Python Course

Hermann Himmelbauer

Himmelbauer IT

Wien, 2024-09-01

Effective Python Software Development

Welcome

Welcome to Effective Software Development with Python

Topics Covered

- Basic software development recommendations
- Coding style, version control, virtual environment
- Software testing, logging, program control
- argument parsing, configuration
- Logging and profiling
- Python packaging and distribution

Introduction

Who is Who

- **Hermann Himmelbauer**
  - Studied Computer Technology at TU-Wien
  - Long-time software developer
  - Started with Python around 1994, loved it right away
- **Short Introduction of Class**
  - Who you are
  - Profession
  - Programming experience
  - Expectations

The course will be held in English.

Introduction - Timetable

Key Points:

- Relatively tight schedule
- Basic info, then hands-on examples
- Breaks between blocks
- Please inform me if you need a break

Time	**Duration**	**Block**
13:30	15 min	Introduction
13:45	105 min	Development Best Practice
15:30	*15 min*	*Break*
15:45	105 min	Testing, Debugging, Profiling
18:00	*60 min*	*Dinner*
19:00	90 min	Hands-On Example
20:30	-	*End*

Introduction - Seminar Concept

Seminar Objectives

- Evolve your Software development skills for bigger projects
  - From simple Jupyter Notebooks to software projects
  - Better collaboration
  - Effective code management
- Provide lots of information - not everything needs to be memorized
- Mastering effective software development takes longer than a day
- Plant ideas and entry points
- Slides and code snippets provided on our website

Participation Tips

Participation Tips

- Please ask questions
- Code snippets will be presented; feel free to copy/paste
- Download all code from here:
  https://helios.himmelbauer-it.at/distrib/solid4fun/
- We will use the code in the PyCharm editor during the course
- If you have problems with your IDE or running code, please ask

Software Development with Python

Basic Software Development Process

Software Development Lifecycle

Classical Waterfall-Model

- **Requirements Gathering**: Understanding the problem and defining what needs to be built.
- **Design**: Planning the architecture and components of the software, write specification.
- **Implementation**: Writing the code to build the software. (Not to early!)
- **Testing**: Verifying that the software works as intended.
- **Deployment**: Releasing the software to users.
- **Maintenance**: Updating and fixing the software post-release.

Software Development Lifecycle

In Reality

- **Iterative**: The steps above will be an iterative process of requirements gathering, design, implementation and testing.
- **Prototyping**: One may start with a prototype / proof of concept already in the design phase and advance from there.
- **Testing by Design**: Possibly create parts of the specification as tests which must then pass during the implementation phase.
- **Alpha/Beta Phasing**: Eventually pre-release your software in an alpha version to gather feedback from users.
- **Feedback Loops**: Get feedback from others, refactoring, and testing.

Basic Design Suggestions

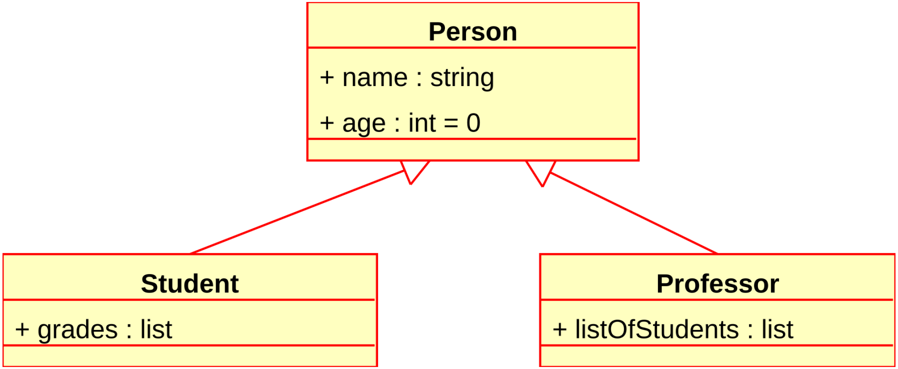
Data Modeling

- Identify key entities and their relationships.
- Use tools like ER diagrams or UML class diagrams to visualize the data structure.
- Define data types, constraints, and primary/foreign keys.

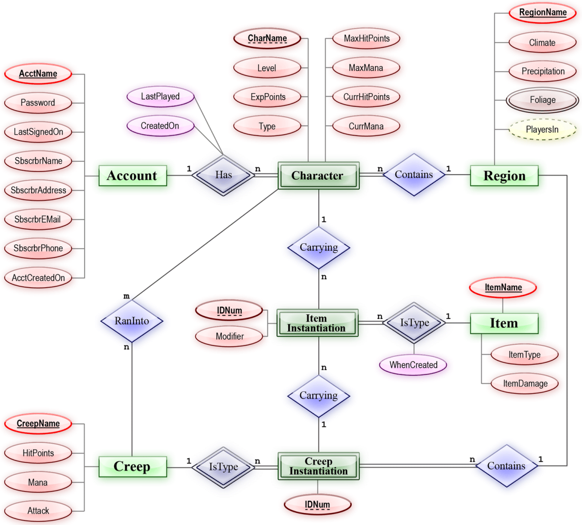
UI Sketching

- Start with low-fidelity sketches to outline basic layouts.
- Focus on user flows and essential components.
- Iterate quickly, gathering feedback to refine designs.
- Example Tools: pen and paper or tools like Balsamiq.

