Using AI Tools and Python to Automate Tasks

A two-day, beginner-friendly journey to streamline workflows, boost productivity, and integrate AI into everyday tasks.

Goals

1. **Understand the basics of Python scripting without prior programming experience**

Goals

1. **Understand the basics of Python scripting without prior programming experience**
2. **Explore ways to leverage LLMs in combination with Python**

Goals

1. **Understand the basics of Python scripting without prior programming experience**
2. **Explore ways to leverage LLMs in combination with Python**
3. **Learn how to automate tasks such as file organization and data entry**

Goals

1. **Understand the basics of Python scripting without prior programming experience**
2. **Explore ways to leverage LLMs in combination with Python**
3. **Learn how to automate tasks such as file organization and data entry**
4. **Use Python scripts to integrate with external LLM APIs**

Why Learn to Automate?

Automation as a General Skill

- **Automate Repetitive Tasks**

Automation as a General Skill

- **Automate Repetitive Tasks**
- **Increase Productivity**

Automation as a General Skill

- **Automate Repetitive Tasks**
- **Increase Productivity**
- **Save Money**:
    - Replace expensive subscription tools with your own scripts

Automation as a General Skill

- **Automate Repetitive Tasks**
- **Increase Productivity**
- **Save Money**:
  - Replace expensive subscription tools with your own scripts
- **Analyse Massive Amounts of Data**

The Real Reason....

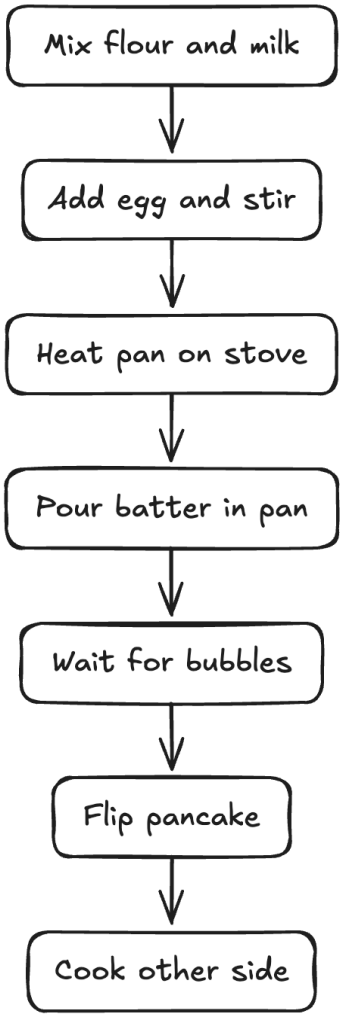
Writing code = the most general skill you can have!

It is a superpower! :)

Scripting vs. Programming

An important distinction!

Both are About Following Recipes



Scripting

- Writing code to full fill a set of pre-defined steps that accomplish a task

Scripting

- Writing code to full fill a set of pre-defined steps that accomplish a task
- Quick, targeted, and task-oriented

Scripting

- Writing code to full fill a set of pre-defined steps that accomplish a task
- Quick, targeted, and task-oriented
- Typically used for automations, data manipulation, or utility tasks

Scripting

- Writing code to full fill a set of pre-defined steps that accomplish a task
- Quick, targeted, and task-oriented
- Typically used for automations, data manipulation, or utility tasks
- Toy example -> Renaming multiple files

```
# Rename all .txt files in a directory to include today's date
import os
from datetime import date

today = date.today().strftime("%Y-%m-%d")

for filename in os.listdir("."):
    if filename.endswith(".txt"):
        new_name = f"{today}_{filename}"
        os.rename(filename, new_name)
```

Programming

- At it's core is also about writing code to perform steps but...

Programming

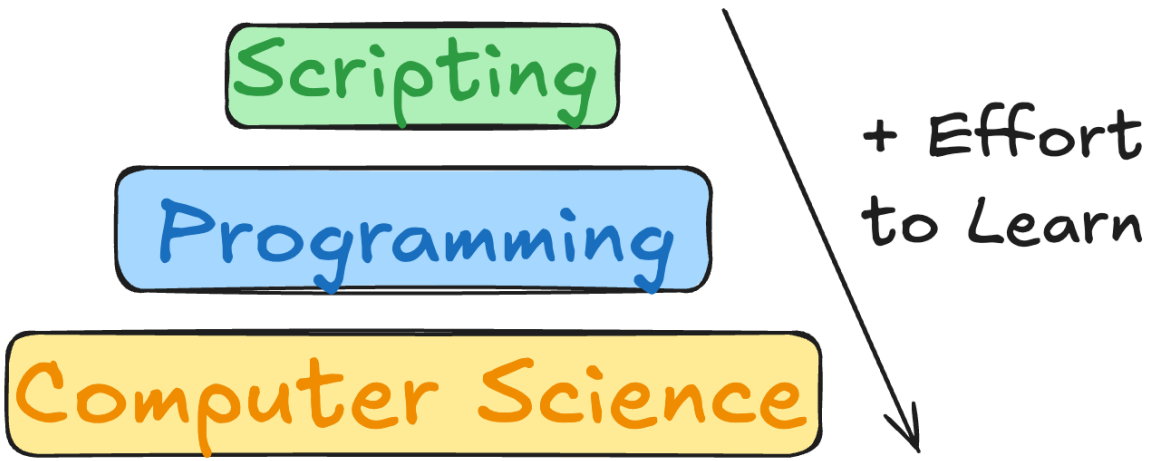
- At it's core is also about writing code to perform steps but...
- It's about developing software systems (implementing engineering practices, etc..)

