたが、これはまさにこの目的のためです—.env ファイルを読み込み、その値を os.environ にロードします。

リクエストの構造

LLM と対話するために、以下の URL に HTTP POST リクエストを送信します:

```
https://api.anthropic.com/v1/messages
```

このリクエストには 3 つの要素が必要です: ヘッダーでの認証(API キー)、ボディでの設定(使用するモデル、トークン数)、そしてメッセージ本体です。

ヘッダー

Anthropic は 3 つのヘッダーを必要とします:

ヘッダー	値	目的
x-api-key	シークレットキー	認証
anthropic-version	2023-06-01	API バージョン
content-type	application/json	フォーマット

ペイロード

「Messages API」はメッセージ辞書のリストを想定しています:

```
"messages": [
    {"role": "user", "content": "Hello, world!"}
]
```

各メッセージにはrole("user" または"assistant" のいずれか)とcontent(テキスト)があります。

コード

test_api.py というファイルを作成します。これは接続が機能することを証明する「スモークテスト」です。このファイルは後で削除します。

コンテキスト:線形の手続き型コードを書いています。関数もクラスも使いません。最も基本的なレベルのコードを確認したいのです。

コード:

```python
import os
import requests
import json
from dotenv import load_dotenv

# 1. Load the vault
load_dotenv()
api_key = os.getenv("ANTHROPIC_API_KEY")

# Basic check so we don't crash with a confusing "NoneType" error later
if not api_key:
    print("Error: ANTHROPIC_API_KEY not found in .env")
    exit(1)

# 2. Define the target
url = "https://api.anthropic.com/v1/messages"

# 3. Authenticate
headers = {
    "x-api-key": api_key,
    "anthropic-version": "2023-06-01",
```

```
    "content-type": "application/json"
}

# 4. Construct the payload
payload = {
    "model": "claude-sonnet-4-6",
    "max_tokens": 4096,
    "messages": [
        {"role": "user", "content": "Hello, are you ready to code?"}
    ]
}

# 5. Fire! (No safety net)
print("📡 Sending request to Claude...")
response = requests.post(url, headers=headers, json=payload, timeout=120)

# 6. Inspect the raw result
print(f"Status: {response.status_code}")

if response.status_code == 200:
    # Success: Print the beautiful JSON
    print("Response:")
    print(json.dumps(response.json(), indent=2))
else:
    # Failure: Print the ugly raw text so we can debug
    print("Error:", response.text)
```

コードの詳細解説:

- **7 行目:** `load_dotenv()` は `.env` ファイルを見つけて変数を `os.environ` に読み込みます。
- **8 行目:** API キーを取得します。これは絶対にハードコーディングしてはいけません。
- **11-13 行目:** 基本的な健全性チェックです。これがないと、キーが存在しない場合にヘッダー辞書で分かりにくい `NoneType` エラーが発生します。
- **21 行目:** `anthropic-version` ヘッダーは必須です。これを省略すると API に拒否されます。
- **27 行目:** `claude-sonnet-4-6` は使用したいモデルを指定します。

- **28 行目:** `max_tokens` は必須パラメータです。応答の長さを制限し、予期せぬコストの発生を防ぎます。
- **36 行目:** 2 分のタイムアウトを設定してリクエストを送信します。try/except は使用していません—ネットワークがダウンしている場合は、Python をクラッシュさせて、どこで失敗したのかを確認する必要があります。
- **41-47 行目:** ステータスコードを確認します。200 は成功を意味し（JSON を整形出力します）。それ以外の場合は、デバッグ用に生のエラーテキストを出力します。

実行方法

```
python test_api.py
```

すべてが正常に動作する場合、以下が表示されるはずです:

```
Status: 200
Response:
{
  "id": "msg_01...",
  "type": "message",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "Hello! Yes, I'm ready to code..."
    }
  ],
  "stop_reason": "end_turn",
  "usage": {
    "input_tokens": 15,
    "output_tokens": 81
  }
}
```

