repo2pdf - repo2pdf

/Users/harissujethan/Desktop/repo2pdf

Generated 2025-09-05 15:50 UTC

Table of Contents

Overview

repo2pdf - repo2pdf

/Users/harissujethan/Desktop/repo2pdf

CLI tool to convert your repositories into clean PDFs and structured JSON outputs, **designed for giving LLMs full context of your codebase**

Key Features

• Convert **local** or **remote GitHub repositories**
• Generate **PDFs** containing full file structures and contents
• Output structured **JSON summaries**
• Exclude unnecessary file types automatically
• **repo_output.pdf**
• **repo_output.json**

Quick Usage

```
pip install repo2pdf
```

Option 2: Install from Source

Clone the repository and install locally:

```bash
git clone https://github.com/haris-sujethan/repo-2-pdf
cd repo-2-pdf
pip install -r requirements.txt
```

Then choose one of the following:

Local development install (recommended):

```bash
pip install -e .
repo2pdf
```

Run without installing:

```bash
python -m repo2pdf.cli
```

Usage

Run the CLI tool:

```bash
repo2pdf
```

Files & Languages

• .py - 10 file(s)
• (no ext) - 1 file(s)

- .md - 1 file(s)
- .toml - 1 file(s)
- .txt - 1 file(s)
- Total files: 14

Dependencies

- fpdf2
- GitPython
- inquirer
- pathspec
- pytest
- pygments>=2.13

.gitignore

Transact-SQL • 9 line(s)

```
repo2pdf.egg-info/
__pycache__/
*.py[cod]
*$py.class
dist/
build/
*.egg-info/
.pytest_cache/
node_modules/
```

README.md

Markdown • 70 line(s)

```
# repo-2-pdf

CLI tool to convert your repositories into clean PDFs and structured JSON outputs, **designed f
or giving LLMs full context of your codebase**

## Features

- Convert **local** or **remote GitHub repositories**
- Generate **PDFs** containing full file structures and contents
- Output structured **JSON summaries**
- Exclude unnecessary file types automatically

## Installation

### Option 1: Install from [PyPI](https://pypi.org/project/repo2pdf/) (Recommended)

```bash
pip install repo2pdf
```

Option 2: Install from Source

Clone the repository and install locally:

```bash
git clone https://github.com/haris-sujethan/repo-2-pdf
cd repo-2-pdf
pip install -r requirements.txt
```

Then choose one of the following:

Local development install (recommended):

```bash
pip install -e .
repo2pdf
```
```

```
39 **Run without installing:**

41 ```bash
42 python -m repo2pdf.cli
43 ```

45 ## Usage

47 Run the CLI tool:

49 ```bash
50 repo2pdf
51 ```

53 **Follow the interactive prompts:**

55 1. Select local or remote repository
56 2. Provide the local repo path or GitHub URL
57 3. Choose an output location
58 4. Exclude any file types you don't want included (e.g., `.png`, `.jpg`)
59 5. Optionally generate a JSON summary alongside the PDF

61 ## Example CLI Flow

63 <img src="https://raw.githubusercontent.com/haris-sujethan/repo-2-pdf/main/repo2pdf/docs/images
63 /example-CLI.png" alt="Example CLI Interface" width="850"/>

65 ## Example Outputs

67 Example outputs are available in the `/examples` folder:

69 - **repo_output.pdf**
70 - **repo_output.json**
```

pyproject.toml

TOML • 23 line(s)

```toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "repo2pdf"
version = "0.1.4"
description = "Convert coding repositories into PDFs and JSON summaries"
authors = [
  { name="Haris Sujethan", email="your-email@example.com" },
]
license = {text = "MIT"}
readme = "README.md"
requires-python = ">=3.7"
dependencies = [
  "fpdf2",
  "GitPython",
  "inquirer",
  "pathspec",
]

[project.scripts]
repo2pdf = "repo2pdf.cli:main"
```

repo2pdf/__init__.py

Python • 3 line(s)

```python
# __init__.py

__version__ = '0.1.0'
```

repo2pdf/cli.py

Python • 50 line(s)

```python
# repo2pdf/cli.py
from __future__ import annotations

import inquirer
from repo2pdf.core import process_local_repo, process_remote_repo

def main():
    ascii_art = r"""
 ______   ______  ______  ______           _____           ______  ______  ______
/_____/\ /_____/\/_____/\/_____/\         /_____/\        /_____/\/_____/\/_____/\
\:::_ \ \\::::_\/\:::_ \ \\:::_ \ \  ______\:::_:\ \ ______\:::_ \ \:::_ \ \\::::_\/_
 \:(_) ) )\:\/___/\:(_) \ \:\ \ \ \ \/_____/\  _\:\|/_____/\:(_) \ \:\ \ \ \:\/___/\
  \: __ `\ \::___\/\: ___\/\:\ \ \ \__::::\/ /::_/_\__::::\/\: ___\/\:\ \ \ \::._\/
   \ \ `\ \ \:\____/\ \ \   \:\_\ \ \        \:\____/\       \ \ \   \:\/.:| \:\ \
    \_\/ \_\/\_____\/\_\/    \_____\/         \_____\/        \_\/    \____/_/\_\/


Welcome to repo2pdf - convert your repositories to PDFs
    """
    print(ascii_art)

    repo_type_q = [
        inquirer.List(
            "repo_type",
            message="Do you want to generate a PDF from a local or remote repo?",
            choices=["Local", "Remote"],
        )
    ]
    repo_type = inquirer.prompt(repo_type_q)["repo_type"]

    json_q = [inquirer.Confirm("json", message="Do you also want to generate a JSON version?",
default=False)]
    want_json = inquirer.prompt(json_q)["json"]

    output_q = [inquirer.Text("output", message="Provide output path for PDF (press enter for d
efault)")]
    output_path = inquirer.prompt(output_q)["output"]
```

```
    exclude_q = [inquirer.Text("exclude", message="Enter file extensions to exclude (e.g. .png,.jpg,.exe), or press enter to skip")]
    exclude_input = inquirer.prompt(exclude_q)["exclude"]
    exclude_list = [e.strip() for e in exclude_input.split(",")] if exclude_input else []

    if repo_type == "Local":
        path_q = [inquirer.Text("path", message="Provide local repo path (or press enter if current directory)")]
        path = inquirer.prompt(path_q)["path"]
        process_local_repo(path, want_json, output_path, exclude_list)
    else:
        url_q = [inquirer.Text("url", message="Provide GitHub repo URL (e.g. https://github.com/user/repo)")]
        url = inquirer.prompt(url_q)["url"]
        process_remote_repo(url, want_json, output_path, exclude_list)

if __name__ == "__main__":
    main()
```

