Tutorial Script 2: How to Use Claude Code Effectively

Duration: 10-12 minutes (Split into Part A and Part B) **Objective:** Teach students how to effectively communicate with Claude Code through file references, context management, commands, and best practices

---

PART A: Core Features (6 minutes)

SECTION 1: Referencing Files (2 minutes)

[SCREEN: Claude Code interface open in minimal-flashcards]

Professor:

"In our last tutorial, we got Claude Code set up and running. Now let's learn how to use it effectively. The first thing you need to understand is how to reference files.

Claude Code can read files from your codebase, but you need to tell it which files to look at. There are several ways to do this.

Method 1: Natural language

You can simply mention the file path in your question:"

[TYPE in Claude Code]:

```
What does the Card model in backend/app/models/card.py look like?
```

[SCREEN: Shows Claude reading and displaying the Card model]

Professor:

"Claude automatically reads the file when you mention it. But sometimes you want to be more explicit.

Method 2: Using @ symbol

Type the @ symbol and start typing a file name:"

[TYPE in Claude Code]:

```
@backend/app/models/card.py explain this model
```

[SCREEN: Shows autocomplete suggestions as typing]

Professor:

"Notice how Claude gives you autocomplete suggestions? This is really helpful when you don't remember the exact path.

Method 3: Asking Claude to find files

If you don't know where a file is, just ask:"

[TYPE in Claude Code]:

```
Find all the API route files in the backend
```

[SCREEN: Shows Claude using Glob to find route files]

Professor:

"Claude will search the codebase and show you the files. This is incredibly useful when you're new to a project.

Let me show you a practical example. Let's say I want to understand how user authentication works:"

[TYPE in Claude Code]:

```
Show me the authentication flow. Start with
@backend/app/api/routes/auth.py and explain how it connects to the
services layer.
```

[SCREEN: Shows Claude reading auth.py and then automatically reading the auth service file]

Professor:

"See how Claude automatically followed the imports and read the related files? This is smart context building."

SECTION 2: Understanding Context (2 minutes)

[SCREEN: Claude Code interface]

Professor:

"Now, here's something crucial to understand: context windows.

Claude can't read your ENTIRE codebase at once. It has a context window - think of it like short-term memory. It can hold about 200,000 tokens, which roughly translates to about 150,000 words or several hundred files.

The good news? For most tasks, you don't need the entire codebase. You need the RELEVANT parts.

Here's how Claude manages context intelligently:"

[TYPE in Claude Code]:

```
I want to add a new field to cards. What files will I need to modify?
```

[SCREEN: Shows Claude's response listing files]

Professor:

"Claude tells us we'd need to modify:

1. The Card model
2. The Pydantic schemas
3. Create a database migration
4. Possibly update the frontend types

This is context planning. Claude is telling us WHAT it needs to read before it starts making changes.

Pro tip: If you're working on a complex feature, you can give Claude context upfront:"

[TYPE in Claude Code]:

```
I'm going to work on the study session functionality. Read these files and
tell me you're ready:
- backend/app/models/study.py
- backend/app/services/study.py
- backend/app/api/routes/study.py
- frontend/src/pages/StudySessionPage.tsx
```

[SCREEN: Shows Claude confirming it has read all files]

Professor:

"Now Claude has all the relevant context loaded and can answer detailed questions about the study system without having to re-read files constantly.

However, be careful not to overload context with irrelevant files. More isn't always better. Think of it like explaining a problem to a colleague - you give them the relevant information, not everything you know."

SECTION 3: Slash Commands (2 minutes)

[SCREEN: Claude Code interface]

Professor:

"Claude Code has special slash commands that trigger specific behaviors. Let me show you the most important ones.

First, /help - Your quick reference guide:"

[TYPE in Claude Code]:

```
/help
```

[SCREEN: Shows help menu with available commands]

Professor:

"This shows you all available commands. Let me demonstrate the most useful ones.

The /init command - This is REALLY important. Run this when you first start working with a new codebase:"

[TYPE in Claude Code]:

```
/init
```

[SCREEN: Shows Claude analyzing the codebase and creating/updating CLAUDE.md]

Professor:

"What just happened? Claude analyzed our entire codebase and created a CLAUDE.md file. This file contains:

- Common development commands
- Architecture overview
- Important patterns and conventions
- Testing strategies

Let me show you what it created:"

[SCREEN: Open CLAUDE.md file in editor]

Professor:

"Look at this! It has specific commands for our project:

- How to run tests with pytest
- How to start the development server
- Database migration commands
- Frontend build commands

This becomes the 'onboarding document' for Claude. Every time you start a new conversation, Claude reads this file first and understands your project conventions.

Other useful slash commands:

/clear - Clears the conversation and starts fresh. Use this when switching to a completely different task:"

[TYPE in Claude Code]:

```
/clear
```

Professor:

"The conversation resets, but Claude still has access to your files.

