[[SYSTEM INSTRUCTIONS]] The following output presents a detailed directory structure and file contents from a specified root path. The file tree includes both excluded and included files and directories, clearly marking exclusions. Each file's content is displayed with comprehensive headings and separators to enhance readability and facilitate detailed parsing for extracting hierarchical and content-related insights. If the data represents a codebase, interpret and handle it as such, providing appropriate assistance as a programmer AI assistant. [[END SYSTEM INSTRUCTIONS]]

Root Path: /home/joost/.USER_SCRIPTS/final/collect-context

========== File Tree ==========

[Excluded] □□ .git [Excluded] □□ .gitignore □ 9 MANIFEST.in □ 956 README.md [Excluded] □□ build □ ccontext □ 0 ccontext/NotoEmoji-VariableFont_wght.ttf □ 0 ccontext/NotoSans-Bold.ttf □ 0 ccontext/NotoSans-MediumItalic.ttf □ 0 ccontext/NotoSans-Regular.ttf □ 0 ccontext/**init**.py □ 18 ccontext/**main**.py [Excluded] □□ ccontext/**pycache** □ 444 ccontext/argument_parser.py □ 79 ccontext/cli.py □ 483 ccontext/clipboard.py □ 209 ccontext/config.json □ 231 ccontext/configurator.py □ 646 ccontext/content_handler.py □ 766 ccontext/file_system.py □ 1026 ccontext/main.py □ 1178 ccontext/md_generator.py □ 750 ccontext/output_handler.py □ 1136 ccontext/pdf_generator.py □ 692 ccontext/tokenizer.py □ 53 ccontext/utils.py [Excluded] □□ ccontext.egg-info [Excluded] □□ dist □ 135 ideas.MD □ 395 output.md □ -1 output.pdf □ 33 requirements.txt □ 87 run_ccontext.sh □ 274 setup.py [Excluded] □□ venv

========== End of File Tree ==========

⚠ **output.pdf**

Contents: Error reading file output.pdf: 'utf-8' codec can't decode byte 0x93 in position 10: invalid start byte

□ **README.md**

Contents: Sure, let's continue updating the `README.md` to include the new PDF generation feature.

□ **README.md**

Updated Contents (continued):

````
3. Generate a PDF of the directory tree and file contents:

   ```sh
   ccontext -p /path/to/directory --generate-pdf
   ```

### Command-Line Arguments
````

````
- `-p, --root_path`: The root path to start the directory tree (default:
current directory).
- `-e, --excludes`: Additional files or directories to exclude, separated
by `|`, e.g., `node_modules|.git`.
- `-i, --includes`: Files or directories to include, separated by `|`,
e.g., `important_file.txt|docs`.
- `-m, --max_tokens`: Maximum number of tokens allowed before chunking.
- `-c, --config`: Path to a custom configuration file.
- `-v, --verbose`: Enable verbose output to stdout.
- `-ig, --ignore_gitignore`: Ignore the `.gitignore` file for exclusions.
- `--generate-pdf`: Generate a PDF of the directory tree and file
contents.

### Example

```sh
ccontext -p /home/user/project -e ".git|build" -i "README.md|src" --
generate-pdf
```
````

Configuration

You can customize the behavior of `ccontext` by creating a configuration file. The default configuration file is `config.json` located in the user's home directory under `.ccontext`. You can also provide a custom configuration file via the `-c` argument.

Sample `config.json`


```
{
  "verbose": false,
  "max_tokens": 120000,
  "model_type": "gpt-4o",
  "buffer_size": 0.05,
  "excluded_folders_files": [
    ".git",
    "bin",
    "build",
    "node_modules",
    "venv",
    "__pycache__",
    "package-lock.json",
    "ccontext.egg-info",
    "dist"
  ],
  "context_prompt": "[[SYSTEM INSTRUCTIONS]] The following output presents
a detailed directory structure and file contents from a specified root
path. The file tree includes both excluded and included files and
directories, clearly marking exclusions. Each file's content is displayed
with comprehensive headings and separators to enhance readability and
facilitate detailed parsing for extracting hierarchical and content-
related insights. If the data represents a codebase, interpret and handle
```

```
it as such, providing appropriate assistance as a programmer AI assistant.
[[END SYSTEM INSTRUCTIONS]]"
}
```

Use Cases

- **Codebase Context**: Send the entire codebase as context to an LLM in one go, avoiding the need to copy and paste snippets manually.
- **PDF Documentation**: Generate a comprehensive PDF documentation of the directory structure and file contents, with links to specific sections.

Contributing

We welcome contributions to `ccontext`! Please follow these steps to contribute:

1. Fork the repository.
2. Create a new branch for your feature or bug fix.
3. Commit your changes and push them to your branch.
4. Submit a pull request with a description of your changes.

License

This project is licensed under the MIT License - see the LICENSE file for details.

Acknowledgments

- Inspired by the need to streamline the process of providing context to LLMs.
- Thanks to the contributors and users who have provided valuable feedback and suggestions.

Future Ideas

Here are some ideas that might be implemented in future versions of `ccontext`:

Future versions of ccontext may include:

- **Document Support:** Incorporate the ability to handle documents such as PDFs and image files in prompts.
- **Binary File Handling:** Introduce mechanisms to manage non-text file types effectively.