Programming

- At it's core is also about writing code to perform steps but...
- It's about developing software systems (implementing engineering practices, etc..)
- Involves structure, design patterns, and often collaborative development

Programming

- At it's core is also about writing code to perform steps but...
- It's about developing software systems (implementing engineering practices, etc..)
- Involves structure, design patterns, and often collaborative development
- *Programming is considerably more effortful than scripting.*



This course focuses on Scripting!

Why focus on scripting here?

Why focus on scripting here?

- Much faster learning curve - *if it works don't change it*

Why focus on scripting here?

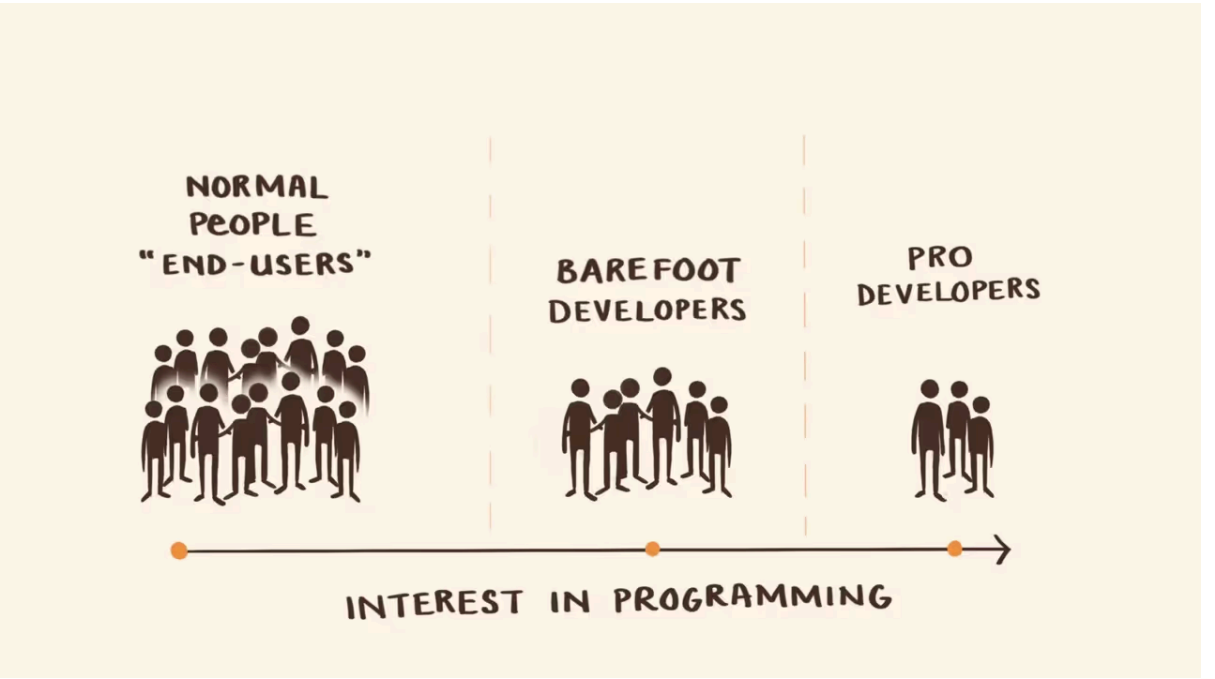
- Much faster learning curve - *if it works don't change it*
- Immediate practical benefits - *makes your life better*

Why focus on scripting here?

- Much faster learning curve - *if it works don't change it*
- Immediate practical benefits - *makes your life better*
- Ideal for nondevelopers who just want to get things done

Why focus on scripting here?

- Much faster learning curve - *if it works don't change it*
- Immediate practical benefits - *makes your life better*
- Ideal for nondevelopers who just want to get things done



Home-Cooked Software and Barefoot Developers

The Ostrich Approach to Learning Python

Ostrich algorithm

From Wikipedia, the free encyclopedia

In computer science, the **ostrich algorithm** is a strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named after the ostrich effect which is defined as "to stick one's head in the sand and pretend there is no problem". It is used when it appears the situation may be more cost-effectively managed by allowing the problem to continue to occur rather than to attempt its prevention.

It's about Learning What We Need for Our Tasks!

It's about Learning What We Need for Our Tasks!

And purposefuly ignoring that which does not seem to have any effect on its success.

Why Python?

Python is Easy and Everywhere

Python is Easy and Everywhere

- Python is a general purpose language (can be used for everything)

Python is Easy and Everywhere

- Python is a general purpose language (can be used for everything)
- Python is used across the board in AI from developing AI models to powering self-driving cars

Python is Easy and Everywhere

- Python is a general purpose language (can be used for everything)
- Python is used across the board in AI from developing AI models to powering self-driving cars
- Python is super easy to learn due to its proximity with natural language

The Busy Person Guide to Python Basics

Python Things We Will Learn

- **Basic Syntax**: Indentation, variable assignments, printing

Python Things We Will Learn

- **Basic Syntax**: Indentation, variable assignments, printing
- **Common Data Types**: Strings, integers, floats, booleans

Python Things We Will Learn

- **Basic Syntax**: Indentation, variable assignments, printing
- **Common Data Types**: Strings, integers, floats, booleans
- **Control Structures**: `if` statements, loops (`for`, `while`)

Python Things We Will Learn

- **Basic Syntax**: Indentation, variable assignments, printing
- **Common Data Types**: Strings, integers, floats, booleans
- **Control Structures**: `if` statements, loops (`for`, `while`)
- **Useful Libraries**: `os` for file operations, `csv` for handling CSVs, `requests` for making web requests

Python Things We Will Learn

- **Basic Syntax**: Indentation, variable assignments, printing
- **Common Data Types**: Strings, integers, floats, booleans
- **Control Structures**: `if` statements, loops (`for`, `while`)
- **Useful Libraries**: `os` for file operations, `csv` for handling CSVs, `requests` for making web requests
- That is a lot! How can we manage?

