ChatGPT

1. Requirements Document: Python Project Guidelines (v1.0)

This document outlines the unified guidelines and best practices for all Python projects in our ecosystem. Adhering to these standards will ensure consistency, maintainability, and ease of integration across projects.

Project Management and Structure

- **Unified Project Tool (`uv`)**: Use the **uv** tool for project management. This ensures consistent creation of project structures, dependency management, virtual environments, and easy cross-project referencing. (If using uv, initialize and manage projects with `uv` commands for consistency across all teams.)
- **Directory Layout**: Follow a consistent directory structure for every project:
- A top-level project folder containing `README.md`, `ai_requirement.md` (this guidelines document), configuration files (like `pyproject.toml` or `setup.cfg`), etc.
- A `src/` directory housing the Python package (named after the project). Inside, organize core logic under sub-packages (e.g., `core/`) as needed.
- Peer directories such as `data/` (for sample or default data files), `tests/` (for unit and integration tests), `scripts/` (for utility scripts or CLI entry points), and `docs/` (for additional documentation).
- **Cross-Platform Design**: Ensure all code is designed with **cross-platform** compatibility. In particular, treat **Windows compatibility** as a first-class concern – avoid assumptions that only work on Unix-like systems. For example, file path handling and newlines should be portable.
- **Versioning**: Adopt semantic versioning, starting every new library at **version 1.0.0**. Maintain version info in a single source (e.g., in `pyproject.toml` and/or an `__version__` variable in the package).
- **Build and Deployment**: Each project should be configured for easy installation and distribution (using `pyproject.toml` with PEP 621 or similar). Leverage uv to build and publish packages when ready.
- **Initial Testing**: Include a `tests/` directory for test code. (While initial prototypes can skip strict test coverage, plan to add unit tests for all critical functionality as the project matures.)

Coding Style and Best Practices

- **Python Version**: Use **Python 3.12+** for all projects. This ensures access to the latest language features and optimizations. All code should be written to be compatible with Python 3.12 or above.
- **Naming Conventions**: Follow PEP 8 style for naming. In particular, use **snake_case** for functions, variables, and module names. (Classes may use CamelCase as per PEP 8, but strive for simplicity – if a class isn’t necessary, use functions.) Favor clarity and descriptiveness in names; avoid abbreviations that aren’t obvious.
- **Minimalistic Design (Unix Philosophy)**: Embrace a minimalistic, **Unix-like design** philosophy. Write simple, modular functions that “do one thing well.” Keep functions and interfaces small and composable, similar to small Linux utilities. Avoid unnecessary complexity – prefer straightforward solutions and lean, efficient code.
- **Code Conciseness**: Use Python’s syntactic sugar and modern features to simplify code when it improves readability. For example, use context managers (`with` statements), comprehensions,

and relevant stdlib utilities to make code more concise and clear. However, do not sacrifice clarity for cleverness.

- **Type Hinting**: All functions must include **type hints** for parameters and return types, using Python 3.12's improved typing syntax where appropriate. Leverage new typing features (like the `type` statement for generic aliases, `Self` type, etc.) for cleaner annotations. This makes code self-documenting and helps with static analysis.
- **Imports**: Use absolute imports for project-internal modules. Avoid relative import paths (e.g., no `from .. import ...`). Instead, import using the top-level package name to prevent confusion and ensure clarity in large projects.
- **Formatting**: Enforce a consistent code style using **Black** (or an equivalent code formatter configured to Black's style). This guarantees uniform formatting (4-space indents, line lengths, etc.) across all projects, reducing diffs and making code reviews easier.
- **Comments and Docstrings**: Write clear, **plain-text docstrings** for all public functions, classes, and modules. The docstrings should describe the purpose, parameters, return values, and exceptions (if any) in simple language without special markup. If a function's behavior is not obvious, include a short usage example in the docstring to illustrate how to call it. (Use a simple text example – no complex formatting – so that anyone reading the source can easily understand the usage.)
- **Example**: For a complex function, a docstring might include:

  ```
  Example:
      result = my_function(foo, bar)
      # where foo is ..., bar is ...
  ```

  This helps users quickly grasp the intended usage.
- **Internal API vs External API**: Within a project, separate internal utility code and the public API. Provide a clear **API layer** for external use – for instance, via an `__init__.py` that exposes the most important classes/functions, or a dedicated `api.py` module. This way, users of the library can import from a single high-level module without needing to know internal module structure. Design this external API for efficiency and ease-of-use, minimizing the overhead or complexity for the caller.