repo2pdf/core.py

Python • 223 line(s)

```python
# repo2pdf/core.py
from __future__ import annotations

import json
import os
import tempfile
from datetime import datetime
from pathlib import Path
from typing import List, Tuple, Dict, Any

from pathspec import PathSpec
from pathspec.patterns.gitwildmatch import GitWildMatchPattern

from repo2pdf.pdf import generate_pdf, PDFMeta  # updated renderer

# Directories we always skip anywhere in the path
EXCLUDE_DIRS = {
    ".git", ".github", "node_modules", "dist", "build", "out", "target",
    "__pycache__", ".mypy_cache", ".pytest_cache", ".venv", "venv",
    ".tox", ".idea", ".vscode"
}

# Files we always skip by name
ALWAYS_SKIP_FILENAMES = {"repo_output.pdf", "repo2pdf.pdf"}

# Obvious binary extensions (expanded)
BINARY_EXTS = {
    ".png", ".jpg", ".jpeg", ".gif", ".webp", ".ico",
    ".pdf", ".zip", ".gz", ".7z", ".tar", ".rar",
    ".woff", ".woff2", ".ttf", ".otf", ".eot",
    ".bmp", ".tiff", ".psd", ".svg",
    ".mp3", ".mp4", ".mov", ".avi", ".mkv",
    ".exe", ".dll", ".so", ".dylib",
    ".bin", ".class", ".o", ".a",
    ".lock",
}

# Max size we'll read as "text"
```

```python
MAX_TEXT_BYTES = 1_000_000  # 1 MB


def _gitignore(root: Path) -> PathSpec:
    gi = root / ".gitignore"
    lines = gi.read_text().splitlines() if gi.exists() else []
    return PathSpec.from_lines(GitWildMatchPattern, lines)


def _skip_dir(p: Path) -> bool:
    return any(part in EXCLUDE_DIRS for part in p.parts)


def _looks_binary(head: bytes) -> bool:
    if b"\x00" in head:
        return True
    if head.startswith(b"%PDF-"):
        return True
    if head.startswith(b"\x1f\x8b"):       # gzip
        return True
    if head.startswith(b"PK\x03\x04"):     # zip/jar/docx/etc.
        return True
    printable = sum(32 <= b <= 126 or b in (9, 10, 13) for b in head)
    return (len(head) - printable) / max(1, len(head)) > 0.20


def _collect_files(root: Path, exclude_exts: set[str]) -> Tuple[List[Tuple[str, str]], Dict[st
r, Any]]:
    spec = _gitignore(root)
    files: List[Tuple[str, str]] = []
    counts = {
        "gitignored": 0,
        "manual_exclude": 0,
        "excluded_dir": 0,
        "binary_ext": 0,
        "binary_magic": 0,
        "too_large": 0,
        "read_errors": 0,
    }
```

```
    for p in root.rglob("*"):
        if p.is_dir():
            if _skip_dir(p):
                # skip entire subtree
                counts["excluded_dir"] += 1
                continue
            continue

        rel = p.relative_to(root).as_posix()

        # .gitignore + manual skips
        if rel.startswith(".git/") or spec.match_file(rel):
            counts["gitignored"] += 1
            continue
        if p.name in ALWAYS_SKIP_FILENAMES:
            counts["manual_exclude"] += 1
            continue
        if _skip_dir(p):
            counts["excluded_dir"] += 1
            continue

        ext = p.suffix.lower()
        if ext in exclude_exts or ext in BINARY_EXTS:
            counts["binary_ext"] += 1
            continue

        try:
            if p.stat().st_size > MAX_TEXT_BYTES:
                counts["too_large"] += 1
                continue
        except Exception:
            pass

        try:
            with p.open("rb") as f:
                head = f.read(4096)
                if _looks_binary(head):
                    counts["binary_magic"] += 1
                    continue
                data = head + f.read()
```

```
            text = data.decode("utf-8", errors="replace")
        except Exception:
            counts["read_errors"] += 1
            continue

        files.append((rel, text))

    files.sort(key=lambda t: t[0])
    summary = {"counts": counts, "notes": [], "packed_small_files": 0}
    return files, summary


def _resolve_output_path(output_path: str | None, root: Path) -> Path:
    """
    If output_path is:
      - empty/None -> use CWD/repo2pdf-<root>-YYYYmmdd-HHMM.pdf
      - a directory -> append repo2pdf-<root>-YYYYmmdd-HHMM.pdf
      - a file path without .pdf -> add .pdf
      - a file path with .pdf -> use as-is
    """
    ts = datetime.now().strftime("%Y%m%d-%H%M")
    default_name = f"repo2pdf-{root.name}-{ts}.pdf"

    if not output_path or output_path.strip() == "":
        return Path(os.getcwd()) / default_name

    p = Path(output_path).expanduser()
    if p.is_dir() or str(output_path).endswith(os.sep):
        return p / default_name

    if p.suffix.lower() != ".pdf":
        p = p.with_suffix(".pdf")
    return p


def _build_json_summary(root: Path, files: List[Tuple[str, str]]) -> dict:
    from datetime import datetime, timezone
    entries = []
    for rel, content in files:
        p = root / rel
```

```
        try:
            size = p.stat().st_size
        except Exception:
            size = len(content.encode("utf-8", errors="ignore"))
        lines = content.count("\n") + (1 if content and not content.endswith("\n") else 0)
        entries.append({
            "path": rel,
            "ext": Path(rel).suffix.lower(),
            "size_bytes": size,
            "line_count": lines,
        })
    return {
        "repo_name": root.name,
        "root": str(root),
        "file_count": len(entries),
        "generated_at": datetime.now(timezone.utc).isoformat(),
        "files": entries,
    }


def _render(root: Path, output_path: str | None, exclude_list: list[str] | None, repo_url: str | None, want_json: bool):
    # Normalize CLI excludes (like ".png,.jpg") into a set of extensions
    exclude_exts = set()
    for item in (exclude_list or []):
        for token in item.split(","):
            token = token.strip()
            if token and token.startswith("."):
                exclude_exts.add(token.lower())

    files, summary = _collect_files(root, exclude_exts)

    meta = PDFMeta(
        title=f"repo2pdf - {root.name}",
        subtitle=str(root),
        repo_url=repo_url,
    )

    out_path = _resolve_output_path(output_path, root)
    out_path.parent.mkdir(parents=True, exist_ok=True)
```

```
    # Generate PDF (summary appended in appendix)
    generate_pdf(files, str(out_path), meta, summary=summary)

    if want_json:
        out_json = _build_json_summary(root, files)
        json_path = out_path.with_suffix(".json")
        json_path.write_text(json.dumps(out_json, indent=2), encoding="utf-8")

    print(f"\nPDF saved to: {out_path}")
    if want_json:
        print(f"JSON saved to: {out_path.with_suffix('.json')}")


# Public entry points expected by cli.py

def process_local_repo(path: str, want_json: bool, output_path: str | None, exclude_list: list
[str]):
    root = Path(path or ".").resolve()
    _render(root, output_path, exclude_list, repo_url=None, want_json=want_json)


def process_remote_repo(url: str, want_json: bool, output_path: str | None, exclude_list: list
[str]):
    from git import Repo  # requires GitPython
    with tempfile.TemporaryDirectory(prefix="repo2pdf_") as tmp:
        tmp_path = Path(tmp)
        Repo.clone_from(url, tmp_path)
        _render(tmp_path, output_path, exclude_list, repo_url=url, want_json=want_json)
```