トラブルシューティング

エラー	原因	解決方法
401 未認証	API キーが無効	`.env` が読み込まれているか確認。`os.environ.get("ANTHROPIC_API_KEY")` を出力して検証。
400 不正なリクエスト	JSON の形式が不正	`max_tokens` の指定を忘れていませんか? `messages` はリスト形式ですか?
429 レート制限	リクエストが多すぎる、またはクレジット不足	待機するか、アカウントにクレジットを追加してください。

クリーンアップ

AI と対話できることが確認できました。`test_api.py` は不要なので削除しましょう。

補足: この`test_api.py` は使い捨てコード(一度きりのスモークテスト)です。本格的な自動テスト(`FakeBrain` と pytest を使用)は第 3 章で扱います。API の接続確認が済んだら、このファイルは必ず削除してください。

開発者向けヒント: API の呼び出しには料金が発生します。入力トークン(送信するもの)は安価です。出力トークン(モデルが生成するもの)は約 5 倍高価です。プロンプトは簡潔に保ちましょう。

補足: 支出を監視するには、Anthropic Console の Usage タブを確認してください。2026 年初頭の時点では、Claude Sonnet を使用した 20-30 回のやり取りを含む一般的なコーディングセッションのコストは 0.10-0.50 ドルです。レスポンス JSON の末尾にある`usage` フィールドには正確なトークン数が表示されます—プログラムでコストを追跡する場合にこれらを記録できます。

まとめ

これが生の API 呼び出しです：ヘッダー、JSON ペイロード、レスポンスの解析。あなたと低レベル通信の間に抽象化は存在しません。何か問題が発生したとき（そして必ず発生します）、レイヤーが 1 つしかないため、どの層で失敗したのかが正確に分かります。

1 つ問題があります：Claude は完全な記憶喪失状態です。リクエストのたびに白紙の状態から始まります。これに対処するため、毎回の対話で会話履歴全体を再送信することで疑似的な記憶を実現します。

第 3 章：無限ループ

問題があります。

第 2 章のスクリプトを 2 回実行してみましょう。「私の名前はアリスです」と言うと、Claude は挨拶を返します。もう一度実行して「私の名前は何ですか?」と尋ねると、Claude は「わかりません」と答えます。

これは、LLM が**ステートレス**だからです。完全な記憶喪失状態なのです。すべてのリクエストが、まるで初めて会ったかのようになります。

エージェントを構築するには、人工的な記憶を作成することでこの問題を解決する必要があります。

記憶の幻想

LLM における「記憶」は、ハードドライブではありません。ログファイルなのです。

ChatGPT とチャットする時、5 分前に何を言ったかを「記憶」しているわけではありません。舞台裏では、新しいメッセージが送られるたびに、コードが**会話履歴全体**をモデルに送り返しているのです。

図 2. コンテキストの蓄積：ターン 1 では「User: Hi」だけが API に送信されます。ターン 2 では履歴全体—「User: Hi」、「Assistant: Hello」、「User: How are you?」—が API に送信されます。

モデルは毎回、完全な会話記録を見ているのです。それがトリックの正体です。

このような**コンテキストループ**を手動で実装していきます。しかし、その前にコードをテスト可能にする必要があります。

テストの課題

ここで厳しい事実を述べましょう：LLM を使用したアプリケーションを、実際に LLM を呼び出してテストすることはできません。

API 呼び出しは遅く（2-10 秒かかる）、高価で（1 回ごとに実際のお金がかかる）、非決定的（毎回異なる応答が得られる可能性がある）です。$5 かかり 20 分もかかるテストスイートを想像してみてください。絶対に実行したくないでしょう。

解決策は**依存性の注入**です。エージェント内に API 呼び出しをハードコーディングする代わりに、「brain」オブジェクトを渡します。本番環境では、この brain は Claude です。テストでは、予測可能な応答を返す偽物の brain を使用します。

これから本番コードを書く前に、このパターンを確立していきます。

レスポンスの型

brain を構築する前に、その戻り値を定義する必要があります。Claude の API は複数のコンテンツブロックを含む複雑な JSON を返します。私たちは扱いやすい Python オブジェクトが必要です。