Feel free to raise issues or contribute to the project. We appreciate your support!

Nicolas Arnouts
arnouts.software@gmail.com

GitHub Repository

Badges

output.md

Contents:

Directory and File Contents

File Tree

========== File Tree ========== [Excluded] .git [Excluded] .gitignore 9 MANIFEST.in 956 README.md [Excluded] build ccontext 0 ccontext/NotoEmoji-VariableFont_wght.ttf 0 ccontext/NotoSans-Bold.ttf 0 ccontext/NotoSans-MediumItalic.ttf 0 ccontext/NotoSans-Regular.ttf 0 ccontext/**init**.py 18 ccontext/**main**.py [Excluded] ccontext/**pycache** 444 ccontext/argument_parser.py 79 ccontext/cli.py 483 ccontext/clipboard.py 209 ccontext/config.json 231 ccontext/configurator.py 646 ccontext/content_handler.py 766 ccontext/file_system.py 1026 ccontext/main.py 1178 ccontext/md_generator.py 750 ccontext/output_handler.py 1136 ccontext/pdf_generator.py 692 ccontext/tokenizer.py 53 ccontext/utils.py [Excluded] ccontext.egg-info [Excluded] dist 135 ideas.MD 393 output.md -1 output.pdf 33 requirements.txt 87 run_ccontext.sh 274 setup.py [Excluded] venv ========== End of File Tree ==========

File Contents

run_ccontext.sh

Contents: #!/bin/bash

Determine the script's directory

SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

Add the script directory to PYTHONPATH

export PYTHONPATH="$SCRIPT_DIR"

Activate the virtual environment

source "$SCRIPT_DIR/venv/bin/activate"

Run the main script with the provided arguments

python3 "$SCRIPT_DIR/ccontext/main.py" "$@"

MANIFEST.in

Contents: include ccontext/config.json include README.md

ideas.MD

Contents:

- add an argument to specify documents like .pdfs, .jpgs, ... to add in a prompt -> u can only upload 10 files in 1 prompt, and there are rate limits regarding file uploads to OpenAI servers
- parse non txt files and/or add them to the clipboard -> same as, before max 10 in 1 prompt. Rate limits apply
- By default: add all binary files specified in "included" arg to the folder

NEW======

- When on ssh, notify user that the ssh connection should be started with -X, check if the user has a DISPLAY var by using:

```
echo $DISPLAY
```

setup.py

Contents: from setuptools import setup, find_packages

setup(name="ccontext", version="0.1.1", author="Nicolas Arnouts", author_email="arnouts.software@gmail.com", description="collect-context: Makes the process of collecting and sending context to an LLM like ChatGPT-4o as easy as possible.", long_description=open("README.md").read(), long_description_content_type="text/markdown", url="https://github.com/NicolasArnouts/ccontext", packages=find_packages(), include_package_data=True, package_data={ "ccontext": ["config.json"], }, install_requires=["colorama==0.4.6", "pyperclip==1.8.2", "tiktoken==0.7.0", "pathspec==0.12.1", "pypdf",], entry_points={ "console_scripts": ["ccontext=ccontext.cli:main", "ccontext-configure=ccontext.configurator:copy_default_config",], }, classifiers=["Programming Language :: Python :: 3", "License :: OSI Approved :: MIT License", "Operating System :: OS Independent",], python_requires=">=3.6",)

requirements.txt

Contents: tiktoken==0.7.0 colorama==0.4.6 pyperclip==1.8.2 pypdf

ccontext/main.py

Contents: from ccontext.cli import main

if **name** == "**main**": main()

ccontext/utils.py

Contents: from colorama import init

def initialize_environment(): """Initialize the environment settings.""" init(autoreset=True)

def format_number(number: int) -> str: """Formats a number with commas as thousands separators.""" return f"{number:,}"

ccontext/configurator.py

Contents: import os import shutil from pathlib import Path

try: from importlib import resources # Python 3.7+ except ImportError: import importlib_resources as resources # Backport for older Python versions

DEFAULT_CONFIG_FILENAME = "config.json" USER_CONFIG_DIR = Path.home() / ".ccontext" USER_CONFIG_PATH = USER_CONFIG_DIR / DEFAULT_CONFIG_FILENAME

def copy_default_config(): """Copy the default configuration file to the user-specific location.""" try: if not USER_CONFIG_DIR.exists(): USER_CONFIG_DIR.mkdir(parents=True, exist_ok=True)

```
    # Correctly use the context manager with resources.path
    with resources.path("ccontext", DEFAULT_CONFIG_FILENAME) as
default_config_path:
        if not USER_CONFIG_PATH.exists():
            shutil.copy(default_config_path, USER_CONFIG_PATH)
            print(f"Copied default config to {USER_CONFIG_PATH}")
        else:
            print(f"Config file already exists at {USER_CONFIG_PATH}")

except Exception as e:
    print(f"Error copying default config: {e}")
```

if **name** == "**main**": copy_default_config()

ccontext/main.py

Contents: import os import json from colorama import Fore, Style from pathlib import Path import importlib.resources as resources

from ccontext.utils import initialize_environment from ccontext.content_handler import (print_file_tree, gather_file_contents, combine_initial_content,) from ccontext.output_handler import handle_chunking_and_output from ccontext.file_system import collect_excludes_includes from ccontext.argument_parser import parse_arguments from ccontext.tokenizer import set_model_type_and_buffer from ccontext.pdf_generator import generate_pdf from ccontext.md_generator import generate_md

