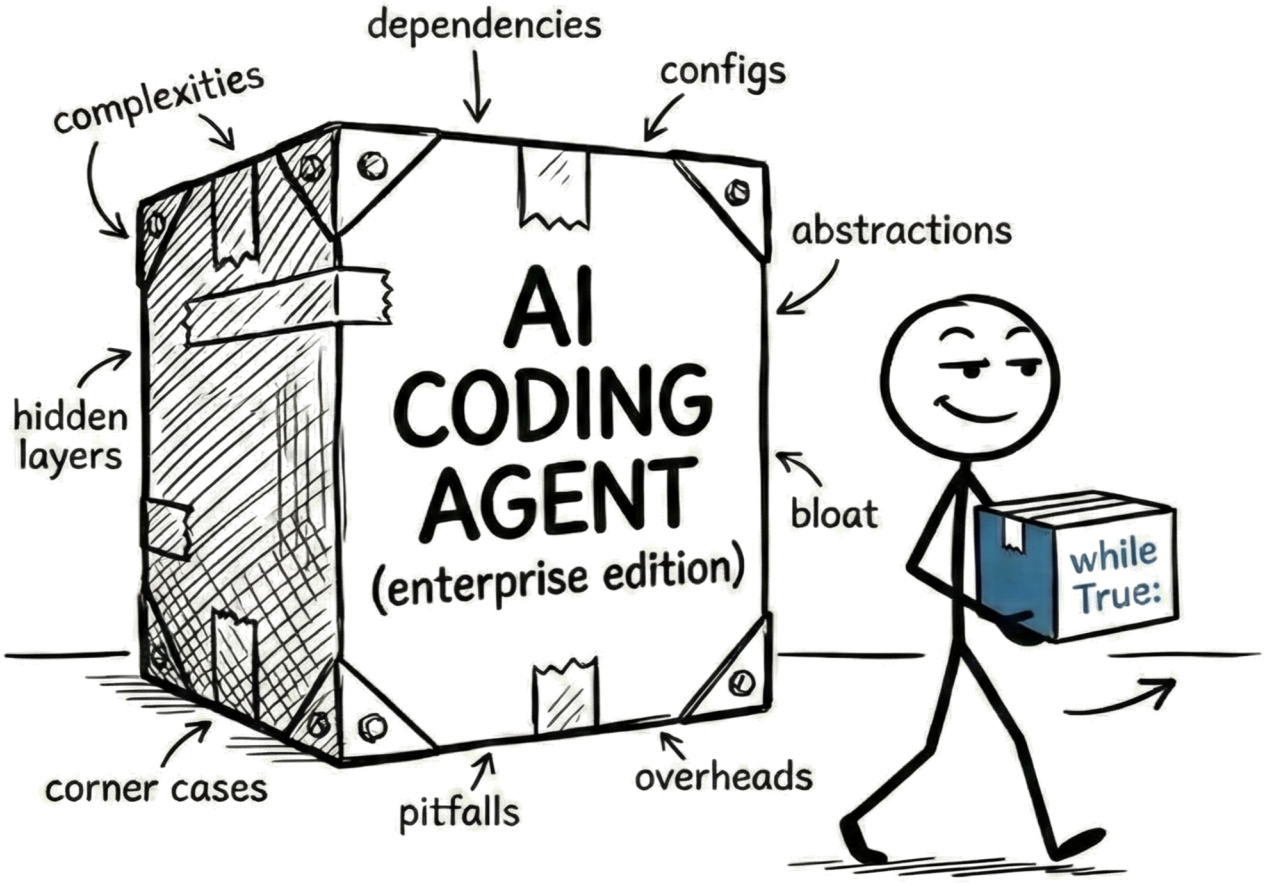
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



J. Owen

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

J. Owen

本書はこちらで販売中です
https://leanpub.com/build-your-own-coding-agent-ja-a7f51c5d

この版は 2026-04-19 に発行されました。

Leanpub

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 J. Owen 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：

Just built a coding agent from scratch with nothing but HTTP calls and a shell. Turns out "AI magic" is just a while loop and an API call.

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

まえがき

この本の対象読者

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。

フレームワークを試したことがあり、LangChain アプリがハルシネーションを起こしながら本番データベースを削除するところを目の当たりにした経験もあるでしょう。そして、こう思うはずです。**「もっとましな方法があるはずだ。」**

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のものです。マーケティング資料のダイアグラムの話ではありません。「推論エンジン」という抽象概念の話でもありません。実際の HTTP リクエスト。実際の`while` ループの話です。

Python が読めて、Web アプリか CLI ツールを作ったことがあれば、必要なものはすべて揃っています。

何を作るか

Nanocode は、ターミナル上で動作するコーディングエージェントです。本書を読み終えるころには、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を読み書きする
- シェルコマンドを実行する
- 純粋な Python でコードを検索する
- セッションをまたいでコンテキストを記憶する
- セーフティモードによって計画と実行を分離する
- ドキュメントや回答をウェブで検索する

`requests`、`python-dotenv`、`pytest` を使ってゼロから構築します。シェルアクセスには Python の標準ライブラリである`subprocess` モジュールを使います。LangChain は使いません。ベクター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、シンプルな Python だけです。