コンテキスト： Claude は 1 つの応答でテキスト、ツール呼び出し、またはその両方を返すことができます。これらの可能性を表現するためのデータクラスが必要です。

コード：

```
17  class ToolCall:
18      """A tool invocation request from the brain."""
19
20      def __init__(self, id, name, args):
21          self.id = id
22          self.name = name
23          self.args = args  # dict
```

`ToolCall` は AI 実行環境がツールの実行を要求する際に使用されます。`id` は追跡用の一意の識別子です（Claude が結果を報告する際に必要とします）。`name` は実行するツールを指定し、`args` はパラメータを含む辞書です。

現時点では AI 実行環境がツールを呼び出すことができないため、`ToolCall` はまだ使用しませんが、これが`Thought` レスポンスタイプの一部となるため、ここで定義しています。ツールを追加すると、Claude がファイルの読み込みやコマンドの実行を要求する際にこれを返すようになります。

```
class Thought:
    """Standardized response from any Brain."""

    def __init__(self, text=None, tool_calls=None):
        self.text = text  # str or None
        self.tool_calls = tool_calls or []  # list of ToolCall
```

Thought は、思考処理後にブレインが返すものです。テキスト、ツールコール、その両方、またはどちらも含まない場合があります。この抽象化により、他のコードを変更することなく、後で Claude を DeepSeek に置き換えることが可能になります。

フェイクブレインパターン

これでテスト用のフェイクブレインを構築できます。

コンテキスト: 予測可能なレスポンスを返し、呼び出された回数を追跡し、受け取った会話を記録するブレインが必要です。

コード:

```
class FakeBrain:
    """Fake brain for testing - returns predictable responses."""

    def __init__(self, responses=None):
        self.responses = responses or [Thought(text="Fake response")]
        self.call_count = 0
        self.last_conversation = None

    def think(self, conversation):
        self.last_conversation = list(conversation)  # Store a copy
        if self.call_count < len(self.responses):
            response = self.responses[self.call_count]
            self.call_count += 1
            return response
        return Thought(text="No more responses")
```

これは本番コードではなく、test_nanocode.py に記述します。FakeBrain は実際のブレインと同じインターフェース、つまり会話を受け取ってThought を返すthink() メソッドを持っていることに注目してください。

補足：このパターン（テスト時に実際の依存関係を予測可能なフェイクで置き換えること）は、デペンデンシーインジェクションと呼ばれています。Martin Fowler の記事「Mocks Aren't Stubs」[1]では、その種類（フェイク、スタブ、モック、スパイ）について説明しています。LLM のテストでは、通常、事前に用意された応答を持つシンプルなフェイクで十分です。

成功の定義

本番コードを書く前に、成功がどのような形になるのかを定義しましょう。これらのテストが実装の指針となります。

テスト 1：ブレインが応答を返す

```
def test_handle_input_returns_brain_response():
    """Verify handle_input returns the brain's response text."""
    brain = FakeBrain(responses=[Thought(text="Hello from brain!")])
    agent = Agent(brain=brain)
    result = agent.handle_input("hi")
    assert result == "Hello from brain!"
```

`brain=brain` を Agent に渡していることに注目してください。これは依存性の注入が実際に動作している例です。

テスト 2：会話の蓄積

```
def test_conversation_accumulates():
    """Verify conversation list grows with each interaction."""
    brain = FakeBrain(responses=[
        Thought(text="Response 1"),
        Thought(text="Response 2")
    ])
    agent = Agent(brain=brain)

    agent.handle_input("First message")
    assert len(agent.conversation) == 2  # user + assistant
```

[1] https://martinfowler.com/articles/mocksArentStubs.html

```
11
12      agent.handle_input("Second message")
13      assert len(agent.conversation) == 4  # 2 users + 2 assistants
```