DEFAULT_CONFIG_FILENAME = "config.json" USER_CONFIG_DIR = Path.home() / ".ccontext" USER_CONFIG_PATH = USER_CONFIG_DIR / DEFAULT_CONFIG_FILENAME CURRENT_CONFIG_FILENAME = ".conti-config.json" DEFAULT_CONTEXT_PROMPT = """[[SYSTEM INSTRUCTIONS]] The following output presents a detailed directory structure and file contents from a specified root path. The file tree includes both excluded and included files and directories, clearly marking exclusions. Each file's content is displayed

with comprehensive headings and separators to enhance readability and facilitate detailed parsing for extracting hierarchical and content-related insights. If the data represents a codebase, interpret and handle it as such, providing appropriate assistance as a programmer AI assistant. [[END SYSTEM INSTRUCTIONS]]" """

def load_config(root_path: str, config_path: str = None) -> dict: """Load configuration from the specified path or use default settings.""" if config_path and os.path.exists(config_path): with open(config_path, "r") as f: print(f"{Fore.CYAN}Using config from provided argument: {config_path}{Style.RESET_ALL}") return json.load(f)

```
current_config_path = os.path.join(root_path, CURRENT_CONFIG_FILENAME)
if os.path.exists(current_config_path):
    with open(current_config_path, "r") as f:
        print(
            f"{Fore.CYAN}Using config found in root_path:
{current_config_path}{Style.RESET_ALL}"
        )
        return json.load(f)

if USER_CONFIG_PATH.exists():
    with open(USER_CONFIG_PATH, "r") as f:
        print(
            f"{Fore.CYAN}Using user config file: {USER_CONFIG_PATH}
{Style.RESET_ALL}"
        )
        return json.load(f)

with resources.open_text("ccontext", DEFAULT_CONFIG_FILENAME) as f:
    print(
        f"{Fore.CYAN}Using default config file: {DEFAULT_CONFIG_FILENAME}
{Style.RESET_ALL}"
    )
    return json.load(f)

print("No configuration file found. Using default settings.")
return {}
```

def main(root_path: str = None, excludes: list = None, includes: list = None, max_tokens: int = None, config_path: str = None, verbose: bool = False, ignore_gitignore: bool = False, generate_pdf_flag: bool = False, generate_md_flag: bool = False, # New argument for generating Markdown): root_path = os.path.abspath(root_path or os.getcwd()) config = load_config(root_path, config_path)

```
excludes, includes = collect_excludes_includes(
    config.get("excluded_folders_files", []),
    excludes,
    includes,
    root_path,
    ignore_gitignore,
)
```

```
    max_tokens = max_tokens or int(config.get("max_tokens", 32000))
    verbose = verbose or config.get("verbose", False)
    context_prompt = config.get(
        "context_prompt",
        DEFAULT_CONTEXT_PROMPT,
    )

    set_model_type_and_buffer(
        config.get("model_type", "gpt-4o"), config.get("buffer_size", 0.05)
    )
    initialize_environment()

    print(f"{Fore.CYAN}Root Path: {root_path}\n{Style.RESET_ALL}")
    tree_output = print_file_tree(
        root_path, excludes, includes, max_tokens, for_preview=True
    )
    print(tree_output)
    file_contents_list, total_tokens = gather_file_contents(
        root_path, excludes, includes
    )
    initial_content = combine_initial_content(
        root_path, excludes, includes, context_prompt, max_tokens
    )

    if generate_pdf_flag:
        generate_pdf(root_path, tree_output, file_contents_list)

    if generate_md_flag:
        generate_md(root_path, tree_output, file_contents_list)

    if not generate_pdf_flag and not generate_md_flag:
        handle_chunking_and_output(
            initial_content, file_contents_list, max_tokens, verbose
        )
```

if **name** == "**main**": args = parse_arguments() main(args.root_path, args.excludes, args.includes, args.max_tokens, args.config, args.verbose, args.ignore_gitignore, args.generate_pdf, # Pass the argument for generating PDF args.generate_md, # Pass the argument for generating Markdown)

ccontext/NotoSans-Bold.ttf

Contents:

ccontext/pdf_generator.py

Contents: from reportlab.lib.pagesizes import letter from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle from reportlab.platypus import (SimpleDocTemplate, Paragraph, Spacer, Table, TableStyle, PageBreak,) from reportlab.lib.units import inch from reportlab.lib import colors import os import re

