AI Web Scraper (main.py) - Complete Line-by-Line Documentation

This document provides an exhaustive explanation of every line, variable, function, module, and component in the `main.py` file of the AI Web Scraper application.

Table of Contents

---

1. File Overview

File Purpose: `main.py` is a comprehensive AI-powered web scraping application that finds and compares product prices across multiple websites with built-in fraud detection and validation.

Architecture: Command-line interface with step-by-step user interaction, Google search integration, multi-layer URL validation, and AI-powered price extraction.

Key Technologies: Python asyncio, Google Gemini AI, browser automation, WHOIS validation, SSL certificate checking.

---

2. Import Statements Analysis

Standard Library Imports

```python
import sys
```

- **Purpose**: System-specific parameters and functions
- **Usage in code**:
  - `sys.stdout` - Standard output stream for logging
  - `sys.exit(1)` - Terminate program with error code
  - Keyboard interrupt handling for graceful exit

```python
import os
```

- **Purpose**: Operating system interface
- **Usage in code**:
  - `os.getenv("GEMINI_API_KEY")` - Retrieve environment variables
  - `os.makedirs(LOGS_DIR, exist_ok=True)` - Create directory structures
  - `os.path.join()` - Cross-platform file path construction

```python
import asyncio
```

- **Purpose**: Asynchronous I/O, event loop, and coroutines
- **Usage in code**:
  - `async def` - Define asynchronous functions
  - `await` - Wait for asynchronous operations
  - `asyncio.run(main())` - Run the main async function

```python
import re
```

- **Purpose**: Regular expressions for pattern matching
- **Usage in code**:
  - `re.sub(r'https?://(www\.)?', '', url)` - Remove URL prefixes
  - `re.search(r'  Result: (\d+(?:\.\d+)?)', terminal_output)` - Extract prices from logs
  - ``re.search(r'```json\n(.*?)```', response_text, re.DOTALL)`` - Parse JSON from AI responses

`import json`

- **Purpose**: JSON data serialization and deserialization
- **Usage in code**:
  - `json.loads(json_str)` - Parse JSON strings from AI responses
  - `json.dumps(json_list, indent=2, ensure_ascii=False)` - Format final results as JSON

`from urllib.parse import urlparse`

- **Purpose**: URL parsing and manipulation
- **Usage in code**:
  - `urlparse(url)` - Break URLs into components (scheme, netloc, path, etc.)
  - `url_parts.netloc` - Extract domain name from URL
  - Domain validation and sanitization

`from datetime import datetime, timedelta`

- **Purpose**: Date and time operations
- **Usage in code**:
  - `datetime.now()` - Current timestamp for logging
  - `(datetime.now() - creation_date).days` - Calculate domain age
  - `datetime.now().strftime("%Y%m%d_%H%M%S")` - Format timestamps for filenames

`import io`

- **Purpose**: Core tools for working with streams
- **Usage in code**:
  - `io.StringIO()` - In-memory string buffer for capturing log output
  - `log_capture_stream.getvalue()` - Retrieve captured log content

`from contextlib import redirect_stdout, redirect_stderr`

- **Purpose**: Context managers for redirecting output streams
- **Usage in code**: Imported but not actively used (legacy import for potential output redirection)

`import logging`

- **Purpose**: Logging facility for Python applications
- **Usage in code**:
  - `logging.basicConfig()` - Configure logging format and handlers
  - `logging.info()`, `logging.error()` - Log messages at different levels
  - `logging.getLogger('browser_use')` - Get specific logger for browser automation

Third-Party Library Imports

`from dotenv import load_dotenv`

- **Module**: python-dotenv
- **Purpose**: Load environment variables from .env files
- **Usage in code**: `load_dotenv()` - Load API keys and configuration from .env file

`from browser_use.llm import ChatGoogle`

- **Module**: browser-use
- **Purpose**: Google Gemini AI integration for browser automation
- **Usage in code**:
  - `ChatGoogle(model="gemini-2.5-flash", api_key=gemini_api_key)` - Initialize AI model
  - `llm.achat(messages)` - Send messages to AI for analysis

`from browser_use import Agent`

- **Module**: browser-use
- **Purpose**: AI-powered browser automation agent
- **Usage in code**:
  - `Agent(task=task, llm=llm, max_loops=8)` - Create browser automation agent
  - `await agent.run()` - Execute browser automation tasks

`import pandas as pd`

- **Module**: pandas
- **Purpose**: Data manipulation and analysis
- **Usage in code**:
  - `pd.DataFrame(display_list)` - Create structured data tables
  - `df.set_index()`, `df.to_string()` - Format and display results

```
from IPython.display import display
```

- **Module**: IPython
- **Purpose**: Rich display system for notebooks and enhanced terminals
- **Usage in code**: `display(df)` - Enhanced table display (with fallback to standard print)

```
import pycountry
```

- **Module**: pycountry
- **Purpose**: ISO country, subdivision, language, currency and script definitions
- **Usage in code**:
  - `pycountry.countries` - Access country database
  - Generate country selection list for user interface

```
from googlesearch import search
```

- **Module**: googlesearch-python
- **Purpose**: Perform Google searches programmatically
- **Usage in code**:
  - `search(product_query, num_results=NUM_SEARCH_RESULTS, ...)` - Find product URLs

```
import whois
```

- **Module**: python-whois
- **Purpose**: WHOIS protocol client for domain information
- **Usage in code**:
  - `whois.whois(domain)` - Get domain registration information
  - `domain_info.creation_date` - Extract domain creation date

```
import ssl
```

- **Module**: Standard library SSL/TLS wrapper
- **Purpose**: SSL/TLS certificate validation
- **Usage in code**:
  - `ssl.create_default_context()` - Create secure SSL context
  - Certificate validation for domain security checks

```
import socket
```

