"""

Lazy Package Loader

This module provides utilities for lazy loading Python packages to improve startup time

and reduce memory usage by only importing packages when they are actually used.

Features:

- Type-safe lazy loading of packages

- Support for nested module imports

- Auto-completion support in IDEs

- Thread-safe implementation

- Comprehensive test coverage

"""

from types import ModuleType

from typing import (

    Optional,

    Dict,

    Any,

    Callable,

    Type,

    TypeVar,

    Union,

    cast,

)

import importlib

```
import functools
import threading
from importlib.util import find_spec
from swarms.utils.auto_download_check_packages import (
    auto_check_and_download_package,
)



T = TypeVar("T")
C = TypeVar("C")



class ImportError(Exception):
    """Raised when a lazy import fails."""

    pass



class LazyLoader:
    """
    A thread-safe lazy loader for Python packages that only imports them when accessed.

    Attributes:
        _module_name (str): The name of the module to be lazily loaded
        _module (Optional[ModuleType]): The cached module instance once loaded
        _lock (threading.Lock): Thread lock for safe concurrent access
```

```
    Examples:
        >>> np = LazyLoader('numpy')
        >>> # numpy is not imported yet
        >>> result = np.array([1, 2, 3])
        >>> # numpy is imported only when first used
    """

    def __init__(self, module_name: str) -> None:
        """
        Initialize the lazy loader with a module name.

        Args:
            module_name: The fully qualified name of the module to lazily load

        Raises:
            ImportError: If the module cannot be found in sys.path
        """
        self._module_name = module_name
        self._module: Optional[ModuleType] = None
        self._lock = threading.Lock()

        auto_check_and_download_package(
            module_name, package_manager="pip"
        )
```

```
    # Verify module exists without importing it
    if find_spec(module_name) is None:
        raise ImportError(
            f"Module '{module_name}' not found in sys.path"
        )

def _load_module(self) -> ModuleType:
    """
    Thread-safe module loading.

    Returns:
        ModuleType: The loaded module

    Raises:
        ImportError: If module import fails
    """
    if self._module is None:
        with self._lock:
            # Double-check pattern
            if self._module is None:
                try:
                    self._module = importlib.import_module(
                        self._module_name
                    )
                except Exception as e:
                    raise ImportError(
```

```
                    f"Failed to import '{self._module_name}': {str(e)}"
                ) 
    return cast(ModuleType, self._module)

def __getattr__(self, name: str) -> Any:
    """
    Intercepts attribute access to load the module if needed.

    Args:
        name: The attribute name being accessed

    Returns:
        Any: The requested attribute from the loaded module

    Raises:
        AttributeError: If the attribute doesn't exist in the module
    """
    module = self._load_module()
    try:
        return getattr(module, name)
    except AttributeError:
        raise AttributeError(
            f"Module '{self._module_name}' has no attribute '{name}'"
        )

def __dir__(self) -> list[str]:
```

```
        """
        Returns list of attributes for autocomplete support.

        Returns:
            List[str]: Available attributes in the module
        """
        return dir(self._load_module())

    def is_loaded(self) -> bool:
        """
        Check if the module has been loaded.

        Returns:
            bool: True if module is loaded, False otherwise
        """
        return self._module is not None


class LazyLoaderMetaclass(type):
    """Metaclass to handle lazy loading behavior"""

    def __call__(cls, *args, **kwargs):
        if hasattr(cls, "_lazy_loader"):
            return super().__call__(*args, **kwargs)
        return super().__call__(*args, **kwargs)
```

```
class LazyClassLoader:
    """
    A descriptor that creates the actual class only when accessed,
    with proper inheritance support.
    """

    def __init__(
        self, class_name: str, bases: tuple, namespace: Dict[str, Any]
    ):
        self.class_name = class_name
        self.bases = bases
        self.namespace = namespace
        self._real_class: Optional[Type] = None
        self._lock = threading.Lock()

    def _create_class(self) -> Type:
        """Creates the actual class if it hasn't been created yet."""
        if self._real_class is None:
            with self._lock:
                if self._real_class is None:
                    # Update namespace to include metaclass
                    namespace = dict(self.namespace)
                    namespace["__metaclass__"] = LazyLoaderMetaclass

                    # Create the class with metaclass
```

```
                new_class = LazyLoaderMetaclass(
                    self.class_name, self.bases, namespace
                )

                # Store reference to this loader
                new_class._lazy_loader = self
                self._real_class = new_class

        return cast(Type, self._real_class)

    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        """Creates an instance of the lazy loaded class."""
        real_class = self._create_class()
        # Use the metaclass __call__ method
        return real_class(*args, **kwargs)

    def __instancecheck__(self, instance: Any) -> bool:
        """Support for isinstance() checks"""
        real_class = self._create_class()
        return isinstance(instance, real_class)

    def __subclasscheck__(self, subclass: Type) -> bool:
        """Support for issubclass() checks"""
        real_class = self._create_class()
        return issubclass(subclass, real_class)
```

```
def lazy_import(*names: str) -> Dict[str, LazyLoader]:
    """
    Create multiple lazy loaders at once.

    Args:
        *names: Module names to create lazy loaders for

    Returns:
        Dict[str, LazyLoader]: Dictionary mapping module names to their lazy loaders

    Examples:
        >>> modules = lazy_import('numpy', 'pandas', 'matplotlib.pyplot')
        >>> np = modules['numpy']
        >>> pd = modules['pandas']
        >>> plt = modules['matplotlib.pyplot']
    """
    return {name.split(".")[-1]: LazyLoader(name) for name in names}


def lazy_import_decorator(
    target: Union[Callable[..., T], Type[C]]
) -> Union[Callable[..., T], Type[C], LazyClassLoader]:
    """
    Enhanced decorator that supports both lazy imports and lazy class loading.
    """
```

```
if isinstance(target, type):
    # Store the original class details
    namespace = {
        name: value
        for name, value in target.__dict__.items()
        if not name.startswith("__")
        or name in ("__init__", "__new__")
    }

    # Create lazy loader
    loader = LazyClassLoader(
        target.__name__, target.__bases__, namespace
    )

    # Preserve class metadata
    loader.__module__ = target.__module__
    loader.__doc__ = target.__doc__

    # Add reference to original class
    loader._original_class = target

    return loader
else:
    # Handle function decoration
    @functools.wraps(target)
    def wrapper(*args: Any, **kwargs: Any) -> T:
```

```
    return target(*args, **kwargs)


return wrapper
```