Resource Management and Reliability

- **Context Managers for Resources**: Use `with` statements (context managers) to manage resources such as files, network connections, or locks. Any operation that acquires a resource should ensure that resource is released promptly after use. For example, if a function opens a file, use a `with open(...) as f:` block so the file closes automatically. If your code acquires any system resource (threads, subprocesses, etc.), ensure there's a clean teardown (possibly in a context manager's `__exit__`).
- **Path Handling**: Represent file system paths with `pathlib.Path` objects rather than raw strings. Create utility functions (e.g., in our common utils library) for path operations like joining paths, creating directories, etc., to encourage consistent usage. This uniform approach avoids common bugs with string paths and makes code more readable.
- **Atomic File Operations**: For operations that write or modify files in multiple steps, adopt an "all-or-nothing" strategy. For example, when writing output to a file, first write to a temporary file, then **atomically replace** the target file (e.g., using `Path.replace()` which uses an atomic rename under the hood). This ensures that if a process is interrupted, partial or corrupt files are not left in place. Provide helper functions if necessary to facilitate this pattern.

- **Asynchronous I/O with Timeouts**: When performing I/O-bound work (file reads/writes, network requests, etc.), consider using `asyncio` for concurrency, especially if the project might benefit from parallel operations. Always set **timeouts** for I/O operations to avoid hanging indefinitely. For example, if using `asyncio`, use `asyncio.wait_for` or timeout parameters in libraries (where available) to bound how long operations can block.
- **Signal Handling (Ctrl+C)**: Long-running or blocking operations should respond promptly to user interrupts (Ctrl+C). Utilize a centralized **signal handling context** (provided by the common utils library) to handle `SIGINT`. Wrapping critical sections in a `with SignalContext():` (or equivalent) will ensure that a Ctrl+C (SIGINT) triggers a graceful interruption. The signal context should be designed to support **nested usage** – i.e., if multiple nested contexts are opened, they don’t interfere with each other and all respect the interrupt. Always ensure that upon receiving such a signal, the program can clean up resources and exit cleanly rather than hanging.
- **Retry Logic**: If implementing retry mechanisms for operations (e.g., retrying a network call), separate the retry logic from the resource handling. Implement a retry decorator or utility that calls a function multiple times on exception. **Within each attempt**, manage resources properly (e.g., open files inside the attempt, not outside, so that each try uses a fresh resource and closes it at the end of the attempt). This prevents resource leakage across retries and keeps concerns separated (retry vs. actual operation).
- **Temporary Changes (Working Dir, Env)**: If code needs to temporarily change the process state (such as current working directory or environment variables), provide context manager utilities to encapsulate these changes. For example, a context manager that changes `os.chdir` to a given path on enter and restores it on exit, or one that sets environment variables and then reverts them. **Never scatter `os.chdir()` or `os.environ` modifications throughout the code** – always use a controlled context so that the original state is restored and side effects don’t leak out. The common utils library (pyu) should offer such context managers (e.g., `with pyu.system.working_directory(path): ...`) to encourage this practice.
- **Resource Handle Exposure**: When writing functions that manage resources (database connections, file handles, threads, etc.), consider providing an option for advanced users to access or control the underlying resource. For instance, a function could accept a parameter like `return_handle=True` which, if set, returns the raw resource (and leaves it open for the caller to manage). By default, the function would abstract this away (managing the resource internally), but this option gives flexibility to those who need it. Any such design must be documented clearly, and when the user takes control of the resource, they are responsible for closing or cleaning it up. This guideline helps balance ease-of-use for most users with flexibility for power users.
- **Error Handling and Cleanup**: Always write error handling with cleanup in mind. Use `try/except/finally` or context managers to ensure that regardless of errors or interrupts, the system can release resources (files, locks, etc.), save necessary state, and either retry or fail gracefully. Logging exceptions with context (see Logging section) is encouraged for easier debugging.

Logging and Monitoring

- **Unified Logging**: All projects should use the **common logging utility (pyu.logger)** to ensure consistent logging behavior and format. Rather than creating separate log configurations in each project, import and use the shared logger. This means every function or module can log messages (info, warnings, errors) in a uniform way, making it easier to trace execution across projects.
- **Logger Configuration**: The default logging configuration should be minimal and focused. By default, log messages should include at least the **severity level** and the **function name** (or context) producing the log, along with the message. For example:

```
[INFO] my_function: Task completed successfully.
```

This concise format keeps logs readable. Additional details (timestamps, module names, etc.) can be configured by the user as needed. The logging utility should provide a simple way for users to change the log format (for instance, a function to set a new format string, perhaps to include timestamps or thread info if desired). By keeping the default simple and offering extensibility, we cater to both straightforward use and advanced customization.