- **Module**: Standard library low-level networking interface
- **Purpose**: Network connections and socket operations
- **Usage in code**:
  - `socket.create_connection((domain, 443), timeout=5)` - Test HTTPS connectivity
  - SSL certificate verification

---

3. Configuration Variables

Search and Processing Settings

```
NUM_SEARCH_RESULTS = 2
```

- **Type**: `int`
- **Purpose**: Limits the number of URLs to process for performance optimization
- **Impact**: Controls Google search results and processing time
- **Reasoning**: Reduced from higher numbers to balance thoroughness with execution speed

```
MINIMUM_DOMAIN_AGE_DAYS = 365
```

- **Type**: `int`
- **Purpose**: Minimum age requirement for domains to be considered trustworthy

- **Security Impact**: Filters out newly created domains that might be fraudulent
- **Calculation**: Domains must be at least 1 year old to pass validation

Security and Fraud Prevention

```
HIGH_RISK_TLDS = ['.zip', '.top', '.xyz', '.club', '.online', '.loan', '.work', '.gq', '.cf', '.tk']
```

- **Type**: `List[str]`
- **Purpose**: Top-level domains commonly associated with fraudulent websites
- **Security Logic**: These TLDs are often cheap/free and used by scammers
- **Validation Impact**: URLs with these TLDs are automatically rejected

```
DOMAIN_BLACKLIST = [
    'youtube.com', 'youtu.be', 'wikipedia.org', 'facebook.com', 'twitter.com', 'instagram.com',
    'pinterest.com', 'linkedin.com', 'reddit.com', 'quora.com', 'google.com', 'amazon.com'
]
```

- **Type**: `List[str]`
- **Purpose**: Domains to exclude from price scraping
- **Categories**:
  - **Social Media**: youtube.com, facebook.com, twitter.com, instagram.com, pinterest.com, linkedin.com
  - **Information Sites**: wikipedia.org, quora.com
  - **Search Engines**: google.com
  - **Complex E-commerce**: amazon.com (too complex for simple scraping)
- **Reasoning**: These sites don't contain direct product pricing or are too complex to scrape reliably

```
EXCLUSION_KEYWORDS = ['review', 'news', 'vs', 'compare']
```

- **Type**: `List[str]`
- **Purpose**: URL keywords that indicate non-shopping content
- **Logic**: URLs containing these keywords likely contain reviews/news rather than product listings
- **Filter Impact**: Helps focus on actual shopping websites

---

4. Helper Functions

Domain Age Validation

```
def check_domain_age(domain):
```

Function Purpose: Validates if a domain is old enough to be trustworthy

Parameters:

- `domain` (str): Domain name without protocol (e.g., "example.com")

Return Values:

- `Tuple[bool, str]`: (is_valid, message)
  - `True, "X days old"` if domain is older than MINIMUM_DOMAIN_AGE_DAYS
  - `False, "Unknown Age"` if creation date cannot be determined
  - `False, "WHOIS Error"` if WHOIS lookup fails

Implementation Details:

```
try:
    domain_info = whois.whois(domain)
```

- **Action**: Performs WHOIS lookup to get domain registration information
- **Error Handling**: Catches any network, DNS, or parsing errors

```
creation_date = domain_info.creation_date
if isinstance(creation_date, list): creation_date = creation_date[0]
```

- **Logic**: WHOIS can return single date or list of dates
- **Handling**: Takes first date if multiple dates returned (common for domains with multiple registrations)

```
if creation_date is None: return False, "Unknown Age"
```

- **Edge Case**: Some domains may not have accessible creation dates
- **Response**: Treats unknown age as failure for security

```
age = (datetime.now() - creation_date).days
return age > MINIMUM_DOMAIN_AGE_DAYS, f"{age} days old"
```

- **Calculation**: Subtracts creation date from current date to get age in days
- **Validation**: Compares against MINIMUM_DOMAIN_AGE_DAYS threshold
- **Return**: Boolean result with descriptive message

URL Structure Validation

```
def check_url_structure(url_parts):
```

Function Purpose: Analyzes URL structure for suspicious patterns

Parameters:

- `url_parts`: Parsed URL object from `urlparse()`

Return Values:

- `Tuple[bool, str]`: (is_valid, reason)

Validation Checks:

```
if any(url_parts.netloc.endswith(tld) for tld in HIGH_RISK_TLDS):
    return False, "High-Risk TLD"
```

- **Check**: Tests if domain ends with any high-risk top-level domain
- **Logic**: Uses `any()` with generator expression for efficient checking
- **Security**: Immediately rejects domains with suspicious TLDs

```
if len(url_parts.netloc.split('.')) > 4:
    return False, "Excessive Subdomains"
```

- **Check**: Counts number of domain parts (subdomains)
- **Logic**: Splits domain by '.' and counts parts
- **Example**: `shop.products.store.example.com` would have 5 parts (suspicious)
- **Security**: Many subdomains can indicate phishing attempts

```
return True, "Looks OK"
```

- **Default**: If all checks pass, domain structure is acceptable

SSL Certificate Validation

```
def check_ssl_certificate(domain):
```

Function Purpose: Verifies SSL/TLS certificate validity and HTTPS support

Parameters:

- `domain` (str): Domain name to check

Return Values:

- `Tuple[bool, str]`: (has_valid_ssl, message)

Implementation Details:

```
try:
    context = ssl.create_default_context()
```

- **Action**: Creates SSL context with default security settings
- **Security**: Uses system's trusted certificate authorities

```
with socket.create_connection((domain, 443), timeout=5) as sock:
```

- **Action**: Opens TCP connection to domain on port 443 (HTTPS)

- **Timeout**: 5-second limit to prevent hanging on slow connections
- **Context Manager**: Ensures connection is properly closed

```
with context.wrap_socket(sock, server_hostname=domain) as ssock:
```

- **Action**: Wraps TCP socket with SSL/TLS encryption
- **Validation**: Automatically validates certificate against domain name
- **Context Manager**: Ensures SSL socket is properly closed

```
cert = ssock.getpeercert()
return bool(cert), "Valid Certificate"
```

