Python Fundamentals for Software Engineers

This preparatory course equips experienced software engineers with the essential Python skills needed to be successful in the *Building LLM-Powered Applications with Open-Source Tools* program. Designed for developers with strong backgrounds in languages like C#, Java, or PHP, it focuses on quickly building fluency in Python's syntax, data structures, idioms, and tooling.

Through hands-on exercises, real-world examples, and progressive projects, learners will master scripting, Jupyter notebooks, debugging, and key libraries relevant to AI and LLM workflows. By the end of the course, participants will be confident writing clean, Pythonic code and leveraging the ecosystem's tools for data processing, API integration, and rapid prototyping.

Module 0 — Environment Setup and Tooling

⏱ **Time budget:** 1–2 hours

🎯 Learning goals:

- Install Python 3.x and verify it runs (command-line and/or IDE)
- Manage packages with pip and isolate environments with venv
- Run Python interactively (REPL) and execute simple scripts
- Set up a code editor or IDE (e.g., VS Code) for Python development

---

📚 Required resources:

- *Python Downloads (Official Site)* — **article**, https://www.python.org/downloads/, **5 min**
  Get the latest Python 3.x installer for your OS; official source for Windows/macOS/Linux distributions.

- *Google's Python Setup Guide* — **article**, https://developers.google.com/edu/python/, **10 min**
  Walks through installation across OSes and running a first `hello.py` both in terminal and interactively.

- *Installing Packages with pip & virtual environments* — **guide**, https://packaging.python.org/tutorials/installing-packages/, **15 min**
  Official guide to pip usage and virtual environments with `python -m venv`,

including activation steps.

- *Getting Started with Python in VS Code* — **tutorial**, https://code.visualstudio.com/docs/python/python-tutorial, **10 min**
  Configure VS Code for Python: extension install, interpreter selection, running/debugging code.

- *Jupyter Notebook Beginner's Tutorial* — **article**, https://www.dataquest.io/blog/jupyter-notebook-tutorial/, **15 min**
  Install and launch Jupyter, create notebooks, run cells, and add Markdown for documentation.

💡 Optional enrichment:

- *Conda Package Manager (Documentation)* — **article**, https://docs.conda.io/projects/conda/en/latest/user-guide/index.html, **N/A**
  Alternative environment/package manager favored in data science; create and manage isolated envs.

📝 Assignment:

1. Install Python 3 and verify via `python --version` (or `python3 --version`).
2. Create and activate a venv; install a test package (e.g., `requests`).
3. Write `hello.py` printing a greeting + CLI args.
4. Run and debug it in VS Code.
5. Launch Jupyter and run a "Hello Jupyter" cell.
6. Bonus: try a few expressions/variables in the notebook.

🤝 How to use the weekly sync:

"Setup show-and-tell": compare environments (OS/IDE), help fix issues, and run each other's `hello.py` and first notebooks.

Module 1 — Python Syntax and Basics

⏱ **Time budget:** ~5 hours

🎯 Learning goals:

- Understand Python syntax (indentation, no semicolons) and case sensitivity
- Use numbers, strings, booleans; perform arithmetic and string ops
- Write control flow with `if/elif/else`, `for/while`
  Define/call functions with arguments and return values

📚 Required resources:

- *Python Tutorial: An Informal Introduction to Python* — **docs**, https://docs.python.org/3/tutorial/introduction.html, **20 min**
  Official intro covering interpreter usage, numbers/strings (incl. slicing), printing, and basic control flow.

- *Google Python Class – Introduction* — **article**, https://developers.google.com/edu/python/introduction, **15 min**
  Beginner-friendly overview highlighting indentation, dynamic typing, comments, and simple branching/loops.

- *LearnPython.org Interactive Tutorial (Basics)* — **interactive course**, https://www.learnpython.org/, **30+ min (self-paced)**
  In-browser practice on variables, data types, string formatting, lists, operators, conditions, and loops.

- *Official Python FAQ: Programming (Beginner)* — **reference**, https://docs.python.org/3/faq/programming.html, **10 min**
  Common Q&A on braces vs indentation, lack of ++, truthiness, and typical beginner pitfalls.

💡 Optional enrichment:

- *Moving to Python from Other Languages* — **wiki guide**, https://wiki.python.org/moin/MovingToPython, **5 min**
  Pythonic idioms and mindset shifts for C#/Java/PHP devs.

- *Codecademy: Learn Python (Free Sections)* — **interactive**, https://www.codecademy.com/learn/learn-python, **N/A**
  Extra guided practice with immediate feedback on basics.