各やり取りの後、会話にはユーザーメッセージとアシスタントの応答の両方が含まれているべきです。

テスト 3:正しいメッセージ構造

```
1   def test_conversation_contains_correct_roles():
2       """Verify conversation has correct role alternation."""
3       brain = FakeBrain(responses=[Thought(text="AI response")])
4       agent = Agent(brain=brain)
5
6       agent.handle_input("User message")
7
8       assert agent.conversation[0]["role"] == "user"
9       assert agent.conversation[0]["content"] == "User message"
10      assert agent.conversation[1]["role"] == "assistant"
11      assert agent.conversation[1]["content"] == "AI response"
```

メッセージは、Claude が期待する正確なフォーマットでなければなりません:`{"role": "user", "content": "..."}`。

テスト 4:Brain が会話を受信する

```
1   def test_brain_receives_conversation():
2       """Verify brain.think is called with the conversation list."""
3       brain = FakeBrain()
4       agent = Agent(brain=brain)
5
6       agent.handle_input("Test message")
7
8       assert brain.last_conversation is not None
9       assert len(brain.last_conversation) == 1
10      assert brain.last_conversation[0]["content"] == "Test message"
```

脳は現在のメッセージだけでなく、会話全体を受け取る必要があります。

これらのテストを今すぐ実行してください—すべて失敗するはずです:

```
pytest test_nanocode.py -v
```

```
FAILED test_nanocode.py::test_handle_input_returns_brain_response
FAILED test_nanocode.py::test_conversation_accumulates
...
```

良いでしょう。では、これらをパスさせていきましょう。

Claude クラス

ここからが本当の中核部分です。

コンテキスト: Claude API をラップするクラスが必要です。このクラスは認証を処理し、会話履歴を送信し、レスポンスをThought に解析する必要があります。

コード:

```
class Claude:
    """Claude API - the brain of our agent."""

    def __init__(self):
        self.api_key = os.getenv("ANTHROPIC_API_KEY")
        if not self.api_key:
            raise ValueError("ANTHROPIC_API_KEY not found in .env")
        self.model = "claude-sonnet-4-6"
        self.url = "https://api.anthropic.com/v1/messages"

    def think(self, conversation):
        headers = {
            "x-api-key": self.api_key,
            "anthropic-version": "2023-06-01",
            "content-type": "application/json"
        }
```

```
52          payload = {
53              "model": self.model,
54              "max_tokens": 4096,
55              "messages": conversation
56          }
57
58          print("(Claude is thinking...)")
59          response = requests.post(self.url, headers=headers, json=payload,
            ↪  timeout=120)
60          response.raise_for_status()
61          return self._parse_response(response.json()["content"])
```

コードの解説:

- **39-41 行目:** API キーをロードし、存在しない場合は即時終了します。
- **43-44 行目:**設定を保存します。後でモデルを設定可能にします。
- **46 行目:** `think()` メソッドは、`FakeBrain` と同様のブレインのインターフェースです。
- **52-55 行目:**ペイロードには`"messages": conversation` が含まれています—現在のメッセージだけでなく、完全な会話履歴です。これがコンテキストループです。
- **61 行目:** Claude の複雑なレスポンス形式を、シンプルな`Thought` に解析します。

次はレスポンスパーサーです:

```
63      def _parse_response(self, content):
64          """Convert Claude's response format to Thought."""
65          text_parts = []
66          tool_calls = []
67
68          for block in content:
69              if block["type"] == "text":
70                  text_parts.append(block["text"])
71              elif block["type"] == "tool_use":
72                  tool_calls.append(ToolCall(
73                      id=block["id"],
74                      name=block["name"],
75                      args=block["input"]
76                  ))
```

```
77
78          return Thought(
79              text="\n".join(text_parts) if text_parts else None,
80              tool_calls=tool_calls
81          )
```

Claude の API は「コンテンツブロック」のリストを返します。各ブロックにはtype があり、"text" か"tool_use" のいずれかです。すべてのテキストブロックを 1 つの文字列にまとめ、tool_use ブロックをToolCall オブジェクトに変換します。

