Análise Inicial do Código

O script Python fornecido é uma ferramenta para geração de artigos, utilizando a API Gemini para criar conteúdo baseado em prompts e configurações pré-definidas. Ele automatiza o processo de criação de artigos, desde a coleta de dados iniciais até a revisão final.

Estrutura Geral:

O código é dividido em várias seções lógicas: 1. **Constantes de Configuração**: Define URLs para prompts e arquivos de configuração hospedados no GitHub. 2. **Configuração da API Gemini**: Inicializa a API Gemini, carregando a chave de API de uma variável de ambiente. 3. **Funções Utilitárias**: Contém funções auxiliares para carregar recursos de URLs (JSON e texto) e interagir com a API Gemini, além de obter input do usuário. 4. **Funções do Fluxo de Geração**: Constituem o cerne da lógica de geração do artigo, divididas em etapas como coleta de dados, busca de palavras-chave, sugestão de título/outline, geração de TOC e construção/revisão do artigo. 5. **Função `main()`**: Orquestra a execução de todas as etapas do fluxo de geração do artigo.

Propósito:

O principal objetivo do script é facilitar a criação de artigos de forma semi-automatizada, permitindo que o usuário forneça um tema e diretrizes, e o script se encarregue de gerar o conteúdo utilizando modelos de linguagem grandes (LLMs) da Google Gemini. Ele busca otimizar o processo de escrita, gerando rascunhos e estruturas que podem ser posteriormente refinados.

Componentes Chave:

- **Prompts Externos**: O script depende fortemente de arquivos Markdown e JSON hospedados no GitHub para seus prompts e configurações, o que permite flexibilidade na atualização da lógica de geração sem alterar o código principal.
- **Integração com Gemini API**: A comunicação com a API Gemini é central para a geração de texto em todas as etapas do processo.

- **Fluxo Sequencial**: O processo de geração do artigo segue uma sequência lógica de passos, onde a saída de uma etapa serve como entrada para a próxima.
- **Geração Modular de Conteúdo**: O artigo é construído seção por seção, permitindo um controle mais granular sobre o conteúdo gerado.

Esta análise inicial fornece uma visão geral do funcionamento do script. As próximas fases se aprofundarão nas funções específicas e identificarão possíveis melhorias.

Análise Detalhada das Funções

Esta seção detalha cada função presente no script `main.py`, descrevendo seu propósito, parâmetros, valores de retorno e como ela se integra ao fluxo geral de geração de artigos.

Funções Utilitárias

`carregar_recurso_url(url: str) -> str`

- **Propósito**: Carrega o conteúdo de um recurso (texto) de uma URL especificada.
- **Parâmetros**:
  - `url` (str): A URL do recurso a ser carregado.
- **Retorno**: `str` - O conteúdo do recurso como uma string, com espaços em branco removidos das extremidades.
- **Detalhes**: Utiliza a biblioteca `requests` para fazer uma requisição GET. Lança uma exceção `HTTPError` para respostas de status HTTP ruins. Essencial para carregar prompts e arquivos de configuração remotos.

`carregar_json_url(url: str) -> Dict[str, Any]`

- **Propósito**: Carrega e parseia um arquivo JSON de uma URL especificada.
- **Parâmetros**:
  - `url` (str): A URL do arquivo JSON a ser carregado.
- **Retorno**: `Dict[str, Any]` - Um dicionário Python representando o conteúdo JSON.

- **Detalhes**: Similar a `carregar_recurso_url`, mas espera e decodifica uma resposta JSON. Usado para carregar o arquivo `article_config.json`.

`gerar_conteudo_gemini(prompt: str, temperatura: float = 0.7, max_tokens: int = 2048) -> Optional[str]`

- **Propósito**: Interage com a API Gemini para gerar conteúdo textual com base em um prompt.
- **Parâmetros**:
  - `prompt` (str): O texto do prompt a ser enviado para o modelo Gemini.
  - `temperatura` (float, opcional): Controla a aleatoriedade da saída. Valores mais altos (e.g., 0.8) tornam a saída mais criativa, enquanto valores mais baixos (e.g., 0.2) a tornam mais focada. Padrão é 0.7.
  - `max_tokens` (int, opcional): O número máximo de tokens na resposta gerada. Padrão é 2048.
- **Retorno**: `Optional[str]` - O texto gerado pelo modelo Gemini, ou `None` se ocorrer um erro durante a geração.
- **Detalhes**: Encapsula a lógica de chamada à API `genai.GenerativeModel`. É a função central para toda a geração de conteúdo do artigo.

