Python Architecture Pillars: Supplementary Guide

Table of Contents

Introduction

This document provides deeper explanations of foundational concepts that support the four pillars of Python architecture: Reusability (Functions), Resilience (Exceptions), Persistence (Files), and Organization (Modules). These topics are essential prerequisites that enhance your understanding of professional Python development.

Understanding Variable Scope in Functions

The PDF introduces functions but doesn't explain how Python determines which variables are accessible where. This concept is critical for avoiding bugs.

The LEGB Rule

Python resolves variable names using the LEGB rule, checking scopes in this order:

1. **Local**: Variables defined inside the current function
2. **Enclosing**: Variables in any enclosing functions (for nested functions)
3. **Global**: Variables defined at the module level
4. **Built-in**: Python's pre-defined names like `print`, `len`, `int`

Practical Example

```
message = "I am global"  # Global scope

def outer_function():
    message = "I am enclosing"  # Enclosing scope

    def inner_function():
        message = "I am local"  # Local scope
        print(message)  # Prints: "I am local"

    inner_function()
    print(message)  # Prints: "I am enclosing"
```

```
outer_function()
print(message)  # Prints: "I am global"
```

The `global` and `nonlocal` Keywords

When you need to modify a variable from an outer scope:

```
counter = 0

def increment():
    global counter  # Declares intent to modify global variable
    counter += 1

increment()
print(counter)  # Output: 1
```

```
def make_counter():
    count = 0

    def increment():
        nonlocal count  # Modifies the enclosing scope's variable
        count += 1
        return count

    return increment

my_counter = make_counter()
print(my_counter())  # Output: 1
print(my_counter())  # Output: 2
```

Why This Matters for Functions

Understanding scope prevents common bugs:

```
# Common mistake: forgetting that assignment creates a local variable
total = 100

def add_to_total(value):
    # This creates a NEW local variable, doesn't modify global
    total = total + value  # UnboundLocalError!
    return total

# Correct approach
def add_to_total(value):
    global total
    total = total + value
```

```
    return total
```

Mutable vs Immutable Default Arguments

The PDF covers default parameters but doesn't address a common pitfall that trips up many Python developers.

The Problem

```
def add_item(item, item_list=[]):  # Dangerous!
    item_list.append(item)
    return item_list

print(add_item("apple"))   # ['apple']
print(add_item("banana"))  # ['apple', 'banana'] - Unexpected!
print(add_item("cherry"))  # ['apple', 'banana', 'cherry'] - Bug!
```

Why This Happens

Default argument values are evaluated **once** when the function is defined, not each time the function is called. Mutable objects (lists, dictionaries, sets) retain modifications between calls.

The Solution

Use `None` as the default and create the mutable object inside the function:

```
def add_item(item, item_list=None):
    if item_list is None:
        item_list = []  # Fresh list created each call
    item_list.append(item)
    return item_list

print(add_item("apple"))   # ['apple']
print(add_item("banana"))  # ['banana'] - Correct!
```

Rule of Thumb

Default arguments should always be immutable types:

- ✅ Safe: `None`, numbers, strings, tuples
- ❌ Dangerous: lists, dictionaries, sets, custom objects

The Exception Hierarchy

The PDF shows common exceptions but doesn't explain how they relate to each other. Understanding the hierarchy helps you write better exception handlers.

Python's Exception Tree (Simplified)

```
BaseException
 ├─── SystemExit
 ├─── KeyboardInterrupt
 ├─── GeneratorExit
 └─── Exception
      ├─── StopIteration
      ├─── ArithmeticError
      │    ├─── ZeroDivisionError
      │    ├─── OverflowError
      │    └─── FloatingPointError
      ├─── LookupError
      │    ├─── IndexError
      │    └─── KeyError
      ├─── OSError
      │    ├─── FileNotFoundError
      │    ├─── PermissionError
      │    └─── FileExistsError
      ├─── ValueError
      ├─── TypeError
      └─── AttributeError
```

Why the Hierarchy Matters

Catching a parent exception also catches all its children:

```
# This catches both IndexError and KeyError
try:
    some_operation()
except LookupError as e:
    print(f"Item not found: {e}")
```