Class Diagram Example



ER Diagram Example



Basic Design Suggestions

Design Best Practices

- Keep designs simple and user-centric.
- Ensure consistency across different parts of the application.
- Consider scalability and future enhancements.

Key Principles of Effective Software Development

Key Principles

- **Modularity**: Break down the software into smaller, manageable pieces.
- **Version Control**: Use tools like Git to track changes and collaborate.
- **Code Quality**: Write readable, maintainable, and efficient code.
- **Testing and Debugging**: Continuously test your code and fix issues early.
- **Documentation**: Keep your code and processes well-documented.

What is Refactoring?

Definition

- Refactoring is the process of restructuring existing code without changing its external behavior.
- It aims to improve the internal structure of the code, making it cleaner, more efficient, and easier to understand.

Why Refactor?

- **Improves Readability:** Makes code easier to read and understand.
- **Enhances Maintainability:** Simplifies the process of updating and extending the code.
- **Reduces Complexity:** Eliminates redundant code and simplifies complex logic.
- **Increases Performance:** Optimizes code for better performance and efficiency.

Software Development with Python

Coding Style Recommendations

PEP 8 - Python Style Guide (1/2)

PEP 8 - Key Guidelines

- **Indentation:** Use 4 spaces per indentation level, no tabs.
- **Line Length:** Limit lines to a maximum of 79 characters.
- **Blank Lines:** Use blank lines to separate functions, classes, and blocks of code inside functions.
- **Imports:** Place imports at the top of the file; group into standard library, third-party, and local imports.
- **Naming Conventions:**
  - `snake_case` for functions and variables.
  - `CamelCase` for classes.
  - UPPERCASE for constants.
- **Whitespace:** Avoid unnecessary whitespace; be consistent with spacing in expressions and statements.

PEP 8 - Python Style Guide (2/2)

PEP 8 - Key Guidelines

- **Comments and Docstrings:**
  - Use # for inline comments, placed at least two spaces after code.
  - Use triple quotes for docstrings in modules, functions, and classes.
- **Code Layout:** Keep function definitions and class declarations separated by two blank lines.
- **Readability:** Prioritize readability and simplicity in code structure and style.
- **Tools:** Tools like “flake8” or “pylint” can be used to check the code style - editors like PyCharm will help, too.

PEP 20 - The Zen of Python

The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.

Beautiful is better than ugly

Do

```python
def calculate_area(radius):
    pi = 3.14159
    return pi * radius ** 2

area = calculate_area(5)
print(f"The area is {area}")
```

Don't

```python
def c(r):
 pi=3.14159
 return pi*r**2
a=c(5)
print('area=',a)
```

Explicit is better than implicit

Do

```python
def multiply_by_two(number):
    return number * 2

result = multiply_by_two(5)
print(result)
```

Don't

```python
result = 5 * 2
print(result)
```

Simple is better than complex

Do

```
def is_even(number):
    return number % 2 == 0
```

Don't

```
def is_even(number):
    return True if number % 2 == 0 else False
```

Complex is better than complicated

Do

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Don't

```
def fib(n):
    a = 0
    b = 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

Flat is better than nested

Do

```python
def process_data(data):
    if data:
        process(data)
    else:
        print("No data to process.")
```

Don't

```python
def process_data(data):
    if data:
        if len(data) > 0:
            process(data)
        else:
            print("Data is empty.")
    else:
        print("No data to process.")
```

Sparse is better than dense

Do

```
def add(a, b):
    return a + b

x = add(1, 2)
y = add(3, 4)
print(x, y)
```

Don't

```
def add(a,b):return a+b;x=add(1,2);y=add(3,4);print(x,y)
```

Readability counts (1/2)

Do

```
def find_max(numbers):
    """Returns the maximum number from a list of numbers."""
    if not numbers:
        return None
    max_number = numbers[0]
    for number in numbers:
        if number > max_number:
            max_number = number
    return max_number
```

Readability counts (2/2)

Don't

```
def fm(l):
    if not l: return None
    m = l[0]
    for n in l:
        if n > m:
            m = n
    return m
```

Software Development with Python

Code Documenting

Why Document Your Code?

Benefits of Documentation

- Enhances code readability and maintainability.
- Facilitates collaboration and onboarding.
- Helps future-proof your code for yourself and others.
- Essential for open-source and academic projects.

Type Annotations

What are Type Annotations?

- Syntax to specify expected data types of variables, function parameters, and return values.
- Improves code clarity and helps with debugging.

Example

```
def add_numbers(x: int, y: int) -> int:
    return x + y
```

Docstrings (1/2)

What are Docstrings?

- Strings used to document modules, classes, functions, and methods.
- They describe the purpose, parameters, and return values.

Docstrings (2/2)

Example

```
def calculate_area(radius: float) -> float:
    """
    Calculate the area of a circle given its radius.

    Parameters:
    radius (float): The radius of the circle.