Python Things We Will Learn

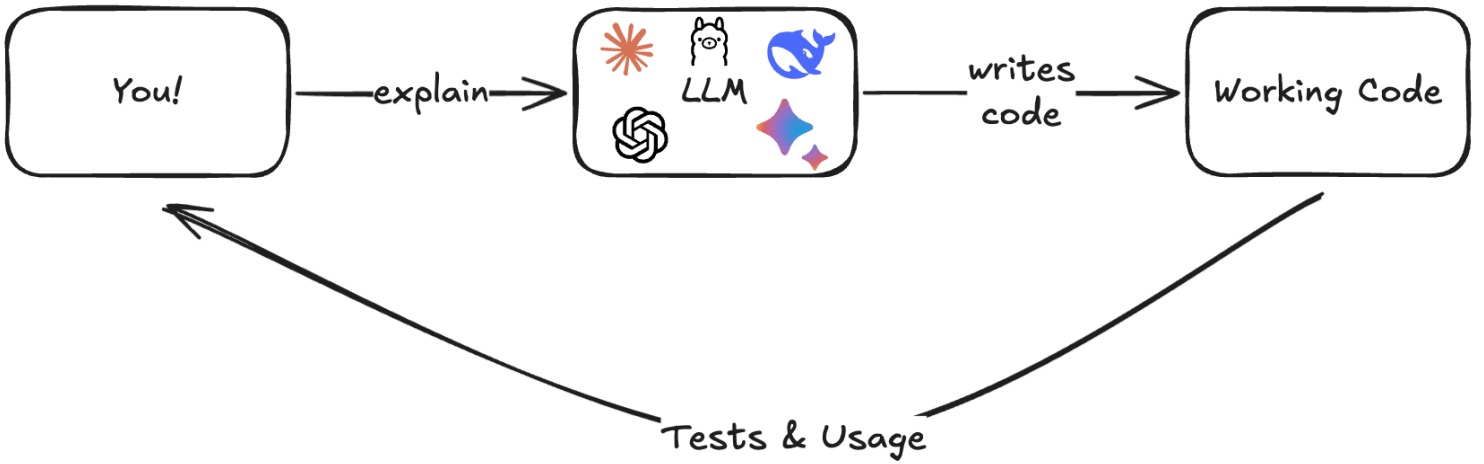
- **Basic Syntax**: Indentation, variable assignments, printing
- **Common Data Types**: Strings, integers, floats, booleans
- **Control Structures**: `if` statements, loops (`for`, `while`)
- **Useful Libraries**: `os` for file operations, `csv` for handling CSVs, `requests` for making web requests
- That is a lot! How can we manage?
- Use AI conversationally! Asking questions and clarifying what you don't know!

How to Use AI to Learn/Use Python

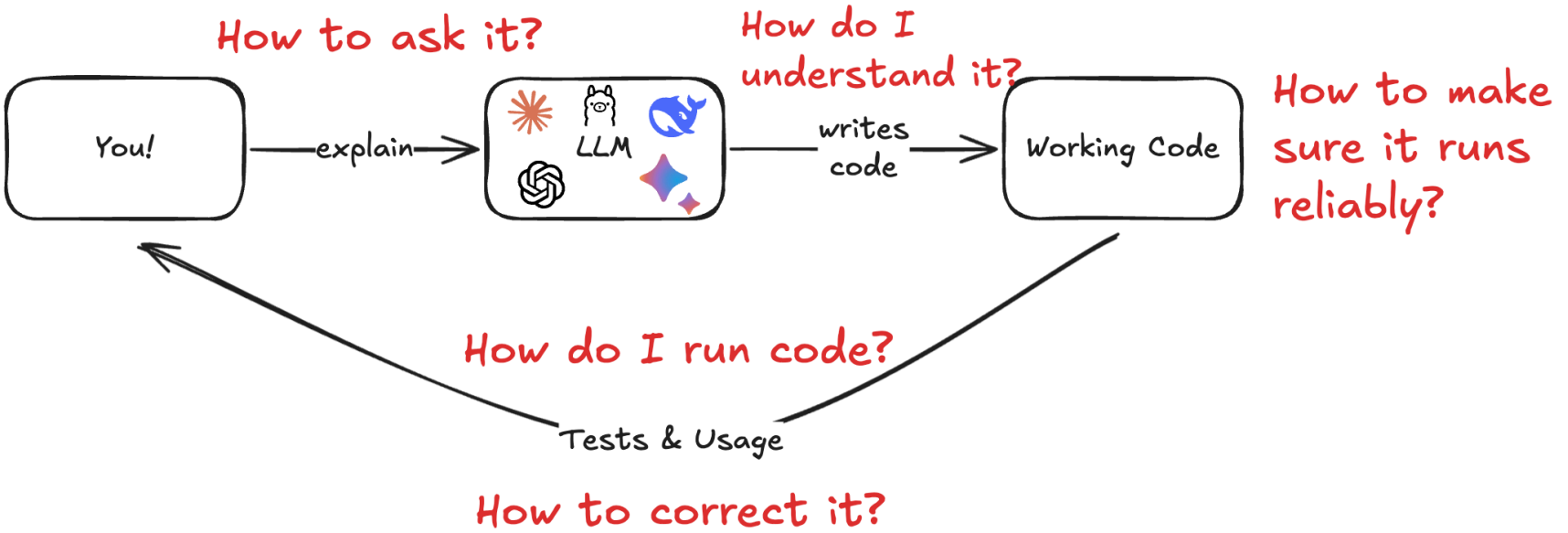
A quick detour to set up an AI toolkit to speed up our Automation skills

The "Just Ask AI" Naive Approach

The "Just Ask AI" Naive Approach



The "Just Ask AI" Naive Approach



Demo - Meta-Strategy for Scripting with AI

Data Types, Operations, Variables

- Writing code is about writing text that changes or manipulates data in some way

Data Types, Operations, Variables

- Writing code is about writing text that changes or manipulates data in some way
- For that we need? ..... (drumbroll) 
- **Data!** Specifically a way to describe it, reference it, talk about it etc...