`obter_input_usuario(rotulo: str) -> str`

- **Propósito**: Solicita e obtém uma entrada de texto do usuário via console.
- **Parâmetros**:
  - `rotulo` (str): O texto a ser exibido ao usuário como uma solicitação de entrada.
- **Retorno**: `str` - A string de entrada fornecida pelo usuário, com espaços em branco removidos das extremidades.
- **Detalhes**: Uma função simples para interatividade com o usuário, usada para coletar o tema e as diretrizes iniciais do artigo.

Funções do Fluxo de Geração

`carregar_prompts() -> Dict[str, str]`

- **Propósito**: Carrega todos os prompts Markdown definidos em `PROMPT_URLS` de suas respectivas URLs.
- **Parâmetros**: Nenhum.
- **Retorno**: `Dict[str, str]` - Um dicionário onde as chaves são os nomes dos prompts (e.g., "COLETA", "SEMANTICA") e os valores são o conteúdo textual de cada prompt.
- **Detalhes**: Utiliza `carregar_recurso_url` para cada URL de prompt, centralizando o carregamento de todos os prompts necessários para o fluxo.

`coleta_dados_iniciais() -> Dict[str, str]`

- **Propósito**: Coleta o tema principal e diretrizes opcionais para o artigo do usuário.
- **Parâmetros**: Nenhum.
- **Retorno**: `Dict[str, str]` - Um dicionário contendo as chaves "tema" e "diretrizes" com as entradas do usuário.
- **Detalhes**: É a primeira etapa interativa do processo de geração do artigo.

`busca_palavras_chave(tema: str, diretrizes: str, prompt_semantica: str) -> Optional[Dict[str, Any]]`

- **Propósito**: Gera palavras-chave semanticamente relevantes com base no tema e diretrizes fornecidos, utilizando o prompt de semântica.
- **Parâmetros**:
  - `tema` (str): O tema principal do artigo.
  - `diretrizes` (str): Diretrizes adicionais ou palavras-chave fornecidas pelo usuário.
  - `prompt_semantica` (str): O prompt específico para a geração de palavras-chave semânticas.

- **Retorno**: `Optional[Dict[str, Any]]` - Um dicionário contendo o `raw_output` (saída bruta do Gemini) e uma lista das `top_palavras_chave` extraídas, ou `None` se a geração falhar.
- **Detalhes**: Formata o prompt de semântica com o tema e diretrizes, envia para o Gemini e extrai as palavras-chave usando expressões regulares. As palavras-chave são usadas para enriquecer prompts subsequentes.

`sugerir_titulo_e_outline(dados_iniciais: dict, dados_pesquisa: dict, prompt_titulo: str, artigo_conf: dict) -> Optional[Tuple[str, str, str]]`

- **Propósito**: Sugere um título, subtítulo e um outline inicial para o artigo.
- **Parâmetros**:
  - `dados_iniciais` (dict): Contém o tema e as diretrizes iniciais.
  - `dados_pesquisa` (dict): Contém a saída bruta da busca de palavras-chave (`raw_output`).
  - `prompt_titulo` (str): O prompt específico para a geração de título e outline.
  - `artigo_conf` (dict): Configurações do artigo, incluindo parâmetros para H1.
- **Retorno**: `Optional[Tuple[str, str, str]]` - Uma tupla contendo o título, subtítulo e outline gerados, ou `None` se a geração falhar.
- **Detalhes**: Formata o prompt com os dados iniciais e de pesquisa, gera o conteúdo com o Gemini e extrai o título, subtítulo e outline usando expressões regulares. O subtítulo é opcional.

`sugerir_toc(titulo: str, subtitulo: str, outline: str, dados_pesquisa: dict, prompt_toc: str, artigo_conf: dict) -> Optional[Tuple[str, List[Dict]]]`

- **Propósito**: Gera uma Tabela de Conteúdo (TOC) estruturada para o artigo, com seções e subseções.
- **Parâmetros**:
  - `titulo` (str): O título principal do artigo.
  - `subtitulo` (str): O subtítulo do artigo.