repo2pdf/pdf.py

Python • 767 line(s)

```python
# repo2pdf/pdf.py
# Clean, readable PDF renderer with *native* syntax highlighting:
# - Cover
# - Table of Contents AT THE START (reserved then backfilled; truncates with a note)
# - Text-only Overview (LLM + human friendly; strips README images)
# - One section per file with Unicode-safe monospaced text
# - Native Pygments token coloring (no HTML), line numbers, light code background
# - Safe soft-wrapping; no empty background bands; robust around page breaks
# - Small-file packing: multiple tiny files share a page when space allows
# - Header shows: path • language • lines (per-page context)
# - Appendix: transparent "Skipped & condensed" summary

from __future__ import annotations

import os
import re
from dataclasses import dataclass
from datetime import datetime
from typing import Iterable, Tuple, Optional, List, Dict, Any

from fpdf import FPDF

# Pygments for lexing & token types
from pygments import lex
from pygments.lexers import get_lexer_for_filename, guess_lexer
from pygments.lexers.special import TextLexer
from pygments.token import Token

# ---------------------------------------------------------------------------
# Configuration
# ---------------------------------------------------------------------------

PACKAGE_DIR = os.path.dirname(__file__)
FONTS_DIR = os.path.join(PACKAGE_DIR, "fonts")

DEJAVU_SANS = os.path.join(FONTS_DIR, "DejaVuSans.ttf")
DEJAVU_SANS_BOLD = os.path.join(FONTS_DIR, "DejaVuSans-Bold.ttf")
DEJAVU_MONO = os.path.join(FONTS_DIR, "DejaVuSansMono.ttf")
```

```python
# Minimal text normalizer so DejaVu can render everything
CHAR_MAP = {
    # arrows, misc
    "⚠": "⚠", "→": "→", "←": "←",
    # smart punctuation -> ASCII
    "‐": "-", "‑": "-", "‒": "-", "–": "-", "—": "-",
    "“": '"', "”": '"', "„": '"', "‘": "'", "’": "'", "‚": "'", "<": "<", ">": ">",
    "\u00A0": " ",  # NBSP
}
```

```python
def normalize_text_for_pdf(s: str) -> str:
    s = (s or "").replace("\uFE0F", "")  # strip variation selector
    for k, v in CHAR_MAP.items():
        s = s.replace(k, v)
    return s
```

```python
@dataclass
class PDFMeta:
    title: str
    subtitle: Optional[str] = None
    repo_url: Optional[str] = None
    generated_at: Optional[datetime] = None
```

```python
class RepoPDF(FPDF):
    """FPDF renderer with a cover, ToC at start, text Overview, and per-file sections."""
```

```python
    def __init__(self, meta: PDFMeta):
        super().__init__(orientation="P", unit="mm", format="A4")
        # Reduced bottom margin from 16 to 10 for tighter spacing
        self.set_auto_page_break(auto=True, margin=10)
        self.meta = meta
        self._toc: List[Tuple[str, int, int]] = []  # (label, level, page)
        self._links: Dict[str, int] = {}
        self._toc_reserved_page: Optional[int] = None
        # Header state (per page)
        self._hdr_path: str = meta.title
        self._hdr_lang: str = ""
        self._hdr_lines: Optional[int] = None
        self._register_fonts()
        self._set_doc_info()
```

```
    # ---------------------------- Fonts & metadata ----------------------------
    def _register_fonts(self):
        for path in (DEJAVU_SANS, DEJAVU_SANS_BOLD, DEJAVU_MONO):
            if not (os.path.exists(path) and os.path.getsize(path) > 50_000):
                raise RuntimeError(
                    f"Missing/invalid font at {path}. Please vendor real DejaVu TTF binaries."
                )
        # Register Unicode-safe fonts (regular + bold only; no italics to prevent errors)
        self.add_font("DejaVu", style="", fname=DEJAVU_SANS, uni=True)
        self.add_font("DejaVu", style="B", fname=DEJAVU_SANS_BOLD, uni=True)
        self.add_font("DejaVuMono", style="", fname=DEJAVU_MONO, uni=True)
        self.set_font("DejaVu", size=11)

    def _set_doc_info(self):
        self.set_title(self.meta.title)
        if self.meta.subtitle:
            self.set_subject(self.meta.subtitle)
        if self.meta.repo_url:
            self.set_author(self.meta.repo_url)
        self.set_creator("repo2pdf")

    # ---------------------------- Header / Footer ----------------------------
    def header(self):
        # Header line + context
        self.set_font("DejaVu", size=9)
        self.set_text_color(60)
        self.set_x(self.l_margin)

        # Trim path to available width
        right_part = ""
        if self._hdr_lang or self._hdr_lines is not None:
            parts = [p for p in [self._hdr_lang, f"{self._hdr_lines} lines" if self._hdr_lines
else None] if p]
            right_part = " • ".join(parts)
        max_w = self.w - self.l_margin - self.r_margin

        left_txt = normalize_text_for_pdf(self._hdr_path)
        if right_part:
            # reserve space for right_part
            rp_w = self.get_string_width(" " + right_part)
```

```
            avail = max_w - rp_w
            # elide left if too long
            while self.get_string_width(left_txt) > avail and len(left_txt) > 4:
                left_txt = "…" + left_txt[1:]
            self.cell(avail, 6, left_txt, ln=0, align="L")
            # right-aligned meta
            self.set_xy(self.w - self.r_margin - rp_w, self.get_y())
            self.cell(rp_w, 6, right_part, ln=1, align="R")
        else:
            self.cell(0, 6, left_txt, ln=1, align="L")

        self.set_draw_color(220)
        self.set_line_width(0.2)
        y = self.get_y()
        self.line(self.l_margin, y, self.w - self.r_margin, y)
        # Reduced from ln(2) to ln(1)
        self.ln(1)
        self.set_text_color(0)

    def footer(self):
        self.set_y(-12)
        self.set_font("DejaVu", size=9)
        self.set_text_color(120)
        self.cell(0, 8, f"Page {self.page_no()}", align="C")
        self.set_text_color(0)

    # ------------------------------ Helpers ---------------------------------
    def _page_width_available(self) -> float:
        return self.w - self.l_margin - self.r_margin

    def _safe_multicell(self, text: str, line_h: float):
        """Reset X to left margin and use explicit width to avoid FPDF width errors."""
        self.set_x(self.l_margin)
        self.multi_cell(self._page_width_available(), line_h, text)

    # ------------------------------ High level ------------------------------
    def add_cover(self):