唯一の例外として、第 11 章ではウェブ検索のために`ddgs` を追加します —— これは軽量な依存関係がひとつ増えるだけです。

テストのアプローチ

本書では「テストと実装を並行して進める」アプローチを採用しています。ほとんどの機能について、次の流れで進めます。

1. コンセプトの紹介 —— なぜこれが必要なのか
2. まずテストを提示し、成功がどのような状態かを明確にする
3. テストをパスするコードを実装する
4. pytest で検証する

これにより、紙面上でレッド・グリーン・リファクタリングのすべての手順を踏まなくても、**テスト駆動開発**の考え方を身につけられます。また、これは重大な問題も解決してくれます。LLM を使ったアプリケーションは、実際に LLM を呼び出してテストすることができません。API コールは遅く、コストがかかり、非決定論的です。

第 3 章からは、`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、API コールを一切行わず、一銭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」アプローチに従っています。各章は前の章の上に積み上げられており、すべてのコード例は実際に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してください。[1]
- **Leanpub：**購入時のダウンロードリソースにも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/`、`ch03/`、`ch04/`など）。また、`appendix/`フォルダもあります。ほとんどのフォルダには以下が含まれています。

- `nanocode.py` — その章の完全な実行可能なエージェント
- `test_nanocode.py` — API コールなしでコードが正しく動作することを検証するテスト

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

例外が 2 つあります。ch02/にはスタンドアロンのtest_api.py スクリプトのみが含まれており（一度使ったら捨てるものです）、ch12/のエージェントのスナップショットは機能的にはch11/と同じで、集大成となる成果物が追加されています。

任意の章のフォルダをコピーして、そこから再開できます。pytest を実行して、コードが期待どおりの動作と一致するか確認してください。

本書で使用する表記規則

本書を通じて、3 種類のコールアウトが登場します。

> **開発のヒント：**このプロジェクトを超えて応用できるアーキテクチャ上の知見です。自分の仕事でも再利用できるパターンです。

警告：セキュリティや安全性に関するリスクです。注意してください —— 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：深掘りや脱線です。有益なコンテキストですが、最初に読む際はスキップしても流れを見失うことはありません。

コードのウォークスルー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なのか）、次にコード本体、そして重要な部分の行ごとの説明、という順序です。

さあ、コードを書きましょう。

パート I：ブレイン

パート I では、エージェントの骨格を組み立てます。生の HTTP を使って Claude と通信する`while` ループです。第 4 章までに、アダプターパターンを用いて複数の LLM プロバイダーをサポートできるようになります。そして、「AI エージェント」とはただのループとブレインとメッセージのリストに過ぎないことを理解できます。

第 1 章：ゼロマジック宣言

過去 2 年間で AI アプリケーションを構築しようとしたことがあるなら、「フレームワーク疲れ」を感じたことがあるはずです。

人気のライブラリをインストールする。`ReasoningEngine` をインポートする。`.run()` を呼び出す。「Hello World」のサンプルでは魔法のように動く。でも、実際に何か意味のあることをしようとした瞬間——たとえば、インポート文を消さずに Python ファイルの特定の行を編集するといった作業——途端に壊れる。

しかも、フレームワークを使っているせいで、自分では直せない。抽象クラス、ファクトリーパターン、「チェーン」の層を掘り進んで、ハルシネーションの原因となっているプロンプトを探し出そうとする羽目になる。

ここではそんなことはしません。

この本は「マジック」への反乱です。「ゼロマジック」アプローチで進めます。つまり、プロダクションレベルのコーディングエージェント **Nanocode** を純粋な Python で構築するのです。LangChain なし、AutoGPT なし、Pydantic なし。

なぜか？　自律型エージェントは魔法ではないからです。ただの `while` ループです。

エージェントとは、本当に何なのか？

ベンチャーキャピタルのマーケティングを取り払えば、「エージェント」とはただの**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）、設定温度と比較し（判断）、ヒーターをオンにする（行動）。そして待機して繰り返す。それだけです。AI エージェントも同じことをします。温度の代わりにテキストを使うだけです。

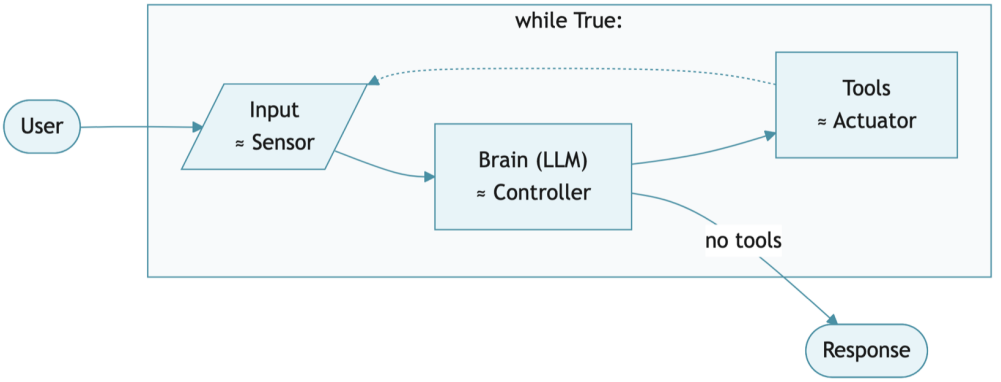


図 1. エージェントループ：ユーザー入力が while ループを流れ、入力（センサー）がブレイン／ LLM（コントローラー）に渡され、ツール（アクチュエーター）を起動し、ブレインが応答を出力するまで入力に戻り続ける。