- **Avoiding Log Handler Leakage**: Configure log handlers (such as console or file handlers) in a centralized way. For example, a function `pyu.logger.configure()` can set up handlers (stream to console, file rotation, etc.) as needed. Ensure that file handlers or other resources attached to the logger are closed properly on program exit or when reconfiguring, to avoid issues like file descriptor leaks. It’ s good practice to use context managers or application shutdown hooks to close log files. If multiple projects or libraries use the same logging system, make sure they don’ t inadvertently add multiple handlers producing duplicate logs – typically, configure each handler once (check if a handler is already present before adding).
- **Progress Indicators**: For long-running operations, provide visual feedback to the user. The projects should incorporate a **progress bar utility** (for example, leveraging `tqdm` or a similar library) for loops or tasks that take noticeable time. The common utils (pyu.progress) can wrap an iterator or provide a context to display progress percentage, elapsed time, etc. This helps users know that the program is active and not frozen. Choose a well-supported progress bar implementation (like `tqdm`) to ensure reliability and nice features (e.g., nested progress bars for nested loops, timing, etc.). Keep the usage simple – e.g., `for item in pyu.progress(my_iterable): ...` or `with pyu.progress(total=n) as p: ...` for manual updates – so that adding a progress bar requires minimal changes to existing code.
- **Monitoring and Alerts**: (For future consideration) Projects should be designed so that they can be monitored in production. This could mean structured logging (machine-parseable logs), metrics collection hooks, or other instrumentation. While not an immediate requirement, keep this in mind when designing the logging and progress features – they should be possible to integrate into a broader monitoring system if needed.

Documentation and Accessibility

- **README**: Every project **must include a** `README.md` at the root. The README should at minimum describe the project’ s purpose, how to set up the development environment, and basic usage examples. If the project produces an installable package, the README should cover installation steps (e.g., via uv, pip, etc.) and include a quickstart example. This file is the first entry-point for new users/developers, so keep it informative and concise.
- **Additional Docs**: For more complex projects, maintain a `docs/` folder with further documentation (which can be in Markdown or generated via tools like Sphinx if needed). However, for small libraries, a thorough README might suffice.
- **Markdown for Docs**: Use **Markdown** for all documentation to ensure it’ s easy to read both as plain text and rendered (e.g., on GitHub). This includes README, this requirements document (`ai_requirement.md`), and any other design docs. Consistent formatting and styles (headings, lists, code fences for examples) should be used as per this document’ s standards.
- **Documentation of Features**: Clearly document any tricky or non-obvious features of the code. For example, if a utility supports **nested signal handling contexts**, explain in the docs how a user should use this (perhaps with a short code snippet demonstrating nesting two `with` contexts). If the logger format can be changed, show an example of customizing it. Users should be able to quickly find how to use a feature by scanning the documentation.

- **Keep It Simple**: In line with the minimalistic philosophy, documentation should also be straightforward. Explain things in simple terms and assume the reader is familiar with Python but not with the internal design decisions. Avoid overly formal or technical jargon where a simple explanation would do.

By following these guidelines, all our Python projects will share a common structure and behavior. This makes it easier to maintain multiple projects simultaneously, enables code reuse, and provides a consistent experience for developers and users interacting with our codebase.

---

2. General Project Template

Below is a general project template structure incorporating the above requirements. This template can be used as a starting point for new Python projects. It includes the standard directories and files, and placeholders for core modules (like logger, progress, etc.) that can be filled in or utilized as needed.

```
<project_name>/
├── README.md                        <- Project overview, setup, and usage
instructions
├── ai_requirement.md                <- Requirements/guide document (like
the one above)
├── pyproject.toml                   <- Project metadata and dependencies
(PEP 621 / Poetry format)
├── src/
│   └── <project_name>/             <- Python package for the project
│        ├── __init__.py            <- Initializes the package (could expose
API here)
│        ├── core/                  <- Core functionality modules
│        │    ├── __init__.py
│        │    ├── logger.py         <- (Logging utilities, or use common
pyu.logger)
│        │    ├── progress.py       <- (Progress bar utilities)
│        │    ├── signals.py        <- (Signal handling utilities)
│        │    └── system.py         <- (System and OS utilities)
├── tests/                          <- Tests for the project (can be
structured similarly to src/)
│   └── __init__.py
├── scripts/                        <- Scripts or entry points (if any,
e.g., CLI scripts)
├── data/                           <- Data files or assets (if
applicable)
└── docs/                           <- Additional documentation (if
needed)
```