class PDFGenerator: def **init**(self, output_path): self.output_path = output_path self.styles = getSampleStyleSheet() self.story = [] self.toc = [] self.custom_styles = self.create_custom_styles()

```
def create_custom_styles(self):
    custom_styles = getSampleStyleSheet()
    custom_styles.add(
        ParagraphStyle(
            name="TOC", fontSize=12, textColor=colors.blue, underline=True
        )
    )
    custom_styles.add(
        ParagraphStyle(name="FileTree", fontSize=10, leading=12,
spaceAfter=6)
    )
    custom_styles.add(
        ParagraphStyle(name="FileContent", fontSize=10, leading=12,
spaceAfter=6)
    )
    return custom_styles

def create_pdf(self, tree_content, file_contents_list):
    self.doc = SimpleDocTemplate(self.output_path, pagesize=letter)
    self.story.append(
        Paragraph("Directory and File Contents", self.styles["Title"])
    )
    self.story.append(Spacer(1, 0.2 * inch))
    self.add_table_of_contents()
    self.add_tree_section(tree_content)
    self.add_file_sections(file_contents_list)
    self.doc.build(
        self.story,
        onFirstPage=self.add_page_number,
        onLaterPages=self.add_page_number,
    )
    print(f"PDF generated at {self.output_path}")

def add_tree_section(self, tree_content):
    self.story.append(Paragraph("File Tree", self.styles["Heading2"]))
    for line in tree_content.splitlines():
        line = (
            line.replace("[DIR]", "□")
            .replace("[FILE]", "□")
            .replace("[EXCLUDED]", "□")
        )
        self.story.append(Paragraph(line, self.custom_styles["FileTree"]))
    self.story.append(PageBreak())

def add_file_sections(self, file_contents_list):
    for file_content in file_contents_list:
        match = re.match(
            r"#### □ (.+)\n\*\*Contents:\*\*\n(.+)", file_content,
re.DOTALL
        )
        if match:
            file_path = match.group(1)
```

```
            content = match.group(2)
            section_anchor = f"section_{len(self.toc)}"
            self.toc.append((file_path, section_anchor))
            self.story.append(
                Paragraph(
                    f'<a name="{section_anchor}">{file_path}</a>',
                    self.styles["Heading2"],
                )
            )
            self.story.append(Spacer(1, 0.1 * inch))
            self.story.append(
                Paragraph(
                    content.replace("\n", "<br />"),
                    self.custom_styles["FileContent"],
                )
            )
            self.story.append(PageBreak())

    def add_table_of_contents(self):
        self.story.append(Paragraph("Table of Contents",
    self.styles["Heading2"]))
        toc_entries = []
        for file_path, section_anchor in self.toc:
            toc_entries.append(
                [
                    Paragraph(
                        f'<a href="#{section_anchor}">{file_path}</a>',
                        self.custom_styles["TOC"],
                    )
                ]
            )
        if toc_entries:
            table = Table(toc_entries, colWidths=[7 * inch])
            table.setStyle(
                TableStyle(
                    [
                        ("TEXTCOLOR", (0, 0), (-1, -1), colors.blue),
                        ("VALIGN", (0, 0), (-1, -1), "TOP"),
                    ]
                )
            )
            self.story.append(table)
            self.story.append(PageBreak())

    def add_page_number(self, canvas, doc):
        canvas.saveState()
        canvas.setFont("Helvetica", 10)
        canvas.drawString(inch, 0.75 * inch, f"Page {doc.page}")
        canvas.restoreState()
```

def generate_pdf(root_path, tree_content, file_contents_list): output_path = os.path.join(root_path, "output.pdf") pdf_gen = PDFGenerator(output_path) pdf_gen.create_pdf(tree_content, file_contents_list)

if **name** == "**main**": import argparse from ccontext.main import load_config, print_file_tree, gather_file_contents

```
parser = argparse.ArgumentParser(
    description="Generate PDF of directory tree and file contents."
)
parser.add_argument(
    "root_path", type=str, help="The root path of the directory to
process."
)
parser.add_argument(
    "-c",
    "--config",
    type=str,
    help="Path to a custom configuration file.",
    default=None,
)
args = parser.parse_args()

config = load_config(args.root_path, args.config)
excludes, includes = config.get("excluded_folders_files", []), []
max_tokens = config.get("max_tokens", 32000)

tree_content = print_file_tree(args.root_path, excludes, includes,
max_tokens)
file_contents_list, _ = gather_file_contents(args.root_path, excludes,
includes)

generate_pdf(args.root_path, tree_content, file_contents_list)
```

ccontext/argument_parser.py

Contents: import argparse import sys from colorama import Fore, Style import os

def parse_arguments(): """Parse command-line arguments.""" parser = argparse.ArgumentParser(description="A script to display a directory structure and file contents with chunking if needed.") parser.add_argument("-p", "--root_path", required=False, default=os.getcwd(), help="The root path to start the directory tree (default: current directory).",) parser.add_argument("-e", "--excludes", required=False, default="", help='Additional files or directories to exclude, separated by "|", e.g. "node_modules|.git"',) parser.add_argument("-i", "--includes", required=False, default="", help='Files or directories to include, separated by "|", e.g. "important_file.txt|docs"',) parser.add_argument("-m", "--max_tokens", required=False, type=int, help="Maximum number of tokens allowed before chunking.",) parser.add_argument("-c", "--config", required=False, default=None, help="Path to a custom configuration file.",) parser.add_argument("-v", "--verbose", action="store_true", help="Enable verbose output to stdout.",) parser.add_argument("-ig", "--ignore_gitignore", action="store_true", help="Ignore the .gitignore file for exclusions.",) parser.add_argument("--generate-pdf", action="store_true", help="Generate a PDF of the directory tree and file contents.",) parser.add_argument("--generate-md", action="store_true", help="Generate a Markdown file of the directory tree and file contents.",)

```
args, unknown = parser.parse_known_args()
if unknown:
    print(f"{Fore.RED}Unrecognized arguments: {' '.join(unknown)}
{Fore.RESET}")
    parser.print_help()
    sys.exit(1)

return args
```