より具体的に言うと、エージェントは 4 つの部品で構成されています。**ブレイン**は LLM です——テキストを送るとテキストが返ってくる、ステートレスな関数です。ブレインは**ツール**を呼び出します——「ファイルを読む（Read File）」「コマンドを実行する（Run Command）」といった関数で、外の世界と対話します。これらすべては**ループ**（`while True`）の中に収まっており、タスクが完了するまで繰り返し続けます。その間、**メモリ**——ただの Python リスト——が会話履歴を蓄積していきます。（リストはプログラムが終了すると消えます。永続ストレージは第 6 章で追加します。）

`while` ループが書けるなら、エージェントは作れます。

ゼロから構築することで、フレームワーク利用者には手に入らないものが手に入ります。それは**制御**です。エージェントがループにはまったとき、どのコード行が原因かを正確に把握できます。API の料金が高くなりすぎたとき、どこでトークンが漏れているかを正確に確認できます。

私たちが構築するもの

Nanocode はターミナルで動く CLI ツールです。同僚に話しかけるように会話できます。ファイルを読み、コマンドを実行し、コードを編集します。

この本を読み終えるころには、Claude Sonnet 4.6（または DeepSeek、あるいは Ollama を経由したローカルモデル）に接続できているでしょう。ファイルの読み込み、書き込み、シェルコマンドの実行といったツール（「手」）を与え、コードベースを検索するための「目」も与えます。そして、誤って `rm -rf /` を実行してしまわないよう、安全装置も構築します。

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、後方互換性を壊していきます。`requests` と `subprocess` だけに依存することで、このエージェントはおそらく 5 年後も動き続けるでしょう。

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 ツールは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、ボットがそれをスクレイピングし、数分以内にアカウントを使い果たしてしまいます。

`.gitignore` を作成してください:

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

AgentStop 例外

イベントループを書く前に、きれいな終了メカニズムが必要です。コードのあちこちに散らばった`break` 文より、例外の方が優れています。

背景: 例外はエラーのためだけにあるのではなく、制御フローのメカニズムとしても使われます。ユーザーが`/q` と入力すると、`AgentStop` が発生します。メインループがそれをキャッチして、きれいに終了します。

コード:

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

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

これは nanocode.py の先頭に記述します。マーカーとなる例外であり、ロジックを持たず、シグナルとして機能するだけです。

Agent クラス

次に、コアとなる抽象化である Agent クラスを見ていきます。状態とロジックをひとつの場所に保持するため、テストが容易になります。

背景: すべてのロジックを main() に実装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() をモックする必要があります。ロジックを Agent.handle_input() に抽出することで、直接テストできるようになります。

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。`brain` と `tools` は後の章で追加します。
- **18〜19 行目:** `/q` コマンドは特別な値を返すのではなく、`AgentStop` を送出します。シャットダウンの処理方法は呼び出し元が決定します。
- **24 行目:** 入力をエコーバックします。これはプレースホルダーで、後ほど Brain に送信するようにします。

テストで成功を定義する

メインループを書く前に、テストを用意する必要があります。

`test_nanocode.py` を作成してください:

```
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  ")
```

テストを実行する:

```
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
```

全てグリーン。エージェントは基本的なケースを正しく処理できています。

補足:なぜ pytest なのか?pytest は`test_`で始まる関数を見つけて実行します。ボイラープレートも不要、クラスも不要。テストコード自体は素の Python——マジックは一切ありません。

メインループ

次に、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()` がブロックし、ユーザーの入力を待ちます。
- **37〜39 行目:** `handle_input()` を呼び出し、出力があれば表示します。
- **41〜43 行目:** `AgentStop`(`/q` による)または `KeyboardInterrupt`(Ctrl+C による)をキャッチし、ループを抜けます。

補足: Python の `input()` は 1 行ずつ読み込みます。本書のプロンプトはすべて 1 行形式です。これによってコードをシンプルに保てます。本番エージェントでは、readline やフル TUI といった、より高度な入力方法を使うことが多いです。

この分離に注目してください:`Agent.handle_input()` にすべてのロジックが含まれており、`main()` は単なる I/O のつなぎ役です。この構造により、stdin/stdout をモック化することなくエージェントをテストできます。

実行してみよう

```
python nanocode.py
```

次のように表示されるはずです:

```
⚡ 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

シャーシはこれで完成です:`Agent` クラス、`handle_input()` メソッド、`while True` ループ。まだ何も役に立つことはできていませんが、これ以降に構築するものはすべてこのスケルトンに接続されていきます。テストによって、作業を進める中ですでに動いている部分を壊さないようにします。

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` を実行するよう指示されています。しかし、本書ではそうしません。

SDK は真実を隠します。抽象化のレイヤーを重ねることで「Hello World」は簡単になりますが、「Error 400」のデバッグは悪夢になります。SDK を学べば、SDK を学んだことになります。生の HTTP 呼び出しを学べば、あらゆる SDK の背後にあるプロトコルそのものを学んだことになります。

私たちは `requests` ライブラリだけを使って Claude にメッセージを送ります。Claude は Anthropic の主力 LLM であり、コーディングタスクにおいて最も優れたモデルの一つです。

API キーを取得する

Claude と通信するには、API キーが必要です。これは、課金アカウントに紐付いたパスワードのような長い文字列です。

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 ドルのクレジットが必要です）。
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：このキーはパスワードと同様に扱ってください。このキーを持つ者は誰でもあなたの資金を使えます。

金庫（.env）

このキーを安全に保管する場所が必要です。キーをコードに直接書き込むことはしません。

プロジェクトのルートに `.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-...
```

python-dotenv は、まさにこの目的のために第 1 章でインストールしました。このライブラリは .env 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 と通信するには、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。ヘッダーに含める認証情報（API キー）、ボディに含める設定（使用するモデルやトークン数）、そしてメッセージそのものです。