- **Action**: Retrieves certificate information from server
- **Validation**: `bool(cert)` checks if certificate was successfully obtained
- **Success**: Returns True with descriptive message

```
except Exception:
    return False, "SSL/Socket Error"
```

- **Error Handling**: Catches all SSL, socket, and network errors
- **Response**: Treats any error as certificate validation failure
- **Security**: Fail-safe approach for security validation

---

5. Core Logic Functions

User Input Handling

```
def get_user_input():
```

Function Purpose: Interactive user interface for collecting product search parameters

Return Values:

- `Tuple[str, str, str]`: (product_query, country_code, country_name)

Implementation Breakdown:

```
print("--- Please provide the product details ---")
product_query = input("Enter the product you want to search for: ")
```

- **Action**: Prompts user for product search terms
- **Input**: Free-form text (e.g., "iPhone 15 Pro 256GB")
- **Storage**: Raw user input stored in `product_query`

```
countries_data = sorted([{'name': c.name, 'code': c.alpha_2} for c in pycountry.countries], key=lambda x: x['name'])
```

- **Data Source**: `pycountry.countries` - ISO country database
- **Processing**: List comprehension creates dictionaries with name and 2-letter code
- **Sorting**: Alphabetical by country name for user convenience
- **Structure**: `[{'name': 'Afghanistan', 'code': 'AF'}, ...]`

```
print("\n--- Please select a country from the list below ---")
for i, country in enumerate(countries_data):
    print(f"{i+1:<4} - {country['name']} ({country['code']})")
```

- **Display**: Numbered list of all countries with codes
- **Formatting**: `{i+1:<4}` left-aligns numbers in 4-character field
- **Information**: Shows both full name and ISO code for clarity

```
selected_country = None
while True:
```

- **Loop**: Continues until valid country selection
- **Initialization**: `selected_country` starts as None to track selection state

```
try:
    choice_str = input(f"\n Enter the number for your desired country (1-{len(countries_data)}): ")
    choice_num = int(choice_str)
```

- **Input**: User enters number corresponding to country
- **Conversion**: String input converted to integer
- **Error Potential**: `int()` can raise `ValueError` for invalid input

```
if 1 <= choice_num <= len(countries_data):
    selected_country = countries_data[choice_num - 1]
    break
```

- **Validation**: Ensures number is within valid range
- **Selection**: Uses 0-based indexing (user sees 1-based numbers)
- **Exit**: `break` exits the while loop on successful selection

```
else:
    print(f"  Error: Please enter a number between 1 and {len(countries_data)}.")
```

- **Error Handling**: Clear message for out-of-range numbers
- **User Experience**: Specific range information helps user correct input

```
except (ValueError, IndexError):
    print("  Error: Invalid input. Please enter a valid number.")
```

- **ValueError**: Handles non-numeric input (e.g., "abc")
- **IndexError**: Handles edge cases in list access
- **User Experience**: Generic error message for invalid input

```
except KeyboardInterrupt:
    print("\nOperation cancelled by user.")
    sys.exit()
```

- **Graceful Exit**: Handles Ctrl+C keyboard interrupt
- **User Experience**: Friendly message instead of error traceback
- **Clean Exit**: `sys.exit()` terminates program normally

```
logging.info("\n" + "="*50)
logging.info("  INPUT CAPTURED SUCCESSFULLY")
logging.info("="*50)
logging.info(f"  -> Product Query: '{product_query}'")
logging.info(f"  -> Search Location: {selected_country['name']} ({selected_country['code']})")
return product_query, selected_country['code'], selected_country['name']
```

- **Logging**: Records successful input capture with formatting
- **Information**: Logs both product and location for debugging
- **Return**: Tuple with product query, country code, and country name

URL Validation Pipeline

```
def validate_search_results(product_query, country_code):
```

Function Purpose: Performs Google search and validates URLs through multi-layer security funnel

Parameters:

- `product_query` (str): Product search terms from user
- `country_code` (str): ISO country code for localized search

Return Values:

- `List[str]`: List of validated, trustworthy URLs

Google Search Implementation:

```
print(f"\n  Starting search for '{product_query}' in region: {country_code.upper()}...")
```

- **User Feedback**: Informs user of search initiation
- **Information**: Shows search terms and target region

```
try:
    search_results = search(
        product_query,
        num_results=NUM_SEARCH_RESULTS,
        lang='en',
        region=country_code.lower(),
        advanced=True
    )
```

- **search()**: Google search function from googlesearch library
- **Parameters**:
  - `product_query`: User's search terms
  - `num_results`: Limits results (from NUM_SEARCH_RESULTS config)
  - `lang='en'`: Forces English results for consistency
  - `region=country_code.lower()`: Localizes results to user's country
  - `advanced=True`: Returns result objects with metadata

```
raw_urls = [result.url for result in search_results]
logging.info(f" Google search found {len(raw_urls)} potential URLs.")
```

- **Extraction**: Gets URL from each result object
- **Logging**: Records number of URLs found for debugging

```
except Exception as e:
    logging.error(f"\n An error occurred during Google Search: {e}")
    return []
```

- **Error Handling**: Catches all Google search errors (rate limiting, network, etc.)
- **Response**: Returns empty list if search fails
- **Logging**: Records error details for debugging

URL Validation Funnel:

```
print(f"\n2. Validating URLs through the Trust Funnel...")
validated_urls = []
checked_domains = set()
```

- **User Feedback**: Informs user of validation phase
- **Storage**: `validated_urls` collects passing URLs
- **Deduplication**: `checked_domains` prevents processing same domain twice

```
for url in raw_urls:
    try:
        url_parts = urlparse(url)
        domain = url_parts.netloc.replace('www.', '')
```