Data Types, Operations, Variables

- Writing code is about writing text that changes or manipulates data in some way
- For that we need? ..... (drumbroll) 
- **Data!** Specifically a way to describe it, reference it, talk about it etc...
- In Python data can be of different **types** (like numbers, text, image etc...)

Data Types, Operations, Variables

- Writing code is about writing text that changes or manipulates data in some way
- For that we need? ..... (drumbroll) 
- **Data!** Specifically a way to describe it, reference it, talk about it etc...
- In Python data can be of different **types** (like numbers, text, image etc...)
- The things we can do to it are called **operations** (like 5+5)

Data Types, Operations, Variables

- Writing code is about writing text that changes or manipulates data in some way
- For that we need? ..... (drumbroll) 
- **Data!** Specifically a way to describe it, reference it, talk about it etc...
- In Python data can be of different **types** (like numbers, text, image etc...)
- The things we can do to it are called **operations** (like 5+5)
- To organize things, we use variables to define what each thing is

Data Types, Operations, Variables - Example

- Here is a piece of code that defines a variable of some type and performs a simple operation on the data stored in the variable

```
# This is data of type string!
name = "Lucas"

# This is data of type integer!
actual_age = 33

# This is data of type integer!
mental_age = 12

# our operation
average_age_between_actual_and_mental = (actual_age + mental_age) / 2

# special function that displays what goes inside of it
print(average_age_between_actual_and_mental)

# this would be a float!
# Output: 22.5
```

Data Types; Operations; Variables - Demo

Data Types, Operations, Variables

- **Core Data Types**:
  - **int**: whole nubmers like 1, 2, 3 ...
  - **float**: decimal numbers like 1.1, 1.0, etc...
  - **string**: text like `'Hello'`
  - **bool**: logical booleans like True or False

Data Types, Operations, Variables

- **Core Data Types**:
  - **int**: whole nubmers like 1, 2, 3 ...
  - **float**: decimal numbers like 1.1, 1.0, etc...
  - **string**: text like `'Hello'`
  - **bool**: logical booleans like True or False

- **Operations**:
  - **Arithmetic** (`+`,`-`,`/`,`*`,`//`,`%`)
  - **String concatenation** (`'Lucas' + ' is' + ' wonderful' = 'Lucas is wonderful'`)
  - **Logical operations** (`and`, `or`, `not`)
  - **Comparison operations** (`>`, `<`, `>=`, `<=`, `==`, `!=`)

Data Types, Operations, Variables

- **Core Data Types**:
  - **int**: whole nubmers like 1, 2, 3 ...
  - **float**: decimal numbers like 1.1, 1.0, etc...
  - **string**: text like `'Hello'`
  - **bool**: logical booleans like True or False

- **Operations**:
  - **Arithmetic** (`+`,`-`,`/`,`*`,`//`,`%`)
  - **String concatenation** (`'Lucas' + ' is' + ' wonderful' = 'Lucas is wonderful'`)
  - **Logical operations** (`and`, `or`, `not`)
  - **Comparison operations** (`>`, `<`, `>=`, `<=`, `==`, `!=`)

- **Variables**: Storing data for reuse, assigning and reassigning values:

```
a = 10
b = 20
```

Functions

Functions

- Functions are a way to group code that performs a specific task

Functions

- Functions are a way to group code that performs a specific task
- Functions can take parameters and return values

Functions

- Functions are a way to group code that performs a specific task
- Functions can take parameters and return values
- For example, a function that calculates the total cost of an item including tax

Functions

- Functions are a way to group code that performs a specific task
- Functions can take parameters and return values
- For example, a function that calculates the total cost of an item including tax
- We could do this by simply writing a script:

Functions

- Functions are a way to group code that performs a specific task
- Functions can take parameters and return values
- For example, a function that calculates the total cost of an item including tax
- We could do this by simply writing a script:

```
# Arithmetic operation: multiplication of parameters
tax = price * tax_rate
# Arithmetic operation: addition of variables
total = price + tax
```

- But what if I want to re-use this code for different prices and tax rates?

Functions

- Here is what it would look like if we define a function to do this:

```
def calculate_total(price, tax_rate):
    tax = price * tax_rate
    total = price + tax
    return total
```

Functions

- Here is what it would look like if we define a function to do this:

```
def calculate_total(price, tax_rate):
    tax = price * tax_rate
    total = price + tax
    return total
```

- Now we can re-use the function for different prices and tax rates:

```
shirt_price = 10
shirt_tax_rate = 0.05
pants_price = 20
pants_tax_rate = 0.1

print(calculate_total(shirt_price, shirt_tax_rate))
# Output: 10.5
print(calculate_total(pants_price, pants_tax_rate))
# Output: 22.0
```

Functions

- **Why Functions?**: Reuse code and break tasks into smaller chunks

Functions

- **Why Functions?**: Reuse code and break tasks into smaller chunks
- **Defining a Function**: `def function_name(parameters):`

