**TaskMasterPy Project Overview**

TaskMasterPy is a Python-based open-source automation framework focused on data operations, designed primarily for data scientists. It provides an extensible, CLI-first, event-driven automation engine that allows creating workflows via YAML/JSON config files or Python APIs.

**Core Components**

**Main Package Structure**

* **taskmaster/init.py**: Package initialization
* **taskmaster/autopilot.py**: Provides simplified interface for running workflows with minimal configuration
* **taskmaster/main.py**: Main entry point for the CLI application

**Core Modules**

* **taskmaster/core/workflow.py**: Defines the

 Workflow

 class that represents a complete workflow with triggers and actions in a DAG

* **taskmaster/core/runner.py**: Contains the WorkflowRunner class for executing and managing workflows

**Actions**

* **taskmaster/actions/base.py**: Base action class that all actions inherit from
* **taskmaster/actions/load\_data.py**: Actions for loading data from various sources (CSV, JSON, Excel, SQL)
* **taskmaster/actions/clean\_data.py**: Actions for data cleaning (DropNA, FixDataTypes, RenameColumns, FilterRows)
* **taskmaster/actions/transform\_data.py**: Actions for data transformation (Normalize, Aggregate, Pivot, Encode)
* **taskmaster/actions/save\_data.py**: Actions for saving data to various formats (CSV, JSON, Excel, SQL)
* **taskmaster/actions/api.py**: Actions for API integration (CallAPI, Webhook)
* **taskmaster/actions/script.py**: Actions for executing scripts (Python, Shell)
* **taskmaster/actions/notify.py**: Actions for notifications (Console, System, Email)
* **taskmaster/actions/email.py**: Actions for email operations

**Triggers**

* **taskmaster/triggers/base.py**: Base trigger class that all triggers inherit from
* **taskmaster/triggers/time\_trigger.py**: Trigger based on time schedules

**CLI**

* **taskmaster/cli/commands.py**: CLI command definitions using Typer

**Utils**

* **taskmaster/utils/config.py**: Utilities for loading and parsing workflow configurations
* **taskmaster/utils/logging.py**: Logging configuration
* **taskmaster/utils/validators.py**: Validation utilities for workflow configurations

**Plugins**

* **taskmaster/plugins/init.py**: Plugin system initialization
* **taskmaster/plugins/loader.py**: Plugin loading functionality

**Example Workflows**

* **example\_workflow.yaml**: Basic example workflow
* **examples/simple\_workflow.yaml**: Simple data processing workflow
* **examples/normalize\_workflow.yaml**: Workflow for normalizing data
* **examples/api\_workflow.yaml**: Workflow for fetching and processing API data
* **examples/etl\_pipeline.yaml**: ETL pipeline example in YAML
* **examples/file\_transform\_workflow.yaml**: Workflow for file transformation
* **examples/script\_workflow.yaml**: Workflow using custom Python scripts

**Example Python Scripts**

* **examples/etl\_pipeline.py**: ETL pipeline example using Python API
* **examples/etl\_pipeline\_fixed.py**: Fixed version with proper imports
* **examples/financial\_etl\_pipeline.py**: Financial data ETL example
* **examples/advanced\_etl\_pipeline.py**: More complex ETL pipeline
* **examples/custom\_transform.py**: Custom transformation functions

**Setup and Configuration**

* **setup.py**: Package setup script with dependencies
* **README.md**: Project documentation
* **test\_taskmaster.py**: Simple test script

**Package Metadata**

* **taskmasterpy.egg-info/**: Package metadata files
  + **PKG-INFO**: Package information
  + **SOURCES.txt**: List of source files
  + **requires.txt**: Package dependencies
  + **top\_level.txt**: Top-level package names

**Key Features**

1. **Event-driven architecture**: Define workflows that respond to various triggers
2. **Data-focused actions**: Built-in actions for common data operations
3. **YAML/JSON configuration**: Define workflows without writing code
4. **Python API**: Programmatically create and run workflows
5. **Plugin system**: Extend with custom triggers and actions
6. **CLI interface**: Run and manage workflows from the command line
7. **Autopilot mode**: One-liner for simple data processing tasks

The project follows a modular design pattern where each component (actions, triggers) inherits from a base class, making it extensible for custom implementations. The workflow system uses a directed acyclic graph (DAG) to manage dependencies between actions.