- **Processing**: Parse each URL into components
- **Domain Extraction**: Get domain name, removing 'www.' prefix
- **Error Handling**: Individual URL processing wrapped in try/except

```
if not domain or domain in checked_domains:
    continue
checked_domains.add(domain)
```

- **Validation**: Skip if domain is empty or already processed
- **Deduplication**: Add domain to checked set
- **Efficiency**: Prevents redundant processing of same domain

Security Filter Chain:

```
if any(blacklisted in domain for blacklisted in DOMAIN_BLACKLIST):
    continue
```

- **Blacklist Check**: Rejects domains in DOMAIN_BLACKLIST
- **Logic**: `any()` with generator expression for efficiency
- **Purpose**: Excludes social media, news sites, complex e-commerce

```
if any(keyword in url.lower() for keyword in EXCLUSION_KEYWORDS):
    continue
```

- **Keyword Filter**: Rejects URLs containing review/news keywords
- **Case Insensitive**: `.lower()` ensures consistent matching
- **Purpose**: Focuses on shopping sites, not review sites

```
if not url.startswith('https://'):
    continue
```

- **Security Check**: Requires HTTPS for secure transactions
- **Purpose**: Ensures encrypted communication for shopping sites

```
is_structured_ok, _ = check_url_structure(url_parts)
if not is_structured_ok:
    continue
```

- **Structure Validation**: Calls helper function for URL analysis
- **Checks**: High-risk TLDs, excessive subdomains
- **Purpose**: Identifies suspicious URL patterns

```
is_old_enough, _ = check_domain_age(domain)
if not is_old_enough:
    continue
```

- **Age Validation**: Ensures domain is older than minimum threshold
- **Security**: New domains more likely to be fraudulent
- **Purpose**: Filters out recently created scam sites

```
is_cert_valid, _ = check_ssl_certificate(domain)
if not is_cert_valid:
    continue
```

- **SSL Validation**: Verifies valid SSL certificate
- **Security**: Ensures encrypted, authenticated connection
- **Purpose**: Confirms legitimate HTTPS implementation

```
validated_urls.append(url)
```

- **Success**: URL passed all security checks
- **Storage**: Added to list of trusted URLs for scraping

```
except Exception:
    continue
```

- **Error Handling**: Skip URL if any validation step fails
- **Robustness**: Prevents single bad URL from breaking entire process

```
logging.info(f" Validation complete. Found {len(validated_urls)} high-trust URLs.")
return validated_urls
```

- **Completion**: Log final count of validated URLs
- **Return**: List of URLs that passed all security checks

---

6. AI Integration Functions

Log Analysis with Large Language Model

```
async def analyze_log_with_llm(terminal_log: str, product_query: str, llm: ChatGoogle):
```

Function Purpose: Uses Google Gemini AI to analyze browser automation logs and extract price/availability information

Parameters:

- `terminal_log` (str): Raw terminal output from browser automation agent
- `product_query` (str): Original product search query for context
- `llm` (ChatGoogle): Initialized Google Gemini AI model instance

Return Values:

- `Dict[str, Any]`: JSON object with 'price' (float or None) and 'status' (str)

AI Prompt Engineering:

```python
print(" Sending terminal log to Gemini for final analysis...")
```

- **User Feedback**: Informs user that AI analysis is starting

```python
analysis_prompt = f"""
You are a data analyst. Your task is to analyze the provided terminal log from a web scraping agent.
The agent's goal was to find the price for the product: "{product_query}".

Based on the log, determine the final outcome. Answer in a JSON format with two keys: 'price' and 'status'.
- 'price': Should be a float number (e.g., 1399.00). If the price is not found or the product is unavailable, set it to null.
- 'status': Should be a string. Possible values are "Available", "Out of Stock", "Not Found", or "Scraping Error".

Look for patterns like:
- "  Result: 119900" or "  Result: 107900" (extract the number)
- "  Task completed successfully" (indicates success)
- "Out of Stock" or "Not Found" (indicates unavailable)
- Any price extraction or currency symbols

Here is the terminal log:
--- LOG START ---
{terminal_log}
--- LOG END ---

Provide only the JSON object in your response.
"""
```

Prompt Analysis:

- **Role Definition**: “You are a data analyst” - Sets AI’s operational context
- **Task Specification**: Clear objective to analyze scraping logs
- **Context Provision**: Includes original product query for relevance
- **Output Format**: Specifies exact JSON structure required
- **Field Definitions**:
  - `price`: Float or null, clear data type specification
  - `status`: Enumerated string values for consistency
- **Pattern Examples**: Specific regex-like patterns the AI should recognize
- **Input Framing**: Clear delimiters for log content
- **Response Constraint**: “Provide only the JSON” prevents extra text

AI Communication:

```python
try:
    messages = [{"role": "user", "content": analysis_prompt}]
    response = await llm.achat(messages)
```

- **Message Format**: OpenAI-compatible chat format
- **Async Call**: `await llm.achat()` for non-blocking AI communication
- **Error Handling**: Wrapped in try/except for robust error management

```python
response_text = response.content if hasattr(response, 'content') else str(response)
```

- **Response Extraction**: Handles different response object formats
- **Compatibility**: Works with various AI library versions
- **Fallback**: `str(response)` if no content attribute

Response Processing:

```python
json_match = re.search(r'```json\n(.*?)```', response_text, re.DOTALL)
if json_match:
    json_str = json_match.group(1).strip()
else:
    json_str = response_text.strip()
```

- **Markdown Handling**: Extracts JSON from code blocks if present
- **Regex Pattern**: r'```json\n(.*?)```' matches markdown code blocks
- **DOTALL Flag**: Allows matching across multiple lines
- **Fallback**: Uses raw response if no code block detected
- **Cleanup**: `.strip()` removes whitespace

```
result = json.loads(json_str)
print(f" Gemini analysis complete: {result}")
return result
```

- **JSON Parsing**: Converts string to Python dictionary
- **User Feedback**: Shows AI analysis result
- **Return**: Parsed JSON object with price and status