Agent クラス（更新版）

ここでは、第 1 章の Agent を更新し、ブレインを受け入れ、会話履歴を保持できるようにします。

コード:

```
86  class Agent:
87      """A coding agent with conversation memory."""
88
89      def __init__(self, brain):
90          self.brain = brain
91          self.conversation = []
92
93      def handle_input(self, user_input):
94          """Handle user input. Returns output string, raises AgentStop to
            ↪  quit."""
95          if user_input.strip() == "/q":
96              raise AgentStop()
97
98          if not user_input.strip():
99              return ""
100
101         self.conversation.append({"role": "user", "content": user_input})
102
103         try:
104             thought = self.brain.think(self.conversation)
```

```
105             text = thought.text or ""
106             self.conversation.append({"role": "assistant", "content": text})
107             return text
108         except Exception as e:
109             self.conversation.pop()  # Remove failed user message
110             return f"Error: {e}"
```

ウォークスルー:

- **89-91 行目:**依存性注入によって brain を受け取ります。空の会話リストを初期化します。
- **101 行目:** brain を呼び出す**前に**ユーザーのメッセージを履歴に追加します。
- **103-107 行目:** brain を呼び出し、テキストを抽出し、応答を履歴に追加します。
- **108-110 行目:** API コールが失敗した場合、追加したばかりのユーザーメッセージを削除します。これにより会話が有効な状態に保たれます。

101 行目に注目してください:brain を呼び出す**前に**ユーザーメッセージを追加します。brain は現在のメッセージを含む会話全体を見る必要があるからです。

> **開発者向けヒント:** 問題が発生した場合、最初のデバッグツールは`print(self.conversation)`です。これにより brain が実際に見ている内容を正確に確認できます。不正な形式のメッセージ、欠落しているロール、切り詰められたコンテンツは、生のリストを調査すると一目瞭然です。

メインループ（更新版）

メインループは現在、単なる薄い I/O ラッパーとなっています:

```
def main():
    brain = Claude()
    agent = Agent(brain)
    print("⚡ Nanocode v0.2 (Conversation Memory)")
    print("Type '/q' to quit.\n")

    while True:
        try:
            user_input = input("  ")
            output = agent.handle_input(user_input)
            if output:
                print(f"\n{output}\n")

        except (AgentStop, KeyboardInterrupt):
            print("\nExiting...")
            break


if __name__ == "__main__":
    main()
```

すべてのロジックは Agent クラスの中にあります。ループは入力を読み取り、handle_input() を呼び出し、結果を出力するだけです。この分離によってエージェントがテスト可能になります—input() や print() をモック化する必要なく、Agent.handle_input() を直接テストできます。

テストが通ることを確認

テストを再度実行してください:

```
pytest test_nanocode.py -v
```

```
test_nanocode.py::test_handle_input_returns_brain_response PASSED
test_nanocode.py::test_conversation_accumulates PASSED
test_nanocode.py::test_conversation_contains_correct_roles PASSED
test_nanocode.py::test_brain_receives_conversation PASSED
```

すべてグリーンです。テストは 1 回の API コールも行うことなく、私たちの実装を検証します。

メモリのテスト

では、実際の本体でテストを行いましょう:

```
python nanocode.py
```

この会話をお試しください:

```
 I am building a Python agent.
(Claude is thinking...)

That sounds exciting! Building a Python agent is a great project...

 What language am I using?
(Claude is thinking...)

You are using Python.
```

会話リストは期待通りに機能しています。

コンテキストウィンドウの問題

「これを永遠に実行し続けることはできるの?」と考えているかもしれません。

いいえ。

ループが 1 回実行されるたびに、messages リストは大きくなっていきます:

ターン	おおよそのトークン数
1	50
10	5,000
100	50,000

最終的に、**コンテキスト制限**に到達します—Claude Sonnet では 200k トークン、DeepSeek では 128k トークン、一部のローカルモデルでは 4k トークンと低くなることもあります。この制限を超えると、API は`400 Bad Request` を返します。108 行目のエラー処理がこれを捕捉してエラーを報告するため、エージェントが静かにクラッシュすることはありません。しかし、会話は実質的に行き詰まります—履歴が長すぎる状態が続くため、その後のメッセージもすべて失敗します。