    Returns:
    float: The area of the circle.
    """
    pi = 3.14159
    return pi * radius ** 2
```

Commenting Best Practices

When to Comment

- Use comments to explain complex logic or important decisions.
- Avoid obvious comments; focus on the why, not the what.
- Keep comments up-to-date with code changes.

Example

```
# Calculate the area of a circle
area = calculate_area(5)

# Avoid using 'magic numbers', use constants
MAX_ITERATIONS = 100  # Maximum number of iterations allowed
```

Combining Annotations, Docstrings, and Comments (1/2)

Best Practices

- Use type annotations for all function signatures.
- Provide detailed docstrings for all public functions, classes, and modules.
- Use comments sparingly to clarify complex parts of the code.

Combining Annotations, Docstrings, and Comments (2/2)

Example

```python
def fetch_data(url: str, timeout: int = 10) -> dict:
    """
    Fetch data from a given URL.

    Parameters:
    url (str): The URL to fetch data from.
    timeout (int, optional): The timeout for the request in seconds.
         Default is 10.

    Returns:
    dict: A dictionary containing the fetched data.
    """
    # Attempt to get data, handle potential network issues
    response = requests.get(url, timeout=timeout)
    return response.json()
```

Software Development with Python

Version Control

Introduction to Version Control

What is Version Control?

- A system for managing changes to files and projects over time.
- Keeps a record of modifications, additions, and deletions.
- Allows multiple people to work on the same project simultaneously.
- Enables rollback to previous versions if needed.

Why Use Version Control?

- Track history and changes in your code.
- Collaborate with others without conflicts.
- Experiment with new features without affecting the main codebase.

Git Basics

What is Git?

- A distributed version control system.
- Keeps track of changes in source code during software development.
- Allows for branching, merging, and collaborative workflows.
- Popular platforms: GitHub, GitLab, Bitbucket.

Key Concepts

- **Repository:** A project's folder tracked by Git.
- **Commit:** A snapshot of changes made to the code.
- **Branch:** A separate line of development.
- **Merge:** Combining changes from different branches.

Common Git Commands

Common Git Commands

- `git init` - Initialize a new Git repository.
- `git clone` - Copy an existing repository to your local machine.
- `git status` - Check the status of your files in the working directory.
- `git add` - Stage changes for the next commit.
- `git commit` - Save staged changes with a message.
- `git pull` - Fetch and merge changes from a remote repository.
- `git push` - Upload local commits to a remote repository.
- `git branch` - List, create, or delete branches.
- `git merge` - Combine changes from different branches.

Version Control Workflow with Git

Basic Git Workflow

1. **Clone** a repository: `git clone <url>`
2. **Create a new branch** for your work: `git checkout -b <branch-name>`
3. **Make changes** and **stage** them: `git add <file>`
4. **Commit** your changes with a message: `git commit -m "Description"`
5. **Push** your changes to the remote repository: `git push`
6. **Merge** changes into the main branch after review.

Tips

- Commit often with clear, concise messages.
- Keep your branches short-lived and focused on specific tasks.
- Regularly pull changes from the main branch to avoid conflicts.

Benefits of Using Git

Benefits of Using Git

- **Collaboration:** Work simultaneously with others on the same project.
- **Code History:** Access previous versions and understand the history of changes.
- **Branching and Merging:** Develop features in isolation and integrate them easily.
- **Backup:** Your code is backed up in the remote repository.
- **Experimentation:** Try out new ideas without affecting the main codebase.

Quote

"Version Control is essential for efficient and organized software development."

Hands On

Version Control

(See file *03-version_control.sh*)

Software Development with Python

Virtual Environments (virtualenv)

What is virtualenv?

Definition

- `virtualenv` is a tool to create isolated Python environments.
- It allows you to manage dependencies for different projects separately.

Why Use virtualenv?

Benefits

- Avoid version conflicts between projects.
- Simplify dependency management.
- Create reproducible environments, enhancing collaboration and deployment.

Key Concepts of virtualenv

Cornerstones

- **Isolation:** Each virtual environment has its own Python interpreter and libraries.
- **Environment-specific packages:** Packages installed in one environment do not affect others.
- **Easy environment management:** Create, activate, deactivate, and delete environments effortlessly.

Creating a Virtual Environment

Steps to Create

```
# Install virtualenv if not already installed
pip install virtualenv

# Create a new virtual environment
virtualenv myenv

# Activate the virtual environment (Linux/Mac)
source myenv/bin/activate

# Activate the virtual environment (Windows)
myenv\Scripts\activate
```

Using a Virtual Environment

Best Practices

- Always activate your environment before running or developing code.
- Use `requirements.txt` to document dependencies:

```
pip freeze > requirements.txt
```

- Reproduce environments easily:

```
pip install -r requirements.txt
```

Deactivating and Removing Virtual Environments

Deactivating

```
# Deactivate the current virtual environment
deactivate
```

Removing

- Simply delete the folder containing the virtual environment.
- Example:

```
rm -rf myenv
```

Hands On

Virtual Environments

(See file *04-virtualenv.sh*)

Software Development with Python

Automated Testing

Importance of Automated Software Testing

Why Automated Testing Matters

- **Efficiency:** Automated tests run faster than manual tests, saving time and resources.
- **Consistency:** Provides reliable and repeatable results, reducing human error.
- **Early Bug Detection:** Identifies issues early in the development cycle, reducing costs of fixing defects.
- **Scalability:** Easily scales to test large codebases or multiple configurations without additional effort.
- **Improved Code Quality:** Encourages developers to write cleaner, more maintainable code
- **Different Teams:** Tests can be created by different teams, which also improves code quality and may catch errors you did not think of
- **Documentation:** Acts as living documentation that demonstrates how the software is expected to behave.

Why Testing is Crucial When Using LLM Code Snippets

Challenges with LLM-Generated Code

- **Potential Errors:** Code generated by LLMs like ChatGPT may contain syntax errors, bugs, or logical flaws.
- **Security Risks:** LLMs might produce code with security vulnerabilities, such as injection flaws or weak validation.
- **Context Misalignment:** Generated snippets may not fully align with the specific requirements or context of your project.
- **Outdated Practices:** LLMs can suggest outdated or deprecated methods and libraries, leading to compatibility issues.
- **Refactoring:** The LLM may break code during refactoring processes (possibly due to limits of the context window).

Doctests in Docstrings

What are Doctests?

- Doctests are a simple way to test code by embedding test cases in docstrings.
- They serve as both documentation and tests.

Example

```
def add(a, b):
    """
    Add two numbers.

    >>> add(2, 3)
    5
    >>> add(-1, 1)
    0
    """
    return a + b
```

Executing Doctests in Docstrings

How to Run Doctests

- Use the Python command line to run doctests directly from the source file.

Command

```
# Run doctests from a Python file
python -m doctest -v your_script.py
```

Details

- `-m doctest`: Specifies the module for running doctests.
- `-v`: Enables verbose output, showing each test and the result.

Doctests in External Text Files

Why Use External Text Files?

- Separate test cases from code, useful for larger test suites.
- Can provide as documentation for users.

Example of a Test File (`test.txt`)

```
>>> from your_script import add
>>> add(2, 3)
5
>>> add(-1, 1)
0
```

Executing Doctests from External Files

How to Run Doctests from a Text File

- Use the Python command line with the filename of the text file.

Command

```
# Run doctests from an external text file
python -m doctest -v test.txt
```

Benefits

- Allows for easier maintenance and separation of test logic.

Integration Testing (1/2)

What is Integration Testing?

- Tests the interaction between different parts of the codebase.
- Ensures that components work together as expected.

Integration Testing (2/2)

Example Using `unittest`

```python
import unittest
from your_script import add, subtract

class TestMathOperations(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(add(2, 3), 5)

    def test_subtraction(self):
        self.assertEqual(subtract(5, 3), 2)

if __name__ == '__main__':
    unittest.main()
```

Running Integration Tests

How to Execute Integration Tests

- Use the `unittest` module to run the test cases.

Command

```
# Run integration tests using unittest
python -m unittest test_module.py
```

Options

- Can run individual test files or discover and run all tests in a directory.

Best Practices for Testing

Tips for Effective Testing

- **Test-Driven Development:** Write tests as you develop your code - you can even start with the tests and then code until all tests pass.
- **Descriptive Test Names:** Use clear and descriptive names for tests to indicate what they verify.
- **Repeatability and Independence:** Ensure tests are repeatable and independent of each other.
- **Regular Testing:** Regularly run tests to catch issues early.

Hands On

Software Testing

(See example package *ibanlib*)

Software Development with Python

Logging

General Ideas of Logging

Why Use Logging?

- Helps in tracking the flow of the program and diagnosing issues.
- Provides insight into application behavior and performance.
- Useful for debugging and monitoring in development and production.
- More flexible and informative than simple print statements.

Simple Print Statements for Logging

Using Print Statements

- Basic way to output messages to the console.
- Useful for quick debugging but not suitable for production code.

Example

```python
def calculate_total(items):
    print("Calculating total...")
    total = sum(items)
    print(f"Total: {total}")
    return total

calculate_total([10, 20, 30])
```

Logging with the logging Module

Advantages over Print Statements

- Configurable logging levels (DEBUG, INFO, WARNING, ERROR, CRITICAL).
- Control over logging output (console, files, remote servers).
- Enables or disables logging easily without changing code logic.

Basic Setup

```python
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

# Logging an informational message
logging.info("This is an info message.")
```

Setting Options for Enabling/Disabling Logging

Controlling Logging Output

- Use different logging levels to control what messages appear.
- Adjust logging configuration to enable or disable logging as needed.

Example

```
import logging

# Set logging level to WARNING (higher severity)
logging.basicConfig(level=logging.WARNING)

logging.debug("This won't show up.")
logging.info("Neither will this.")
logging.warning("This is a warning message.")
```

Logging to a File

Persisting Logs

- Logs can be saved to files for later analysis.
- Useful for long-running applications and production environments.

Example