        # Header state for this page
        self._hdr_path = normalize_text_for_pdf(self.meta.title)
        self._hdr_lang = ""
```

```
        self._hdr_lines = None

        self.add_page()
        self.set_font("DejaVu", "B", 22)
        self.ln(25)  # Reduced from 30
        self._safe_multicell(normalize_text_for_pdf(self.meta.title), line_h=12)
        self.ln(3)  # Reduced from 4
        self.set_font("DejaVu", size=12)
        sub = self.meta.subtitle or "Repository to PDF"
        self._safe_multicell(normalize_text_for_pdf(sub), line_h=8)
        self.ln(3)  # Reduced from 4
        if self.meta.repo_url:
            url = normalize_text_for_pdf(self.meta.repo_url)
            self.set_text_color(60, 90, 200)
            self.set_x(self.l_margin)
            self.cell(self._page_width_available(), 8, url, align="C", ln=1, link=self.meta.re
po_url)
            self.set_text_color(0)
        self.ln(4)  # Reduced from 6
        when = (self.meta.generated_at or datetime.utcnow()).strftime("%Y-%m-%d %H:%M UTC")
        self.set_text_color(120)
        self.set_x(self.l_margin)
        self.cell(self._page_width_available(), 8, f"Generated {when}", align="C")
        self.set_text_color(0)

    def reserve_toc_page(self):
        """Reserve a page right after the cover for the ToC and remember its number."""
        # Header state for ToC page
        self._hdr_path = "Table of Contents"
        self._hdr_lang = ""
        self._hdr_lines = None

        self.add_page()
        self._toc_reserved_page = self.page_no()

    def render_toc_on_reserved_page(self):
        if not self._toc_reserved_page:
            return
        # Jump to the reserved page and render
        current_page = self.page_no()
```

```
199         current_x, current_y = self.get_x(), self.get_y()

201         self.page = self._toc_reserved_page
202         self.set_xy(self.l_margin, self.t_margin)

204         self.set_font("DejaVu", "B", 16)
205         self._safe_multicell("Table of Contents", line_h=10)
206         self.ln(1)  # Reduced from 2

208         # Guard: don't let ToC overflow this single page (truncate gracefully)
209         bottom_limit = self.h - self.b_margin
210         self.set_font("DejaVu", size=11)
211         truncated = False
212         for label, level, page in self._toc:
213             if self.get_y() + 8 > bottom_limit:
214                 truncated = True
215                 break
216             indent = " " * level
217             text = f"{indent}{normalize_text_for_pdf(label)}"
218             link_id = self._links.get(label)
219             y_before = self.get_y()
220             self.set_x(self.l_margin)
221             self.cell(self._page_width_available(), 7, text, ln=0, link=link_id)
222             self.set_xy(self.l_margin, y_before)
223             self.cell(self._page_width_available(), 7, str(page), align="R", ln=1)

225         if truncated:
226             self.ln(1)
227             self.set_font("DejaVu", "B", 10)
228             self._safe_multicell("… ToC truncated", line_h=6)

230         # Return to where we were (append mode)
231         self.page = current_page
232         self.set_xy(current_x, current_y)

234     def toc_add(self, label: str, level: int = 0):
235         self._toc.append((label, level, self.page_no()))
236         # Internal link target bookkeeping
237         try:
238             link_id = self.add_link()
```

```
            self._links[label] = link_id
            self.set_link(link_id, y=self.get_y(), page=self.page_no())
        except Exception:
            pass

    # ------------------------------ Sections ----------------------------------
    def add_overview_section(self, overview: Dict[str, object]):
        """Overview section summarizing repo for humans & LLMs (text only)."""
        # Header state for this page
        self._hdr_path = "Overview"
        self._hdr_lang = ""
        self._hdr_lines = None

        self.add_page()
        title = "Overview"
        self.set_font("DejaVu", "B", 16)
        self._safe_multicell(title, line_h=10)
        self.ln(0.5)  # Reduced from 1
        self.toc_add(title, level=0)

        self.set_font("DejaVu", size=11)
        line_h = 5.5  # Reduced from 6

        def p(text: str = ""):
            self._safe_multicell(normalize_text_for_pdf(text), line_h=line_h)
            if text:
                self.ln(0.2)  # Add minimal spacing only for non-empty text

        def bullet(text: str):
            self._safe_multicell(f"• {normalize_text_for_pdf(text)}", line_h=line_h)

        title_text = overview.get("title") or ""
        subtitle_text = overview.get("subtitle") or ""
        desc = overview.get("description") or ""
        features: List[str] = overview.get("features") or []
        usage = overview.get("usage") or ""
        exts: List[Tuple[str, int]] = overview.get("ext_counts") or []
        total_files: int = overview.get("total_files") or 0
        deps: List[str] = overview.get("dependencies") or []
```

```
        if title_text:
            self.set_font("DejaVu", "B", 12)
            p(str(title_text))
            self.set_font("DejaVu", size=11)
        if subtitle_text:
            p(str(subtitle_text))
        if desc:
            p(str(desc))

        if features:
            self.ln(0.6)  # Reduced from 1
            self.set_font("DejaVu", "B", 12)
            p("Key Features")
            self.set_font("DejaVu", size=11)
            for f in features[:8]:
                bullet(str(f))

        if usage:
            self.ln(0.6)  # Reduced from 1
            self.set_font("DejaVu", "B", 12)
            p("Quick Usage")
            self.set_font("DejaVuMono", size=10)
            self._safe_multicell(str(usage), line_h=5)  # Reduced from 5.5
            self.set_font("DejaVu", size=11)

        if exts:
            self.ln(0.6)  # Reduced from 1
            self.set_font("DejaVu", "B", 12)
            p("Files & Languages")
            self.set_font("DejaVu", size=11)
            for ext, cnt in exts[:8]:
                bullet(f"{ext} - {cnt} file(s)")
            bullet(f"Total files: {total_files}")

        if deps:
            self.ln(0.6)  # Reduced from 1
            self.set_font("DejaVu", "B", 12)
            p("Dependencies")
            self.set_font("DejaVu", size=11)