    - `outline` (str): O outline inicial gerado.
    - `dados_pesquisa` (dict): Contém a saída bruta da busca de palavras-chave (`raw_output`).
    - `prompt_toc` (str): O prompt específico para a geração da TOC.
    - `artigo_conf` (dict): Configurações do artigo, incluindo parâmetros para H2 e H3.
- **Retorno**: `Optional[Tuple[str, List[Dict]]]` - Uma tupla contendo a saída bruta da TOC e uma lista de dicionários representando a estrutura da TOC (seções e subseções), ou `None` se a geração falhar.
- **Detalhes**: Formata o prompt com as informações do artigo e de pesquisa, gera a TOC com o Gemini e parseia a saída para criar uma estrutura hierárquica de seções e subseções. Esta estrutura é crucial para a construção modular do artigo.

`_gerar_conteudo_secao(titulo_artigo: str, subtitulo_artigo: str, dados_pesquisa: dict, secao_info: dict, prompt_bloco: str, artigo_conf: dict) -> Optional[str]`

- **Propósito**: Função auxiliar para gerar o conteúdo de uma seção ou subseção específica do artigo.
- **Parâmetros**:
    - `titulo_artigo` (str): O título principal do artigo.
    - `subtitulo_artigo` (str): O subtítulo do artigo.
    - `dados_pesquisa` (dict): Contém a saída bruta da busca de palavras-chave (`raw_output`).
    - `secao_info` (dict): Um dicionário contendo informações sobre a seção (nível, título, introdução).
    - `prompt_bloco` (str): O prompt específico para a geração de blocos de conteúdo.
    - `artigo_conf` (dict): Configurações do artigo, incluindo parâmetros para parágrafos.
- **Retorno**: `Optional[str]` - O conteúdo gerado para a seção, ou `None` se a geração falhar.

- **Detalhes**: Formata o prompt com base nas informações da seção e do artigo, e gera o conteúdo usando o Gemini. Ajusta `max_tokens` com base no nível da seção (H2 ou H3).

`construir_artigo(titulo: str, subtitulo: str, toc_struct: list, dados_pesquisa: dict, prompts: dict, artigo_conf: dict) -> str`

- **Propósito**: Monta o artigo completo, gerando o conteúdo para cada seção e subseção da TOC.
- **Parâmetros**:
  - `titulo` (str): O título principal do artigo.
  - `subtitulo` (str): O subtítulo do artigo.
  - `toc_struct` (list): A estrutura da Tabela de Conteúdo gerada por `sugerir_toc`.
  - `dados_pesquisa` (dict): Contém a saída bruta da busca de palavras-chave (`raw_output`).
  - `prompts` (dict): Dicionário contendo todos os prompts carregados.
  - `artigo_conf` (dict): Configurações do artigo.
- **Retorno**: `str` - O artigo completo em formato Markdown.
- **Detalhes**: Itera sobre a estrutura da TOC, chamando `_gerar_conteudo_secao` para cada seção e subseção. Concatena o conteúdo gerado e, finalmente, chama `revisar_e_finalizar`.

`revisar_e_finalizar(artigo_parcial: str, prompt_revisao: str, artigo_conf: dict) -> str`

- **Propósito**: Realiza uma revisão final do artigo gerado, aplicando diretrizes de revisão.
- **Parâmetros**:
  - `artigo_parcial` (str): O artigo gerado até o momento.
  - `prompt_revisao` (str): O prompt específico para a revisão do artigo.
  - `artigo_conf` (dict): Configurações do artigo, incluindo parâmetros para revisão geral, imagens e links.

- **Retorno**: `str` - O artigo revisado e finalizado.
- **Detalhes**: Envia o artigo parcial para o Gemini com o prompt de revisão. Se a revisão for bem-sucedida, retorna o artigo revisado; caso contrário, retorna o artigo parcial original.

Função Principal

`main()`

- **Propósito**: Orquestra todo o fluxo de geração do artigo.
- **Parâmetros**: Nenhum.
- **Retorno**: Nenhum.
- **Detalhes**: Gerencia a sequência de chamadas para as funções utilitárias e de fluxo de geração. Inclui tratamento básico de exceções para erros de requisição e outros erros gerais. É o ponto de entrada do script.