```
import logging

# Configure logging to write to a file
logging.basicConfig(filename='app.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)
                        s')

logging.info("Logging to a file.")
logging.error("An error occurred.")
```

Best Practices for Logging

Tips for Effective Logging

- Use appropriate logging levels (DEBUG for development, WARNING/ERROR for production).
- Format logs to include timestamps, log levels, and contextual information.
- Avoid logging sensitive information like passwords or personal data.
- Regularly review and manage log files to avoid excessive storage usage.

Software Development with Python

Controlling your Program

Controlling Python Programs: Changing Program Code

Changing Program Code

- Directly modifying global variables or constants within the script.
- Common for quick changes or when the program has a limited scope.

Pros and Cons

Pros:

- Simple and straightforward.
- No additional setup required.

Cons:

- Not fit in case non-programmers should use your program
- Not scalable for complex or frequently changing settings.
- Changes require modifying and re-deploying code.
- Increases risk of introducing bugs.

Controlling Python Programs: Command Line Options

Specifying Options on the Command Line

- Use modules like `argparse` to specify options and arguments.
- Allows users to control program behavior without changing code.

Pros and Cons

Pros:

- Flexibility to change behavior at runtime.
- Easy to document and provide help messages.
- Supports both mandatory and optional parameters.

Cons:

- Command line length can be a limitation for complex configurations.
- Requires users to understand the command-line interface.

Controlling Python Programs: Configuration Files

Writing Configuration Files

- Use configuration files (e.g., INI, YAML, JSON) to manage settings.
- Python libraries like `configparser`, `PyYAML`, and `json` can be used to read these files.

Pros and Cons

Pros:

- Centralizes configuration management.
- Easy to change without modifying code.
- Suitable for complex settings with many parameters.
- Configuration frontends can be created for simple management

Cons:

- Requires parsing and validation logic in the program.

Summary: Controlling Python Programs

Choosing the Right Approach

- **Changing Program Code:** Best for quick changes or prototyping.
- **Command Line Options:** Ideal for flexible, user-driven control.
- **Configuration Files:** Best for managing complex or numerous settings.

Considerations

- Evaluate the complexity and frequency of changes.
- Consider user expertise and accessibility needs.
- Balance between flexibility, maintainability, and ease of use.

Software Development with Python

Argument Parsing

Why Use Argument Parsing?

Benefits of Argument Parsing

- **Improves Flexibility:** Allows users to provide input dynamically without modifying the code.
- **Avoids Global Variables:** Reduces reliance on global variables and hard-coded values in scripts.
- **Easier Configuration:** Simplifies script configuration for different environments and use cases.
- **Enhances Reusability:** Makes scripts more reusable by externalizing inputs and parameters.
- **Automatic Documentation:** Provides built-in help and usage messages, making scripts more user-friendly.

Introduction to Argparse

What is Argparse?

- A Python module for parsing command-line arguments.
- Enables adding, parsing, and validating arguments easily.
- Automatically generates help and usage messages.
- Supports various argument types and default values.

How to Use Argparse

Steps to Implement Argparse

1. **Create a Parser:** Initialize with `argparse.ArgumentParser()`.
2. **Add Arguments:** Use `add_argument()` to define expected inputs.
3. **Parse Arguments:** Call `parse_args()` to process the inputs.
4. **Use Parsed Data:** Access the parsed arguments via the returned object.

Common Argument Types

- `positional arguments`: Required inputs like filenames or values.
- `optional arguments`: Optional flags like `-h` or `--verbose`.
- `type`: Specifies the data type (e.g., `int`, `float`, `str`).
- `default`: Provides a default value if the argument is not supplied.

Using argparse for Command Line Arguments

Simple argparse Example

```python
import argparse

def main():
    parser = argparse.ArgumentParser(description="A simple argparse
        example")
    parser.add_argument('--name', type=str, help="Your name")
    parser.add_argument('--age', type=int, help="Your age")

    args = parser.parse_args()
    print(f"Hello, {args.name}! You are {args.age} years old.")

if __name__ == "__main__":
    main()
```

Argparse --help Output

--help Output

```
$ python script.py --help
usage: script.py [-h] [--name NAME] [--age AGE]

A simple argparse example

optional arguments:
  -h, --help      show this help message and exit
  --name NAME     Your name
  --age AGE       Your age
```

Usage Examples for `argparse`

Usage Examples

```
# Provide name and age arguments
$ python script.py --name Alice --age 30
Hello, Alice! You are 30 years old.

# Missing arguments
$ python script.py --name Bob
usage: script.py [-h] [--name NAME] [--age AGE]
script.py: error: argument --age is required
```

Benefits of Using Argparse

Why Choose Argparse?

- **Ease of Use:** Simplifies handling command-line arguments.
- **Built-In Help:** Automatically generates help and usage messages.
- **Error Handling:** Provides user-friendly error messages for invalid inputs.
- **Flexibility:** Supports complex argument parsing needs, including subcommands.

Software Development with Python

Configuration

Configuration Files in Python

What Are Configuration Files?

- Configuration files are external files that define settings and options for a program.
- They allow changing program behavior without modifying the code.

Common Formats

- **INI files**: Simple, key-value pairs grouped by sections.
- **YAML files**: Human-readable, supports complex data structures.
- **JSON files**: Lightweight, commonly used for data exchange.
- **TOML files**: Simple, similar to INI but with better data type support.

Using INI Files 1/2

INI Files with configparser

- INI files use sections, keys, and values.
- Python's configparser module reads and writes INI files.

Example: INI File (config.ini)

