Wikipedia Research Assistant

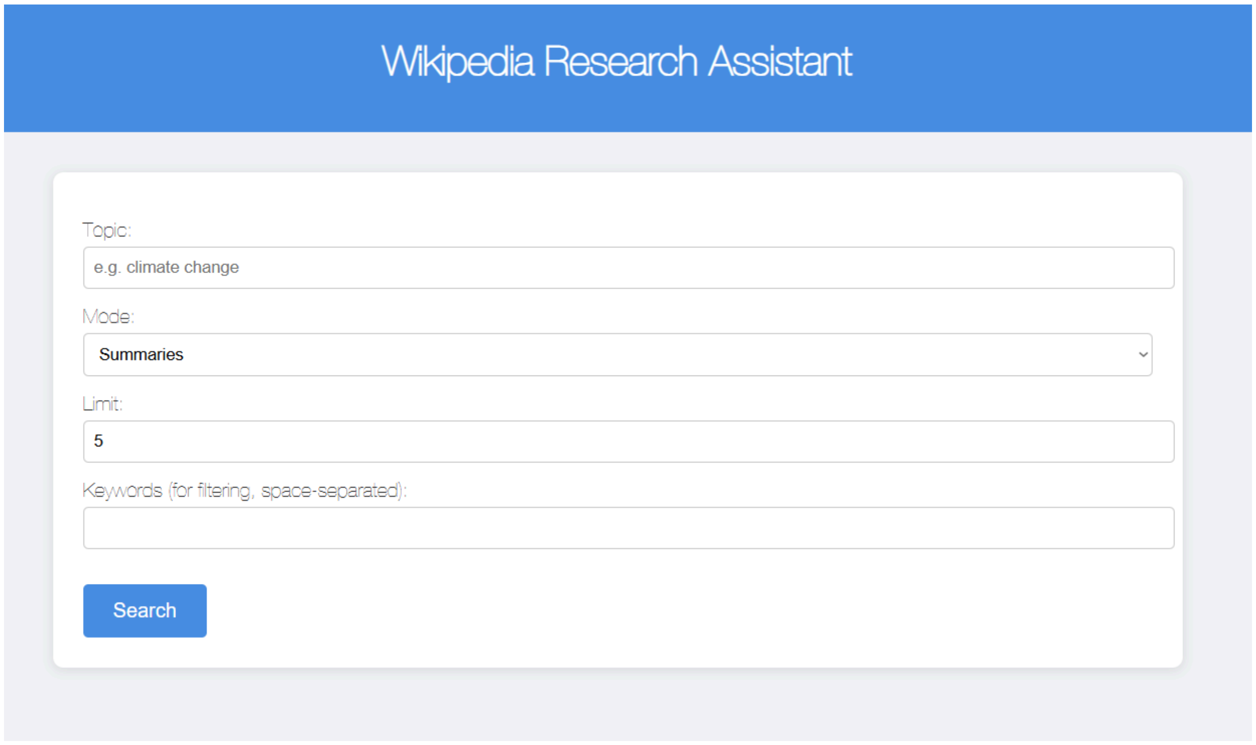
Markus Rabanes & Hana Thahireen

The Wikipedia Research Assistant is a handy Python tool that makes digging into any topic on Wikipedia incredibly easy. You just type in a topic (even if it's not an exact match), and behind the scenes it uses Requests and BeautifulSoup to grab the main article plus all its internal links. It can even pull out a quick summary of each page, tag them by category, and let you filter by keywords if you want to zero in on something specific. You can run it straight from the command line or fire up a simple Flask web app to click through everything in your browser. In our tests, it nails exact and fuzzy matches (thanks to a MediaWiki API fallback), skips over any broken links without crashing, and delivers a neat, organized view of both high-level overviews and deeper related content.

Wikipedia Research Assistant

Please scroll down for further information on our application!

Here is a preview of our program running through Flask:



Wikipedia Research Assistant

Topic:

e.g. climate change

Mode:

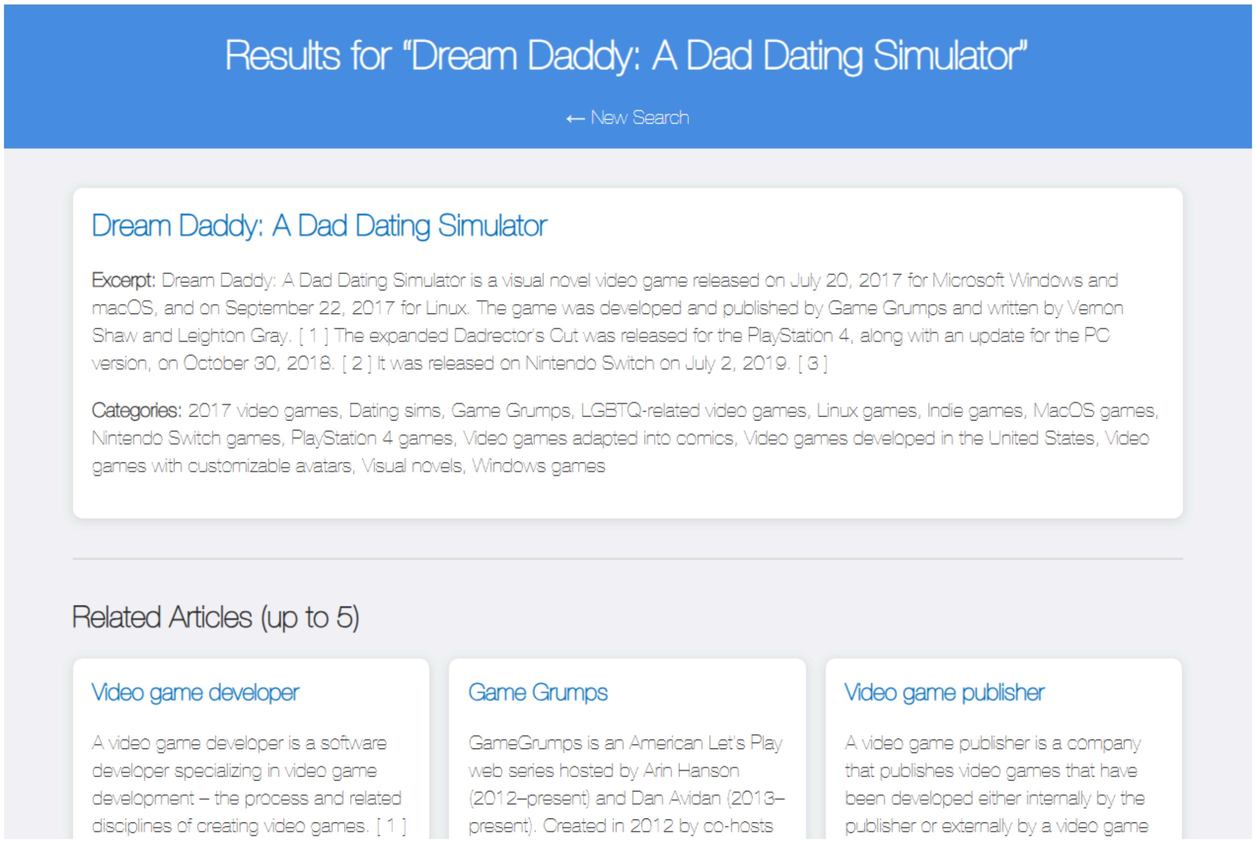
Summaries

Limit:

5

Keywords (for filtering, space-separated):

Search



Results for "Dream Daddy: A Dad Dating Simulator"

← New Search

Dream Daddy: A Dad Dating Simulator

Excerpt: Dream Daddy: A Dad Dating Simulator is a visual novel video game released on July 20, 2017 for Microsoft Windows and macOS, and on September 22, 2017 for Linux. The game was developed and published by Game Grumps and written by Vernon Shaw and Leighton Gray. [1] The expanded Dadrector's Cut was released for the PlayStation 4, along with an update for the PC version, on October 30, 2018. [2] It was released on Nintendo Switch on July 2, 2019. [3]

Categories: 2017 video games, Dating sims, Game Grumps, LGBTQ-related video games, Linux games, Indie games, MacOS games, Nintendo Switch games, PlayStation 4 games, Video games adapted into comics, Video games developed in the United States, Video games with customizable avatars, Visual novels, Windows games

Related Articles (up to 5)

Video game developer

A video game developer is a software developer specializing in video game development – the process and related disciplines of creating video games. [1]

