Python Agent Component

This component extends the AI Agent Master Definition with Python-specific guidelines, best practices, and tooling instructions.

Python Environment Management

Virtual Environments

- **Always Use Virtual Environments**: Use `venv` to isolate dependencies
- **Detect Existing Environments**: Check for `.venv/`, `venv/`, or `conda` environments before creating new ones
- **Activation**: Ensure virtual environment is activated before running Python commands
- **Documentation**: Document environment setup in README or project documentation

Dependency Management

- **requirements.txt**: Use for simple projects and compatibility
- **pyproject.toml**: Prefer for modern projects (PEP 518/621)
- **Pin Versions**: Specify exact versions for reproducibility in production
- **Separate Dev Dependencies**: Keep development/test dependencies separate from production

Code Style and Formatting

PEP 8 Compliance

- **Follow PEP 8**: Adhere to Python's official style guide
- **Line Length**: Maximum 88 characters (Black) or 79 (PEP 8), follow project convention
- **Indentation**: Use 4 spaces, never tabs
- **Naming Conventions**:
  - `snake_case` for functions, variables, and module names
  - `PascalCase` for class names
  - `UPPER_SNAKE_CASE` for constants
  - `_leading_underscore` for internal/private members

Automatic Formatting

- **Black**: Use for opinionated, consistent formatting
- **autopep8**: Alternative for PEP 8 compliant formatting
- **isort**: Organize imports alphabetically and by type
- **Check Existing Config**: Look for `pyproject.toml`, `.flake8`, or `setup.cfg`

Linting Tools

- **pylint**: Comprehensive code analysis and quality checks
- **flake8**: Fast style guide enforcement
- **ruff**: Modern, fast linter (combines multiple tools)

- **mypy**: Static type checking for type-annotated code
- **Run Before Commit**: Always run configured linters before committing

Type Annotations

Type Hinting Best Practices

- **Use Type Hints**: Add type annotations to function signatures
- **Modern Syntax**: Use `list[str]` instead of `List[str]` (Python 3.9+)
- **Optional Types**: Use `Optional[T]` or `T | None` for nullable types
- **Return Types**: Always annotate function return types
- **Complex Types**: Use `TypedDict`, `Protocol`, and `Generic` appropriately

```python
from typing import Optional, TypedDict
from collections.abc import Sequence

class UserDict(TypedDict):
    name: str
    age: int
    email: Optional[str]

def process_users(users: Sequence[UserDict]) -> list[str]:
    return [user["name"] for user in users]
```

Type Checking

- **mypy**: Run `mypy .` to check type consistency
- **pyright**: Alternative type checker with better performance
- **Configuration**: Respect `mypy.ini` or `[tool.mypy]` in `pyproject.toml`
- **Strict Mode**: Enable strict mode for new projects when appropriate

Testing

Testing Frameworks

- **pytest**: Preferred modern testing framework
- **unittest**: Use when already established in project
- **doctest**: Use for simple example-based tests in docstrings
- **Test Discovery**: Follow naming conventions (`test_*.py` or `*_test.py`)

Test Structure

```python
# tests/test_module.py
import pytest
from mymodule import MyClass

class TestMyClass:
    """Test suite for MyClass"""

    def test_initialization(self):
        """Test object creation"""
        obj = MyClass(value=42)
        assert obj.value == 42

    def test_method_behavior(self):
        """Test specific method behavior"""
        obj = MyClass(value=10)
        result = obj.process()
        assert result == expected_value

    @pytest.mark.parametrize("input,expected", [
        (1, 2),
        (2, 4),
        (3, 6),
    ])
    def test_multiple_cases(self, input, expected):
        """Test multiple input/output pairs"""
        assert MyClass.double(input) == expected
```

Test Coverage

- **coverage.py**: Measure test coverage with `coverage run -m pytest`
- **Minimum Coverage**: Aim for 80%+ coverage, follow project standards
- **Coverage Reports**: Generate HTML reports with `coverage html`
- **CI Integration**: Run coverage checks in continuous integration

Testing Best Practices

- **Arrange-Act-Assert**: Structure tests clearly (AAA pattern)
- **One Assertion Focus**: Each test should verify one specific behavior
- **Fixtures**: Use pytest fixtures for setup/teardown and shared data
- **Mocking**: Use `unittest.mock` or `pytest-mock` for external dependencies
- **Fast Tests**: Keep tests fast; use markers for slow integration tests