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	あなたのシークレットキー	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、メッセージの辞書のリストを受け取ります:

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

各メッセージには role(`"user"` または `"assistant"`)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てください。これは接続が正しく動作することを証明するための「スモークテスト」です。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。関数もクラスもありません。ベアメタル(素の実装)を直接見たいのです。

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)
```

ウォークスルー:

- **7 行目:** `load_dotenv()` が `.env` ファイルを見つけ、変数を `os.environ` に読み込みます。
- **8 行目:** API キーを取得します。絶対にハードコードしないこと。
- **11〜13 行目:** 基本的な確認処理です。これがないと、キーが見つからない場合にヘッダーの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。省略すると、API からリクエストが拒否されます。
- **27 行目:** `claude-sonnet-4-6` で使用するモデルを指定します。
- **28 行目:** `max_tokens` は必須項目です。レスポンスの長さを制限し、コストが際限なく膨らむのを防ぎます。

- **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ください。どこで失敗している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 なら成功（JSON を整形して表示します）。それ以外の場合は、デバッグのために生のエラーテキストを出力します。

実行

```
python test_api.py
```

すべてうまくいけば、次のように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	対処法
401 Unauthorized	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")` を出力して検証してください。
400 Bad Request	JSON の形式が不正	`max_tokens` を忘れていませんか?`messages` はリストになっていますか?
429 Rate Limit	リクエスト過多またはクレジット不足	しばらく待つか、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

脳と通信できることを証明しました。`test_api.py` は削除してしまいましょう——もう必要ありません。

補足: この `test_api.py` は使い捨てコードです——一回限りのスモークテストに過ぎません。本格的な自動テスト(`FakeBrain` と pytest を使ったもの)は第 3 章で登場します。API 接続が正常に動作することを確認したら、必ずこのファイルを削除してください。

開発者向け Tip: API を呼び出すたびにコストがかかります。入力トークン(送信する内容)は安価ですが、出力トークン(モデルが生成する内容)はおよそ 5 倍高くなります。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 の使用状況タブを確認してください。2026 年初頭時点では、Claude Sonnet を使った 20〜30 回のやり取りを含む典型的なコーディングセッションのコストは $0.10〜$0.50 程度です。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ており、これをログに記録してコストをプログラム的に追跡することもできます。

まとめ

これが生の API コールです——ヘッダー、JSON ペイロード、レスポンスの解析。あなたとワイヤーの間に抽象化レイヤーは一切ありません。何か問題が起きたとき（必ず起きます）、どのレイヤーで失敗したかが正確にわかります。なぜなら、レイヤーは一つしかないからです。

一つ問題があります。Claude には記憶が一切ありません。リクエストのたびに白紙の状態から始まります。これを解決するために、毎回のやり取りで会話履歴全体を送り直すことで、偽の記憶を実現していきます。

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 回実行する場面を想像してください。「私の名前は Alice 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?」と聞くと、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。LLM には記憶がまったくありません。すべてのリクエストにおいて、LLM にとってあなたは初対面です。

エージェントを構築するには、人工的なメモリを作成してこの問題を解決する必要があります。

メモリという幻想

LLM における「メモリ」はハードドライブではありません。ログファイルです。

ChatGPT とチャットするとき、5 分前に言ったことを「覚えている」わけではありません。舞台裏では、新しいメッセージのたびに**会話履歴全体**がモデルに送信されています。

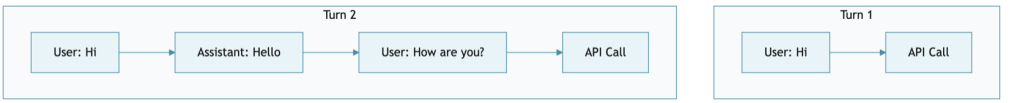


図 2. コンテキストの蓄積：ターン 1 では API に「User: Hi」だけを送信する。ターン 2 では API に「User: Hi」「Assistant: Hello」「User: How are you?」という全履歴を送信する。

モデルは毎回、会話の全文を見ています。それがこの仕掛けです。

この**コンテキストループ**を手動で実装してみましょう。ただし、その前にコードをテスト可能な状態にする必要があります。

テストの問題

厳しい現実をお伝えします：LLM を実際に呼び出して LLM 搭載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（1 回あたり 2〜10 秒）、コストがかかり（呼び出しのたびに課金されます）、非決定的です（毎回異なるレスポンスが返ってくる可能性があります）。5 ドルのコストで 20 分かかるテストスイートの実行を想像してみてください。とても実行する気にはなれないでしょう。

解決策は**依存性の注入**です。API 呼び出しをエージェントの内部にハードコーディングする代わりに、「brain」オブジェクトを外部から渡します。本番環境では、brain は Claude です。テストでは、brain は予測可能なレスポンスを返すフェイクです。

このパターンを今のうちに確立しておきましょう。本番コードをこれ以上書く前に。

レスポンスの型

brain を構築する前に、それが返す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。扱いやすいシンプルな Python オブジェクトが必要です。

コンテキスト： Claude は 1 回のレスポンスで、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シンプルなデータオブジェクトが必要です。（`@dataclass` は意図的に使用しません —— これらのクラスはシンプルなので、デコレーターを使えば数行は省略できますが、`__init__`が実際に何をしているかが見えにくくなります。）

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、ブレインがツールの実行を私たちに依頼していることを表します。`id` は追跡用の一意の識別子です（結果を返す際に Claude が必要とします）。`name` は実行するツールの名前で、`args` はパラメータの辞書です。

`ToolCall` はまだ使用しません——ブレインはまだツールを呼び出せないからです——しかし、`Thought` レスポンス型の一部であるため、ここで定義しておきます。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行いたいときに、これら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, thinking=None):
        self.text = text  # str or None
        self.tool_calls = tool_calls or []  # list of ToolCall
        self.thinking = thinking  # str or None
```