Functions

- **Why Functions?**: Reuse code and break tasks into smaller chunks
- **Defining a Function**: `def function_name(parameters):`
- **Return Values**: Make your functions flexible and reusable

Functions

- **Why Functions?**: Reuse code and break tasks into smaller chunks
- **Defining a Function**: `def function_name(parameters):`
- **Return Values**: Make your functions flexible and reusable
- **Best Practices**: Keep them short, descriptive, and single-purpose

Demo Functions

Lists & Loops

- **Lists**: Python's go-to data structure for ordered collections

Lists & Loops

- **Lists**: Python's go-to data structure for ordered collections

```
# Define a list of tasks
tasks = ["Buy groceries", "Finish project", "Call the bank"]
```

- **Access Elements**: Indexing and slicing

Lists & Loops

- **Lists**: Python's go-to data structure for ordered collections

```
# Define a list of tasks
tasks = ["Buy groceries", "Finish project", "Call the bank"]
```

- **Access Elements**: Indexing and slicing

```
print(tasks[0])
# Output: Buy groceries

print(tasks[1])
# Output: Finish project
```

Lists & Loops

- **Lists**: Python's go-to data structure for ordered collections

```
# Define a list of tasks
tasks = ["Buy groceries", "Finish project", "Call the bank"]
```

- **Access Elements**: Indexing and slicing

```
print(tasks[0])
# Output: Buy groceries

print(tasks[1])
# Output: Finish project
```

- **Slice**: Get a range of elements

Lists & Loops

- **Lists**: Python's go-to data structure for ordered collections

```
# Define a list of tasks
tasks = ["Buy groceries", "Finish project", "Call the bank"]
```

- **Access Elements**: Indexing and slicing

```
print(tasks[0])
# Output: Buy groceries

print(tasks[1])
# Output: Finish project
```

- **Slice**: Get a range of elements

```
print(tasks[0:2])
# Output: ['Buy groceries', 'Finish project']
```

Lists & Loops

- **Loops**: `for` loops to iterate over items

Lists & Loops

- **Loops**: `for` loops to iterate over items

```
tasks = ["Buy groceries", "Finish project", "Call the bank"]

for task in tasks:
    print(task)
```

Lists & Loops

- **Loops**: `for` loops to iterate over items

```
tasks = ["Buy groceries", "Finish project", "Call the bank"]

for task in tasks:
    print(task)
```

- **Processing List Elements**: With `for` loops we can perform the same operation on each element of the list

Lists & Loops

- **Loops**: `for` loops to iterate over items

```
tasks = ["Buy groceries", "Finish project", "Call the bank"]

for task in tasks:
    print(task)
```

- **Processing List Elements**: With `for` loops we can perform the same operation on each element of the list

```
# Below we use an imaginary function that asks a robot to do a task
for task in tasks:
    ask_robot_to_do(task)
```

Lists & Loops

- **Loops**: `for` loops to iterate over items

```
tasks = ["Buy groceries", "Finish project", "Call the bank"]

for task in tasks:
    print(task)
```

- **Processing List Elements**: With `for` loops we can perform the same operation on each element of the list

```
# Below we use an imaginary function that asks a robot to do a task
for task in tasks:
    ask_robot_to_do(task)
```

Common Uses: Batch renaming files in a directory, processing data from tables, etc...

Demo - Lists & Loops

Dictionaries & Tabular Data

- **Dictionaries**: Key-value pairs for storing related data

Dictionaries & Tabular Data

- **Dictionaries**: Key-value pairs for storing related data

```
# Dictionary of product prices
prices = {
    "apple": 0.50,
    "banana": 0.75,
    "orange": 0.60
}
```

Dictionaries & Tabular Data

- **Dictionaries**: Key-value pairs for storing related data

```
# Dictionary of product prices
prices = {
    "apple": 0.50,
    "banana": 0.75,
    "orange": 0.60
}
```

- **Accessing Values**: Use keys to lookup values quickly

Dictionaries & Tabular Data

- **Dictionaries**: Key-value pairs for storing related data

```
# Dictionary of product prices
prices = {
    "apple": 0.50,
    "banana": 0.75,
    "orange": 0.60
}
```

- **Accessing Values**: Use keys to lookup values quickly

```
print(prices["apple"])
# Output: 0.50

print(prices["banana"])
# Output: 0.75
```

Dictionaries & Tabular Data

- **Dictionaries**: Key-value pairs for storing related data

```
# Dictionary of product prices
prices = {
    "apple": 0.50,
    "banana": 0.75,
    "orange": 0.60
}
```

- **Accessing Values**: Use keys to lookup values quickly

```
print(prices["apple"])
# Output: 0.50

print(prices["banana"])
# Output: 0.75
```

- **Common Uses**: Storing configurations, mapping relationships, caching data

Reading/Writing

- **Tabular Data**: Data that is organized in a table format

Reading/Writing

- **Tabular Data**: Data that is organized in a table format
- **Common Formats**: `.csv`, `.json`, `.xlsx`, `.xls`

Reading/Writing

- **Tabular Data**: Data that is organized in a table format
- **Common Formats**: `.csv`, `.json`, `.xlsx`, `.xls`
- CSV Files store data in rows and columns via comma-separated values:

```
Name,Age,City
John Smith,32,New York
Jane Doe,28,San Francisco
```

Name	Age	City
John Smith	32	New York
Jane Doe	28	San Francisco

Demo - Dictionaries & Tabular Data

Comparators & Conditionals

- **Boolean Comparisons**: ==, !=, >, <, >=, <=

Comparators & Conditionals

- **Boolean Comparisons**: ==, !=, >, <, >=, <=

```
# Example of a boolean comparison
a = 10
b = 20
print(a == b)  # Output: False
print(a != b)  # Output: True
print(a > b)   # Output: False
print(a < b)   # Output: True
print(a >= b) # Output: False
print(a <= b) # Output: True
```