Notes:
- In the template, the `core` subpackage is shown containing modules for logger, progress, signals, and system utilities. In practice, if this project is the common utils library (pyu), those modules will be implemented. For other projects, you may not need to implement new logger/progress functionalities;

instead, you would import and use them from the common library. The template includes them for completeness – they can be left empty or serve as integration points depending on the project’ s nature.
- The `ai_requirement.md` should contain the guidelines from section 1 (possibly adapted to project-specific needs). This helps anyone working on the project to understand the agreed standards.
- `pyproject.toml` should declare the project name, version (start at 1.0.0), Python requirement (>=3.12), and dependencies (for example, the common utils library `pyu` itself when other projects use it, or `tqdm` if the project uses progress bars, etc.). If using **uv**, much of this is handled via uv’ s commands, but the toml file remains as the source of truth for project metadata.
- The `README.md` in the template should be edited to include project-specific information. At minimum, it should have instructions to set up the environment (perhaps using uv: e.g., “`uv install` to install dependencies in a virtual environment”), and examples of basic usage of the project.
- The `tests/` directory can house test files mirroring the structure of `src/`. Initially, you might skip writing tests (during rapid prototyping), but the structure is in place to add them later.
- All Python code files in `src/<project_name>/` and subdirectories should adhere to the guidelines (proper naming, type hints, docstrings, etc.). The template’ s placeholder modules (logger.py, etc.) can be implemented as needed. In many cases, other projects will rely on the common `pyu` library for these features, so they might not implement those modules themselves. However, having the files present (even if just with a basic pass or import statements) can serve as a reminder of available functionality.

This template is meant to be a living scaffold – feel free to add more structure (for example, submodules for different components of the project) as long as the overall organization remains clear and consistent.

3. Example Implementation: pyu Utility Library (v1.0.0)

To illustrate the above template and guidelines in action, here is the first version of **pyu**, a Python utilities library that provides common functionality for all our projects. This project is created following the template structure and implements the core features (system operations, logging, progress bar, signal handling) discussed earlier.

Project Structure

```
pyu/
├── README.md
├── ai_requirement.md              <- Contains the unified guidelines (from
section 1 above)
├── pyproject.toml
├── src/
│   └── pyu/
│        ├── __init__.py
│        └── core/
│            ├── __init__.py
│            ├── system.py
│            ├── logger.py
│            ├── progress.py
│            └── signals.py
└── tests/
    └── __init__.py
    └── (tests would go here, omitted for brevity)
```

pyproject.toml (excerpt): This defines the project metadata and dependencies. For example:

```
[project]
name = "pyu"
version = "1.0.0"
description = "Python Utilities Library for system ops, logging, progress bars, and signal handling"
requires-python = ">=3.12"
authors = [
  { name="Your Name", email="you@example.com" }
]
license = "MIT"
readme = "README.md"

# Specify the project's dependencies
dependencies = [
  "tqdm >=4.0.0"  # used for progress bar functionality
]
```

(Using uv: if you are using the uv tool to manage this project, uv will handle virtual environment setup and dependency installation as per this configuration.)

Source Code

Below, we provide the source code for the main components of the **pyu** library. All code is written to reflect the guidelines: clear naming, type hints, context managers for critical sections, etc., and includes docstrings with usage examples where appropriate.

File: `src/pyu/__init__.py`

```
"""pyu - Python Utilities Library

This package provides common utilities for system operations, logging, progress bars,
and signal handling, following the unified project guidelines.
"""

# Expose key functionality at the package level for convenience (API layer)
from pyu.core.system import ensure_dir, join_path, atomic_write, working_directory,
temporary_env
from pyu.core.logger import configure_logger, set_log_format, debug, info, warning, error,
critical
from pyu.core.progress import progress
from pyu.core.signals import SignalContext

# Package version
__version__ = "1.0.0"

# Initialize default logger configuration on import (with a simple format)
configure_logger()
```

Explanation: The `__init__.py` acts as an API layer, importing commonly used functions and classes from the `core` submodules so that users can import them directly from `pyu`. For example, `from pyu import info, ensure_dir, SignalContext`. The default logger is configured at package import with a basic format, so logging is ready to use immediately.