現時点では、エージェントを再起動することで履歴がクリアされ、再び動作するようになります。第 9 章でフィードバックループを構築する際に、適切な**コンテキスト圧縮**—API レスポンスからトークン使用量を追跡し、オーバーフローする前に古いメッセージを自動的に要約する機能—を追加します。そこで実際に会話が膨らみ、その解決策が真価を発揮することになります。

> **開発者向けヒント：** `print("(Claude is thinking...)")` に注目してください。ネットワークコールには 2〜10 秒かかります。Enter キーが機能したことを即座にフィードバックしないと、ユーザーはフリーズしたと思って Ctrl+C を押してしまいます。

まとめ

Claude は今や記憶を持っています—より正確には、記憶があるように見せかけることができました。会話リストは各ターンで成長し、`FakeBrain` を使えば 1 セントも使わずにすべてをテストできます。

これらのパターンは本書全体を通して継続します。私たちが構築するすべてのブレイン（Claude、DeepSeek、Ollama）は同じ`think()` インターフェースを実装し、`FakeBrain` がそれらすべてをテストします。

未解決の問題が 1 つあります：現在のコードは Anthropic の API に強く結びついています。DeepSeek やローカルモデルを追加しようとすると、多くのコードを複製しなければならないでしょう。

第 4 章：ユニバーサルアダプター

アダプターパターン

HTTP レジリエンス

Brain インターフェース

FakeBrain（更新版）

Claude ブレイン（リファクタリング後）

DeepSeek Brain

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

BRAINS レジストリ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

エージェントクラス（更新版）

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

マルチブレインサポートのテスト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

メインループ（更新済み）

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

DeepSeek のセットアップ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

試してみましょう

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

「コードを移動しただけ」

まとめ

パート II：実行能力

第 5 章：ツールプロトコル

ツールの実際の仕組み

ツールインターフェースの定義

ReadFile ツール

ファイル書き込みツール

ツールヘルパー

Thought クラスの更新

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

Claude クラスの更新

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

ツールを備えた Agent クラス

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

メインループ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

テストする

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

まとめ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

第 6 章：スクラッチパッド（メモリー）

「ゼロマジック」メモリー

メモリークラス

ToolContext クラス

SaveMemory ツール

Claude クラスの更新

システムプロンプトの作成

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

エージェントクラスの更新

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

メインループ（更新版）

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

永続性のテスト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

まとめ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます
https://leanpub.com/build-your-own-coding-agent-ja.

第 7 章：安全装置（プランモード）

コンセプト

まずはテストから

ToolContext の拡張

保護された **WriteFile** ツール

エージェントクラス（更新版）

メインループ（更新済み）

ハーネスのテスト

「計画」の心理学

まとめ

第 8 章：コンテキストパイプライン（マップと検索）

ListFiles ツール

コードベース検索ツール

ツールリストの更新

「ズームイン」テスト

これは RAG なのでしょうか？

まとめ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます https://leanpub.com/build-your-own-coding-agent-ja.

第 9 章：現実確認（コードの実行）

フィードバックループ

最初にテスト

コマンド実行ツール

対話型の罠

自己修復のデモ

TDD のワークフロー

外科手術的な編集

クローズドループ

コンテキストの圧縮

セキュリティに関する考慮事項

まとめ

パート III：フロンティア

第 10 章：ダークサイドへ（ローカルモデル）

トレードオフ

Ollama のインストール

Ollama ブレインクラス

Ollama での実行

「無限ループ」実験

実用面での違い

ハイブリッドワークフロー

モデル選択

Ollama のトラブルシューティング

まとめ

第 11 章：拡張機能（ウェブ検索）

ステップ 1：メタプロンプト

ステップ 2：実行手順

ステップ 3：リファレンス実装

ステップ 4：テスト

自己修正

まとめ

第 12 章：総仕上げ（ゲームの制作）

Nanocode は実際に何かを作れるのでしょうか？

「ゼロコードチャレンジ」: Python と Pygame を使用して、古典的なスネークゲームを作ります。ルールは単純で、Python のコードを 1 行も書いてはいけません。エージェントとは英語でのみコミュニケーションを取ります。