Comparators & Conditionals

- **Boolean Comparisons**: `==`, `!=`, `>`, `<`, `>=`, `<=`

```
# Example of a boolean comparison
a = 10
b = 20
print(a == b)  # Output: False
print(a != b)  # Output: True
print(a > b)   # Output: False
print(a < b)   # Output: True
print(a >= b) # Output: False
print(a <= b) # Output: True
```

- **Logical Operators**: `and`, `or`, `not`

Comparators & Conditionals

- **Boolean Comparisons**: `==`, `!=`, `>`, `<`, `>=`, `<=`

```
# Example of a boolean comparison
a = 10
b = 20
print(a == b)  # Output: False
print(a != b)  # Output: True
print(a > b)   # Output: False
print(a < b)   # Output: True
print(a >= b) # Output: False
print(a <= b) # Output: True
```

- **Logical Operators**: `and`, `or`, `not`

```
# Example of a logical operator
print(True and False)  # Output: False
print(True or False)   # Output: True
print(not True)         # Output: False
```

Comparators & Conditionals

- **Conditionals**: Using conditionals `if/elif/else` to perform different actions based on data

Comparators & Conditionals

- **Conditionals**: Using conditionals `if/elif/else` to perform different actions based on data

```
# Example of an if statement
if a > b:
    print("a is greater than b")
```

Comparators & Conditionals

- **Conditionals**: Using conditionals `if/elif/else` to perform different actions based on data

```
# Example of an if statement
if a > b:
    print("a is greater than b")
```

```
# Example of an if/else statement
if a > b:
    print("a is greater than b")
else:
    print("a is less than or equal to b")
```

Comparators & Conditionals

- **Conditionals**: Using conditionals `if/elif/else` to perform different actions based on data

```
# Example of an if statement
if a > b:
    print("a is greater than b")
```

```
# Example of an if/else statement
if a > b:
    print("a is greater than b")
else:
    print("a is less than or equal to b")
```

```
# Example of an if/elif/else statement
if a > b:
    print("a is greater than b")
elif a == b:
    print("a is equal to b")
else:
    print("a is less than b")
```

Demo - Working with Data

APIs, Packages, and AI as an API

- **APIs 101**: Application Programming Interface—send a request, get a response
- **Python Packages**: Example: `requests` library to make HTTP calls
- **AI as an API**: Connect to services like OpenAI or Claude for text generation, content analysis, and more

The Ostrich Approach to AI Assistants

Automating Data Extraction

- **Target Websites or Documents**: Identify patterns or structures (tables, IDs, HTML tags)
- **Techniques**: Using Beautiful Soup, Pandas, or request-response cycles
- **Practical Examples**: Extracting data from a CSV, scraping a simple webpage for product listings

Automating Basic Data Analysis

- **Data Loading**: Reading CSV files, Excel sheets, or database tables with Pandas
- **Basic Analysis**: Calculating averages, sums, or finding patterns in data
- **Visualization**: Creating simple charts or graphs to represent findings

Automating Slides

- Effective data wrangling for high quality slides
- A Hybrid Approach: AI + Python Scripts
- Bulk Processing to save time

Automating the Browser

- **Tools**: `selenium`, `playwright`
- **Common Tasks**: Logging in to websites, navigating pages, clicking buttons, scraping dynamic elements
- **Why Automate Browser Tasks?**: Speed up online research, data entry, or repetitive website interactions

Automating Filling Out Forms

- **Form Fields**: Identifying input boxes, radio buttons, checkboxes in HTML
- **Scripts**: Using `selenium` to locate elements by ID/class/xpath and input data
- **Real-World Use**: Automating repetitive website sign-up processes, survey completion, or internal data-entry forms

AI Hacks

1. **Prompt Templates**: Keep a library of common prompt structures for repetitive tasks

AI Hacks

1. **Prompt Templates**: Keep a library of common prompt structures for repetitive tasks
2. **Chaining Tools**: Use AI outputs as inputs to other scripts or services for multi-step automations

AI Hacks

1. **Prompt Templates**: Keep a library of common prompt structures for repetitive tasks
2. **Chaining Tools**: Use AI outputs as inputs to other scripts or services for multi-step automations
3. **Hybrid Approach**: Combine AI text generation with Python logic to automate tasks like content creation or bulk editing