Error Handling:

```
except Exception as e:
    print(f" Error during Gemini log analysis: {e}")
    return {"price": None, "status": "Analysis Error"}
```

- **Comprehensive Catch**: Handles JSON parsing, network, AI model errors
- **User Feedback**: Clear error message with details
- **Fallback Response**: Standard error format maintains system consistency

---

7. Web Scraping Functions

AI-Powered Price Scraping

```
async def scrape_prices_with_agent(product_query, urls, llm):
```

Function Purpose: Orchestrates AI browser automation to extract prices from validated URLs

Parameters:

- `product_query` (str): Original product search terms
- `urls` (List[str]): List of validated, trustworthy URLs to scrape
- `llm` (ChatGoogle): Initialized Google Gemini AI model

Return Values:

- `List[Dict[str, Any]]`: List of scraping results with price, status, and metadata

Initialization and Setup:

```
scraped_data = []
total_urls = len(urls)
MODEL_OUTPUT_DIR = os.path.join("logs", "model_output")
COMPLETE_VIEW_DIR = os.path.join("logs", "complete_view")
os.makedirs(MODEL_OUTPUT_DIR, exist_ok=True)
os.makedirs(COMPLETE_VIEW_DIR, exist_ok=True)
```

- **Data Storage**: `scraped_data` accumulates results from all URLs
- **Progress Tracking**: `total_urls` for user feedback
- **Log Directories**:
  - `MODEL_OUTPUT_DIR`: Stores detailed agent terminal logs
  - `COMPLETE_VIEW_DIR`: Stores complete agent execution history
- **Directory Creation**: `exist_ok=True` prevents errors if directories exist

URL Processing Loop:

```
for i, url in enumerate(urls, 1):
    logging.info("\n" + "-"*60)
    logging.info(f" Processing URL {i}/{total_urls}: {url}")
```

- **Enumeration**: `enumerate(urls, 1)` provides 1-based indexing for user-friendly progress
- **Progress Logging**: Shows current URL and position in queue

- **Formatting**: Dashed line separators for log readability

Log File Naming:

```
sanitized_url = re.sub(r'https?://(www\.)?', '', url).replace('/', '_').replace(':', '_').replace('?', '_').replace('=', '_').replace('&', '_')[:100]
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
log_base_name = f"{sanitized_url}_{timestamp}"
model_output_log_path = os.path.join(MODEL_OUTPUT_DIR, f"{log_base_name}.log")
complete_view_log_path = os.path.join(COMPLETE_VIEW_DIR, f"{log_base_name}.log")
```

- **URL Sanitization**:
  - `re.sub(r'https?://(www\.)?', '', url)`: Removes protocol and www prefix
  - `.replace()` chain: Converts URL special characters to underscores
  - `[:100]`: Limits filename length to prevent OS path length errors
- **Timestamp**: `strftime("%Y%m%d_%H%M%S")` creates unique identifier
- **File Paths**: Constructs absolute paths for both log types

AI Task Definition:

```
task = f"""
Go to the website: {url}
Your task is to find the price for the product: "{product_query}".
1. Navigate the page. If there are product options (e.g., color, storage, model), select the ones that best match the query.
2. Find the final price.
3. If the product is out of stock, unavailable, or a price cannot be found, state that clearly.
Your final response must be concise. Return ONLY the price as a number (e.g., '1099.99') OR the status 'Out of Stock' or 'Not Found'.
Do not include currency symbols or any other text.
"""
```

Task Prompt Analysis:

- **Target Specification**: Clear URL and product identification
- **Navigation Instructions**: Handles product variants and options
- **Price Extraction**: Focuses on final, user-facing price
- **Error Cases**: Explicit handling of stock and availability issues
- **Output Format**: Strict format requirements for easy parsing
- **Constraint**: Prohibits extra text that could complicate parsing

Logging Configuration:

```
log_capture_stream = io.StringIO()
agent_logger = logging.getLogger('browser_use')
stream_handler = logging.StreamHandler(log_capture_stream)
original_handlers = agent_logger.handlers[:]
original_level = agent_logger.level
agent_logger.addHandler(stream_handler)
agent_logger.setLevel(logging.INFO)
```

- **Log Capture**: `io.StringIO()` creates in-memory buffer for logs
- **Logger Access**: Gets specific logger used by browser automation library
- **Handler Management**:
  - Saves original handlers and level for restoration
  - Adds new handler to capture logs
  - Sets INFO level to capture detailed output

AI Agent Execution:

```
terminal_output = ""
raw_agent_output = None
agent_result_obj = None

try:
    agent = Agent(task=task, llm=llm, max_loops=8)
    agent_result_obj = await agent.run()
```

- **Variable Initialization**: Prepares storage for agent results
- **Agent Creation**:
  - `task`: AI instructions for browser automation
  - `llm`: Google Gemini model for decision making
  - `max_loops=8`: Limits agent iterations to prevent infinite loops
- **Execution**: `await agent.run()` performs actual browser automation

Result Extraction:

```
if agent_result_obj:
    if hasattr(agent_result_obj, 'final_result'):
        raw_agent_output = agent_result_obj.final_result
    elif hasattr(agent_result_obj, 'result'):
        raw_agent_output = agent_result_obj.result
    elif hasattr(agent_result_obj, 'last_result'):
        raw_agent_output = agent_result_obj.last_result
```

- **Result Handling**: Different agent versions may store results in different attributes
- **Attribute Checking**: `hasattr()` safely checks for attribute existence
- **Priority Order**: Checks most likely attribute names first

Complex Result Extraction:

```
elif hasattr(agent_result_obj, '__iter__') and not isinstance(agent_result_obj, str):
    try:
        results = list(agent_result_obj)
        if results:
            for result in reversed(results):
                if hasattr(result, 'extracted_content'):
                    raw_agent_output = result.extracted_content
                    break
                elif hasattr(result, 'long_term_memory'):
                    if 'Result:' in str(result.long_term_memory):
                        raw_agent_output = result.long_term_memory
                        break
    except:
        pass
```