Best Practice: Be Specific

```
# Too broad - hides bugs!
try:
    value = data[key]
except Exception:
    value = default
```

```
# Better - only catches what you expect
try:
    value = data[key]
except KeyError:
    value = default
```

Never Catch BaseException

```
# WRONG - prevents Ctrl+C from stopping your program
try:
    while True:
        do_work()
except BaseException:
    pass

# CORRECT - allows KeyboardInterrupt to propagate
try:
    while True:
        do_work()
except Exception:
    handle_error()
```

File Encodings Explained

The PDF covers file operations but doesn't address character encoding, which causes many real-world bugs.

What is Encoding?

Computers store text as numbers. An encoding is the mapping between characters and numbers. Different encodings use different mappings.

Common Encodings

Encoding	Description	When to Use
UTF-8	Universal, variable-width, ASCII-compatible	Default choice for most applications
ASCII	Original 7-bit encoding, English only	Legacy systems, simple text
Latin-1 (ISO-8859-1)	Western European characters	Older web pages, some Windows files
UTF-16	Windows native, uses 2+ bytes per character	Windows APIs, some legacy systems

Always Specify Encoding

```
# Risky - uses system default encoding
with open("data.txt", "r") as f:
    content = f.read()

# Safe - explicit UTF-8
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()
```

Handling Encoding Errors

```
# Strict (default) - raises error on invalid characters
with open("data.txt", "r", encoding="utf-8", errors="strict") as f:
    content = f.read()

# Replace - substitutes invalid characters with ?
with open("data.txt", "r", encoding="utf-8", errors="replace") as f:
    content = f.read()

# Ignore - silently skips invalid characters
with open("data.txt", "r", encoding="utf-8", errors="ignore") as f:
    content = f.read()
```

Detecting Unknown Encodings

When you don't know a file's encoding:

```
def read_with_fallback(filepath):
    encodings = ["utf-8", "latin-1", "cp1252"]

    for encoding in encodings:
        try:
            with open(filepath, "r", encoding=encoding) as f:
                return f.read()
        except UnicodeDecodeError:
            continue

    raise ValueError(f"Could not decode {filepath}")
```

Understanding Context Managers Beyond Files

The PDF introduces the `with` statement for files, but this pattern is far more powerful and widely applicable.

What Context Managers Do

A context manager guarantees that setup and cleanup code runs, even if errors occur. The pattern is:

1. **Enter**: Acquire resource or set up state
2. **Execute**: Run your code block
3. **Exit**: Clean up, always runs (even on exceptions)

Common Built-in Context Managers

```
# Database connections
import sqlite3
with sqlite3.connect("database.db") as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    # Connection automatically closed

# Thread locks
import threading
lock = threading.Lock()
with lock:
    # Only one thread can execute this at a time
    shared_resource.modify()

# Temporary directory changes
import os
from contextlib import chdir
with chdir("/tmp"):
    # Working directory is /tmp here
    pass
# Back to original directory
```

Creating Your Own Context Manager

Using a class:

```
class Timer:
```

```
    def __enter__(self):
        import time
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        import time
        self.elapsed = time.time() - self.start
        print(f"Elapsed: {self.elapsed:.2f} seconds")
        return False  # Don't suppress exceptions

with Timer():
    # Code to time
    sum(range(1000000))
```

Using the decorator approach (simpler):

```
from contextlib import contextmanager

@contextmanager
def timer():
    import time
    start = time.time()
    yield  # Code in 'with' block runs here
    elapsed = time.time() - start
    print(f"Elapsed: {elapsed:.2f} seconds")

with timer():
    sum(range(1000000))
```

Import System Deep Dive

The PDF covers basic imports but doesn't explain how Python finds modules or the different import styles.