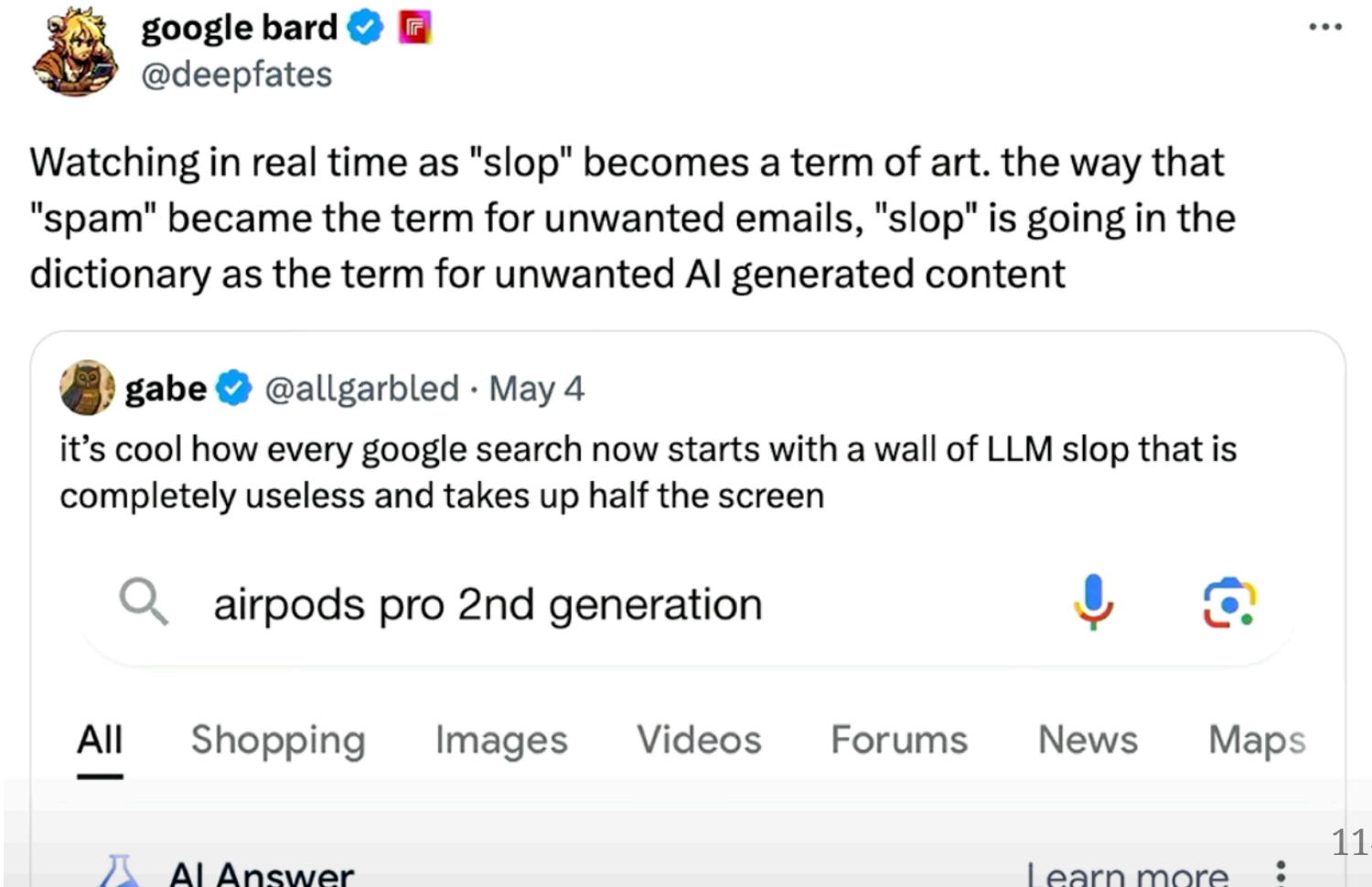
A System for Writing Your Scripts

- **Plan**: Define the scope and goal of your automation
- **Prototype**: Write a small test script or snippet to handle one core task
- **Expand**: Add new features or handle edge cases incrementally
- **Refine**: Use AI assistance to improve structure, fix bugs, and add clarity