            for d in deps[:12]:
```

```
                bullet(d)

    # ---- Code rendering with native syntax highlighting, background, line numbers
    def _ensure_lexer(self, rel_path: str, content: str):
        try:
            return get_lexer_for_filename(rel_path, stripnl=False)
        except Exception:
            try:
                return guess_lexer(content)
            except Exception:
                return TextLexer()

    def _write_code_with_highlighting(
        self,
        rel_path: str,
        content: str,
        *,
        line_numbers: bool = True,
        font_size: int = 9,
    ):
        """
        Write code using token-by-token coloring. Avoids drawing an empty band:
        we only draw the background after we know we'll print text on the line.
        """
        content = content.replace("\t", "    ")  # Normalize tabs
        lexer = self._ensure_lexer(rel_path, content)

        self.set_font("DejaVuMono", size=font_size)
        # Reduced line height for tighter spacing
        line_h = max(4.0, font_size * 0.38 + 3.2)

        # Layout geometry
        left_x = self.l_margin
        right_x = self.w - self.r_margin
        bottom_limit = self.h - self.b_margin
        lines_total = (content.count("\n") + 1) if content else 1

        gutter_w = (self.get_string_width(str(lines_total)) + 4) if line_numbers else 0.0
        code_x = left_x + gutter_w
```

```
359         # State for current visual line
360         cur_line_no = 1
361         at_line_start = True  # start of a visual line (no text yet)
362         drew_band_this_line = False  # background band drawn?
363         wrote_line_number = False  # line number drawn?

365         def start_new_visual_line(new_logical: bool = False):
366             nonlocal at_line_start, drew_band_this_line, wrote_line_number, cur_line_no
367             # Move down a line; auto page break is on
368             self.ln(line_h)
369             at_line_start = True
370             drew_band_this_line = False
371             wrote_line_number = False
372             # If this is because we finished a logical line, increment number now
373             if new_logical:
374                 cur_line_no += 1

376         def ensure_band_and_gutter():
377             """Draw background + gutter only once, right before first text on the visual line.

379             IMPORTANT: Guard against page bottom *before* drawing anything to avoid blank page
379 s.
380             """
381             nonlocal drew_band_this_line, wrote_line_number, at_line_start
382             if drew_band_this_line:
383                 return
384             y = self.get_y()
385             # If not enough space for this line, force a page break first
386             if y + line_h > bottom_limit:
387                 # Explicitly add a page so the band/text draw on the *new* page
388                 self.add_page()
389                 # Reset per-line state at new page top
390                 at_line_start = True
391                 drew_band_this_line = False
392                 wrote_line_number = False
393                 y = self.get_y()

395             # Draw band
396             self.set_fill_color(248, 248, 248)
397             self.rect(left_x, y, right_x - left_x, line_h, style="F")
```

```
398             # Gutter
399             if line_numbers and not wrote_line_number:
400                 self.set_text_color(150, 150, 150)
401                 self.set_xy(left_x, y)
402                 self.cell(gutter_w, line_h, str(cur_line_no).rjust(len(str(lines_total))), ali
402 gn="R")
403                 wrote_line_number = True

405             # Move to code start
406             self.set_xy(code_x, y)
407             drew_band_this_line = True

409         # Begin at current Y; do not pre-draw anything
410         if at_line_start:
411             # just position cursor at code area before first text
412             self.set_x(code_x)

414         # Render each logical line with wrapping
415         for logical_line in (content.splitlines(True) or [""]):
416             pieces = list(lex(logical_line, lexer))

418             for tok_type, txt in pieces:
419                 # Split into printable and whitespace chunks to allow wrapping at spaces
420                 for chunk in re.split(r"(\s+)", txt):
421                     if chunk == "":
422                         continue
423                     if chunk == "\n":
424                         # finish logical line: advance to next visual line as a new logical li
424 ne
425                         start_new_visual_line(new_logical=True)
426                         continue

428                     # We are about to print something: ensure band & gutter once
429                     ensure_band_and_gutter()
430                     at_line_start = False

432                     # Soft wrap if needed
433                     piece = chunk
434                     while piece:
435                         available = right_x - self.get_x()
```

```
436                         piece_w = self.get_string_width(piece)

438                         if piece_w <= available:
439                             r, g, b = _rgb_for(tok_type)
440                             self.set_text_color(r, g, b)
441                             self.cell(piece_w, line_h, piece, ln=0)
442                             piece = ""
443                         else:
444                             # Need to break piece - largest prefix that fits
445                             lo, hi = 0, len(piece)
446                             while lo < hi:
447                                 mid = (lo + hi + 1) // 2
448                                 if self.get_string_width(piece[:mid]) <= available:
449                                     lo = mid
450                                 else:
451                                     hi = mid - 1
452                             prefix = piece[:lo] if lo > 0 else ""
453                             rest = piece[lo:] if lo < len(piece) else ""
454                             if prefix:
455                                 r, g, b = _rgb_for(tok_type)
456                                 self.set_text_color(r, g, b)
457                                 self.cell(self.get_string_width(prefix), line_h, prefix, ln=0)