How Python Finds Modules

Python searches these locations in order:

1. The directory containing the input script
2. `PYTHONPATH` environment variable (if set)
3. Installation-dependent default paths (site-packages)

You can inspect the search path:

```
import sys
```

```
for path in sys.path:
    print(path)
```

Import Styles and When to Use Them

```
# Import entire module - clearest, best for readability
import json
data = json.loads(text)

# Import specific items - good for frequently used items
from datetime import datetime, timedelta
now = datetime.now()

# Import with alias - useful for long names or conventions
import numpy as np
import pandas as pd

# Import all (avoid this!) - pollutes namespace, hides dependencies
from math import *  # Which functions came from math?
```

Relative vs Absolute Imports

In packages (directories with `init.py`):

```
# Absolute import - always works, explicit
from mypackage.utils import helper

# Relative import - only in packages, shows relationship
from .utils import helper      # Same directory
from ..other import something  # Parent directory
```

The `init.py` File

This file makes a directory a Python package. It can be empty or define what's exported:

```
# mypackage/__init__.py

# Make these available when someone imports mypackage
from .core import main_function
from .utils import helper_function

# Define what 'from mypackage import *' exports
__all__ = ["main_function", "helper_function"]
```

Docstrings: Documenting Your Code

The PDF emphasizes descriptive function names but doesn't cover documentation strings, which are essential for maintainable code.

What is a Docstring?

A docstring is a string literal that appears as the first statement in a function, class, or module. Python stores it in the `__doc__` attribute.

Basic Format

```
def calculate_area(length, width):
    """Calculate the area of a rectangle.

    Args:
        length: The length of the rectangle (positive number).
        width: The width of the rectangle (positive number).

    Returns:
        The area as a float.

    Raises:
        ValueError: If length or width is not positive.
    """
    if length <= 0 or width <= 0:
        raise ValueError("Dimensions must be positive")
    return float(length * width)
```

Accessing Docstrings

```
# In code
print(calculate_area.__doc__)

# In interactive Python
help(calculate_area)
```

Popular Docstring Styles

Google Style (shown above) - readable, popular in open source

NumPy Style - common in scientific Python

```
def calculate_area(length, width):
```

```
    """
    Calculate the area of a rectangle.

    Parameters
    ----------
    length : float
        The length of the rectangle.
    width : float
        The width of the rectangle.

    Returns
    -------
    float
        The calculated area.
    """
    return float(length * width)
```

Practical Exception Handling Patterns

The PDF shows try/except basics. Here are real-world patterns you'll use frequently.

Pattern 1: Retry Logic

```
import time

def fetch_with_retry(url, max_attempts=3, delay=1):
    """Attempt an operation multiple times before giving up."""
    for attempt in range(max_attempts):
        try:
            return fetch_url(url)
        except ConnectionError:
            if attempt < max_attempts - 1:
                time.sleep(delay)
            else:
                raise
```

Pattern 2: Exception Chaining

Preserve the original error while adding context:

```
def load_config(filepath):
    try:
        with open(filepath, "r") as f:
            return parse_config(f.read())
    except FileNotFoundError as e:
        raise ConfigError(f"Config file missing: {filepath}") from e
```

```
    except ValueError as e:
        raise ConfigError(f"Invalid config format in {filepath}") from e
```

Pattern 3: Logging Exceptions

```
import logging

logger = logging.getLogger(__name__)

def process_data(data):
    try:
        return transform(data)
    except Exception:
        logger.exception("Failed to process data")  # Logs full traceback
        raise  # Re-raises the original exception
```

Pattern 4: Cleanup with Exception Info

```
def process_file(filepath):
    file = None
    try:
        file = open(filepath, "r")
        return process(file.read())
    except IOError as e:
        print(f"IO Error: {e}")
        return None
    finally:
        if file is not None:
            file.close()
        print("Cleanup complete")
```

Summary

Understanding these foundational concepts will help you:

- Write functions that behave predictably (scope, mutable defaults)
- Handle errors more precisely (exception hierarchy)
- Work with text files reliably (encoding)
- Apply the `with` pattern beyond files (context managers)
- Structure larger projects properly (import system)
- Document code for future maintainability (docstrings)
- Handle real-world error scenarios (exception patterns)

These concepts bridge the gap between writing code that works and writing code that is robust, maintainable, and professional.