best practices for Python projects and Git

Here are the **best practices** for **Python projects and Git** to maintain clean, professional, and efficient development workflows.

1. Naming Conventions in Python

- **Variables & Functions**: Use **snake_case**
  Example: calculate_total_price()
- **Classes**: Use **PascalCase**
  Example: UserProfile
- **Constants**: Use **UPPERCASE_WITH_UNDERSCORES**
  Example: MAX_CONNECTIONS = 100
- **Private Variables (for internal use in classes)**: Prefix with **a single underscore _**
  Example: _internal_cache
- **Strongly Private Variables (name mangling in classes)**: Prefix with **double underscores __**
  Example: __secret_key

2. Best Practices for Git

1. **Descriptive Commit Messages**: Keep commit messages concise and informative.
2. **Use Separate Branches**: Create a new branch for each feature or bugfix to maintain clarity and traceability.
3. **Pull Recent Changes**: Always ensure your local repository is up to date before merging or rebasing.
4. **Avoid Rebasing Public History**: If commits have been pushed to a shared repository, avoid rebasing unless you have team consensus.

Branch Naming Conventions

- **For feature development:**
  - feature/123-add-login

  (A feature branch linked to task 123 for adding login functionality.)
- **For bug fixes:**
  - bugfix/456-fix-crash
- **For hotfixes (urgent fixes in production):**
  - hotfix/urgent-login-issue
- **For experimental branches:**
  - experiment/new-cache-strategy

Writing Git Commit Messages

✅ **Recommended commit message format:**

Feat: Add login functionality #123

or

Fix issue with user authentication in login API

✅ **Commit message rules:**

- **Be concise and descriptive**.
- **Use present tense** (e.g., Add feature instead of Added feature).
- **Separate detailed explanations with a blank line** after the first line if necessary.
- **Make small, meaningful commits** rather than one large commit.

Merging Rules

✅ **Before merging:**

- Always **pull the latest changes** from main or develop.
- **Test your changes** in a testing environment before merging.
- **Use Pull Requests (PRs) or Merge Requests (MRs)** for code reviews.

✅ **Merge Strategies:**

- **For completed features:** Use **Squash and Merge** (to keep a clean history).
- **For important changes with full history:** Use **Merge Commit**.

3. Writing Comments & Documentation in Python

Use docstrings for functions and classes:

```
def calculate_total(price: float, tax: float) -> float:
    "Calculate the final price including tax.
    Args:
        price (float): The base price.
        tax (float): The tax percentage.
    Returns:
        float: The final price after applying tax.
    "
    return price + (price * tax)
```

Use inline comments for short explanations:

```
# Stores the default tax rate
DEFAULT_TAX_RATE = 0.1
```

Avoid useless comments!

❌ Bad:

```
x = 10  # x is set to 10
```

✅ Correct:

```
max_retry_count = 5  # Maximum number of retry attempts for reconnecting to the server
```

4. Recommended Project Structure for Python

```
my_project/
|— src/              # Main project source code
|   ├— main.py         # Entry point of the project
|   ├— utils.py        # Helper functions
|   ├— models/         # Data models
|   ├— services/       # Business logic
|— tests/            # Unit tests
|— docs/             # Documentation
|— requirements.txt    # Python dependencies
|— .gitignore          # Git ignore rules
|— README.md           # Project description
|— LICENSE           # License file
```

5. Best Practices for Writing Tests

✅ Use **pytest** or **unittest** for writing tests.

✅ **Use descriptive test names:**

```
def test_calculate_total_with_valid_inputs():
    ...
```

✅ Store tests in the **tests/** directory.

Following these best practices will make your project **clean, structured, and professional**.