```
[database]
host = localhost
port = 3306
user = admin
password = secret
```

Using INI Files 2/2

Example: INI File (config.ini)

```
[database]
host = localhost
port = 3306
user = admin
password = secret
```

Example: Reading INI File

```
import configparser

config = configparser.ConfigParser()
config.read('config.ini')

host = config['database']['host']
port = config['database'].getint('port')
```

Using YAML Files 1/2

YAML Files with `PyYAML`

- YAML files are easy to read and support nested data structures.
- Python's `PyYAML` library is used for parsing YAML files.

Example: YAML File (`config.yaml`)

```
database:
  host: localhost
  port: 3306
  user: admin
  password: secret
```

Using YAML Files 2/2

Example: YAML File (`config.yaml`)

```
database:
  host: localhost
  port: 3306
  user: admin
  password: secret
```

Example: Reading YAML File

```
import yaml

with open('config.yaml', 'r') as file:
    config = yaml.safe_load(file)

host = config['database']['host']
port = config['database']['port']
```

Using JSON Files 1/2

JSON Files with `json`

- JSON files are widely used for data interchange and support nested data.
- Python's built-in `json` module handles JSON files.

Example: JSON File (`config.json`)

```
{
  "database": {
    "host": "localhost",
    "port": 3306,
    "user": "admin",
    "password": "secret"
  }
}
```

Using JSON Files 2/2

Example: Reading JSON File

```
import json

with open('config.json', 'r') as file:
    config = json.load(file)

host = config['database']['host']
port = config['database']['port']
```

Using TOML Files 1/2

TOML Files with toml

- TOML is a configuration format similar to INI but supports richer data types.
- Python's toml module can read and write TOML files.

Example: TOML File (config.toml)

```
[database]
host = "localhost"
port = 3306
user = "admin"
password = "secret"
```

Using TOML Files 2/2

Example: TOML File (`config.toml`)

```
[database]
host = "localhost"
port = 3306
user = "admin"
password = "secret"
```

Example: Reading TOML File

```
import toml

config = toml.load('config.toml')

host = config['database']['host']
port = config['database']['port']
```

Configuration File Formats Comparison

Comparison of Configuration File Formats

Feature	INI	TOML	YAML	JSON
Human-Readable	Yes	Yes	Yes	Moderate
Supports Comments	Yes	Yes	Yes	No
Data Types	Basic	Rich	Rich	Basic
Hierarchical Structure	Limited	Good	Excellent	Good
Ease of Use	Easy	Easy	Moderate	Easy
Parsing Library	`configparser`	`toml`	`PyYAML`	`json`
Supported by Python	Built-in	External	External	Built-in
File Size	Small	Small	Medium	Small
Standardization	No	Yes	No	Yes

Software Development with Python

Debugging

Why is Debugging Important?

Purpose of Debugging

- Identify and fix errors or unexpected behavior in code.
- Improve the reliability and performance of your software.
- Essential for understanding code flow and logic.

Best Practices for Debugging

General Tips

- Reproduce the issue consistently.
- Simplify the problem: isolate the code causing the bug.
- Use version control (e.g., Git) to track changes and identify when bugs were introduced.
- Write tests to catch errors early and prevent regressions.

Using Print Statements for Debugging

Quick and Simple

- Use print statements to inspect variables and program flow.
- Useful for simple or quick debugging, but can clutter the code.

Example

```python
def calculate_area(radius):
    print(f"Debug: radius = {radius}")
    pi = 3.14159
    area = pi * radius ** 2
    print(f"Debug: area = {area}")
    return area

calculate_area(5)
```

Using Logging for Debugging (1/2)

Logging Instead of Prints

- Use the logging module for more control over output.
- Log at different levels (DEBUG, INFO, WARNING, ERROR) to categorize messages.
- You can specify the loglevel in your code, but also on the commandline like "–debug", moreover, you can put the loglevel in your configuration file.

Using Logging for Debugging (2/2)

Example

```python
import logging

logging.basicConfig(level=logging.DEBUG)

def calculate_total(items):
    logging.debug(f"Items: {items}")
    total = sum(items)
    logging.info(f"Total: {total}")
    return total

calculate_total([10, 20, 30])
```

Using the Built-in Debugger (pdb)

What is pdb?

- Python's built-in debugger that allows step-by-step execution.
- Inspect variables, set breakpoints, and navigate through code.

Basic Commands

- `l`: List code.
- `b`: Set a breakpoint.
- `c`: Continue execution until the next breakpoint.
- `s`: Step into.
- `n`: Execute the next line of code.
- `q`: Quit the debugger.
- All Python code can be executed here, too, e.g., `print(a)`

Using pdb Example

Example of Using pdb