Thought は、ブレインが思考した後に返すものです。テキスト、ツール呼び出し、その両方、またはどちらも含まない場合があります。thinking フィールドはモデルの推論の要約を記録します。それがどこから来るのかは、以下で Claude クラスを構築する際に確認します。この抽象化により、他のコードを一切変更せずに、後で Claude を DeepSeek に差し替えることができます。

FakeBrain 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。

背景: 予測可能な応答を返し、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test_nanocode.py に記述するもので、プロダクションコードには書きません。FakeBrain は、実際のブレインが持つことになるのと同じインターフェースを持っていることに注目してください。つまり、会話を受け取ってThought を返すthink() メソッドです。

補足:実際の依存オブジェクトを、テスト用の予測可能なフェイクに置き換えるこのパターンは、**テストダブル**と呼ばれます。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)について詳しく説明されています。LLM のテストでは、固定レスポンスを持つシンプルなフェイクがあれば、たいていの場合は十分です。

成功の定義

プロダクションコードを書く前に、成功とはどのような状態かを定義しておきましょう。これらのテストが実装の指針となります。

テスト 1:ブレインがレスポンスを返す

```
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。これが依存性の注入の実例です。

テスト 2:会話が蓄積される

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

```python
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

    agent.handle_input("Second message")
    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必要があります。

テスト 3:正しいメッセージ構造

```python
def test_conversation_contains_correct_roles():
    """Verify conversation has correct role alternation."""
    brain = FakeBrain(responses=[Thought(text="AI response")])
    agent = Agent(brain=brain)

    agent.handle_input("User message")

    assert agent.conversation[0]["role"] == "user"
    assert agent.conversation[0]["content"] == "User message"
    assert agent.conversation[1]["role"] == "assistant"
    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ある必要があります:`{"role": "user", "content": "..."}`。

テスト 4:Brain が会話を受け取る

```python
def test_brain_receives_conversation():
    """Verify brain.think is called with the conversation list."""
    brain = FakeBrain()
    agent = Agent(brain=brain)

    agent.handle_input("Test message")

    assert brain.last_conversation is not None
    assert len(brain.last_conversation) == 1
    assert brain.last_conversation[0]["content"] == "Test message"
```

ブレイン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。

今すぐこれらのテストを実行してください——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
...
```

よし。では、テストをパスさせましょう。

Claude クラス

さて、いよいよ本物の頭脳部分です。

背景: 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` にパースする役割を担います。また、**拡張思考**も有効にします。これは、モデルが回答する前に内部的なメモ書きを行う機能です。モデルが話す前にメモ帳で自分自身に語りかけるイメージです。追加のトークンが必要になりますが、品質の向上は顕著です。特に第 5 章でツールを追加した後、モデルがどのツールをなぜ呼び出すべきかを推論する必要が生じる場面では、その効果が際立ちます。

コード:

```
37 class Claude:
38     """Claude API - the brain of our agent."""
39
40     def __init__(self):
41         self.api_key = os.getenv("ANTHROPIC_API_KEY")
42         if not self.api_key:
43             raise ValueError("ANTHROPIC_API_KEY not found in .env")
44         self.model = "claude-sonnet-4-6"
45         self.url = "https://api.anthropic.com/v1/messages"
46
47     def think(self, conversation):
48         headers = {
49             "x-api-key": self.api_key,
50             "anthropic-version": "2023-06-01",
51             "content-type": "application/json"
52         }
53         payload = {
54             "model": self.model,
55             "max_tokens": 16000,
56             "thinking": {
57                 "type": "enabled",
58                 "budget_tokens": 10000
59             },
60             "messages": conversation
61         }
62
63         response = requests.post(self.url, headers=headers, json=payload,
           ↪ timeout=120)
64         response.raise_for_status()
65         return self._parse_response(response.json()["content"])
```

ウォークスルー:

- **41〜43 行目:** API キーを読み込み、見つからない場合は即座に失敗します。
- **44〜45 行目:**設定を格納します。モデルは後で設定可能にします。
- **47 行目:** `think()` メソッドはブレインのインターフェースで、`FakeBrain` と同じです。
- **55〜59 行目:** *extended thinking*(拡張思考)を有効にします。これにより、モデルは応答する前に推論の要約を生成し、複雑なタスクの品質が向上する一方、より多く

のトークンを消費します。budget_tokens は、モデルが推論に使用できるトークン数の上限を設定します(ここでは 10,000)。これらのトークンは出力トークンと同様に料金の対象となります。今回の設定では、max_tokens は思考と応答を含む合計出力をカバーするため、Anthropic は max_tokens が budget_tokens を超えるよう要求しています。思考トークンが 10,000、最大トークンが 16,000 の場合、応答自体には最大 6,000 トークンを使用できます。

- **60 行目:**ペイロードには "messages": conversation が含まれます。これは現在のメッセージだけでなく、会話の全履歴です。これがコンテキストループです。
- **65 行目:** Claude の複雑なレスポンス形式をパースして、シンプルな Thought に変換します。

次はレスポンスパーサーです:

```
67      def _parse_response(self, content):
68          """Convert Claude's response format to Thought."""
69          text_parts = []
70          tool_calls = []
71          thinking = None
72
73          for block in content:
74              if block["type"] == "thinking":
75                  thinking = block["thinking"]
76              elif block["type"] == "text":
77                  text_parts.append(block["text"])
78              elif block["type"] == "tool_use":
79                  tool_calls.append(ToolCall(
80                      id=block["id"],
81                      name=block["name"],
82                      args=block["input"]
83                  ))
84
85          return Thought(
86              text="\n".join(text_parts) if text_parts else None,
87              tool_calls=tool_calls,
88              thinking=thinking
89          )
```

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、

`"thinking"`、`"text"`、`"tool_use"` のいずれかです。thinking ブロックが最初に届き、モデルの推論の要約が含まれています。これは呼び出し元が表示できるよう `Thought` に保存します。text ブロックはレスポンスになり、`tool_use` ブロックは `ToolCall` オブジェクトになります。パーサーは何も出力せず、生の JSON をきれいな `Thought` に変換するだけです。

Agent クラス（更新版）

第 1 章の `Agent` を更新して、ブレインを引数として受け取り、会話履歴を保持できるようにします。

コード：

```
94  class Agent:
95      """A coding agent with conversation memory."""
96
97      def __init__(self, brain):
98          self.brain = brain
99          self.conversation = []
100
101     def handle_input(self, user_input):
102         """Handle user input. Returns output string, raises AgentStop to
            ↪ quit."""
103         if user_input.strip() == "/q":
104             raise AgentStop()
105
106         if not user_input.strip():
107             return ""
108
109         self.conversation.append({"role": "user", "content": user_input})
110
111         try:
112             thought = self.brain.think(self.conversation)
113             if thought.thinking:
114                 lines = thought.thinking.strip().split("\n")[:5]
115                 for i, line in enumerate(lines):
116                     prefix = "  💭 " if i == 0 else "     "
117                     print(f"\033[2m{prefix}{line}\033[0m")
118             text = thought.text or ""
```

```
            self.conversation.append({"role": "assistant", "content": text})
            return text
        except Exception as e:
            self.conversation.pop()  # Remove failed user message
            return f"Error: {e}"
```

ウォークスルー:

- **97〜99 行目:**依存性の注入によってブレインを受け取ります。空の会話リストを初期化します。
- **109 行目:**ブレインを呼び出す**前に**、ユーザーのメッセージを履歴に追加します。
- **112〜120 行目:**ブレインを呼び出し、最大 5 行の思考内容をディム(薄暗く)表示します(`\033[2m` は dim 表示の ANSI エスケープコード、`\033[0m` でリセット)。レスポンスを抽出し、履歴に追加します。
- **121〜123 行目:** API 呼び出しが失敗した場合、直前に追加した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。

109 行目に注目してください。ブレインを呼び出す**前に**ユーザーのメッセージを追加しています。ブレインは、現在のメッセージを含む会話全体を参照する必要があるためです。

> **開発者向け Tip:** うまくいかないとき、最初のデバッグツールは`print(self.conversation)` です。ブレインが何を見ているかを正確に確認できます。生のリストを調べると、不正形式のメッセージ、ロールの欠如、切り詰められたコンテンツがすぐにわかります。

メインループ(更新版)

メインループは、現在は薄い I/O ラッパーにすぎません:

```
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()
```

ロジックはすべて Agent クラスに含まれています。ループは入力を読み取り、handle_input() を呼び出して、結果を出力するだけです。この分離により、エージェントがテストしやすくなります——input() や print() をモックする必要なく、Agent.handle_input() を直接テストできます。

テストが通ることを確認する

テストを再度実行してみましょう:

```
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
```

全てグリーン。テストは API コールを一切行わずに、実装を検証します。

メモリをテストする

次は本物のブレインでテストしてみましょう:

```
python nanocode.py
```

次の会話を試してみましょう:

```
 I am building a Python agent.
  💭 The user is telling me about their project. They want to build
     a Python agent. I should respond helpfully and ask what kind
     of agent they're building.

That sounds exciting! What kind of agent are you building?

 What language am I using?
  💭 The user previously said they are building a Python agent.
     The answer is Python.

You are using Python.
```