458                             # move to next visual line (continuation, same logical line number
458 )
459                             start_new_visual_line(new_logical=False)
460                             ensure_band_and_gutter()
461                             piece = rest

463             # NOTE: Removed the unconditional advance for non-terminated lines.
464             # Previously this could contribute to stray blank lines/pages at boundaries.

466         # Reset color
467         self.set_text_color(0, 0, 0)

469     def _detect_language_label(self, rel_path: str, content: str) -> str:
470         # Try pygments lexer name
471         try:
472             lexer = get_lexer_for_filename(rel_path, stripnl=False)
473             return getattr(lexer, "name", "Text")
474         except Exception:
```

```
            try:
                lexer = guess_lexer(content)
                return getattr(lexer, "name", "Text")
            except Exception:
                # Fall back to extension
                ext = os.path.splitext(rel_path)[1].lower() or "(no ext)"
                return {"": "Text"}.get(ext, ext or "Text")

    def _estimate_block_height(self, line_count: int, font_size: int = 9) -> float:
        """Rough height estimate for small-file packing (title + meta + lines)."""
        title_h = 8.0  # Reduced from 9.0
        meta_h = 5.0   # Reduced from 5.5
        line_h = max(4.0, font_size * 0.38 + 3.2)
        return title_h + 0.5 + meta_h + 0.5 + line_count * line_h + 1

    def _set_header_context(self, path: str, lang: str, lines: int):
        self._hdr_path = path
        self._hdr_lang = lang
        self._hdr_lines = lines

    def add_file_section(self, rel_path: str, content: str, *, force_new_page: bool = True):
        """Render a file. If force_new_page=False we try to keep adding on the same page."""
        # Body (code with native highlighting)
        content = normalize_text_for_pdf(content)
        # Safety: soft-wrap pathological long lines before rendering
        if content and len(max(content.splitlines() or [""], key=len)) > 2000:
            content = "\n".join(_soft_wrap(line, width=200) for line in content.splitlines())

        lang = self._detect_language_label(rel_path, content)
        line_count = content.count("\n") + (1 if content and not content.endswith("\n") else 0
)
        line_count = max(1, line_count)

        # Page decision for small files
        est_h = self._estimate_block_height(min(line_count, 40))
        bottom_limit = self.h - self.b_margin
        need_new_page = force_new_page or (self.get_y() + est_h > bottom_limit)

        if need_new_page:
            # Update header state for this page
```

```
514             self._set_header_context(rel_path, lang, line_count)
515             self.add_page()
516         else:
517             # Update header context to reflect the first file on this page
518             if self.page_no() == 0:
519                 self.add_page()
520             if self._hdr_path == self.meta.title:
521                 self._set_header_context(rel_path, lang, line_count)

523         # File title
524         self.set_font("DejaVu", "B", 14)
525         self._safe_multicell(normalize_text_for_pdf(rel_path), line_h=8)  # Reduced from 9

527         # File meta line: language + line count
528         self.set_font("DejaVu", size=9)
529         self.set_text_color(110)
530         meta_line = f"{lang} • {line_count} line(s)"
531         self._safe_multicell(meta_line, line_h=5)  # Reduced from 5.5
532         self.set_text_color(0)
533         self.ln(0.4)  # Reduced from 1

535         # ToC + link
536         self.toc_add(rel_path, level=0)

538         # Code
539         self._write_code_with_highlighting(rel_path, content, line_numbers=True, font_size=9)

541     # ------------------------------ Appendix ---------------------------------
542     def add_appendix(self, summary: Optional[Dict[str, Any]]):
543         if not summary:
544             return

546         self._hdr_path = "Appendix"
547         self._hdr_lang = ""
548         self._hdr_lines = None

550         self.add_page()
551         self.set_font("DejaVu", "B", 16)
552         self._safe_multicell("Appendix: Skipped & condensed", line_h=10)
553         self.ln(1)  # Reduced from 2
```

```
        self.set_font("DejaVu", size=11)

        def row(label: str, value: Any):
            self.set_font("DejaVu", "B", 11)
            self._safe_multicell(label, line_h=5.5)  # Reduced from 6
            self.set_font("DejaVu", size=11)
            self._safe_multicell(str(value), line_h=5.5)  # Reduced from 6
            self.ln(0.3)  # Reduced from 1

        counts = summary.get("counts", {})
        notes = summary.get("notes", [])
        packed = summary.get("packed_small_files", 0)

        row("Skipped (gitignored)", counts.get("gitignored", 0))
        row("Skipped (excluded dirs)", counts.get("excluded_dir", 0))
        row("Skipped (manual excludes)", counts.get("manual_exclude", 0))
        row("Skipped (binary by extension)", counts.get("binary_ext", 0))
        row("Skipped (binary by magic/heuristic)", counts.get("binary_magic", 0))
        row("Skipped (too large)", counts.get("too_large", 0))
        row("Read/decoding errors", counts.get("read_errors", 0))
        row("Packed small files (co-located per page)", packed)

        if notes:
            self.ln(1)  # Reduced from 2
            self.set_font("DejaVu", "B", 12)
            self._safe_multicell("Notes", line_h=6)  # Reduced from 7
            self.set_font("DejaVu", size=11)
            for n in notes:
                self._safe_multicell(f"• {n}", line_h=5.5)  # Reduced from 6

# ---------------------------------------------------------------------------
# Public API
# ---------------------------------------------------------------------------

def generate_pdf(
    files: Iterable[Tuple[str, str]],
    output_path: str,
    meta: Optional[PDFMeta] = None,
    summary: Optional[Dict[str, Any]] = None,
) -> str:
```

```
    """
    Generate a polished PDF from an iterable of (relative_path, content).

    Adds:
    - Cover
    - Table of Contents (at the start; one page, truncated if needed)
    - Text Overview section (LLM + human friendly)