```
import pdb

def divide(a, b):
    pdb.set_trace()  # Start debugger here
    return a / b

divide(10, 2)
```

Run the Example

```
# Execute the script normally
python your_script.py
```

Using IDE Debugging Tools

Integrated Debugging

- Most modern IDEs (e.g., PyCharm, VSCode) have built-in debugging tools.
- Provides a graphical interface for setting breakpoints, inspecting variables, and stepping through code.

Advantages

- Easier to visualize program flow.
- More features like variable watches, call stacks, and conditional breakpoints.

Choosing the Right Debugging Approach (1/2)

Debugging Tools vs. Logging/Print Statements

Scenario	**Use Debugging Tools**	**Use Logging/Print**
Interactive Development	IDE Debugger, PDB	Logging for Context
Complex Code Navigation	IDE Debugger, PDB	Not Ideal
Remote Servers	Difficult to Use	Ideal for Tracking Issues
Production Environments	Not Recommended	Logging (Various Levels)
Real-time Monitoring	Not Suitable	Use Logging
Long-Running Processes	Limited Use	Essential for Insights
Quick Checks	Overhead	Print Statements

Choosing the Right Debugging Approach (2/2)

Guidelines for Choosing the Right Tool

- **Debugging Tools:** Best for local development, step-by-step code execution, and complex code navigation.
- **Logging/Print Statements:** Ideal for server-side debugging, monitoring, production use, and environments where debuggers are impractical.
- **Balance:** Use logging with appropriate levels (DEBUG, INFO, ERROR) to capture necessary insights without overwhelming the system.

Handling Exceptions

Using `try` and `except`

- Catch and handle exceptions to prevent crashes and provide useful error messages.
- Use specific exceptions rather than catching all with `except Exception:`.

Example

```python
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        print(f"Error: {e}")
        return None

safe_divide(10, 0)
```

Debugging Best Practices

Key Takeaways

- Start with a clear understanding of the problem.
- A top-to-bottom approach is a good idea: outrule bugs in your code piece by piece
- Use the right tool for the situation: print, logging, pdb, or an IDE.
- Keep your code clean: remove or disable debug statements once fixed.
- Regularly review your debugging process to improve efficiency.

Practice Debugging

Debugging Exercise

- **Objective:** Practice debugging skills using prepared code.
- **Buggy Code File:** `16-faulty_code.py`

Tasks

- Debug and fix the code using:
  - **PDB (Python Debugger):** Use the command-line interface to step through the code.
  - **PyCharm Debugger:** Utilize the graphical debugging tools provided by PyCharm.

Software Development with Python

Profiling

What is Profiling?

Definition

- Profiling is the process of measuring the performance of a program.
- It helps identify bottlenecks and areas of code that consume the most resources (time, memory).

Types of Profiling

- **CPU Profiling:** Analyzes the time spent in each function or line of code.
- **Memory Profiling:** Monitors memory usage and identifies leaks or excessive allocations.

Why Profiling is Necessary

Benefits of Profiling

- Helps optimize performance by pinpointing slow or inefficient code.
- Reduces resource consumption, making applications faster and more efficient.
- Essential for scaling applications and improving user experience.
- Provides data-driven insights rather than relying on assumptions.

Simple Profiling with `timeit`

Using `timeit` for Quick Timing

- The `timeit` module measures the execution time of small code snippets.
- Useful for comparing different implementations of the same functionality.

Example

```python
import timeit

# Measure the time to execute a statement
execution_time = timeit.timeit('sum(range(1000))', number=1000)
print(f"Execution time: {execution_time:.4f} seconds")
```

Profiling with `cProfile`

Using `cProfile` for More Detailed Analysis

- `cProfile` provides a detailed report on the time spent in each function.
- Useful for profiling larger applications and understanding function-level performance.

Example

```
import cProfile

def expensive_function():
    result = sum([i ** 2 for i in range(10000)])
    return result

cProfile.run('expensive_function()')
```

Advanced Profiling with `line_profiler`

Line-by-Line Profiling

- `line_profiler` allows line-by-line analysis of code execution time.
- Requires installation via `pip install line_profiler`.

Advanced Profiling with `line_profiler`

Example

```python
# Decorate functions with @profile for line profiling
@profile
def calculate():
    total = 0
    for i in range(10000):
        total += i ** 2
    return total

calculate()
```

Run with

```
kernprof -l -v your_script.py
```

Using Profiling Results to Improve Performance

Interpreting Results

- Focus on functions with the highest cumulative time.
- Look for inefficient algorithms, unnecessary computations, or slow I/O operations.
- Check for excessive memory usage and optimize data structures.

Improvement Strategies

- Optimize algorithms or use more efficient libraries.
- Reduce the complexity of loops and recursive calls.
- Cache results of expensive calculations (memoization).
- Parallelize or asynchronously run independent tasks.

Best Practices for Profiling

Key Takeaways

- Profile regularly during development, not just at the end.
- Use profiling tools that match the scale and complexity of your project.
- Avoid premature optimization; profile first to identify actual bottlenecks.
- Validate improvements with before-and-after profiling comparisons.
- Automated testing will prove helpful here too!

Software Development with Python

Packaging Python Modules

What is a Python Package?

Definition

- A Python package is a collection of modules organized in a directory structure that includes an `__init__.py` file.
- Packages allow you to organize your code into reusable and distributable components.

Why Create a Package?

- Facilitates code reuse and modularity.
- Simplifies distribution and installation.
- Helps in versioning and dependency management.

Necessary Files for a Python Package

Key Files

- `setup.py`: Configuration file for building and distributing the package.
- `__init__.py`: Indicates that the directory is a Python package.
- `README.md`: Provides an overview of the package, how to install and use it.
- `LICENSE`: Specifies the licensing terms of the package.
- `pyproject.toml`: (Optional) Modern configuration file for build systems.
- `MANIFEST.in`: (Optional) Specifies additional files to include in the package.

Directory Structure of a Python Package

Typical Directory Layout