- **Iterable Handling**: Processes result objects that contain multiple results
- **Reverse Iteration**: Starts from most recent results
- **Content Search**: Looks for specific result attributes
- **Memory Search**: Checks long-term memory for result patterns
- **Error Safety**: Nested try/except prevents processing failures

Fallback and Error Handling:

```
else:
    raw_agent_output = str(agent_result_obj)
else:
    raw_agent_output = "No result returned"

print(f"  Agent task finished. Raw Output: '{raw_agent_output}'")
```

- **String Conversion**: Last resort - convert entire result to string
- **Null Handling**: Manages case where agent returns nothing
- **User Feedback**: Shows extracted result to user

Agent Execution Error Handling:

```
except Exception as e:
    print(f"  Agent execution failed for {url}: {e}")
    log_capture_stream.write(f"\n--- AGENT EXECUTION FAILED ---\nURL: {url}\nERROR: {e}\n")
    raw_agent_output = f"Agent Execution Error: {e}"
```

- **Error Display**: Shows user which URL failed and why

- **Log Recording**: Writes error details to log capture stream
- **Error Result**: Creates standardized error output

Log Management and Cleanup:

```
finally:
    agent_logger.handlers[:] = original_handlers
    agent_logger.setLevel(original_level)
    terminal_output = log_capture_stream.getvalue()

    with open(model_output_log_path, 'w', encoding='utf-8') as f:
        f.write(terminal_output)
    print(f"  Verbose agent logs saved to: {model_output_log_path}")

    if agent_result_obj:
        with open(complete_view_log_path, 'w', encoding='utf-8') as f:
            f.write(str(agent_result_obj))
        print(f"  Complete agent history saved to: {complete_view_log_path}")
```

- **Logger Restoration**: Returns logger to original state
- **Log Extraction**: Gets all captured log content
- **File Writing**:
  - `encoding='utf-8'`: Ensures proper Unicode handling
  - Saves both terminal output and complete agent history
- **User Feedback**: Confirms log file locations

AI-Powered Log Analysis:

```
analysis_result = await analyze_log_with_llm(terminal_output, product_query, llm)
price = analysis_result.get('price')
status = analysis_result.get('status', 'Analysis Error')
```

- **AI Analysis**: Sends captured logs to Gemini for interpretation
- **Result Extraction**: Gets price and status from AI analysis
- **Default Status**: Uses 'Analysis Error' if status not provided

Fallback Price Extraction:

```
if price is None:
    result_match = re.search(r'  Result: (\d+(?:\.\d+)?)', terminal_output)
    if result_match:
        try:
            price = float(result_match.group(1))
            status = "Available"
            print(f"  Extracted price from terminal log: {price}")
        except:
            pass
```

- **Manual Extraction**: If AI fails, use regex to find price patterns
- **Pattern Matching**: Looks for " Result: [number]" format
- **Type Conversion**: Converts string to float
- **Status Update**: Sets status to "Available" if price found
- **User Feedback**: Shows manual extraction success

Additional Fallback Logic:

```
elif "  Task completed successfully" in terminal_output:
    if raw_agent_output and isinstance(raw_agent_output, (int, float)):
        price = float(raw_agent_output)
        status = "Available"
    elif raw_agent_output and str(raw_agent_output).replace('.', '').replace(',', '').isdigit():
        price = float(str(raw_agent_output).replace(',', ''))
        status = "Available"
```

- **Success Detection**: Looks for task completion indicator

- **Numeric Validation**: Checks if raw output is already numeric
- **String Processing**: Handles comma-separated numbers (e.g., "1,299.99")
- **Type Safety**: Multiple validation steps prevent conversion errors

Status Detection:

```
elif "Not Found" in terminal_output:
    status = "Not Found"
elif "Out of Stock" in terminal_output:
    status = "Out of Stock"
else:
    status = "Scraping Error"
```

- **Text Pattern Matching**: Searches for specific status phrases
- **Status Assignment**: Maps log content to standardized status values
- **Default Error**: Assumes scraping error if no clear status found

Currency Detection:

```
currency = "USD"  # default
if " " in terminal_output or "INR" in terminal_output.upper():
    currency = "INR"
elif "£" in terminal_output or "GBP" in terminal_output.upper():
    currency = "GBP"
elif "€" in terminal_output or "EUR" in terminal_output.upper():
    currency = "EUR"
```

- **Default Currency**: Assumes USD if no other currency detected
- **Symbol Detection**: Looks for currency symbols in logs
- **Text Detection**: Also searches for currency codes
- **Case Handling**: `.upper()` ensures consistent text matching

Result Compilation:

```
scraped_data.append({
    "url": url,
    "price": float(price) if price is not None else None,
    "status": status,
    "raw_output": raw_agent_output,
    "terminal_log": terminal_output,
    "currency": currency
})
```

- **Data Structure**: Creates comprehensive result dictionary
- **Price Conversion**: Ensures price is float or None
- **Complete Information**: Includes all relevant data for analysis
- **Log Preservation**: Keeps terminal logs for debugging

Function Return:

```
return scraped_data
```

- **Result**: List of dictionaries containing all scraping results
- **Data Completeness**: Each result includes price, status, logs, and metadata

---

8. Display & Output Functions

Results Presentation

```
def display_final_results(data, country_name):
```

Function Purpose: Presents scraped price data in multiple formats (table, detailed text, JSON)

Parameters:

- `data` (List[Dict[str, Any]]): List of scraping results from all URLs

- `country_name` (str): Country name for report header

Data Validation and Sorting:

```
if not data:
    print("\nNo price data could be collected.")
    return
```

- **Empty Check**: Handles case where no data was successfully scraped
- **User Feedback**: Clear message explaining why no results are shown
- **Early Return**: Prevents further processing of empty data

```
sorted_data = sorted(data, key=lambda x: x['price'] if x['price'] is not None else float('inf'))
```