**Detailed Code Analysis of TaskMasterPy**

**Core Components**

**taskmaster/core/workflow.py**

This file defines the

Workflow

class, which is the central component of the system.

workflow.py

TaskMasterPy/taskmaster/core

Copy

class Workflow:

    """A workflow is a collection of triggers and

    actions arranged in a DAG."""

    def \_\_init\_\_(self, name: str = None, description:

    str = None):

        self.id = str(uuid.uuid4())

        self.name = name or f"Workflow\_{self.id[:8]}"

        self.description = description or ""

        self.triggers: List[BaseTrigger] = []

        self.actions: Dict[str, BaseAction] = {}

        self.context: Dict[str, Any] = {}

        self.is\_running = False

        self.logger = logging.getLogger(f"taskmaster.

        workflow.{self.name}")

Key features:

* Uses UUID for unique workflow identification
* Maintains collections of triggers and actions
* Implements a DAG (Directed Acyclic Graph) for action dependencies
* Provides methods for adding triggers, actions, and dependencies
* Includes execution logic that respects dependencies
* Uses Python's logging module for tracking workflow execution

**taskmaster/core/runner.py**

The WorkflowRunner class manages workflow execution:

* Registers and stores workflows
* Provides methods to run workflows on demand or via triggers
* Handles workflow lifecycle (activation/deactivation)

**taskmaster/autopilot.py**

autopilot.py

TaskMasterPy/taskmaster

Copy

"""

Autopilot mode for TaskMasterPy.

This module provides a simplified interface for running

workflows.

"""

import os

import pandas as pd

from typing import Dict, Any, Optional, List, Union

from taskmaster.core.workflow import Workflow

from taskmaster.core.runner import WorkflowRunner

This module provides a simplified API for common data operations:

* Uses type hints for better IDE support and documentation
* Imports core components from other modules
* Likely provides one-liner functions for common data tasks
* Abstracts away the complexity of creating workflows manually

**Actions**

**taskmaster/actions/base.py**

Defines the

BaseAction

abstract class that all actions inherit from:

* Includes common properties like ID, name, status
* Defines the execution lifecycle (setup, execute, cleanup)
* Manages dependencies between actions
* Implements error handling

**taskmaster/actions/load\_data.py**

load\_data.py

TaskMasterPy/taskmaster/actions

Copy

class LoadDataAction(BaseAction):

    """Base class for actions that load data from

    various sources."""

    def \_\_init\_\_(self, name: str = None, config: Dict

    [str, Any] = None):

        """Initialize a new load data action."""

        super().\_\_init\_\_(name, config)

        self.data = None

    def execute(self, context: Dict[str, Any] = None)

    -> pd.DataFrame:

        """Execute the action to load data."""

        raise NotImplementedError("Subclasses must

        implement execute()")

This file contains classes for loading data:

* LoadDataAction: Abstract base class for all data loading actions
* LoadCSVAction: Loads data from CSV files using pandas
* LoadJSONAction: Loads data from JSON files
* LoadExcelAction: Loads data from Excel files
* LoadSQLAction: Loads data from SQL databases

Each class:

* Inherits from BaseAction
* Implements the execute() method for its specific data source
* Uses pandas for data manipulation
* Handles file existence checks and error conditions
* Returns a pandas DataFrame

**taskmaster/actions/save\_data.py**

save\_data.py

TaskMasterPy/taskmaster/actions

Copy

def execute(self, context: Dict[str, Any] = None) ->

str:

    """Execute the action to save data to a JSON file.

    """

    context = context or {}

    # Get the input DataFrame

    df = self.\_get\_input\_dataframe(context)

    # Get parameters from config

    file\_path = self.config.get("file\_path", "")

    orient = self.config.get("orient", "records")

    indent = self.config.get("indent", 2)

    encoding = self.config.get("encoding", "utf-8")

    # Create directory if it doesn't exist

    os.makedirs(os.path.dirname(os.path.abspath

    (file\_path)), exist\_ok=True)

    # Save the DataFrame to JSON

    df.to\_json(

        file\_path,

        orient=orient,

        indent=indent,

        force\_ascii=False

    )

    return file\_path

Contains classes for saving data:

* SaveDataAction: Base class for all data saving actions
* SaveCSVAction, SaveJSONAction, SaveExcelAction, SaveSQLAction

Features:

* Creates directories if they don't exist
* Configurable output options (encoding, format-specific options)
* Returns the path where data was saved
* Uses pandas' built-in export functions

**taskmaster/actions/transform\_data.py**

Contains data transformation actions:

* NormalizeAction: Normalizes data using various methods (z-score, min-max)
* AggregateAction: Performs groupby operations
* PivotAction: Creates pivot tables
* EncodeAction: Encodes categorical data

**taskmaster/actions/clean\_data.py**

Contains data cleaning actions:

* DropNAAction: Removes rows with missing values
* FixDataTypesAction: Corrects data types
* RenameColumnsAction: Renames columns
* FilterRowsAction: Filters rows based on conditions

**taskmaster/actions/api.py**

Contains API-related actions:

* CallAPIAction: Makes HTTP requests to APIs
* WebhookAction: Sends webhook notifications

**taskmaster/actions/notify.py**

Contains notification actions:

* ConsoleNotifyAction: Prints messages to console
* SystemNotifyAction: Sends system notifications

**Triggers**

**taskmaster/triggers/base.py**

Defines the BaseTrigger abstract class:

* Includes activation/deactivation methods
* Implements callback registration
* Provides event firing mechanism

**taskmaster/triggers/time\_trigger.py**

Implements time-based triggers:

* Uses scheduling libraries for time-based execution
* Supports cron-like expressions
* Handles recurring and one-time schedules

**Utils**

**taskmaster/utils/config.py**

config.py

TaskMasterPy/taskmaster/utils

Copy

# Import all action types

from taskmaster.actions.load\_data import LoadCSVAction,

LoadJSONAction, LoadExcelAction, LoadSQLAction

from taskmaster.actions.clean\_data import DropNAAction,

FixDataTypesAction, RenameColumnsAction,

FilterRowsAction

from taskmaster.actions.transform\_data import

NormalizeAction, AggregateAction, PivotAction,

EncodeAction

from taskmaster.actions.save\_data import SaveCSVAction,

SaveJSONAction, SaveExcelAction, SaveSQLAction

This file:

* Imports all action types for dynamic instantiation
* Provides functions to load workflow configurations from YAML/JSON
* Includes factory methods to create actions based on type strings
* Maps configuration keys to action parameters

**Example Files**

**examples/etl\_pipeline.py**

etl\_pipeline.py

TaskMasterPy/examples

Copy

"""

ETL Pipeline using TaskMasterPy

This script demonstrates how to use TaskMasterPy to

create an ETL pipeline

that extracts data from an API, transforms it, and

loads it into a CSV file.

"""

from taskmaster.core.workflow import Workflow

from taskmaster.core.runner import WorkflowRunner

from taskmaster.triggers.time\_trigger import TimeTrigger

from taskmaster.actions.api import CallAPIAction

from taskmaster.actions.transform\_data import

NormalizeAction, AggregateAction

from taskmaster.actions.save\_data import SaveCSVAction

from taskmaster.actions.notify import

ConsoleNotifyAction

def create\_etl\_pipeline():

    """Create an ETL pipeline workflow."""

    # Create a workflow

    workflow = Workflow(

        name="API ETL Pipeline",

        description="Extract data from an API,

        transform it, and load it into a CSV file"

    )

This example:

* Demonstrates creating a workflow programmatically
* Shows how to add triggers and actions
* Establishes dependencies between actions
* Implements a complete ETL pipeline
* Uses Python's docstrings for documentation

**example\_workflow.yaml**

example\_workflow.yaml

TaskMasterPy

Copy

name: Example Data Processing Workflow

description: A simple workflow to process a CSV file

triggers:

  - type: time

    name: Manual Trigger

    config:

      schedule\_str: every 1 hour

actions:

  - id: load\_data

    type: load\_csv

    name: Load CSV Data

    config:

      file\_path: ./data/test.csv

This YAML file:

* Defines a workflow configuration
* Uses a declarative approach to workflow definition
* Specifies triggers, actions, and dependencies
* Includes configuration parameters for each component
* Demonstrates the YAML-based workflow definition

**test\_taskmaster.py**

test\_taskmaster.py

TaskMasterPy

Copy

def test\_workflow():

    """Create and run a simple workflow."""

    # Create test data

    test\_file = create\_test\_data()

    # Create the workflow

    workflow = Workflow(name="Test Workflow")

    # Create actions

    load\_action = LoadCSVAction(

        name="Load Test Data",

        config={"file\_path": test\_file}

    )

This test file:

* Creates a simple workflow for testing
* Generates test data
* Demonstrates the basic API usage
* Verifies core functionality works correctly

**Technical Implementation Details**

The project uses:

1. **Python's type hints** for better code documentation and IDE support
2. **Object-oriented design** with inheritance for extensibility
3. **Pandas** for data manipulation
4. **YAML/JSON** for configuration
5. **Logging** for execution tracking
6. **UUID** for unique identifiers
7. **Factory pattern** for creating actions from configuration
8. **Strategy pattern** for different action implementations
9. **Observer pattern** for trigger callbacks
10. **Dependency injection** for workflow context

Each file follows a consistent structure with imports, class definitions, docstrings, and type annotations, making the codebase maintainable and extensible.