Documentation

Docstrings

- **Always Document**: Add docstrings to all public modules, classes, and functions
- **Format**: Use Google, NumPy, or Sphinx style consistently
- **Include Examples**: Add usage examples in docstrings when helpful

```
def calculate_average(numbers: list[float]) -> float:
    """Calculate the arithmetic mean of a list of numbers.

    Args:
        numbers: A list of numeric values

    Returns:
        The arithmetic mean of the input numbers

    Raises:
        ValueError: If the input list is empty

    Examples:
        >>> calculate_average([1, 2, 3, 4, 5])
        3.0
    """
    if not numbers:
        raise ValueError("Cannot calculate average of empty list")
    return sum(numbers) / len(numbers)
```

Documentation Tools

- **Sphinx**: Generate comprehensive documentation from docstrings
- **mkdocs**: Modern documentation with Markdown support
- **pdoc**: Lightweight auto-documentation tool
- **Read the Docs**: Host documentation for open-source projects

Python-Specific Patterns

Pythonic Idioms

- **List Comprehensions**: Use instead of `map`/`filter` when clear

```
# Good
squares = [x**2 for x in range(10)]
# Avoid
squares = list(map(lambda x: x**2, range(10)))
```

- **Context Managers**: Use `with` for resource management

```
# Good
with open("file.txt") as f:
    data = f.read()
# Avoid manually closing files
```

- **Enumerate and Zip**: Use built-in iteration helpers

```
# Good
for i, item in enumerate(items):
    print(f"{i}: {item}")

# Good
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

- **Dictionary Methods**: Use `.get()`, `.setdefault()`, and `defaultdict`

```
# Good
value = my_dict.get(key, default_value)
# Avoid KeyError handling for simple cases
```

Modern Python Features

- **F-strings**: Use for string formatting (Python 3.6+)
- **Dataclasses**: Use `@dataclass` for simple data containers (Python 3.7+)
- **Pattern Matching**: Use `match`/`case` for complex conditionals (Python 3.10+)
- **Walrus Operator**: Use `:=` for assignment expressions when clear (Python 3.8+)

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float

    def distance(self) -> float:
        return (self.x**2 + self.y**2)**0.5
```

Error Handling

- **Specific Exceptions**: Catch specific exceptions, not bare `except:`
- **Custom Exceptions**: Create domain-specific exception classes
- **EAFP**: "Easier to Ask for Forgiveness than Permission" - use try/except
- **Context**: Provide clear error messages with context

```
class ValidationError(Exception):
    """Raised when input validation fails"""
    pass

def validate_age(age: int) -> None:
    """Validate age is within acceptable range"""
    try:
        if not 0 <= age <= 150:
            raise ValidationError(f"Age {age} is outside valid range [0, 150]")
    except TypeError:
        raise ValidationError(f"Age must be an integer, got {type(age)}")
```

Package Structure

Project Layout

```
myproject/
├── src/
│   └── mypackage/
│       ├── __init__.py
│       ├── module1.py
│       └── subpackage/
│           ├── __init__.py
│           └── module2.py
├── tests/
│   ├── __init__.py
│   ├── test_module1.py
│   └── test_module2.py
├── docs/
│   └── conf.py
├── .gitignore
├── README.md
├── LICENSE
├── pyproject.toml
└── requirements.txt
```

Module Organization

- `__init__.py`: Define public API, import key components
- **Single Responsibility**: Each module should have a clear, focused purpose
- **Avoid Circular Imports**: Structure dependencies as a DAG
- **Relative Imports**: Use for intra-package imports in larger projects

Performance Optimization

Profiling

- **cProfile**: Built-in profiler for performance analysis
- **line_profiler**: Line-by-line profiling for bottlenecks
- **memory_profiler**: Track memory usage
- **timeit**: Benchmark small code snippets

Common Optimizations

- **Use Built-ins**: Built-in functions are implemented in C and faster
- **List Comprehensions**: Faster than explicit loops for simple operations

- **Generator Expressions**: Use for large datasets to save memory
- **Local Variables**: Faster lookup than globals or attributes
- `__slots__`: Reduce memory for classes with many instances
- **NumPy**: Use for numerical operations on arrays
- **Caching**: Use `@lru_cache` for expensive, pure functions

```python
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n: int) -> int:
    """Calculate Fibonacci number with memoization"""
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Security Best Practices

Input Validation

- **Sanitize Input**: Never trust user input
- **Type Validation**: Use type hints and runtime validation
- **SQL Injection**: Use parameterized queries, never string concatenation
- **Command Injection**: Avoid `shell=True` in `subprocess`, use argument lists

Secrets Management

- **Environment Variables**: Store secrets in environment, never in code
- **python-dotenv**: Use for local development secrets
- **Secrets Detection**: Never commit `.env` files with real secrets
- **Cryptography**: Use `cryptography` library, not deprecated `pycrypto`