File: `src/pyu/core/system.py`

```
"""System and OS utilities (path operations, environment, etc.)"""

import os
from pathlib import Path
from contextlib import contextmanager
from typing import Any, Iterator, Union

def join_path(base: Union[str, Path], *others: Union[str, Path]) -> Path:
  """Join multiple path components intelligently into a single Path object.

  This function ensures that all parts are joined using pathlib, resulting in a Path.
  Example:
    path = join_path("/home/user", "projects", "myapp")
    # path is Path("/home/user/projects/myapp")
  """
    base_path = Path(base)
    # Join subsequent parts
    if others:
        return base_path.joinpath(*others)
    return base_path

def ensure_dir(path: Union[str, Path]) -> Path:
  """Ensure that the given directory exists. If it doesn't, create it (including parents).

  Returns a Path object for the directory. If the directory already exists, nothing happens.
  Example:
    data_dir = ensure_dir("output/data")
    # Now the "output/data" directory exists on disk and can be used.
  """
    p = Path(path)
    p.mkdir(parents=True, exist_ok=True)
    return p

def atomic_write(path: Union[str, Path], data: Union[str, bytes], mode: str = 'w', encoding: str
= 'utf-8') -> None:
  """Write data to a file atomically. The data is first written to a temp file, then renamed to target
path.

  This ensures that either the full data is written or, in case of error, the original file remains
unchanged.
  Usage:
    atomic_write("result.txt", "important results")
    # "result.txt" will either contain the full new content, or remain as it was (if an error occurred).
```

```python
    """
    target = Path(path)
    tmp_path = target.with_suffix(target.suffix + ".tmp")
    # Determine write mode based on data type (if not already specified appropriately)
    binary = isinstance(data, (bytes, bytearray))
    write_mode = mode
    if binary and 'b' not in write_mode:
        write_mode = write_mode.replace('w', 'wb')
    elif not binary and 'b' in write_mode:
        # Ensure text data is not written in binary mode accidentally
        write_mode = write_mode.replace('b', '')
    # Write to a temporary file first
    with open(tmp_path, write_mode, encoding=(None if binary else encoding)) as f:
        f.write(data)
        f.flush()
        os.fsync(f.fileno())   # ensure data is flushed to disk
    # Atomically replace the target file with the temp file
    tmp_path.replace(target)

@contextmanager
def working_directory(path: Union[str, Path]) -> Iterator[None]:
    """Context manager to temporarily change the working directory.

    Usage:
        with working_directory("/tmp"):
            # Inside this block, CWD is "/tmp".
            # After the block, CWD reverts to its original value.
    """
    new_dir = Path(path).resolve()
    prev_dir = Path.cwd()
    try:
        os.chdir(new_dir)
        yield
    finally:
        os.chdir(prev_dir)

@contextmanager
def temporary_env(new_env: dict[str, str]) -> Iterator[None]:
    """Context manager to temporarily set environment variables.

    Provide a dictionary of environment variable names and values. These will be set upon entering the context.
    The original environment values are restored upon exiting.
    Example:
        with temporary_env({"DEBUG": "1"}):
            # Inside this block, os.environ["DEBUG"] == "1"
        # Outside, the original os.environ["DEBUG"] value is restored (or removed if it wasn't set).
    """
    # Save old values
    old_env: dict[str, Any] = {}
    for key, value in new_env.items():
            old_env[key] = os.environ.get(key)  # save current value (or None if not set)
            os.environ[key] = value             # set new value
    try:
        yield
    finally:
        # Restore original values
        for key, old_value in old_env.items():
            if old_value is None:
                # If it was not originally set, remove it
                os.environ.pop(key, None)
            else:
                os.environ[key] = old_value
```

Explanation: The `system.py` module provides OS-level helpers. `join_path` and `ensure_dir` wrap common `pathlib.Path` usage for convenience. `atomic_write` implements the transaction-style file write (write to temp and rename) to avoid partial writes. `working_directory` and `temporary_env` are context managers for temporarily changing the working directory and environment variables, respectively, ensuring these changes do not leak outside the `with` block. All functions use type hints and have docstring examples.