There's also **/commit** for making git commits with AI-generated messages, and **/review-pr** for reviewing pull requests, but we'll cover those in later tutorials.

The key point: slash commands give you superpowers for specific development tasks."

PART B: Best Practices (6 minutes)

SECTION 4: The CLAUDE.md File (2 minutes)

[SCREEN: Split view - CLAUDE.md on left, Claude Code on right]

Professor:

"Let's talk more about this CLAUDE.md file because it's crucial for getting good results from Claude.

When we ran /init earlier, Claude created this file automatically. But you can also create and customize it yourself. Think of it as instructions you're leaving for a new developer joining your project - except that developer is an AI.

Let me show you what makes a good CLAUDE.md by looking at ours:"

[SCREEN: Scroll through CLAUDE.md highlighting sections]

Professor:

"See how it's structured?

Section 1: Repository Overview

- What is this project?
- What makes it unique?
- Key simplifications or differences

Section 2: Technology Stack

- Frontend and backend technologies
- Not just listing them, but noting what they're used for

Section 3: Common Development Commands

- This is GOLD. Specific, copy-paste ready commands

- Not 'run the tests' but 'DATABASE_URL="sqlite:///./test.db" pytest'
- This saves Claude from having to figure out your specific setup

Section 4: Architecture Overview

- How does the request flow work?
- What patterns are we using?
- This helps Claude understand the 'why' not just the 'what'

Now here's why this matters. Let me demonstrate:"

[TYPE in Claude Code]:

```
How do I run the backend tests?
```

[SCREEN: Shows Claude's response with exact command from CLAUDE.md]

Professor:

"Claude gave us the EXACT command from our CLAUDE.md file:

```
DATABASE_URL=\"sqlite:///./test.db\" pytest
```

Without CLAUDE.md, Claude might have guessed 'pytest' or 'python -m pytest', which might work but might not use our specific test database configuration.

Pro tip: As you work on the project and discover patterns or conventions, UPDATE the CLAUDE.md file. Add sections like:

- 'When adding a new API endpoint, always...'
- 'Our error handling pattern is...'
- 'Testing conventions...'

Think of it as pair programming with future-you and future-Claude."

SECTION 5: Best Practices - Don't Vibe Code (2 minutes)

[SCREEN: Claude Code interface]

Professor:

"Now let's talk about what NOT to do. I call this 'vibe coding' - when you just have a vague idea and ask Claude to 'make it work' without really understanding what you want.

Let me show you a BAD example:"

[TYPE in Claude Code]:

```
Add some cool features to make the app better
```

[SCREEN: Shows Claude asking clarifying questions]

Professor:

"See what happened? Claude is confused. 'Cool features' isn't specific. Claude will either:

1. Ask you a bunch of questions, or
2. Make assumptions that might not match what you want

This wastes time and leads to code you don't understand.

Now let me show you a GOOD example:"

[TYPE in Claude Code]:

```
I want to add a 'multiple choice' card type to the flashcard app.
Currently, we only support basic question-and-answer cards. A multiple
choice card should have:
1. A question (prompt)
2. Four answer options
3. One correct answer
4. Optional explanation shown after answering

I need to:
- Update the database model
- Modify the API to support creating these cards
- Update the frontend form
- Modify the study session to display multiple choice cards differently

Can you help me plan this implementation?
```

[SCREEN: Shows Claude giving a detailed implementation plan]

Professor:

"THIS is a good prompt. Why?

1. **Specific goal**: Multiple choice card type
2. **Context**: We currently only have basic cards
3. **Requirements**: Exactly what fields we need
4. **Scope**: What parts of the system need to change
5. **Asking for help planning**: Not just 'do it for me'

The difference is night and day. With the good prompt, Claude can:

- Verify your approach
- Suggest potential issues
- Give you a step-by-step plan

- Point out files you might have forgotten

With the vague prompt, Claude is just guessing what you want.

Remember: Claude is a TOOL, not a magic wand. The better your input, the better the output. Garbage in, garbage out applies to AI too.

Another anti-pattern to avoid:"

[TYPE in Claude Code]:

```
This doesn't work, fix it
```

Professor:

"What doesn't work? What error did you get? What did you expect to happen? What actually happened?

Instead, do this:"

[TYPE in Claude Code]:

```
I'm trying to create a new deck using the POST /api/v1/decks endpoint. I'm
getting a 422 Unprocessable Entity error. Here's the request body I'm
sending:

{
  \"name\": \"Test Deck\"
}

The error message says \"field required\" for \"description\". Looking at
backend/app/schemas/deck.py, I see the DeckCreate schema. Is description
required or optional?
```

[SCREEN: Shows Claude giving a clear answer]

Professor:

"See the difference? I provided:

- What I'm trying to do
- The specific error
- What I've already investigated
- A specific question

This lets Claude immediately help instead of playing 20 questions with you."