    - File sections (syntax-highlighted, small-file packing)
    - Appendix with skip/condense summary
    """
    meta = meta or PDFMeta(title="Repository Export", generated_at=datetime.utcnow())
    files = list(files)  # iterate twice safely
    pdf = RepoPDF(meta)

    # 1) Cover
    pdf.add_cover()

    # 2) Reserve a page for the ToC (at the start). We fill it later.
    pdf.reserve_toc_page()

    # 3) Overview
    overview = _build_overview_data(files, meta)
    pdf.add_overview_section(overview)

    # 4) Sections with small-file packing
    SMALL_LINE_THRESHOLD = 40  # Increased from 30 to pack more files together
    current_page_small_lines = 0
    for rel_path, content in files:
        # Safety for pathological lines (still soft wrap later)
        if content and len(max(content.splitlines() or [""], key=len)) > 4000:
            content = "\n".join(_soft_wrap(line, width=200) for line in content.splitlines())

        line_count = content.count("\n") + (1 if content and not content.endswith("\n") else 0
)
        line_count = max(1, line_count)

        if line_count <= SMALL_LINE_THRESHOLD:
            # Try to keep adding on same page until space runs out
            pdf.add_file_section(rel_path, content, force_new_page=False)
            current_page_small_lines += line_count
```

```
633         else:
634             # Large file: force a new page
635             current_page_small_lines = 0
636             pdf.add_file_section(rel_path, content, force_new_page=True)

638     # 5) Go back and render ToC on the reserved page (truncate if too long)
639     pdf.render_toc_on_reserved_page()

641     # 6) Appendix
642     pdf.add_appendix(summary)

644     # 7) Save
645     os.makedirs(os.path.dirname(output_path) or ".", exist_ok=True)
646     pdf.output(output_path)
647     return output_path

649 # -----------------------------------------------------------------------------
650 # Helpers
651 # -----------------------------------------------------------------------------

653 def _soft_wrap(line: str, width: int) -> str:
654     if len(line) <= width:
655         return line
656     return "\n".join(line[i:i+width] for i in range(0, len(line), width))

658 def _strip_readme_images(text: str) -> str:
659     # Remove markdown image syntax ![alt](url) and <img ...> HTML tags
660     text = re.sub(r"!\[[^\]]*\]\([^)]+\)", "", text)
661     text = re.sub(r"<img\s+[^>]*>", "", text, flags=re.IGNORECASE)
662     return text

664 def _build_overview_data(files: List[Tuple[str, str]], meta: PDFMeta) -> Dict[str, object]:
665     """
666     Build a compact, LLM-friendly + human-friendly overview using repo content:
667     - Name, purpose (from README if present)
668     - Headline features (from README bullets)
669     - Usage (from README or CLI hints)
670     - Language & file stats
671     - Dependencies (requirements.txt, pyproject)
672     """
```

```
673     file_map: Dict[str, str] = {p.lower(): c for p, c in files}

675     # README
676     readme_name = next((p for p, _ in files if os.path.basename(p).lower() in {"readme.md", "r
676 eadme"}), None)
677     readme = file_map.get(readme_name.lower(), "") if readme_name else ""
678     readme = _strip_readme_images(readme)

680     title = meta.title or "Repository"
681     subtitle = meta.subtitle or ""

683     # Description: first paragraph of README (strip headings)
684     desc = ""
685     if readme:
686         text = re.sub(r"^#{1,6}\s+.*$", "", readme, flags=re.MULTILINE).strip()
687         parts = [p.strip() for p in text.split("\n\n") if p.strip()]
688         if parts:
689             desc = parts[0][:800]

691     # Features: README bullet list (first 5-8)
692     features: List[str] = []
693     if readme:
694         for line in readme.splitlines():
695             if re.match(r"^\s*[-*]\s+", line):
696                 features.append(re.sub(r"^\s*[-*]\s+", "", line).strip())
697             if len(features) >= 8:
698                 break

700     # Usage: a code snippet containing 'repo2pdf'
701     usage = ""
702     if readme:
703         m = re.search(r"```(?:bash|sh)?\s*([^`]*repo2pdf[^\n`]*\n(?:.*?\n)*)```", readme, flag
703 s=re.IGNORECASE)
704         if m:
705             usage = m.group(1).strip()
706         if not usage:
707             usage = "repo2pdf # Follow interactive prompts"

709     # Language & file stats
710     from collections import Counter
```

```
    ext_counts = Counter()
    for p, _ in files:
        ext = os.path.splitext(p)[1].lower() or "(no ext)"
        ext_counts[ext] += 1
    top_exts = sorted(ext_counts.items(), key=lambda kv: kv[1], reverse=True)[:8]
    file_count = sum(ext_counts.values())

    # Dependencies
    deps: List[str] = []
    req = file_map.get("requirements.txt", "")
    if req:
        for line in req.splitlines():
            line = line.strip()
            if line and not line.startswith("#"):
                deps.append(line)
    pyproject = file_map.get("pyproject.toml", "")
    if pyproject and not deps:
        for name in ("fpdf2", "GitPython", "inquirer", "pathspec", "pygments", "pytest"):
            if name in pyproject and name not in deps:
                deps.append(name)

    return {
        "title": title,
        "subtitle": subtitle,
        "description": desc,
        "features": features,
        "usage": usage,
        "ext_counts": top_exts,
        "total_files": file_count,
        "dependencies": deps,
    }

# --- token color theme -------------------------------------------------------

# Simple light theme for tokens (tweak as you like)
THEME = {
    Token.Comment: (120, 120, 120),
    Token.Keyword: (170, 55, 140),
    Token.Keyword.Namespace: (170, 55, 140),
    Token.Name.Function: (30, 120, 180),
```

```
    Token.Name.Class: (30, 120, 180),
    Token.Name.Decorator: (135, 110, 180),
    Token.String: (25, 140, 65),
    Token.Number: (190, 110, 30),
    Token.Operator: (90, 90, 90),
    Token.Punctuation: (90, 90, 90),
    Token.Name.Builtin: (30, 120, 180),
    Token.Name.Variable: (0, 0, 0),
    Token.Text: (0, 0, 0),
}
```

```
def _rgb_for(tok_type):
    # Find first mapping that contains this token type, else default black
    for t, rgb in THEME.items():
        if tok_type in t:
            return rgb
    return (0, 0, 0)
```