File: `src/pyu/core/logger.py`

```
"""Logging utilities with a simple default configuration."""

import logging
import sys
from typing import Optional

# Create a module-level logger instance (will be configured in configure_logger)
_logger: Optional[logging.Logger] = None

def configure_logger(level: int = logging.INFO, fmt: Optional[str] = None) -> logging.Logger:
    """Configure the default logger for pyu.

    Sets up a StreamHandler to stdout with a simple format by default.
    `level` determines the logging level threshold.
    `fmt` can be provided to customize the log message format.
    If called multiple times, it will not add duplicate handlers.
    """
    global _logger
    if _logger is None:
        # Initialize logger only once
        _logger = logging.getLogger("pyu")
        _logger.setLevel(level)
        handler = logging.StreamHandler(sys.stdout)
        handler.setLevel(level)
        # Default format: "[LEVEL] function_name: message"
        if fmt is None:
            fmt = "[%(levelname)s] %(funcName)s: %(message)s"
        formatter = logging.Formatter(fmt)
        handler.setFormatter(formatter)
        _logger.addHandler(handler)
        # Avoid propagating to root (to prevent double logging if root logger is configured)
        _logger.propagate = False
    else:
        # If logger already exists, allow updating level or format
        _logger.setLevel(level)
        if fmt is not None:
            set_log_format(fmt)
    return _logger

def set_log_format(fmt: str) -> None:
  """Update the logging format for the default pyu logger.

  This function allows users to add additional info to logs (e.g., time, module).
  Example:
    set_log_format("[%(asctime)s] [%(levelname)s] %(funcName)s: %(message)s")
  """
    if _logger is None:
        # If logger isn't configured yet, configure with this format
        configure_logger(fmt=fmt)
    else:
        for handler in _logger.handlers:
            handler.setFormatter(logging.Formatter(fmt))

def debug(msg: str, *args: object, **kwargs: object) -> None:
  """Log a DEBUG level message."""
    if _logger is None:
        configure_logger()
    _logger.debug(msg, *args, stacklevel=2, **kwargs)

def info(msg: str, *args: object, **kwargs: object) -> None:
  """Log an INFO level message."""
    if _logger is None:
        configure_logger()
    _logger.info(msg, *args, stacklevel=2, **kwargs)

def warning(msg: str, *args: object, **kwargs: object) -> None:
  """Log a WARNING level message."""
    if _logger is None:
        configure_logger()
    _logger.warning(msg, *args, stacklevel=2, **kwargs)

def error(msg: str, *args: object, **kwargs: object) -> None:
  """Log an ERROR level message."""
    if _logger is None:
        configure_logger()
    _logger.error(msg, *args, stacklevel=2, **kwargs)

def critical(msg: str, *args: object, **kwargs: object) -> None:
  """Log a CRITICAL level message."""
```

```
    if _logger is None:
        configure_logger()
    _logger.critical(msg, *args, stacklevel=2, **kwargs)
```

Explanation: The `logger.py` module uses Python’ s built-in `logging` library to create a default logger `_logger`. `configure_logger()` sets up the logger with a console handler (printing to stdout) and a default format. It ensures the logger isn’ t reconfigured multiple times (to avoid duplicate handlers). The default format is minimal: `[LEVEL] funcName: message`. The `set_log_format` function allows the user to change the format (for example, to include timestamp or other details) on the fly. The module also provides convenience functions `debug, info, warning, error, critical` which log messages at the respective levels. Internally, these use `stacklevel=2` so that the `%(funcName)s` in the log format will reflect the caller of these functions (i.e., the function in the user’ s code that is logging) rather than always showing “info” or “error” (the name of our wrapper function). This way, logs will indicate the actual source function of the log message. All functions ensure that the logger is configured (initializing it on first use if not already done).

With this logger utility, a user can simply call `pyu.info("message")` in their code, and by default see output like `[INFO] calling_function: message`. If they want timestamps or other context, they can call `pyu.set_log_format(...)` to adjust globally. The logger configuration is shared across all uses of `pyu` in the application.

File: `src/pyu/core/progress.py`

```
"""Progress bar utility using tqdm for visual feedback on long operations."""
from typing import Iterable, TypeVar, Optional
from tqdm.auto import tqdm

T = TypeVar('T')

def progress(iterable: Optional[Iterable[T]] = None, total: Optional[int] = None, desc:
Optional[str] = None, **kwargs) -> Iterable[T]:
  """Wrap an iterable with a progress bar, or create a manual progress bar context.

  If an iterable is provided, yields items from it while displaying a progress bar.
  If no iterable is provided, returns a tqdm progress bar instance that can be manually updated.

  Examples:
    # Example 1: Progress for an iterable
    for item in progress(range(100), desc="Processing"):
      # do work with item
      ...

    # Example 2: Manual progress bar usage
    with progress(total=50, desc="Uploading") as pbar:
      for i in range(50):
        # perform chunk upload
        pbar.update(1)
  """
    if iterable is not None:
        # Create a tqdm iterator for the iterable
        return tqdm(iterable, total=total, desc=desc, **kwargs)
    else:
        # Return a tqdm object to be used as a context manager manually
        return tqdm(total=total, desc=desc, **kwargs)
```

Explanation: The `progress` function is a light wrapper around `tqdm.auto.tqdm`. If you pass an `iterable`, it simply returns `tqdm(iterable, ...)` so you can iterate with a progress bar. If you call `progress()` with no iterable (but with a `total`), it returns a `tqdm` progress bar which you can use as a context manager (`with progress(total=n) as p:`) for manual control (calling `p.update()` as needed inside the block). This design gives flexibility for different use cases while maintaining simplicity. By default, `tqdm.auto` will automatically choose between console or notebook widgets depending on the environment, providing a nice experience in most cases.

Users can customize `desc` (description text) or other `tqdm` parameters via kwargs. The library ensures that all projects use a consistent approach to progress bars, and by centralizing this in `pyu`, we can easily swap out the backend or adjust settings if needed in the future.