会話リストはその役割を果たしています。

コンテキストウィンドウの問題

「これをずっと動かし続けられるの?」と思っているかもしれません。

いいえ。

ループが反復するたびに、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

やがて、**コンテキスト上限**に達します——Claude Sonnet では 200k トークン、DeepSeek では 128k、ローカルモデルによっては 4k という低さのものもあります。上限を超えると、API は `400 Bad Request` を返します。121 行目のエラーハンドリングがこれを捕捉してエラーを報告するので、エージェントが気づかないまま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——履歴がまだ長すぎるため、その後のメッセージもすべて失敗し続けるからです。

今のところ、エージェントを再起動すれば履歴がクリアされ、作業を再開できます。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——は、第 9 章でフィードバックループを構築する際に追加します。会話が実際に膨れ上がるのはそこであり、その修正が真価を発揮するのもそこです。

> **Dev Tip:** ネットワーク呼び出しには 2〜10 秒かかります。113〜117 行目の思考表示は、エージェントが動作中であることをユーザーに即座に視覚的フィードバックとして伝えます。これがなければ、ユーザーはフリーズしたプロンプトをじっと見つめ、Ctrl+C に手を伸ばすことになるでしょう。

まとめ

Claude はこれで記憶を持つようになりました——正確には、記憶を持っているかのように錯覚させることに成功しました。会話リストはターンごとに大きくなり、`FakeBrain` を使えば一切コストをかけずに全体をテストできます。

どちらのパターンも本書の残りの部分でも引き続き使用します。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。

一つ未解決の問題があります：私たちのコードは Anthropic の API にハードコードされています。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りません。

第 4 章：ユニバーサルアダプター

アダプターパターン

HTTP の堅牢性

Brain インターフェース

FakeBrain（更新版）

リファクタリング後の Claude Brain

DeepSeek ブレイン

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-a7f51c5d.

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-a7f51c5d.

Agent 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-a7f51c5d.

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-a7f51c5d.

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-a7f51c5d.

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-a7f51c5d.

試してみよう

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-a7f51c5d.

「コードを動かしただけ」

まとめ

第 II 部：手

第 5 章：ツールプロトコル

ツールが実際にどう機能するか

ツールインターフェースの定義

ReadFile ツール

WriteFile 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-a7f51c5d.

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-a7f51c5d.

ツールを使った 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-a7f51c5d.

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-a7f51c5d.

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-a7f51c5d.

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-a7f51c5d.

第 6 章：スクラッチパッド（メモリ）

「ゼロマジック」メモリ

Memory 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-a7f51c5d.

Agent 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-a7f51c5d.

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-a7f51c5d.

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-a7f51c5d.

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-a7f51c5d.

第 7 章：安全ハーネス（プランモード）

コンセプト

まずテストから

WritePlan ツール

1 つのリスト、2 つのビュー

ブレインに現在のモードを伝える

Agent 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-a7f51c5d.

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-a7f51c5d.

ハーネスのテスト

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-a7f51c5d.

「計画」の心理学

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-a7f51c5d.

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-a7f51c5d.

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

SearchCodebase ツール

ツールリストの更新

「ズームイン」テスト

これって RAG じゃないの？

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-a7f51c5d.

第 9 章：現実確認（コードを実行する）

フィードバックループ

テストファースト

RunCommand ツール

インタラクティブな罠

自己修復デモ

TDD のワークフロー

外科的編集

クローズドループ

ループの堅牢化

コンテキスト圧縮

セキュリティに関する考慮事項

まとめ

第 III 部：フロンティア

第 10 章：闇に潜る（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 で実行する

「無限ループ」実験

実際の違い

ハイブリッドワークフロー

モデルの選択

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：手術

ステップ 3: リファレンス実装

ステップ 4：テスト

自己改変

まとめ

第 12 章：総仕上げ（ゲームを作る）

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 を使ってクラシックな Snake ゲームを作ります。ルールは一つ——Python を一行も書いてはいけません。エージェントへの指示は英語で話しかけるだけです。

補足: このデモでは多くの API コールが発生します。レート制限（HTTP 429）に達した場合、エージェントは自動的にリトライします。長いセッションでは、制限を完全に回避するために Ollama を使ったローカルモデルの利用も検討してみてください。

ステップ 1：準備

プロジェクトのルート（`nanocode.py` と `.env` が含まれるディレクトリ）から、ゲーム用の作業ディレクトリを作成します：

補足: サブディレクトリで作業することで、エージェントの`list_files` がゲームファイルだけを参照し、自身のソースコードが見えないようになります。そして、まっさらなメモリの状態から始められます。

```
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、代わりに 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 をインストールする:

```
pip install pygame
```

補足: 上記のシェルコマンドは macOS/Linux 用です。Windows の場合は、代わりに `mkdir snake_game`、`copy nanocode.py snake_game\`、`copy .env snake_game\` を使用してください。macOS で `pip install pygame` が失敗する場合は、`brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf` が必要になることがあります。Linux の場合は、SDL 開発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`。Windows では、`pip install pygame` にはすべてが同梱されています。

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。レンガを積む前に設計図を用意したいので、プランモードで始めます。

```
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_plan` を使って`PLAN.md` を作成します。読んでください。Snake クラス、Food クラス、そしてゲームループがすべて単一ファイルにまとめられた構成が概説されているはずです。