📝 Assignment:

1. REPL experiments (dynamic typing, string concatenation/repetition, `/` vs `//`, `len()` on str/list).
2. Create `basics.py` with variables, `if/elif/else`, `for` and `while`.
3. Implement FizzBuzz 1–20.
4. Intentionally cause/fix an indentation or syntax error.

🤝 How to use the weekly sync:

Compare script solutions and REPL snippets; refactor non-Pythonic constructs (index-based loops) to idiomatic patterns (`for name in names:`). Clarify truthiness and common gotchas.

Module 2 — Data Structures and Collections

⏱ **Time budget:** ~5 hours

🎯 Learning goals:

- Master lists, tuples, sets, dictionaries
- Use sequence ops (indexing, slicing, `in`, `len`)
- Understand ordered vs unordered; mutable vs immutable
- Learn common methods per type and choose structures appropriately
- Create and use list/dict/set comprehensions to transform and filter collections

---

📚 Required resources:

- *Python Tutorial: Data Structures* — **docs**, https://docs.python.org/3/tutorial/datastructures.html, **30 min**
  Lists (methods/slicing), tuples (packing/unpacking), sets (ops like union/intersection), dicts; intro to comprehensions.

- *Google Python Lists* — **tutorial**, https://developers.google.com/edu/python/lists, **15 min**
  List literals, indexing, slicing, concatenation, mutability, and a taste of comprehensions.

- *Google Python Dicts and Files (Dict section)* — **tutorial**, https://developers.google.com/edu/python/dicts, **20 min**

Dict syntax, `d[k]` vs `d.get`, membership, iterating with `.items()`, insertion order (Py 3.7+).

- *LearnPython.org – Data Structures Practice* — **interactive**, https://www.learnpython.org/, **20+ min**
  Practice tasks for list/tuple/set/dict creation and manipulation.

💡 Optional enrichment:

- *For Loop vs List Comprehension (Performance)* — **blog post**, https://switowski.com/blog/for-loop-vs-list-comprehension/, **5 min**
  When comprehensions are faster/cleaner vs when loops aid clarity.

- *Python Collections Module* — **article**, https://realpython.com/python-collections-module/, **8 min**
  Useful specialized containers: `namedtuple`, `defaultdict`, `Counter`, etc.

📝 Assignment:

1. List ops: sort/append/remove/slice, explain effect each time.
2. Build letter-frequency dict via `get`.
3. Use set to dedupe a list; demo union/intersection/difference.
4. Tuple unpacking in a loop for `(name, age)`
5. List comprehension: extract all even numbers from a list.
6. Dict comprehension: map words to their lengths.

🤝 How to use the weekly sync:

Each person demos one structure; share pitfalls (mutating lists while iterating), alternative solutions (e.g., `collections.Counter`), and clarify mutability aliasing (`list2 = list1`).

Module 3 — Functions, Modules, and Packages

⏱ **Time budget:** ~5 hours

🎯 Learning goals:

- Define functions (`def`, defaults, returns) and understand scope

- Use modules: organize into `.py` files, import patterns, module search path
- Understand packages and install third-party libs with pip
- Skim standard library breadth; use a third-party lib (`requests`)
- Write generator functions using `yield` for efficient iteration over large or streaming datasets

---

📚 Required resources:

- *Defining Functions (Python Tutorial §4.7)* — **docs**, https://docs.python.org/3/tutorial/controlflow.html#defining-functions, **15 min**
  Function syntax, defaults, `*args`/`**kwargs`, returns (including `None`), and local scope rules.

- *Modules and Importing (Python Tutorial §6)* — **docs**, https://docs.python.org/3/tutorial/modules.html, **10 min**
  Create/import modules, namespaces, `import` vs `from import`, `if __name__ == "__main__":`.

- *Official Python Standard Library: 'Tour'* — **docs**, https://docs.python.org/3/library/, **15 min (skim)**
  Survey of stdlib modules (e.g., `math`, `datetime`, `os`, `json`) to know what's built-in.

- *Requests Quickstart* — **docs**, https://requests.readthedocs.io/en/latest/user/quickstart/, **10 min**
  Install/use `requests`, send GET, check status, parse JSON via `.json()`.

💡 Optional enrichment:

- *Python Packaging Tutorial* — **guide**, https://packaging.python.org/tutorials/packaging-projects/, **20 min**
  Structure a project as a package; `pyproject.toml`, building, and publishing basics.

- *The Python import System* — **article**, https://realpython.com/python-import/, **8 min**
  Under-the-hood import mechanics: `sys.path`, bytecode caching, module caching.

📝 Assignment:

1. Implement `greet`, `factorial`, and `find_max` in `utility.py` with a `__main__` test block.
2. Import from `use_util.py`.