ccontext/file_system.py

Contents: import os from typing import List, Tuple import pathspec from ccontext.tokenizer import tokenize_text

def parse_gitignore(gitignore_path: str) -> List[str]: """Parses the .gitignore file and returns a list of patterns.""" if not os.path.exists(gitignore_path): return [] with open(gitignore_path, "r") as file: patterns = file.read().splitlines() return patterns

def is_excluded(path: str, excludes: List[str], includes: List[str]) -> bool: """Checks if a path should be excluded using pathspec.""" spec = pathspec.PathSpec.from_lines("gitwildmatch", excludes) for include_pattern in includes: if spec.match_file(include_pattern): return False return spec.match_file(path)

def get_file_token_length(file_path: str) -> int: """Returns the token length of a file.""" try: with open(file_path, "rb") as f: header = f.read(64) if b"\x00" in header: # if binary data return 0 f.seek(0) contents = f.read().decode("utf-8") tokens = tokenize_text(contents) return len(tokens) except Exception as e: return -1

def collect_excludes_includes(default_excludes: List[str], additional_excludes: List[str], additional_includes: List[str], root_path: str, ignore_gitignore: bool,) -> Tuple[List[str], List[str]]: """Combines default excluded items with additional exclusions and includes, and parses .gitignore.""" if isinstance(additional_excludes, str): additional_excludes = additional_excludes.split("|") if isinstance(additional_includes, str): additional_includes = additional_includes.split("|")

```
excludes = default_excludes + (additional_excludes if additional_excludes
else [])
includes = additional_includes if additional_includes else []

if not ignore_gitignore:
    gitignore_patterns = parse_gitignore(os.path.join(root_path,
".gitignore"))
    excludes.extend(gitignore_patterns)

return excludes, includes
```

def print_tree(root: str, root_path: str, excludes: List[str], includes: List[str], max_tokens: int, indent: str = "",) -> str: """Prints the file structure of the directory tree.""" items = sorted(os.listdir(root)) tree_output =

"" for item in items: full_path = os.path.join(root, item) relative_path = os.path.relpath(full_path, start=root_path)

```
    if os.path.isdir(full_path):
        if is_excluded(relative_path, excludes, includes):
            tree_output += f"{indent}[Excluded] □□ {relative_path}\n"
        else:
            tree_output += f"{indent}□ {relative_path}\n"
            tree_output += print_tree(
                full_path,
                root_path,
                excludes,
                includes,
                max_tokens,
                indent + "    ",
            )
    else:
        token_length = get_file_token_length(full_path)
        if is_excluded(relative_path, excludes, includes):
            tree_output += f"{indent}[Excluded] □□ {relative_path}\n"
        else:
            if token_length > max_tokens:
                tree_output += f"{indent}□ {Fore.RED}{token_length}
{Style.RESET_ALL} {relative_path}\n"
            else:
                tree_output += f"{indent}□ {token_length}
{relative_path}\n"
    return tree_output
```

□ ccontext/cli.py

Contents: from ccontext.argument_parser import parse_arguments from ccontext.main import main as actual_main

def main(): args = parse_arguments() actual_main(root_path=args.root_path, excludes=args.excludes, includes=args.includes, max_tokens=args.max_tokens, config_path=args.config, verbose=args.verbose, ignore_gitignore=args.ignore_gitignore,)

□ ccontext/NotoSans-Regular.ttf

Contents:

□ ccontext/content_handler.py

Contents: import os import re from ccontext.file_system import is_excluded, print_tree from ccontext.tokenizer import tokenize_text

def print_file_tree(root_path: str, excludes: list, includes: list, max_tokens: int, for_preview: bool = False,) -> str: """Print and capture the file tree section.""" tree_output = print_tree(root_path, root_path, excludes, includes, max_tokens)

```
header = (
    "========== File Tree ==========\n"
    if for_preview
    else "### ========== File Tree ==========\n"
)
footer = (
    "========== End of File Tree ==========\n"
    if for_preview
    else "### ========== End of File Tree ==========\n"
)

return f"{header}{tree_output}{footer}"
```