Game Grumps

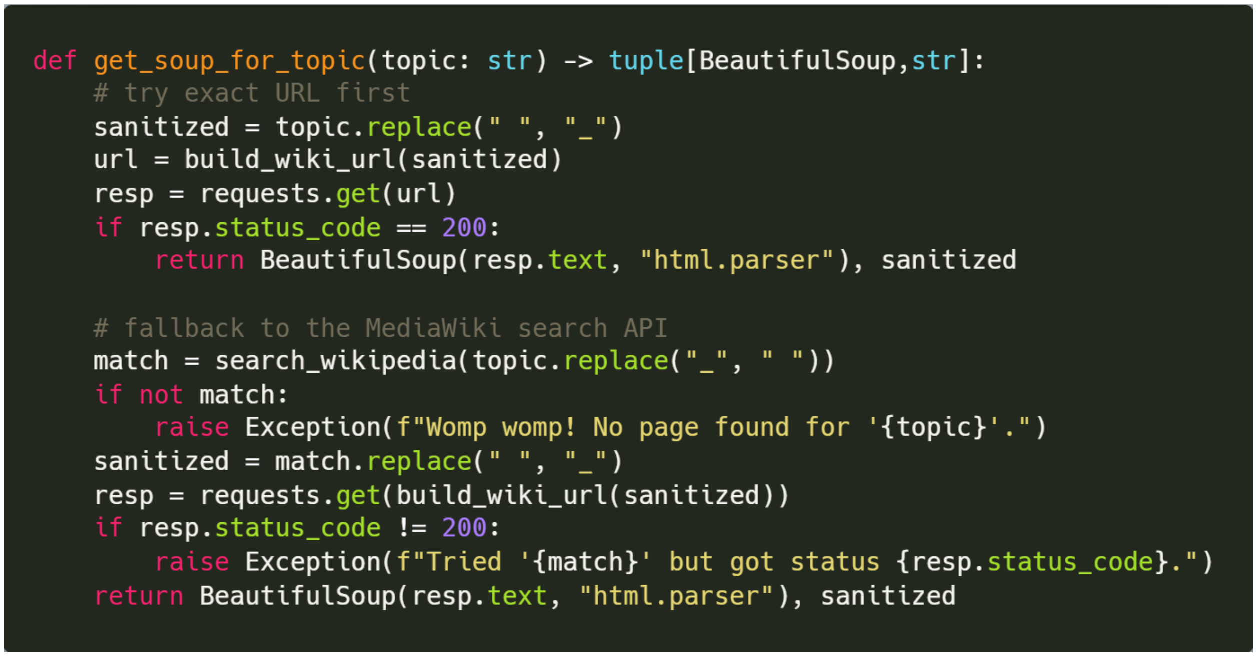
GameGrumps is an American Let's Play web series hosted by Arin Hanson (2012–present) and Dan Avidan (2013–present). Created in 2012 by co-hosts

Video game publisher

A video game publisher is a company that publishes video games that have been developed either internally by the publisher or externally by a video game

Here are five important pieces of our code:

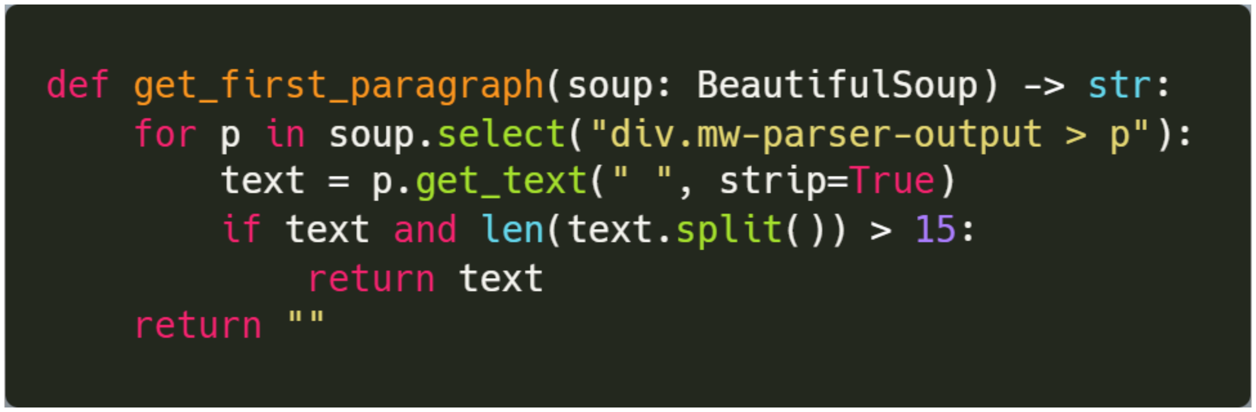
1): This is the “fuzzy-match” engine. It first attempts the user’s exact page; if that 404s, it falls back to a search via the Wikipedia API, grabs the top hit, and re-fetches that page. Without this, anything but an exact title would break.



```python
def get_soup_for_topic(topic: str) -> tuple[BeautifulSoup,str]:
    # try exact URL first
    sanitized = topic.replace(" ", "_")
    url = build_wiki_url(sanitized)
    resp = requests.get(url)
    if resp.status_code == 200:
        return BeautifulSoup(resp.text, "html.parser"), sanitized

    # fallback to the MediaWiki search API
    match = search_wikipedia(topic.replace("_", " "))
    if not match:
        raise Exception(f"Womp womp! No page found for '{topic}'.")
    sanitized = match.replace(" ", "_")
    resp = requests.get(build_wiki_url(sanitized))
    if resp.status_code != 200:
        raise Exception(f"Tried '{match}' but got status {resp.status_code}.")
    return BeautifulSoup(resp.text, "html.parser"), sanitized
```