File: `src/pyu/core/signals.py`

```
"""Signal handling utilities to manage system signals (e.g., graceful Ctrl+C handling)."""
import signal

class SignalContext:
  """A context manager to handle signals (like SIGINT for Ctrl+C) gracefully.

  When this context is active, a Ctrl+C (SIGINT) will raise a KeyboardInterrupt immediately,
  allowing the program to break out of long operations. This context manager supports nesting.

  Usage:
    with SignalContext():  # enter signal-handling region
      # long running code
      with SignalContext():  # nested context is also allowed
        # more code
        ...
      # If Ctrl+C is pressed anywhere in the above block, a KeyboardInterrupt is raised,
      # and the contexts will clean up properly on exit.
  """
    def __init__(self, signum=signal.SIGINT):
        self.signum = signum
        self._prev_handler = None

    def __enter__(self):
        # Save current signal handler and set our own
        self._prev_handler = signal.signal(self.signum, self._handle_signal)
        return self

    def _handle_signal(self, signum, frame):
        # Simply raise KeyboardInterrupt to interrupt execution
```

```
            raise KeyboardInterrupt

    def __exit__(self, exc_type, exc_val, exc_tb):
            # Restore the previous signal handler
            signal.signal(self.signum, self._prev_handler)
            # Do not suppress any exceptions; if KeyboardInterrupt was raised, it will propagate
            return False
```

Explanation: `SignalContext` is a context manager class designed to ensure that a SIGINT (Ctrl+C) is handled immediately and predictably. When you enter the context (`with SignalContext():`), it installs a signal handler that raises `KeyboardInterrupt` as soon as SIGINT is received. Upon exiting the context (either normally or due to an interrupt), it restores the original signal handler. This class **supports nested usage**: if you enter a second `SignalContext` inside an outer one, it will stack another handler and restore the previous (which may be the outer handler) on exit. In practice, this means you can use `SignalContext` in any function that might run long enough to warrant interruption, without worrying if an outer scope also has one – Ctrl+C will unwind out of all contexts properly. The user doesn’ t typically need to interact with the `SignalContext` object itself; it’ s just used via the `with` statement. If a KeyboardInterrupt occurs, it will propagate out of the context, allowing any outer logic or the top-level script to catch it if needed. This tool helps prevent situations where a program becomes unresponsive to Ctrl+C (for instance, if some library call ignores interrupts). By peppering long-running operations with `with SignalContext():` (or by having one wrapping the highest-level loop), we ensure the program can always be interrupted and exit cleanly.

README.md (for pyu)

Below is the content of the `README.md` for the **pyu** library, which provides an overview, installation instructions, and usage examples for each major feature:

~~~
# pyu – Python Utilities Library

**pyu** is a lightweight Python utility library that provides common
functionality needed across projects. It includes:
- Convenient file system operations (path handling, atomic writes, etc.)
- A simple, extensible logging setup
- Progress bar utilities for long-running operations
- Signal handling context managers for graceful interruption (Ctrl+C)

The goal is to unify and simplify how these tasks are done in all our Python
projects, following a consistent design philosophy.

## Installation

**Prerequisites:** Python 3.12 or above.

You can install `pyu` in your environment using `pip`:

```bash
pip install pyu
~~~

(If you are using the uv tool to manage projects, you can add `pyu` as a dependency in your `pyproject.toml` or use `uv add pyu` to include it. Then run `uv install` to install all dependencies.)

Usage

After installation, you can import and use the functions from **pyu** in your project. Below are examples demonstrating the major features:

1. File System Utilities

Ensure a directory exists:

```python
from pyu import ensure_dir

data_dir = ensure_dir("output/data")
print(f"Data directory is at: {data_dir}")
# The directory "output/data" is now guaranteed to exist.
```

Join paths reliably:

```python
from pyu import join_path

config_path = join_path("/home/user", "myproject", "config.yaml")
# config_path is a Path object: "/home/user/myproject/config.yaml"
```

Atomic file write:

```python
from pyu import atomic_write

# Write data to a file safely (either fully writes or not at all)
atomic_write("results.txt", "All results completed.\n")
# If an error occurs during write, "results.txt" remains unchanged.
```

Temporarily change working directory:

```python
from pyu import working_directory

print("Current directory before:", os.getcwd())
with working_directory("/tmp"):
    # Inside this block, the current working directory is /tmp
    print("Current directory inside context:", os.getcwd())
# Outside the block, back to original directory
print("Current directory after:", os.getcwd())
```

Temporarily set environment variables:

```
from pyu import temporary_env
import os

print("DEBUG before:", os.environ.get("DEBUG"))
with temporary_env({"DEBUG": "1"}):
    # Inside context, environment variable DEBUG is set to "1"
    print("DEBUG inside context:", os.environ.get("DEBUG"))  # "1"
# After context, environment variable is restored to previous state
print("DEBUG after:", os.environ.get("DEBUG"))
```

2. Logging

Basic usage:

```
from pyu import info, warning, error

info("This is an info message.")
warning("This is a warning.")
error("This is an error.")
```

By default, these will print to stdout with a format `[LEVEL] function_name: message`. For example, calling `info` inside a function named `process_data` will output:

```
[INFO] process_data: This is an info message.
```

If you call it at the top level (not inside any function), the `funcName` will appear as `<module>`.

Customizing log format:

```
from pyu import set_log_format

# Add timestamp to the log format
set_log_format("[%(asctime)s] [%(levelname)s] %(funcName)s: %(message)s")
```

Now, subsequent log messages will include timestamps. You can adjust the format string using any of Python’s logging format variables. The logging level is set to INFO by default, but you can change it when configuring the logger if needed:

```
from pyu import configure_logger, debug

configure_logger(level=logging.DEBUG)  # switch to DEBUG level logging
debug("This is a debug message, now visible because level is DEBUG.")
```

(Note: The first call to any logging function will auto-configure the logger. If you need to change settings, call `configure_logger` early in your program.)

Logging to a file: By default, `pyu` logs to the console. If you need to log to a file or another destination, you can obtain the underlying logger via `configure_logger()` and add handlers as you would with the standard `logging` module. For example:

```python
import logging
from pyu import configure_logger

logger = configure_logger()   # get the logger instance
file_handler = logging.FileHandler("app.log")
logger.addHandler(file_handler)
```

(Be sure to close file handlers appropriately when the application exits to avoid resource leaks.)

3. Progress Bars

For any long loops or processing tasks, you can use `pyu.progress` to get a progress bar.

Iterating with a progress bar:

```python
from pyu import progress
import time

for i in progress(range(100), desc="Processing"):
    # simulate work
    time.sleep(0.1)
```

This will display a live updating progress bar in the console, with the description "Processing", the progress percentage, elapsed time, etc., courtesy of `tqdm`. It helps in knowing that your program is actively working and approximately how far along it is.

Manual control of progress:

If you have a process where you know the total steps but cannot use a simple loop, you can manually update the progress bar:

```python
from pyu import progress

total_steps = 5
with progress(total=total_steps, desc="Complex Task") as pbar:
    # Imagine these are various independent steps:
    step_names = ["load data", "process data", "train model", "evaluate", "cleanup"]
    for step in step_names:
        # ... perform the step ...
        print(f"Starting {step}...")
```

```
            # simulate work
            time.sleep(1)
            # update progress by one step
            pbar.update(1)
```

Inside the `with` block, we perform each step and call `pbar.update(1)` to increment the progress. The progress bar will update accordingly and close itself when the context is exited.

4. Signal Handling (Graceful Ctrl+C)

When running long computations or waiting on I/O, pressing Ctrl+C (KeyboardInterrupt) might sometimes not immediately stop the program if those operations don’ t regularly check for interrupts. `pyu` provides a context manager to ensure your program can break out promptly.

Basic usage of `SignalContext`:

```
from pyu import SignalContext

print("Starting long operation. Press Ctrl+C to interrupt.")
try:
    with SignalContext():
        # Simulate a long operation
        while True:
            time.sleep(5)   # sleeping to mimic a blocking task
            print("Still working... (press Ctrl+C to stop)")
except KeyboardInterrupt:
    print("Operation was interrupted by user.")
```

In this example, no matter when you press Ctrl+C during the loop, the `KeyboardInterrupt` will be raised and caught by the `except` block, allowing the program to print a message and exit cleanly. Without `SignalContext`, certain blocking calls might ignore the interrupt until they complete, which is not ideal.

Nested SignalContext usage:

You typically only need one `SignalContext` active at a time (for instance, around your main loop). However, it’ s safe to nest them if you have a function deep in your call stack that also uses `with SignalContext():`. The nested usage will still respect the interrupt and unwind correctly. You don’ t need to do anything special for nested contexts; just be aware that the deepest context will handle the signal first, and when it exits, the outer one continues to remain in effect.

Conclusion

pyu is intended to be your go-to toolkit for the patterns and utilities described above. Instead of rewriting path utils, logging setups, or progress loops in each project, you can rely on this library to provide a consistent, well-tested implementation. This not only speeds up development but also ensures all projects behave similarly (same logging format, same interrupt behavior, etc.).

For any questions or further examples, please refer to the `docs/` directory or the in-line docstrings for each function. We encourage contributions and suggestions – as we use **pyu** in more projects, it will evolve to cover more common needs while keeping with the minimalist design philosophy.

Happy coding! ```