Using AI != Slop

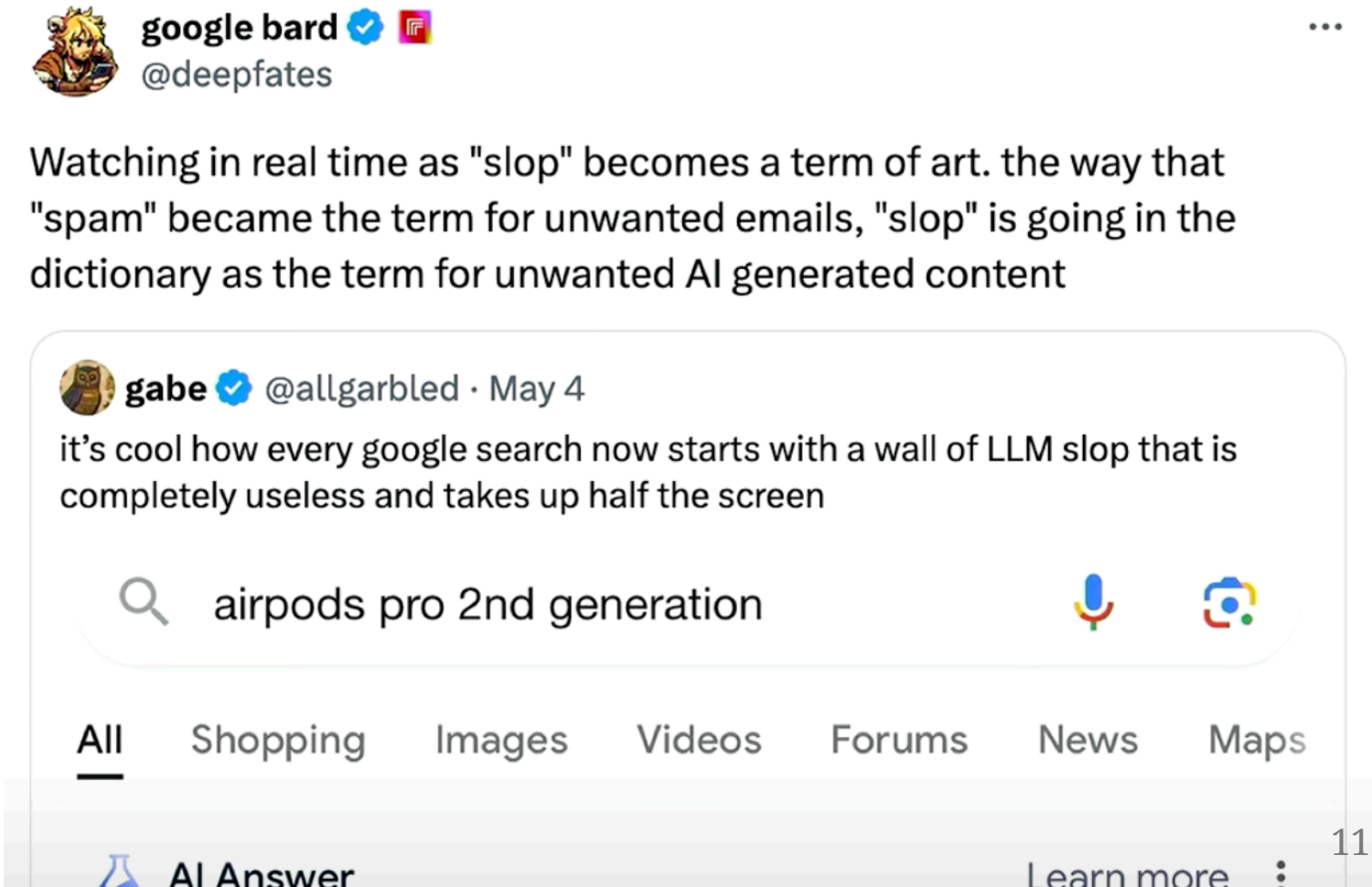
Using AI != Slop

Slop is using unreviewed code from AI models



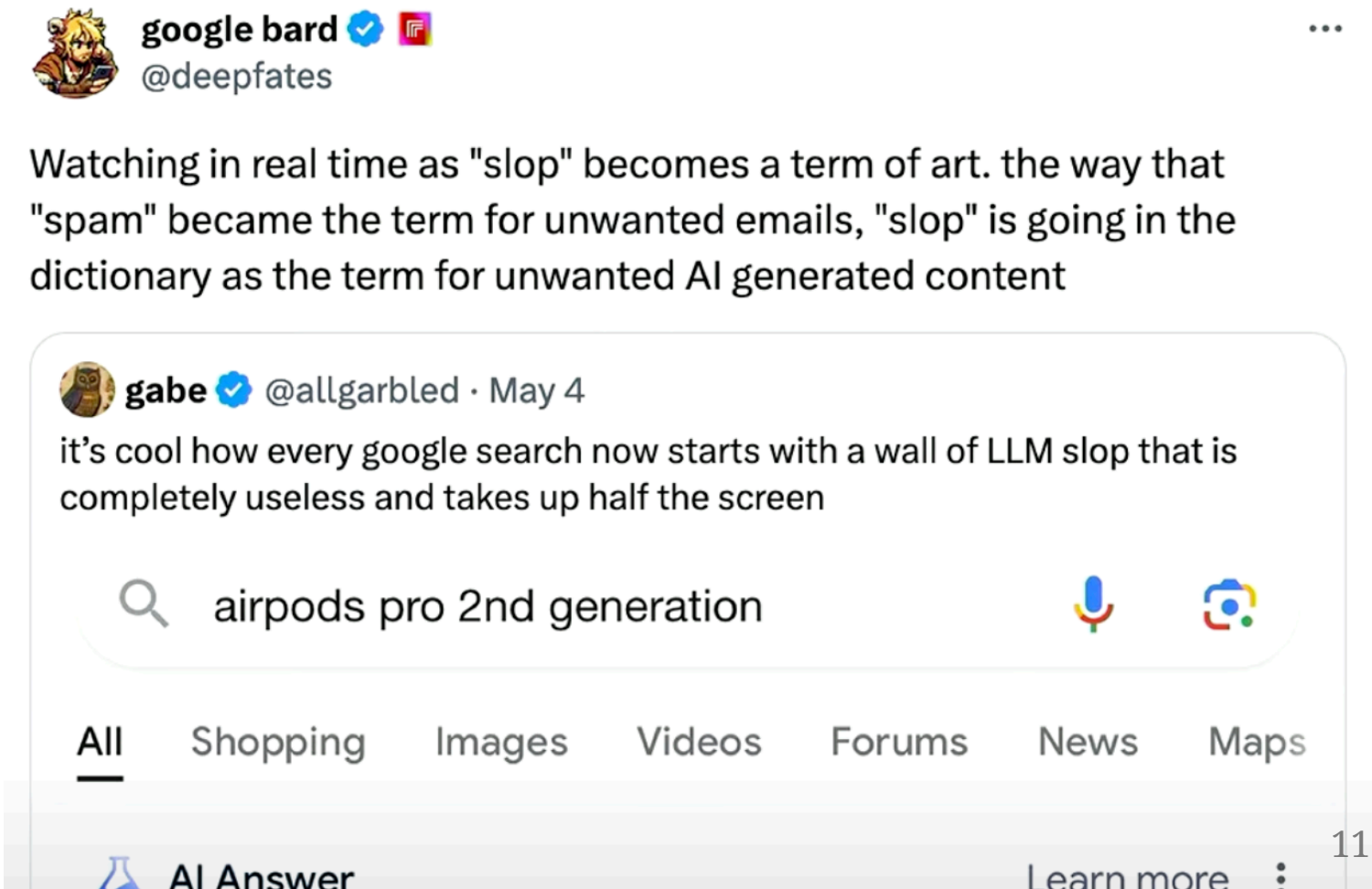
Using AI != Slop

Slop is using unreviewed code from AI models



Using AI != Slop

Slop is using unreviewed code from AI models



Using AI != Slop

Slop is using unreviewed code from AI models