2) This sifts through all the `<p>` tags in the main content and picks the first one that’s “long enough”—so you don’t return tiny boilerplate lines or empty text. It’s the lightweight way of getting a usable summary from any article.



```python
def get_first_paragraph(soup: BeautifulSoup) -> str:
    for p in soup.select("div.mw-parser-output > p"):
        text = p.get_text(" ", strip=True)
        if text and len(text.split()) > 15:
            return text
    return ""
```

3) After you pull raw HTML text, you often end up with weird newlines or stray spaces around commas and periods. This formats the data into easily readable text.

```python
def clean_text(text: str) -> str:
    # 1) collapse multiple whitespace into one
    s = re.sub(r"\s+", " ", text).strip()
    # 2) remove spaces before punctuation
    s = re.sub(r"\s+([.,!?;:])", r"\1", s)
    return s
```

4) This is the “narrow the field” tool. It builds one big lowercase string from the title, excerpt, and categories, then checks each keyword against it—either requiring *all* keywords (`and` mode) or *any* of them (`or` mode).

```python
def filter_by_keyword(items: list[dict], keywords: list[str], mode="or"):
    results = []
    for item in items:
        text = " ".join([item["title"], item["excerpt"], *item["categories"]]).lower()
        if mode == "and":
            if all(kw.lower() in text for kw in keywords):
                results.append(item)
        else:  # OR logic
            if any(kw.lower() in text for kw in keywords):
                results.append(item)
    return results
```

5) This ties everything together in the web app—you see the POST/check logic, how the main article is fetched, and how related links are built and filtered.

```python
@app.route("/", methods=["GET","POST"])
def index():
    if request.method == "POST":
        # 1) resolve main article
        soup, actual = get_soup_for_topic(request.form["topic"])
        main = { ... }            # build main article dict
        # 2) fetch related links (sliced by limit)
        links = find_internal_links(soup)[:int(request.form["limit"])]
        related = [ ... ]         # loop and build each related dict
        # 3) optional filtering
        if request.form["mode"] == "filtered":
            related = filter_by_keyword(related, request.form["keywords"].split())
        return render_template("results.html", main=main, related=related, ...)
    return render_template("index.html")
```

Code Strengths and Weaknesses (not so fun stuff)

Here are some files we analyzed and thought could use improvement.

scraper.py

Strength: Robust fuzzy matching via exact URL first, then MediaWiki API fallback.
Weakness: Can issue up to three HTTP requests per topic and has no caching, so it can be slow and redundant.

summarizer.py

Strength: Zero-dependency, regex-based cleaning and sentence splitting for quick, readable summaries.
Weakness: Naïve split on punctuation may mis-handle abbreviations or inline citations, leading to awkward fragments.

filter.py

Strength: Simple AND/OR keyword filtering over title, excerpt, and categories in one pass.
Weakness: Substring matching can over-match (e.g. “art” in “start”), with no whole-word or boundary enforcement.

Practical Use Case Highlights

Student Literature Surveys
A student beginning a term paper can punch in their broad topic and instantly get the main article plus a handful of related entries. That lets them sketch out a reading list in minutes rather than hours of manual browsing.

Journalist Quick-Briefs
Reporters racing to get background on a breaking story can use the CLI or web version to pull in the core Wikipedia article plus top related pages (e.g. “climate change,” then “Kyoto Protocol,” “IPCC,” etc.). The auto-summaries help them build a bullet-point primer before more in-depth fact-checking.

Casual Deep-Dives
Lifelong learners or hobbyists who want to “go down the rabbit hole” on subjects like photography, cooking techniques, or video games can jump between related pages via the web UI and filter by keywords like “technique” or “history” to keep the journey focused.

Future ideas?

In the next version of Wikipedia Research Assistant, we are planning to let people bookmark and revisit their favorite searches so they don’t have to start over each time, or add simple settings so users can choose how much detail they see by default, whether that is just links, short snippets, or full paragraphs. We could build a more mobile-friendly layout so it works nicely on phones or tablets, and offer easy export options like PDF or Excel so students can drop it right into reports. Another idea is to let users save custom keyword filters, or add a way to share your findings by generating a one-click share link or embedding your results in a webpage. There’s a lot of ideas we are brainstorming, we hope to build and develop them in the near future!

Markus Rabanes & Hana Thahireen