def gather_file_contents(root_path: str, excludes: list, includes: list) -> list: """Gather individual file contents for chunking.""" file_contents_list = [] total_tokens = 0

```
for dirpath, dirs, files in os.walk(root_path, topdown=True):
    dirs[:] = [
        d
        for d in dirs
        if not is_excluded(
            os.path.relpath(os.path.join(dirpath, d), start=root_path),
            excludes,
            includes,
        )
    ]
    for file in files:
        full_path = os.path.join(dirpath, file)
        relative_file_path = os.path.relpath(full_path, start=root_path)
        if is_excluded(relative_file_path, excludes, includes):
            continue  # Skip excluded files
        try:
            with open(full_path, "rb") as f:
                header = f.read(64)
                if b"\x00" in header:  # if binary data
                    outputString = f"\n#### 
{relative_file_path}\n**Contents:**\n<Binary data>\n"
                    file_contents_list.append(outputString)
                else:  # if text data
                    f.seek(0)
                    contents = f.read().decode("utf-8")
                    tokens = tokenize_text(contents)
                    total_tokens += len(tokens)
                    outputString = f"\n#### 
{relative_file_path}\n**Contents:**\n{contents}\n"
                    file_contents_list.append(outputString)
        except Exception as e:
            file_contents_list.append(
                f"\n#### ⚠ {relative_file_path}\n**Contents:**\nError
reading file {relative_file_path}: {e}\n"
```

```
            )
    return file_contents_list, total_tokens
```

def combine_initial_content(root_path: str, excludes: list, includes: list, context_prompt: str, max_tokens: int) -> str: """Combine the initial content for the output."""" context_prompt = f"## {context_prompt}\n\n" header = f"## Root Path: {root_path}\n\n" tree_output = print_file_tree(root_path, excludes, includes, max_tokens)

```
    return f"{context_prompt}{header}{tree_output}"
```

□ ccontext/NotoSans-MediumItalic.ttf

Contents:

□ ccontext/config.json

Contents: { "verbose": false, "max_tokens": 32000, "model_type": "gpt-4o", "buffer_size": 0.05, "excluded_folders_files": [".git", "bin", "build", "node_modules", "venv", "**pycache**", "package-lock.json", "ccontext.egg-info", "dist/", "**tests**", "coverage", ".next"], "context_prompt": "[[SYSTEM INSTRUCTIONS]] The following output presents a detailed directory structure and file contents from a specified root path. The file tree includes both excluded and included files and directories, clearly marking exclusions. Each file's content is displayed with comprehensive headings and separators to enhance readability and facilitate detailed parsing for extracting hierarchical and content-related insights. If the data represents a codebase, interpret and handle it as such, providing appropriate assistance as a programmer AI assistant. [[END SYSTEM INSTRUCTIONS]]" }

□ ccontext/md_generator.py

Contents: import os import re

def generate_md(root_path, tree_content, file_contents_list): output_path = os.path.join(root_path, "output.md")

```
    with open(output_path, "w", encoding="utf-8") as md_file:
        md_file.write("# Directory and File Contents\n\n")

        # File Tree
        md_file.write("## File Tree\n\n")
        md_file.write(f"{tree_content}\n\n")

        # File Contents
        md_file.write("## File Contents\n\n")
        for file_content in file_contents_list:
            match = re.match(
                r"#### □ (.+)\n\*\*Contents:\*\*\n(.+)", file_content,
    re.DOTALL
            )
```

```
        if match:
            file_path = match.group(1)
            content = match.group(2)
            md_file.write(f"### {file_path}\n\n")
            md_file.write("#### Contents\n\n")
            md_file.write("```\n")
            md_file.write(content)
            md_file.write("\n```\n\n")

print(f"Markdown file generated at {output_path}")
```

def gather_file_contents(root_path: str, excludes: list, includes: list) -> list: """Gather individual file contents for chunking.""" file_contents_list = [] total_tokens = 0

```
for dirpath, dirs, files in os.walk(root_path, topdown=True):
    dirs[:] = [
        d
        for d in dirs
        if not is_excluded(
            os.path.relpath(os.path.join(dirpath, d), start=root_path),
            excludes,
            includes,
        )
    ]
    for file in files:
        full_path = os.path.join(dirpath, file)
        relative_file_path = os.path.relpath(full_path, start=root_path)
        if is_excluded(relative_file_path, excludes, includes):
            continue  # Skip excluded files
        try:
            with open(full_path, "rb") as f:
                header = f.read(64)
                if b"\x00" in header:  # if binary data
                    outputString = f"\n#### 
{relative_file_path}\n**Contents:**\n<Binary data>\n"
                    file_contents_list.append(outputString)
                else:  # if text data
                    f.seek(0)
                    contents = f.read().decode("utf-8")
                    outputString = f"\n#### 
{relative_file_path}\n**Contents:**\n{contents}\n"
                    file_contents_list.append(outputString)
        except Exception as e:
            file_contents_list.append(
                f"\n#### ⚠ {relative_file_path}\n**Contents:**\nError
reading file {relative_file_path}: {e}\n"
            )
return file_contents_list, total_tokens
```

def is_excluded(path: str, excludes: list, includes: list) -> bool: """Checks if a path should be excluded using pathspec.""" spec = pathspec.PathSpec.from_lines("gitwildmatch", excludes) for include_pattern in includes: if spec.match_file(include_pattern): return False return spec.match_file(path)