補足: このデモには多くの API 呼び出しが含まれています。レート制限（HTTP 429）に達した場合、エージェントは自動的に再試行します。長時間のセッションでは、制限を完全に回避するために Ollama を通じてローカルモデルの使用を検討してください。

ステップ 1：準備

プロジェクトルート（`nanocode.py` と `.env` を含むディレクトリ）から、ゲーム用の作業ディレクトリを作成します：

```
mkdir -p snake_game
cp nanocode.py snake_game/
cp .env snake_game/
cd snake_game
```

本書のコードリポジトリを使用している場合は、ch11 からコピーしてください：

```
mkdir -p snake_game
cp resources/code/ch11/nanocode.py snake_game/
cp .env snake_game/
cd snake_game
```

Pygame をインストール:

```
pip install pygame
```

補足: Windows では、すぐに使えるはずです。macOS の場合、最初に SDL ライブラリをインストールする必要があるかもしれません:`brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf`。Linux の場合、SDL の開発用パッケージをインストールしてください:`sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`。

ステップ 2:アーキテクト(プランモード)

エージェントを起動します。設計図を描く前にレンガを積むことはできないので、まずはプランモードから始めます。

```
python nanocode.py
```

プロンプト:

```
Build a classic Snake game using Pygame. Include a score counter and Game
↪ Over screen with a restart option. Put ALL code in ONE file: snake.py.
↪ Write the plan in PLAN.md.
```

エージェントは `write_file` を使用して `PLAN.md` を作成します。その内容を確認してください。Snake クラス、Food クラス、そしてゲームループ—これらすべてを 1 つのファイルにまとめた概要が記載されているはずです。

内容が適切であれば、承認してください。

ステップ 3:ビルダー(実行モード)

実行モードに切り替えます:

```
/mode act
```

プロンプト内容:

```
Implement the plan in snake.py. All code in one file.
```

ターミナルを確認してください:

```
  Writing snake.py
```

エージェントは PLAN.md に保存されているコンテキストに基づいてコードを生成しています。

ステップ 4:現実確認

ゲームを実行する前に、タイムアウトの値を上げましょう。nanocode.py を開いて、RunCommand.execute() の timeout=30 を timeout=300 に変更してください。デフォルトの 30 秒ではゲームを実際にプレイするのに十分な時間がありません。(これはゼロコードチャレンジにおける唯一の例外です。)

プロンプト:

```
Run the game with: python snake.py
```

エージェントが run_command を実行します。ウィンドウが表示されます。あなたはスネークゲームをプレイします。

> **開発者向けヒント:** ゲームウィンドウはエージェントをブロックします。python snake.py を実行すると、ゲームウィンドウを閉じるまでエージェントは待機状態となります。これは正常な動作です—Pygame のメインループがターミナルを保持しているためです。

クラッシュした場合: LLM は非決定的です。エージェントは最初の試行でバグを生成する可能性があります。`AttributeError: 'Snake' object has no attribute 'draw'` のようなエラーでゲームがクラッシュした場合、自分で修正しないでください。エージェントに標準エラー出力を確認させてください。

プロンプト:

```
The game crashed. Read the error and fix it.
```

エージェントはトレースバックを読み、`read_file` を使用してバグを見つけ、`edit_file` を使用して修正し、再度実行します。

> **開発者向けヒント:** これは第 9 章で説明したフィードバックループが実際に動いている例です。エージェントがコードを書き、実行し、エラーを読み、修正する — あなたはただ見ているだけです。

ステップ 5:方向転換(機能の拡張)

ゲームは動作しますが、見た目が良くありません。スネークは単なる緑の四角形の集まりです。エージェントのリファクタリング能力をストレステストしてみましょう。

プロンプト:

```
The game looks boring. Make the snake change color as it eats food, increase
↪ speed every 5 points, and search the web for 'cool retro game color
↪ palettes' to apply.
```