3. Use `requests` to fetch a simple API (e.g., random joke) and print JSON fields.
4. Explore two stdlib modules you haven't used before.
5. **Generator task:** Write `read_lines(file_path)` that yields one stripped line at a time; print the first 5 lines.

🤝 How to use the weekly sync:

"Teach one thing" round: defaults & mutable defaults, module structuring + `__main__`, favorite stdlib functions, and `requests` tips/troubleshooting.

Module 4 — Object-Oriented Programming in Python

⏱ **Time budget:** ~4 hours

🎯 Learning goals:

- Define classes and instantiate objects
- Use instance vs class attributes; understand `self`
- Apply inheritance, polymorphism, and `super()`
- Implement special ("dunder") methods like `__init__`, `__str__`

📚 Required resources:

- *Classes (Python Tutorial §9.1–9.5)* — **docs**, https://docs.python.org/3/tutorial/classes.html, **30 min**
  Class definitions, methods with `self`, class variables, inheritance, and simple examples.

- *Object-Oriented Programming in Python (Real Python)* — **article**, https://realpython.com/python3-object-oriented-programming/, **20 min**
  Practical OOP walkthrough, including `__str__`, `__eq__`, and duck typing concepts.

- *Inheritance and Composition in Python (Corey Schafer)* — **video**, https://www.youtube.com/watch?v=RSl87lqOXDE, **15–16 min**
  Concrete example of subclassing vs composition and when to use each.

💡 Optional enrichment:

- *Data Classes in Python 3.7+* — **article**, https://realpython.com/python-data-classes/, **12–15 min**
  `@dataclass` to auto-generate `__init__`, `__repr__`, `__eq__` for data-centric classes.

- *The Python Data Model (dunder methods)* — **docs**, https://docs.python.org/3/reference/datamodel.html, **skim**
  How Python protocols work (`__len__`, `__add__`, `__iter__`, etc.) and integrating objects with built-ins

---

📝 Assignment:

Implement `Book` with `title/author/pages` and `__str__`. Add class attr `library`, then shadow it on one instance; inspect behavior. Create `EBook(Book)` adding `file_size`, call `super().__init__`, override `__str__`. Write `show_book_info` to demonstrate duck typing; (bonus) implement `__eq__`/`__add__`.

🤝 How to use the weekly sync:

Whiteboard class layouts & inheritance diagrams. Compare `__str__` designs and the instance-vs-class attribute behavior. Try duck typing examples (objects with a `.write()` method).

Module 5 — Error Handling and File I/O

⏱ **Time budget:** ~4 hours

🎯 Learning goals:

- Use `try/except/else/finally` and raise exceptions
- Apply EAFP vs LBYL appropriately
- Read/write files with `open()` and `with` context manager
- Handle common I/O exceptions cleanly
- Use Python's built-in debugging tools (`pdb`, `breakpoint()`, `%debug`) to inspect and fix issues

---

📚 Required resources:

- *Errors and Exceptions (Python Tutorial §8)* — **docs**, https://docs.python.org/3/tutorial/errors.html, **20 min**
  Exception model, catching specific types, `else`/`finally`, and `raise`.

- *EAFP vs LBYL: Pythonic Style* — **article**, https://realpython.com/python-eafp-vs-lbyl/, **8–10 min**
  When to "just try and handle errors" vs pre-checking conditions; idiomatic patterns.

- *Reading and Writing Files (Python Tutorial §7.2)* — **docs**, https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files, **15 min**
  File modes, reading strategies, writing, text vs binary, and iteration.

- *Why You Should Use 'with' to Open Files* — **article**, https://realpython.com/python-with-statement/, **5–8 min**
  Context managers for safe resource cleanup even on error.

💡 Optional enrichment:

- *Logging in Python (Basic)* — **docs**, https://docs.python.org/3/howto/logging.html, **10–15 min**
  Configure basic logging and log at levels (DEBUG/INFO/WARNING/ERROR).

- *Contextlib and Custom Context Managers* — **article**, https://realpython.com/python-with-statement/#custom-context-managers, **10 min**
  Build custom context managers with `__enter__`/`__exit__` or `contextlib`.

📝 Assignment:

1. Implement `divide(x, y)` handling `ZeroDivisionError` and `TypeError`.
2. Write `wordcount.py` to read a file and count lines/words; use `try/except`, `with`, and add `else/finally`.
3. Implement LBYL vs EAFP for dict lookups; compare.
4. Create `safe_write` demonstrating `finally`-based cleanup.
5. **Debugging task:** Add `breakpoint()` to `wordcount.py` and inspect variables mid-run. If using Jupyter, run `%debug` after an error to explore the traceback.

🤝 How to use the weekly sync:

Share “bug war stories” and compare `divide()` strategies (handle vs re-raise). Live demo that `with` closes files even if inner code errors.