def print_tree(root: str, root_path: str, excludes: list, includes: list, max_tokens: int, indent: str = "",) -> str: """Prints the file structure of the directory tree.""" items = sorted(os.listdir(root)) tree_output = "" for item in items: full_path = os.path.join(root, item) relative_path = os.path.relpath(full_path, start=root_path)

```
    if os.path.isdir(full_path):
        if is_excluded(relative_path, excludes, includes):
            tree_output += f"{indent}[Excluded] □□ {relative_path}\n"
        else:
            tree_output += f"{indent}□ {relative_path}\n"
            tree_output += print_tree(
                full_path,
                root_path,
                excludes,
                includes,
                max_tokens,
                indent + "    ",
            )
    else:
        token_length = get_file_token_length(full_path)
        if is_excluded(relative_path, excludes, includes):
            tree_output += f"{indent}[Excluded] □□ {relative_path}\n"
        else:
            if token_length > max_tokens:
                tree_output += f"{indent}□ {Fore.RED}{token_length}
{Style.RESET_ALL} {relative_path}\n"
            else:
                tree_output += f"{indent}□ {token_length}
{relative_path}\n"
return tree_output
```

if **name** == "**main**": import argparse from ccontext.main import load_config, print_file_tree, gather_file_contents

```
parser = argparse.ArgumentParser(
    description="Generate Markdown file of directory tree and file
contents."
)
parser.add_argument(
    "root_path", type=str, help="The root path of the directory to
process."
)
parser.add_argument(
    "-c",
    "--config",
    type=str,
    help="Path to a custom configuration file.",
```

```
    default=None,
)
args = parser.parse_args()

config = load_config(args.root_path, args.config)
excludes, includes = config.get("excluded_folders_files", []), []
max_tokens = config.get("max_tokens", 32000)

tree_content = print_file_tree(args.root_path, excludes, includes,
max_tokens)
file_contents_list, _ = gather_file_contents(args.root_path, excludes,
includes)

generate_md(args.root_path, tree_content, file_contents_list)
```

ccontext/output_handler.py

Contents: from colorama import Fore, Style from ccontext.utils import format_number from ccontext.tokenizer import chunk_text, tokenize_text from ccontext.clipboard import copy_to_clipboard

def handle_chunking_and_output(initial_content: str, file_contents_list: list, max_tokens: int, verbose: bool,): """Calculate token length and handle chunking if necessary.""" end_marker = "### ========== End of Detailed File Contents ==========\n" full_output = initial_content + "".join(file_contents_list) + end_marker total_tokens = len(tokenize_text(full_output))

```
token_info = f"\nTokens: {Fore.GREEN if total_tokens <= max_tokens else
Fore.RED}{format_number(total_tokens)}
{Style.RESET_ALL}/{format_number(max_tokens)}"

if total_tokens > max_tokens:
    print(
        f"{Fore.RED}The output exceeds the token limit and will need to be
chunked.{Style.RESET_ALL}"
    )
    print(f"\n{token_info}")

    chunks = chunk_text(
        [initial_content] + file_contents_list + [end_marker], max_tokens
    )

    # Print chunk sizes
    chunk_sizes = [len(tokenize_text(chunk)) for chunk in chunks]
    for i, size in enumerate(chunk_sizes):
        print(f"Chunk {i + 1}: {size} tokens")

    for i, chunk in enumerate(chunks):
        chunk_header = f"### Chunk {i + 1} of {len(chunks)}"
        if i == 0:
            chunk = f"""## Initialization\nThe following content will be
delivered in multiple chunks. This is to ensure all data is processed
correctly. There will be a total of {len(chunks)} chunks. Thoroughly read
```

```
the chunk and reply with a short summary of the content that was inserted.
Until you receive the final chunk, this will be marked by '###This is the
final chunk.###', you will have to make a summary of all the summaries
that you gave. Once you have received the final chunk, reply with the
final summary. '\n\n{chunk_header}: File Tree and Initial File
Contents\n{chunk}\n###More chunks to follow...###"""
        elif i == len(chunks) - 1:
            chunk = f"{chunk_header}\n{chunk}\n###This is the final
chunk.###"
        else:
            previous_chunk_summary = "Previous chunk ended with:\n" +
"\n".join(
                chunks[i - 1].splitlines()[-10:]
            )
            chunk = f"{chunk_header} (continued from Chunk
{i})\n{previous_chunk_summary}\n{chunk}\n###More chunks to follow...###"

        print(
            f"{Fore.MAGENTA}(Chunk {i + 1}/{len(chunks)}){Style.RESET_ALL}
{Fore.CYAN}Press Enter to continue or type 'q' to abort:
{Style.RESET_ALL}",
            end="",
        )
        user_input = input()
        if user_input.lower() == "q":
            print(f"{Fore.YELLOW}Operation aborted by user.
{Style.RESET_ALL}")
            break

        if verbose:
            print(f"\n{chunk_header}:")
            print(chunk)
        copy_to_clipboard(chunk)

    if verbose:
        print(
            f"{Fore.MAGENTA}\nSuccessfully finished all chunks
({len(chunks)}/{len(chunks)}){Style.RESET_ALL}"
        )
else:
    print(token_info)
    if verbose:
        print(full_output)
    copy_to_clipboard(full_output)
```