SECTION 6: Prompt Hygiene (2 minutes)

[SCREEN: Claude Code interface]

Professor:

"Finally, let's talk about 'prompt hygiene' - writing clean, effective prompts.

Rule 1: One task at a time

Bad:"

[TYPE in Claude Code but don't send]:

```
Add multiple choice cards and also refactor the database queries for
better performance and update the UI theme and fix that bug in the login
page
```

Professor:

"This is four different tasks! Claude will either:

- Get overwhelmed and do a mediocre job on all of them
- Focus on one and ignore the others
- Ask you to break it down

Instead, do ONE thing at a time:

- First: Add multiple choice cards
- Then: Optimize database queries
- Then: Update UI theme
- Then: Fix login bug

Each as a separate conversation or clear checkpoint.

Rule 2: Provide examples when relevant

If you want a specific format, SHOW Claude an example:"

[TYPE in Claude Code]:

```
I want to add a new API endpoint for getting a user's study statistics. It
should return data in this format:

{
  \"total_cards_studied\": 150,
  \"study_streak_days\": 7,
  \"average_session_duration_minutes\": 12,
  \"cards_due_today\": 23
}

Can you help me implement this endpoint?
```

Professor:

"By showing the exact response format you want, Claude knows exactly what to build.

Rule 3: Specify your constraints

If you have requirements or limitations, state them upfront:"

[TYPE in Claude Code]:

```
I want to add a search feature to find cards within a deck. Constraints:
- Must work with our existing PostgreSQL database
- Should be case-insensitive
- No external search services (like Elasticsearch)
- Should search both the question and answer fields
- Performance is important - we have decks with 1000+ cards

What's the best approach?
```

Professor:

"By stating constraints upfront, Claude won't suggest solutions that won't work for your situation.

Rule 4: Ask Claude to explain its reasoning

Don't just accept code blindly. Ask WHY:"

[TYPE in Claude Code]:

```
Before you implement the search feature, explain:
1. Why you chose this approach
2. What the trade-offs are
3. Are there any potential issues I should be aware of?
```

Professor:

"This does two things:

1. You learn WHY the code works, not just THAT it works
2. You catch potential issues before writing code

Remember: The goal isn't to have Claude write all your code. The goal is to learn faster and build better software. Understanding the 'why' is crucial."

CLOSING (1 minute)

[SCREEN: Back to terminal with running app]

Professor:

"Let's recap what we learned about using Claude Code effectively:

1. **Reference files** explicitly using file paths or @ mentions
2. **Manage context** by loading only relevant files
3. **Use slash commands** like /init to set up project documentation
4. **Maintain CLAUDE.md** as your project's AI guidebook
5. **Don't vibe code** - be specific about what you want
6. **Practice prompt hygiene** - one task at a time, provide examples, state constraints, ask for explanations

In our next tutorial, we'll put these skills into practice. We'll use Claude to deeply understand this flashcard application's codebase - how the authentication works, how the spaced repetition algorithm is implemented, and how the frontend and backend communicate.

The key takeaway from today: Claude Code is as effective as the questions you ask it. Learn to communicate clearly and specifically, and Claude becomes an incredible force multiplier for your development work.

See you in the next tutorial!"

INSTRUCTOR NOTES:

Timing Breakdown:

- **Part A: Core Features (6 min)**
  - Section 1 (File References): 2 min
  - Section 2 (Context): 2 min
  - Section 3 (Slash Commands): 2 min
- **Part B: Best Practices (6 min)**
  - Section 4 (CLAUDE.md): 2 min
  - Section 5 (Don't Vibe Code): 2 min
  - Section 6 (Prompt Hygiene): 2 min
- Closing: 1 min
- **Total: ~12 minutes (Can be split into two 6-minute videos if needed)**

Key Points to Emphasize:

1. File references are flexible - natural language, @ symbol, or asking Claude to find
2. Context management is about relevance, not quantity
3. CLAUDE.md is a living document that improves over time
4. Specificity in prompts leads to better results
5. Claude is a learning tool, not just a code generator

Preparation Needed:

- Have CLAUDE.md file already created (from /init command)
- Prepare examples of good vs bad prompts
- Have backend and frontend running to show live examples
- Create a "cheat sheet" slide of good prompt patterns

Common Student Questions to Address:

- Q: "How much context is too much?" A: If Claude starts getting slow or says it's hitting limits, you've added too much. Start with the core files for your task.
- Q: "Should I update CLAUDE.md manually?" A: Yes! As you discover patterns or make architectural decisions, add them. The file is yours to customize.
- Q: "What if Claude suggests something different from CLAUDE.md?" A: Ask Claude why! Maybe it has a better approach, or maybe it missed the context from CLAUDE.md.

Suggested Split Point if Making Two Videos:

- **Video 2A: Core Features** (Sections 1-3, ~6 min)
- **Video 2B: Best Practices** (Sections 4-6, ~6 min)