Identificação de Melhorias

Com base na análise detalhada do script `main.py`, as seguintes melhorias podem ser consideradas para aumentar a robustez, eficiência, legibilidade e manutenibilidade do código:

1. Tratamento de Erros e Robustez

- **Tratamento de Exceções Mais Granular**: Atualmente, a função `main()` e `gerar_conteudo_gemini` usam blocos `try-except` genéricos que capturam `Exception`. Isso pode mascarar problemas específicos e dificultar a depuração. Seria benéfico capturar exceções mais específicas (e.g., `requests.exceptions.RequestException` para problemas de rede, `json.JSONDecodeError` para JSON inválido, `ValueError` para problemas de configuração da API) e fornecer mensagens de erro mais informativas ao usuário.
- **Validação de Retorno da API**: As funções que chamam `gerar_conteudo_gemini` (e.g., `busca_palavras_chave`, `sugerir_titulo_e_outline`) verificam se o retorno é `None`. No entanto, a saída do Gemini pode ser uma string vazia ou conter texto que não corresponde ao formato esperado (e.g., regex falha).

Adicionar validações mais robustas ao conteúdo retornado pela API pode evitar erros subsequentes.

- **Retries para Requisições de Rede**: Requisições HTTP (para carregar prompts ou chamar a API Gemini) podem falhar devido a problemas temporários de rede. Implementar um mecanismo de *retries* com *backoff* exponencial para `requests.get` e chamadas à API Gemini pode tornar o script mais resiliente.
- **Variáveis de Ambiente para URLs**: Embora as URLs dos prompts estejam em constantes, seria mais flexível se a `PROMPTS_BASE_URL` pudesse ser configurada via variável de ambiente, permitindo fácil alternância entre ambientes de desenvolvimento/produção ou diferentes conjuntos de prompts.

2. Legibilidade e Manutenibilidade

- **Constantes para Regex**: As expressões regulares usadas para extrair título, subtítulo, outline e palavras-chave estão embutidas nas funções. Definir essas regex como constantes no início do arquivo (ou em um arquivo de configuração) melhoraria a legibilidade e facilitaria a manutenção se os formatos de saída do Gemini mudarem.
- **Documentação (Docstrings)**: Embora a análise já tenha detalhado as funções, adicionar docstrings padrão (e.g., Sphinx, Google Style) a cada função no próprio código melhoraria a documentação interna e facilitaria a compreensão para outros desenvolvedores.
- **Tipagem Mais Específica**: A tipagem já está presente, mas em alguns casos, como `Dict[str, Any]`, poderia ser mais específica se a estrutura dos dicionários fosse bem definida (e.g., usando `TypedDict` do módulo `typing`).
- **Refatoração de `sugerir_toc`**: A lógica de parsing da saída do `sugerir_toc` é um pouco complexa com múltiplos `if/elif`. Embora funcional, poderia ser refatorada para uma abordagem mais clara, talvez com uma máquina de estados simples ou funções auxiliares para cada nível de cabeçalho.
- **Separação de Responsabilidades**: A função `main()` atualmente faz muitas coisas: carrega configurações, prompts, coleta dados, chama todas as etapas de geração e imprime o resultado. Considerar a criação de uma classe `ArticleGenerator` ou funções auxiliares para agrupar lógicas relacionadas (e.g., `_load_configurations`, `_run_generation_flow`) pode melhorar a organização.

3. Funcionalidade e Experiência do Usuário

- **Feedback ao Usuário**: Durante a execução, o script não fornece muito feedback sobre o que está acontecendo (e.g., "Gerando palavras-chave...", "Construindo artigo..."). Adicionar mensagens de progresso melhoraria a experiência do usuário, especialmente para operações que demoram.
- **Opções de Saída**: Atualmente, o artigo final é apenas impresso no console. Adicionar opções para salvar o artigo em um arquivo (e.g., Markdown, PDF) seria uma melhoria significativa na usabilidade.
- **Configuração de Parâmetros da API**: A temperatura e `max_tokens` são fixos ou definidos por padrão. Permitir que o usuário configure esses parâmetros (talvez via argumentos de linha de comando ou um arquivo de configuração local) ofereceria mais controle.
- **Revisão Interativa**: A função `revisar_e_finalizar` é automatizada. Uma melhoria avançada seria permitir uma revisão interativa, onde o usuário pode aceitar/rejeitar sugestões ou fazer edições manuais antes da finalização.
- **Suporte a Mais Formatos de Prompt**: Atualmente, os prompts são carregados como strings. Se houver a necessidade de prompts mais complexos com variáveis que não são apenas `format()`, considerar um motor de *templating* (e.g., Jinja2) pode ser útil, embora possa adicionar complexidade.

Essas melhorias visam tornar o script mais robusto, fácil de usar e manter, e mais flexível para futuras expansões.