ccontext/NotoEmoji-VariableFont_wght.ttf

Contents:

ccontext/clipboard.py

Contents: import subprocess import platform from colorama import Fore, Style import pyperclip

def is_wsl2() -> bool: """Detect if running under WSL2.""" try: with open("/proc/version", "r") as file: version_info = file.read().lower() return "microsoft" in version_info except FileNotFoundError: return False

def check_and_install_utf8clip(): """Check if utf8clip is installed, and install it if necessary.""" try: result = subprocess.run(["utf8clip.exe"], stdin=subprocess.DEVNULL) if result.returncode != 0: raise FileNotFoundError except FileNotFoundError: print("utf8clip is not installed. Attempting to install...") install_process = subprocess.Popen(["powershell.exe", "dotnet", "tool", "install", "--global", "utf8clip"], text=True,) install_process.communicate() if install_process.returncode != 0: raise RuntimeError("Error installing utf8clip using PowerShell.") except Exception as e: print(f"An error occurred: {e}") raise RuntimeError("Failed to check or install utf8clip.")

def copy_to_clipboard(text: str): """Copies the given text to the clipboard.""" system = platform.system()

```
try:
    if system == "Windows":
        pyperclip.copy(text)
    elif system == "Darwin":
        pyperclip.copy(text)
    elif system == "Linux":
        if (
            is_wsl2()
        ):  # WSL2 requires utf8clip,
https://github.com/asweigart/pyperclip/issues/244
            check_and_install_utf8clip()
            process = subprocess.Popen(
                "utf8clip.exe", stdin=subprocess.PIPE, shell=True
            )
            process.communicate(text.encode("utf-8"))
        else:
            pyperclip.copy(text)
    else:
        # Fallback to pyperclip for other systems
        pyperclip.copy(text)

    print(f"{Fore.GREEN}\nOutput copied to clipboard!{Style.RESET_ALL}")
except Exception as e:
    print(f"{Fore.RED}\nAn error occurred: {e}{Style.RESET_ALL}")
```

□ ccontext/tokenizer.py

Contents: import tiktoken

def set_model_type_and_buffer(model_type: str, buffer_size: float): """ Sets the model type and buffer size for tokenization.

```
Args:
    model_type (str): The type of model to use for encoding.
    buffer_size (float): The buffer size as a fraction of max_tokens.
"""
```

```
    global MODEL_TYPE, BUFFER_SIZE
    MODEL_TYPE = model_type
    BUFFER_SIZE = buffer_size
```

def tokenize_text(text: str) -> list: """ Tokenizes the given text using the specified model type.

```
    Args:
        text (str): The text to be tokenized.

    Returns:
        list: A list of token ids.
    """
    encoding = tiktoken.encoding_for_model(MODEL_TYPE)
    return encoding.encode(text)
```

def chunk_text(file_contents: list, max_tokens: int) -> list: """ Splits the file contents into chunks that fit within the max_tokens limit, considering a buffer size.

```
    Args:
        file_contents (list): A list of strings representing file contents.
        max_tokens (int): The maximum number of tokens allowed per chunk.

    Returns:
        list: A list of strings, each representing a chunk.
    """
    # Calculate the number of tokens to reserve as a buffer
    buffer_tokens = int(max_tokens * BUFFER_SIZE)
    available_tokens = max_tokens - buffer_tokens

    current_chunk = ""  # The current chunk being built
    current_chunk_tokens = 0  # The token count of the current chunk
    chunks = []  # List to store all the chunks

    def add_chunk():
        """
        Adds the current chunk to the list of chunks and resets the current
    chunk.
        """
        nonlocal current_chunk, current_chunk_tokens
        if current_chunk.strip():  # Check if the current chunk is not empty
            chunks.append(current_chunk.strip())
        current_chunk = ""
        current_chunk_tokens = 0

    for file_content in file_contents:
        # Ensure the file content is a string
        if not isinstance(file_content, str):
            raise ValueError(f"Expected a string but got
    {type(file_content)}")
```

```
    # Tokenize the current file content
    tokens = tokenize_text(file_content)
    token_count = len(tokens)

    # If the file content exceeds the available tokens, split it into
smaller pieces
    if token_count > available_tokens:
        split_contents = [
            file_content[i : i + available_tokens]
            for i in range(0, len(file_content), available_tokens)
        ]
        for split_content in split_contents:
            split_tokens = tokenize_text(split_content)
            split_token_count = len(split_tokens)
            if current_chunk_tokens + split_token_count >
available_tokens:
                add_chunk()
            current_chunk += split_content
            current_chunk_tokens += split_token_count
    else:
        # If adding the current file content exceeds the available tokens,
create a new chunk
        if current_chunk_tokens + token_count > available_tokens:
            add_chunk()
        current_chunk += file_content
        current_chunk_tokens += token_count

# Add the final chunk if it contains any content
if current_chunk.strip():
    add_chunk()

return chunks
```

Set the default model type and buffer size

set_model_type_and_buffer("gpt-4", 0.05)

□ ccontext/init.py

Contents:

========== End of Detailed File Contents ==========