エージェントは以下を実行する必要があります:

1. `search_web` を使用してカラーパレットを検索
2. `read_file` を使用して現在のレンダリングロジックを理解
3. `edit_file` を使用して新機能を注入
4. ゲームを実行して検証

何が問題になるか

結果は私の場合と異なるでしょう—LLM は非決定的です。しかし、一般的に起こることと、注意すべき点を説明します。

最初の実行で起きがちな失敗:

- `ModuleNotFoundError: No module named 'pygame'` — エージェントがインストールが必要なことを忘れているか、異なる環境でスクリプトを実行しています。最初に`pip install pygame` を実行するよう指示してください。
- エージェントが定義したメソッドを呼び出し箇所でスペルミスしたことによる`AttributeError`。エラートレースバックを確認すると、エージェントはこれらをすぐに修正します。
- 衝突判定での境界値エラー。蛇が壁をすり抜けたり、1 ピクセル早く死んでしまったりします。これらの修正には 2-3 回の編集-実行-修正のイテレーションが必要です。

典型的なセッションの流れ:

私のテストでは、エージェントは通常 2-4 回のイテレーションで動作する(ただし見た目の悪い)ゲームを作成します。最初の記述ではクラッシュするものが生成されます。2 回目か 3 回目の修正で動作するようになります。機能追加(色の変更、速度の調整など)のステップでは、エージェントが自身のコードを読み、的確な編集を行い、各変更を検証するため、さらに 3-5 回のイテレーションが必要になります。

最終的に、会話は 15-20 ラウンドに及びます。Claude を使用している場合、圧縮のトリガーに注意してください—12-15 ラウンド付近で、トークン数が 75% のしきい値に近づくと「(会話を圧縮中...)」と表示されます。圧縮後、エージェントは初期ラウンドの詳細の一部を失いますが、作業は継続します。これは第 9 章のシステムが真価を発揮している例です。

エージェントが苦戦する部分:

Pygame の座標系とイベントループは扱いが難しいものです。エージェントは時々、正しくレンダリングされるものの、キーボード入力を適切に処理できないコードや、蛇の頭が体の後ろに表示されてしまうような順序で描画するコードを書きます。これらは人間なら一目で気付くバグですが、エージェントには分かりません—視覚的なフィードバックがなく、標準出力とエラー出力しか見えないためです。ゲームがエラーなく実行されても見た目がおかしい場合は、視覚的なバグを説明する必要があります:「蛇が逆向きにレンダリングされています—頭が前面にあるべきです。」

重要なのは最初から完璧を目指すことではありません。エージェントが収束すること—書いて、実行して、エラーを読んで、修正して、繰り返す—が重要で、これは 11 章にわたって構築してきたすべてのツールを活用しています。

まとめ

計画、実装、クラッシュ、デバッグ、修正、再実行—これが各章で学んだことの集大成です。

では、ここからどこへ向かうのでしょうか？

エピローグ

全体で約 750 行の Python コードです。フレームワークは使用していません。`nanocode.py` は自由に使えます。以下のようなことが可能です：

- テストが成功した後に自動コミットする Git 統合
- フロントエンド作業のためのスクリーンショットベースのデバッグ（Claude は画像を読み取れます）
- Whisper を使用した音声入力で、タイプする代わりに話しかけることができます
- チームが既に使用している外部サービスに接続するための MCP

Claude Code、Cursor、Copilot のような本番環境のエージェントはこれ以上のことができます—ストリーミングレスポンス、並列ツール実行、tree-sitter 解析、サンドボックス実行環境、数千ファイルにまたがるマルチファイルコンテキストウィンドウなどです。750 行と 750,000 行の差は確かに存在します。しかし、アーキテクチャは同じです：脳、ループ、ツール、メモリ、そして安全性のためのハーネス。これで舞台裏で何が起きているのかが分かりましたね。

モデルは今後も改善を続けるでしょう。ハーネス—ループ、ツール、安全性チェック—その部分はエンジニアリングです。そしてその部分はこれからも変わることはありません。

謝辞