```
mypackage/
|
+-- mypackage/          # Package directory
|   +-- __init__.py     # Makes this a package
|   +-- module1.py      # Module file
|   +-- module2.py      # Another module
|
+-- tests/              # Directory for test files
|   +-- __init__.py
|   +-- test_module1.py
|
+-- setup.py            # Setup script
+-- README.md           # Project description
+-- LICENSE             # License file
+-- pyproject.toml      # Optional modern build system config
```

Contents of `setup.py` (1/2)

Purpose

- `setup.py` is the script used to build, package, and distribute the package.

Contents of `setup.py` (2/2)

Example

```
from setuptools import setup, find_packages

setup(
    name='mypackage',
    version='0.1',
    packages=find_packages(),
    install_requires=[
        'numpy',    # Example dependency
    ],
    author='Your Name',
    author_email='your.email@example.com',
    description='A simple Python package example',
    long_description=open('README.md').read(),
    long_description_content_type='text/markdown',
    url='https://github.com/yourusername/mypackage',
    classifiers=[
        'Programming Language :: Python :: 3',
        'License :: OSI Approved :: MIT License',
    ],
)
```

Contents of `pyproject.toml` (1/2)

Purpose

- `pyproject.toml` is a configuration file for specifying build system requirements.
- Modern alternative to `setup.py` for defining package metadata and dependencies.

Contents of `pyproject.toml` (2/2)

Example

```
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

[tool.setuptools]
packages = ["mypackage"]

[project]
name = "mypackage"
version = "0.1.0"
description = "A simple Python package example"
authors = [
    {name = "Your Name", email = "your.email@example.com"}
]
license = {file = "LICENSE"}
```

Contents of `MANIFEST.in`

Purpose

- Specifies additional files to include in the package distribution.
- Useful for including non-Python files like documentation, data, or configuration files.

Example

```
include README.md
include LICENSE
include mypackage/data/*.csv  # Example of including data files
```

Best Practices for Packaging

Key Takeaways

- Use virtual environments to manage dependencies during development.
- Test your package locally before distribution.
- Include comprehensive documentation in your package.
- Regularly update package dependencies and version information.

Software Development with Python

Code Distribution

Distributing Python Code

Platforms for Distributing Python Code

- **GitHub:** A platform for hosting and sharing code repositories, supporting collaboration and version control.
- **PyPI (Python Package Index):** The official repository for Python packages, used for distributing and installing Python software.

What is GitHub?

GitHub Overview

- A web-based platform that uses Git for version control.
- Facilitates code sharing, collaboration, and project management.
- Supports public and private repositories.
- Integrated tools for issue tracking, code review, and more.

Benefits of GitHub

- **Version Control:** Tracks changes and manages different versions of code.
- **Backup:** Simple cloud backup of your project code.
- **Collaboration:** Enables multiple developers to work on the same project.
- **Open Source Community:** Share your projects and contribute to others.
- **Documentation Hosting:** Supports Markdown for README and project documentation.

What is PyPI?

PyPI Overview

- The Python Package Index (PyPI) is the official repository for Python packages.
- Allows developers to distribute Python software easily.
- Users can install packages using `pip`.

Benefits of PyPI

- **Easy Distribution:** Simplifies the distribution and installation of Python packages.
- **Dependency Management:** Handles package dependencies automatically.
- **Visibility:** Increases the visibility of your packages to the Python community.
- **Versioning:** Supports multiple versions of a package for compatibility.

Choosing the Right Platform

GitHub vs PyPI

Feature	GitHub	PyPI
Version Control	Yes	No
Collaboration	Yes	No
Code Sharing	Yes	No
Package Distribution	No	Yes
Installation with pip	No	Yes
Project Documentation	Yes	Limited

Key Takeaways

- **Use GitHub** for collaboration, version control, and sharing code.
- **Use PyPI** for distributing Python packages to users and the broader Python community.

Software Development with Python

Hands-On Example

Hands-On Example

Summing All Up

- **Data Processing Package:** Some prototyping code is available
- **Version Control:** Put it under version control
- **Virtual Environment:** Create a virtual environment
- **Doctests:** Write some doctests
- **Logging:** Include some logging to a file
- **Scripting:** Make it a script with options
- **Configuration:** Add a simple configuration
- **Packaging:** Make a Python package

Hands On

hands-on-example

(See file *21-hands-on-example.sh*)

Thank You

Thank you for your interest!