```python
import os
from pathlib import Path

# Good
API_KEY = os.getenv("API_KEY")
if not API_KEY:
    raise ValueError("API_KEY environment variable not set")

# Bad - Never do this
API_KEY = "sk-1234567890abcdef"  # Never hardcode secrets
```

Dependency Security

- **pip-audit**: Scan dependencies for known vulnerabilities
- **safety**: Check for security issues in requirements

- **Dependabot**: Enable for automatic security updates
- **Review Dependencies**: Audit new dependencies before adding

Common Python Tools

Build and Package

- **setuptools**: Traditional build backend
- **build**: PEP 517 build frontend
- **twine**: Upload packages to PyPI
- **wheel**: Create binary distributions

Code Quality

- **pre-commit**: Automate checks before commits
- **tox**: Test across multiple Python versions
- **nox**: Flexible test automation
- **bandit**: Security-focused static analysis

Development Workflow

```
# Setup
python -m venv .venv
source .venv/bin/activate  # or `.venv\Scripts\activate` on Windows
pip install -e ".[dev]"

# Code quality
black .
isort .
flake8 .
mypy .

# Testing
pytest
coverage run -m pytest
coverage report

# Security
bandit -r src/
pip-audit
```

Python Version Compatibility

Version Support

- **Specify Minimum Version**: Document and enforce minimum Python version
- **Active Support**: Target actively supported Python versions (currently 3.9+)
- **EOL Awareness**: Plan migration from end-of-life versions

- `python_requires`: Set in `pyproject.toml` or `setup.py`

Compatibility Checks

- **pyupgrade**: Automatically upgrade syntax to newer Python versions
- **vermin**: Detect minimum Python version required
- **tox**: Test across multiple Python versions
- **CI Matrix**: Run tests on all supported Python versions

Async Python

Asyncio Best Practices

- **async/await**: Use for I/O-bound concurrent operations
- **Not for CPU-bound**: Use multiprocessing for CPU-bound tasks
- **Consistent Colors**: Don't mix sync and async without proper bridging
- **Libraries**: Use async-compatible libraries (aiohttp, asyncpg, etc.)

```python
import asyncio
from typing import List

async def fetch_data(url: str) -> dict:
    """Fetch data asynchronously"""
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.json()

async def fetch_all(urls: List[str]) -> List[dict]:
    """Fetch multiple URLs concurrently"""
    tasks = [fetch_data(url) for url in urls]
    return await asyncio.gather(*tasks)
```

Data Science and ML Considerations

Common Libraries

- **NumPy**: Numerical computing with arrays
- **pandas**: Data manipulation and analysis
- **scikit-learn**: Machine learning algorithms
- **matplotlib/seaborn**: Data visualization
- **Jupyter**: Interactive notebooks for exploration

Best Practices

- **Reproducibility**: Set random seeds for reproducible results
- **Data Validation**: Validate input data shape and types
- **Memory Efficiency**: Use chunking for large datasets
- **Vectorization**: Prefer NumPy operations over Python loops

Agent-Specific Python Guidelines

Investigation Commands

```bash
# Check Python version
python --version

# Check installed packages
pip list
pip show <package-name>

# Check project structure
find . -name "*.py" | head -20
cat pyproject.toml || cat setup.py || cat requirements.txt

# Check for tests
pytest --collect-only

# Check linting setup
cat .flake8 || grep -A 10 "\[tool.flake8\]" pyproject.toml
```

Common Task Patterns

Adding a New Feature

1. Identify module location following existing structure
2. Add type-annotated function/class with docstring
3. Add corresponding tests in `tests/` directory
4. Run linters and formatters
5. Run test suite
6. Update documentation if needed

Fixing a Bug

1. Write a failing test that reproduces the bug
2. Fix the minimal code to make test pass
3. Run full test suite to prevent regression
4. Verify linting passes
5. Document fix in commit message

Refactoring

1. Ensure tests exist and pass before refactoring
2. Make small, incremental changes
3. Run tests after each change
4. Keep functionality identical
5. Update docstrings if signature changes

Success Criteria for Python Tasks

A Python task is complete when:

- Code follows PEP 8 and project style guidelines
- All functions have type hints and docstrings
- Tests pass with adequate coverage
- Linters (flake8, pylint, ruff) pass without errors
- Type checker (mypy) passes if configured
- Security scanners show no new vulnerabilities
- Dependencies are properly specified
- Documentation is updated for API changes

This component should be referenced by agents working on Python projects. It extends the core agent principles with Python-specific expertise.