プランが妥当であれば、ステップ 3 に進んでください。

ステップ 3：ビルダー（アクトモード）

アクトモードに切り替えます:

```
/mode act
```

プロンプト:

```
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` に保存したコンテキストをもとに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、タイムアウトを増やしてください。デフォルトの 30 秒では実際にゲームをプレイするには不十分です——Pygame のメインループはウィンドウを閉じるまで処理をブロックし続けます。起動する前に環境変数を設定してください:

```
export NANOCODE_TIMEOUT=300
```

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。ウィンドウがポップアップします。スネークをプレイします。

> **Dev Tip**: 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し続けます。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 は非決定論的です。エージェントが最初の試みでバグを出してしまうこともあります。`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ても、自分で修正しないでください。stderr をエージェントに確認させましょう。

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取り、`read_file` でバグを特定し、`edit_file` でパッチを当て、再度実行します。

ステップ 5：ピボット（フィーチャークリープ）

ゲームは動いていますが、見栄えがしません。スネークはただの緑の四角形です。エージェントのリファクタリング能力を徹底的にテストしましょう。

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.
```

エージェントがすべきこと:

1. `search_web` を使ってカラーパレットを探す
2. `read_file` を使って現在のレンダリングロジックを理解する
3. `edit_file` を使って新機能を組み込む
4. ゲームを実行して動作を確認する

うまくいく場合とうまくいかない場合

LLM は非決定論的なので、あなたの結果は私のものとは異なるでしょう。ただ、典型的に起こりがちなことと、注意すべきポイントを紹介します。

初回実行時によくある失敗：

- `ModuleNotFoundError: No module named 'pygame'` — エージェントがインストールの必要性を忘れていたか、別の環境でスクリプトを実行してしまった場合です。エージェントに先に `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側でスペルミスしたことによる `AttributeError`。トレースバックを確認すれば、エージェントはすぐに修正できます。
- 衝突検出のオフバイワンエラー。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。これは編集・実行・修正を 2〜3 回繰り返すことになります。

典型的なセッションの流れ：

私のテストでは、エージェントはたいてい 2〜4 回の反復で動作する（ただし見た目は粗い）ゲームを完成させます。最初の出力はクラッシュするものになり、2 回目か 3 回目の修正でようやく動くようになります。機能拡張のステップ（色の変更、速度の段階的な上昇）では、エージェントが自分のコードを読み返してピンポイントの編集を行い、各変更を確認するため、さらに 3〜5 回の反復が加わります。

最終的には、会話が 15〜20 ラウンドにも及びます。Claude を使っている場合は、コンパクションが発動するタイミングに注意してください。トークン数が 75% のしきい値に近づく 12〜15 ラウンド目あたりで「(Compacting conversation...)」と表示されます。コンパクション後、エージェントは序盤のラウンドの細かい情報を一部失いますが、作業は続けます。これこそ第 9 章で作り上げたシステムが本領を発揮する瞬間です。

エージェントが苦手とするところ：

Pygame の座標系とイベントループはやっかいです。エージェントは、レンダリングは正しくてもキーボード入力を適切に処理しないコードを書いてしまうことがあります。また、蛇の描画順序が逆になって頭が胴体の後ろに表示されてしまうこともあります。こういったバグは人間がすぐに気づけるものですが、エージェントには視覚的なフィードバックがなく、stdout と stderr しか見えません。エラーなしにゲームが起動しても見た目がおかしい場合は、視覚的なバグを言葉で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る——頭は前に来るべきだ」といった具合に。

最初から完璧にいく必要はありません。大切なのは、エージェントが**収束する**ことです——書く、実行する、エラーを読む、修正する、繰り返す——11 章にわたって構築したすべてのツールを活用しながら。

最終的な snake.py をリリースする前に、コードを読んでみてください。エージェントは**動く**コードを書きますが、テストループでは検出できないことを人間が確認する必要があります。ハードコードされたマジックナンバー、見落とされたエッジケース（ウィンドウがリサイズされたらどうなる?）、そしてコードの構造が保守しやすい形になっているかどうかです。エージェントは素早い下書き役であって、最終レビュアーではありません。

まとめ

計画して、実装して、クラッシュして、デバッグして、修正して、また実行する——これがすべての章の本領を発揮する瞬間です。

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っていません。nanocode.py はあなたのものです。好きなように使ってください：

- テストがグリーンになった後に自動コミットする Git 連携
- フロントエンド作業向け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）
- タイピングの代わりに話せるよう、Whisper による音声入力
- チームがすでに使っている外部サービスへの接続のための MCP
- 会話をフォークしてサブタスクを並行処理するサブエージェント

Claude Code、Cursor、Copilot といったプロダクションエージェントはこれよりもさらに多くのことをこなします。ストリーミングレスポンス（付録 A を参照）、並列ツール実行、tree-sitter による構文解析、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。750 行と 75 万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。ブレイン、ループ、ツール、メモリ、そして安全ハーネス。これで舞台裏に何があるかがわかったはずです。

モデルは進化し続けるでしょう。ハーネス——ループ、ツール、安全チェック——それはエンジニアリングの領域です。そしてその部分は、これからもなくなることはありません。

付録 A：ストリーミングレスポンス

ストリーミングの仕組み

実装

第 11 章からの変更点

トレードオフ

コードの実行

謝辞