repo2pdf/utils.py

Python • 83 line(s)

```python
import os
import mimetypes
import json

EXTENSION_LANGUAGE_MAP = {
    # Programming languages
    '.py': 'Python',
    '.js': 'JavaScript',
    '.ts': 'TypeScript',
    '.java': 'Java',
    '.c': 'C',
    '.cpp': 'C++',
    '.cs': 'C#',
    '.rb': 'Ruby',
    '.go': 'Go',
    '.rs': 'Rust',
    '.php': 'PHP',
    '.swift': 'Swift',
    '.kt': 'Kotlin',
    '.m': 'Objective-C',
    '.scala': 'Scala',
    '.sh': 'Shell Script',
    '.bat': 'Batch Script',
    '.ps1': 'PowerShell',
    '.pl': 'Perl',
    '.r': 'R',

    # Web & markup
    '.html': 'HTML',
    '.htm': 'HTML',
    '.css': 'CSS',
    '.scss': 'SCSS',
    '.sass': 'SASS',
    '.less': 'LESS',
    '.json': 'JSON',
    '.xml': 'XML',
    '.yml': 'YAML',
    '.yaml': 'YAML',
```

```
    '.md': 'Markdown',

    # Config & data
    '.env': 'Environment Config',
    '.ini': 'INI Config',
    '.conf': 'Config',
    '.cfg': 'Config',
    '.toml': 'TOML Config',
    '.gradle': 'Gradle Build File',
    '.dockerfile': 'Dockerfile',

    # Text & miscellaneous
    '.txt': 'Plain Text',
    '.log': 'Log File',
    '.csv': 'CSV',
    '.tsv': 'TSV',
}


def output_json(files, output_path):
    data = []
    for filename, content in files:
        ext = os.path.splitext(filename)[1]
        language = EXTENSION_LANGUAGE_MAP.get(ext)

        if not language:
            # Fall back to mimetypes
            mime_type, _ = mimetypes.guess_type(filename)
            if mime_type:
                # Use the subtype (e.g. 'plain' from 'text/plain') or mime_type as fallback
                language = mime_type.split('/')[1] if '/' in mime_type else mime_type
            else:
                language = 'Unknown'

        data.append({
            "path": filename,
            "language": language,
            "content": content
        })
```

```
    json_path = output_path.replace(".pdf", ".json")
    with open(json_path, 'w') as f:
        json.dump({"files": data}, f, indent=2)

    print(f"  JSON saved to {json_path}")
```

requirements.txt

Text only • 6 line(s)

```
fpdf2
GitPython
inquirer
pathspec
pytest
pygments>=2.13
```

setup.py

Python • 17 line(s)

```
from setuptools import setup, find_packages

setup(
    name='repo2pdf',
    version='0.1.0',
    packages=find_packages(),
    install_requires=[
        'fpdf2',
        'GitPython',
        'inquirer'
    ],
    entry_points={
        'console_scripts': [
            'repo2pdf=repo2pdf.cli:main',
        ],
    },
)
```

tests/__init__.py

Python • 1 line(s)

tests/test_core.py

Python • 86 line(s)

```python
import os
import tempfile
from repo2pdf.core import traverse_repo
import os
import tempfile
from repo2pdf.core import process_local_repo

def test_traverse_repo_reads_files():
    with tempfile.TemporaryDirectory() as tmpdir:
        # Create a dummy file
        file_path = os.path.join(tmpdir, "test.py")
        with open(file_path, "w") as f:
            f.write("print('hello')")

        files = traverse_repo(tmpdir)

        assert len(files) == 1
        assert files[0][0] == "test.py"
        assert "print('hello')" in files[0][1]

def test_traverse_repo_excludes_specified_files():
    with tempfile.TemporaryDirectory() as tmpdir:
        # Create two files: one .py and one .png
        py_path = os.path.join(tmpdir, "test.py")
        png_path = os.path.join(tmpdir, "image.png")

        with open(py_path, "w") as f:
            f.write("print('hello')")

        with open(png_path, "w") as f:
            f.write("binarydata")

        from repo2pdf.core import traverse_repo
        files = traverse_repo(tmpdir)

        # Default traverse_repo (no exclude param) should return both files
        assert any(f[0] == "test.py" for f in files)
```

```
        # Now test excluding .png
        files_exclude = traverse_repo(tmpdir, exclude_list=[".png"])
        assert any(f[0] == "test.py" for f in files_exclude)
        assert not any(f[0] == "image.png" for f in files_exclude)

def test_process_remote_repo_clones_and_generates(monkeypatch):
    from repo2pdf.core import process_remote_repo
    import tempfile
    import os

    # Use a very small public GitHub repo for testing
    test_repo_url = "https://github.com/octocat/Hello-World.git"

    with tempfile.TemporaryDirectory() as tmpdir:
        output_path = os.path.join(tmpdir, "output.pdf")

        # Monkeypatch os.getcwd to tmpdir so output is saved there
        monkeypatch.setattr(os, "getcwd", lambda: tmpdir)

        # Run process_remote_repo with delete=True to clean up after test
        process_remote_repo(test_repo_url, want_json=True, output_path=output_path, exclude_lis
t=[], delete=True)

        assert os.path.exists(output_path)
        assert os.path.getsize(output_path) > 0

        json_path = output_path.replace(".pdf", ".json")
        assert os.path.exists(json_path)

def test_process_local_repo_creates_outputs(monkeypatch):
    with tempfile.TemporaryDirectory() as tmpdir:
        # Create a dummy local repo file
        file_path = os.path.join(tmpdir, "test.py")
        with open(file_path, "w") as f:
            f.write("print('hello')")

        output_path = os.path.join(tmpdir, "repo_output.pdf")

        # Monkeypatch os.getcwd to tmpdir so outputs are saved there
        monkeypatch.setattr(os, "getcwd", lambda: tmpdir)
```

```
        # Run process_local_repo with JSON generation
        process_local_repo(tmpdir, want_json=True)

        assert os.path.exists(output_path)
        assert os.path.getsize(output_path) > 0

        json_path = output_path.replace(".pdf", ".json")
        assert os.path.exists(json_path)
```

tests/test_pdf.py

Python • 13 line(s)

```
import os
import tempfile
from repo2pdf.pdf import generate_pdf

def test_generate_pdf_creates_file():
    with tempfile.TemporaryDirectory() as tmpdir:
        output_path = os.path.join(tmpdir, "output.pdf")
        files = [("test.py", "print('hello')")]

        generate_pdf(files, output_path)

        assert os.path.exists(output_path)
        assert os.path.getsize(output_path) > 0
```

tests/test_utils.py

Python • 20 line(s)

```python
import os
import tempfile
import json
from repo2pdf.utils import output_json

def test_output_json_creates_valid_file():
    with tempfile.TemporaryDirectory() as tmpdir:
        output_path = os.path.join(tmpdir, "output.pdf")
        files = [("test.py", "print('hello')")]

        output_json(files, output_path)

        json_path = output_path.replace(".pdf", ".json")
        assert os.path.exists(json_path)

        with open(json_path) as f:
            data = json.load(f)
            assert "files" in data
            assert data["files"][0]["path"] == "test.py"
            assert "print('hello')" in data["files"][0]["content"]
```

Appendix: Skipped & condensed

Skipped (gitignored)
162
Skipped (excluded dirs)
112
Skipped (manual excludes)
0
Skipped (binary by extension)
6
Skipped (binary by magic/heuristic)
2
Skipped (too large)
0
Read/decoding errors
0
Packed small files (co-located per page)
0