- **Sorting Logic**: Orders results by price, ascending
- **None Handling**: `float('inf')` puts failed scrapes at the end
- **User Experience**: Shows cheapest options first

Table Display Format:

```
print("\n" + "="*80)
print(f"  FINAL PRICE COMPARISON REPORT (Location: {country_name})")
print("="*80)
```

- **Visual Separation**: 80-character border for clear section breaks
- **Header Information**: Includes location context for pricing
- **Emoji**: Shopping cart emoji for visual appeal

```
display_list = []
for item in sorted_data:
    if item['price'] is not None:
        currency_symbol = " " if item.get('currency') == "INR" else "$"
        price_display = f"{currency_symbol}{item['price']:.2f}"
    else:
        price_display = "N/A"

    display_list.append({
        "Retailer URL": item['url'],
        "Price": price_display,
        "Availability": item['status'],
    })
```

- **Currency Formatting**:
  - Detects INR for Rupee symbol ( )
  - Defaults to Dollar symbol ($) for other currencies
  - `.2f` formats to 2 decimal places
- **None Handling**: Shows "N/A" for failed price extractions
- **Data Structure**: Creates clean dictionary for DataFrame creation

```
df = pd.DataFrame(display_list)
df.set_index('Retailer URL', inplace=True)
try:
    display(df)
except Exception:
    print(df.to_string())
```

- **DataFrame Creation**: Converts list to pandas DataFrame for formatting
- **Index Setting**: Uses URL as index for better readability
- **Display Strategy**:
  - Try `IPython.display` for rich formatting (notebooks)
  - Fallback to `to_string()` for plain text terminals

Detailed Text Output:

```
print("\n" + "="*80)
print("  DETAILED TEXT OUTPUT (per URL)")
print("="*80)
for item in sorted_data:
    if item['price'] is not None:
        currency_symbol = " " if item.get('currency') == "INR" else "$"
        price_display = f"{currency_symbol}{item['price']:.2f}"
    else:
        price_display = "N/A"

    print(f"URL: {item['url']}")
    print(f"  - Price: {price_display}")
    print(f"  - Availability: {item['status']}")
    print(f"  - Raw Agent Output: {item.get('raw_output', 'N/A')}")
    print(f"  - Currency: {item.get('currency', 'N/A')}")
    print("-"*60)
```

- **Section Header**: Clear identification of detailed output section
- **Per-Item Display**: Shows complete information for each URL
- **Indentation**: Two-space indentation for item details
- **Separator**: 60-character dashes between items
- **Safe Access**: `.get()` method prevents KeyError for missing keys

JSON Output Format:

```
print("\n" + "="*80)
print("  JSON OUTPUT (full data)")
print("="*80)

json_list = []
for item in sorted_data:
    if item['price'] is not None:
        currency_symbol = " " if item.get('currency') == "INR" else "$"
        price_display = f"{currency_symbol}{item['price']:.2f}"
    else:
        price_display = "N/A"

    json_list.append({
        "Retailer URL": item['url'],
        "Price": price_display,
        "Availability": item['status'],
        "Raw Output": str(item.get('raw_output', '')),
        "Currency": item.get('currency', 'N/A'),
        "Scraped Successfully": item['price'] is not None
    })
print(json.dumps(json_list, indent=2, ensure_ascii=False))
```

- **JSON Structure**: Creates clean JSON representation of results
- **Success Indicator**: Boolean field showing if scraping succeeded
- **String Conversion**: Ensures raw output is JSON-serializable
- **Formatting**:
  - `indent=2`: Pretty-prints with 2-space indentation
  - `ensure_ascii=False`: Preserves Unicode characters (currency symbols)

---

9. Main Application Logic

Asynchronous Main Function

```
async def main():
```

Function Purpose: Orchestrates the entire application workflow from start to finish

Return Values: None (prints results and saves logs)

Logging Configuration:

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(sys.stdout)
    ]
)
```

- **Log Level**: INFO captures informational messages and above (WARNING, ERROR, CRITICAL)
- **Format Components**:
  - `%(asctime)s`: Timestamp of log message
  - `%(name)s`: Logger name (useful for debugging different modules)
  - `%(levelname)s`: Log level (INFO, ERROR, etc.)
  - `%(message)s`: Actual log message content
- **Handler**: `StreamHandler(sys.stdout)` outputs to console

Directory Setup:

```
LOGS_DIR = "logs"
os.makedirs(LOGS_DIR, exist_ok=True)
logging.info(f" Log directory ensured at: ./{LOGS_DIR}/")
```

- **Directory Creation**: Creates logs directory if it doesn't exist
- **Error Prevention**: `exist_ok=True` prevents exception if directory exists
- **Logging**: Records directory creation for debugging

Environment and API Validation:

```
load_dotenv()
gemini_api_key = os.getenv("GEMINI_API_KEY")
if not gemini_api_key or gemini_api_key == "YOUR_GEMINI_API_KEY":
    logging.error(" CRITICAL ERROR: GEMINI_API_KEY is not set.")
    logging.error("   Please create a .env file and add your key: GEMINI_API_KEY='...'")
    sys.exit(1)
```

- **Environment Loading**: `load_dotenv()` reads .env file
- **API Key Retrieval**: Gets Gemini API key from environment
- **Validation Checks**:
  - Key exists (`not gemini_api_key`)
  - Key is not placeholder (`== "YOUR_GEMINI_API_KEY"`)
- **Error Handling**:
  - Clear error messages for user
  - `sys.exit(1)` terminates with error code
- **Security**: Prevents running without valid API credentials

Step 1: User Input Collection:

```
product_query, country_code, country_name = get_user_input()
```

- **Function Call**: Executes interactive user input collection
- **Return Values**: Unpacks tuple into three variables
- **Data Flow**: User input becomes foundation for all subsequent operations

Step 2: URL Validation:

```
validated_urls = validate_search_results(product_query, country_code)

if not validated_urls:
    logging.info("\nNo credible URLs found. Exiting.")
    return
```

- **Search and Validation**: Combines Google search with security validation
- **Early Exit**: Returns if no trustworthy URLs found
- **Logging**: Records reason for early termination

URL Logging:

```
logging.info("\n--- Final List of High-Trust URLs to Scrape ---")
for u in validated_urls:
    logging.info(f"  -> {u}")
```

- **Documentation**: Records which URLs will be processed
- **Debugging**: Helpful for analyzing why certain sites were chosen
- **Transparency**: Shows user which sites passed validation

Step 3: AI Model Initialization:

```
logging.info("\n" + "#"*80)
logging.info("  INITIALIZING AI BROWSER AGENT FOR PRICE SCRAPING")
logging.info("#"*80)

llm = ChatGoogle(model="gemini-2.5-flash", api_key=gemini_api_key)
```

- **Section Marking**: Hash line separators for major workflow sections
- **Model Creation**: Initializes Google Gemini AI with specific model version
- **Model Selection**: "gemini-2.5-flash" balances performance and cost

Price Scraping Execution:

```
scraped_data = await scrape_prices_with_agent(product_query, validated_urls, llm)
```

- **Async Execution**: Uses await for asynchronous browser automation
- **Data Collection**: Gathers price and availability information
- **AI Integration**: Passes LLM instance for intelligent scraping

Results Display:

```
display_final_results(scraped_data, country_name)
```

- **Multi-Format Output**: Shows results in table, text, and JSON formats
- **Context Inclusion**: Passes country name for report headers

Run Log Generation:

```
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
run_log_path = os.path.join(LOGS_DIR, f"run_{timestamp}.log")
with open(run_log_path, 'w', encoding='utf-8') as f:
    f.write(f"Product Query: {product_query}\n")
    f.write(f"Country: {country_name} ({country_code})\n")
    f.write(f"URLs Processed: {len(validated_urls)}\n")
    f.write(f"Results:\n")
    for item in scraped_data:
        if item['price'] is not None:
            currency_symbol = " " if item.get('currency') == "INR" else "$"
            price_str = f"{currency_symbol}{item['price']:.2f}"
        else:
            price_str = "N/A"
        f.write(f"  - {item['url']}: {price_str} ({item['status']})\n")
```

- **Timestamp Creation**: Unique filename with date and time
- **Summary Generation**: Records key parameters and results
- **File Writing**: Saves complete run summary for later reference
- **Price Formatting**: Consistent currency display in logs

Completion Logging:

```
logging.info(f" Script finished. Full log saved to {run_log_path}")
print(f" Script finished. Full log saved to {run_log_path}")
```

- **Dual Output**: Both logging system and direct print for visibility
- **File Reference**: Shows user where complete log is saved
- **Success Indicator**: Checkmark emoji confirms successful completion

10. Entry Point & Error Handling

Program Entry Point

```
if __name__ == "__main__":
```

- **Purpose**: Ensures code only runs when script is executed directly
- **Import Safety**: Prevents execution when file is imported as module
- **Standard Python Practice**: Common pattern for executable scripts

Asyncio Event Loop Management

```
try:
    asyncio.run(main())
```

- **Event Loop**: `asyncio.run()` creates and manages event loop for async functions
- **Main Execution**: Calls the main() async function
- **Python 3.7+**: Uses modern asyncio API (older versions need `loop.run_until_complete()`)

Keyboard Interrupt Handling

```
except KeyboardInterrupt:
    print("\n\nProcess interrupted by user. Exiting.")
```

- **User Control**: Handles Ctrl+C gracefully
- **Clean Exit**: Provides friendly message instead of stack trace
- **Process Termination**: Allows user to stop long-running operations

---

Configuration Summary

Critical Constants

Variable	Value	Purpose	Impact
`NUM_SEARCH_RESULTS`	2	Limits URLs processed	Performance vs thoroughness balance
`MINIMUM_DOMAIN_AGE_DAYS`	365	Domain age threshold	Security filtering
`HIGH_RISK_TLDS`	10 TLDs	Suspicious domain extensions	Fraud prevention
`DOMAIN_BLACKLIST`	11 domains	Known non-shopping sites	Focus on e-commerce
`EXCLUSION_KEYWORDS`	4 keywords	Review/news filtering	Shopping site focus

Security Validation Pipeline

1. **Blacklist Check** → Removes known non-shopping domains
2. **Keyword Filter** → Excludes review and news content
3. **HTTPS Requirement** → Ensures secure connections
4. **Structure Validation** → Detects suspicious URL patterns
5. **Domain Age Check** → Filters new/potentially fraudulent domains
6. **SSL Certificate Verification** → Confirms legitimate HTTPS implementation

AI Integration Points

1. **Log Analysis** → Gemini analyzes browser automation logs
2. **Price Extraction** → AI identifies prices in complex web pages
3. **Status Determination** → AI categorizes availability and errors
4. **Browser Automation** → AI navigates websites intelligently

Data Flow Architecture

```
User Input → Google Search → URL Validation → AI Scraping → Result Analysis → Multi-Format Output
     ↓              ↓              ↓              ↓              ↓              ↓
Product Query   Raw URLs    Trusted URLs   Browser Logs   Structured Data   Reports
```

Error Handling Strategy

- **Network Errors**: Graceful handling of connection failures
- **API Errors**: Fallback to manual extraction if AI fails
- **Validation Errors**: Continue processing other URLs if one fails
- **User Interruption**: Clean exit on Ctrl+C
- **Missing Data**: Default values and clear error messages

This comprehensive documentation covers every aspect of the main.py file, from individual line explanations to architectural decisions and data flow patterns. Each function, variable, and configuration choice is explained with its purpose, implementation details, and impact on the overall system.