Module 6 — Writing Pythonic and Idiomatic Code

⏱ **Time budget:** ~3 hours

🎯 Learning goals:

- Apply idioms like `enumerate`, `zip`, truthiness, and chaining comparisons
- Follow PEP 8 code style and use linters/formatters (`flake8`, `black`)
- Prefer clarity over over-compression; know when to break from idioms for readability

---

📚 Required resources:

- *Python Best Practices: Writing Clean, Idiomatic Code* — **article**, https://dev.to/pythonic/python-best-practices-writing-clean-idiomatic-code-2761, **10 min**
  Checklist-style examples of better naming, built-in usage, comprehensions, and DRY.

- *The Hitchhiker’s Guide to Python: Code Style* — **guide**, https://docs.python-guide.org/writing/style/, **~20 min (skim/reference)**
  Opinionated style guidance beyond PEP 8: naming, layout, and idioms.

- *PEP 8 – Style Guide for Python Code* — **PEP**, https://peps.python.org/pep-0008/, **15–20 min (reference)**
  Official style rules: indentation, line length, naming, imports, whitespace.

- *Idiomatic Python: Intermediate Pythonic Practices* — **article**, https://realpython.com/idiomatic-python-advanced/, **10–15 min**
  Practical idioms like `enumerate`, dict comprehensions, chained comparisons, `any()`/`all()`.

💡 Optional enrichment:

- *PEP 20 – The Zen of Python* — **PEP**, https://peps.python.org/pep-0020/, **2–5 min**
  19 aphorisms capturing Python's philosophy ("Readability counts").

- *Refactoring to Pythonic Code (Video)* — **video**, https://www.youtube.com/watch?v=OSGv2VnC0go, **~30 min**
  Live refactors: comprehensions, `with`, better naming, removing redundancy.

📝 Assignment:

1. Refactor index-based loops to use `enumerate`.
2. Replace nested loops with `zip` for parallel iteration.
3. Simplify conditionals with truthiness and chained comparisons.
4. Apply `black` or `flake8` to your code and fix issues.

🤝 How to use the weekly sync:

Run a "style clinic": each person brings a snippet; group refactors to Pythonic form; discuss when comprehensions harm readability.

Module 7 — Using Pip and Common Libraries

⏱ **Time budget:** ~4 hours

🎯 Learning goals:

- Install and manage dependencies with pip; capture versions via `requirements.txt`
- Use `requests`, `json`, and explore typing/type checkers (`mypy`)
- Optionally use `dataclasses` and `argparse` for cleaner code/CLIs
- Understand semantic versioning and virtual environments

📚 Required resources:

- *Installing Packages with pip (User Guide)* — **docs**, https://packaging.python.org/en/latest/tutorials/installing-packages/, **10 min**

Installing, upgrading, pinning versions, and using requirements files.

- *Making HTTP Requests in Python* — **article**, https://realpython.com/python-requests/, **15 min**
  GET/POST with params and JSON, sessions, errors (`raise_for_status()`), and patterns.

- *Python Type Hints (Guide)* — **article**, https://codefinity.com/blog/A-Comprehensive-Guide-to-Python-Type-Hints/, **~7–12 min**
  Basics of annotations, `typing` constructs, and static checking with tools like `mypy`.

- *Data Classes in Python* — **article**, https://realpython.com/python-data-classes/, **~10 min**
  Cleaner class definitions for data containers with sensible defaults and comparisons.

💡 Optional enrichment:

- *Argparse Tutorial* — **docs**, https://docs.python.org/3/howto/argparse.html, **15 min**
  Build CLIs with positional/optional args, help text, and validation.

- *Virtual Environments and pipx* — **article**, https://realpython.com/pipx-guide/, **8–10 min**
  Use `pipx` to install CLI tools in isolated environments; manage multiple projects cleanly

---

📝 Assignment:

Generate `requirements.txt` with `pip freeze`. Upgrade a package and verify version. Use `requests` for a parametric GET (e.g., `agify.io`), a JSON POST to `httpbin`, and handle an error via `try/except` or `raise_for_status`. Add type hints to functions and run `mypy`. Convert a simple class to a `@dataclass`.

🤝 How to use the weekly sync:

“Show & tell” a useful library; rehearse dependency updates via `requirements.txt`; share a case where `mypy` caught an issue.

Module 8 — Scripting, Jupyter, and Final Practice

⏱ **Time budget:** ~3 hours

🎯 Learning goals:

- Decide between scripts vs notebooks based on workflow
- Version-control notebooks properly (strip outputs)
- Use Jupyter conveniences (Markdown, magics, shell commands)
- Build a small capstone-style mini-project in both script and notebook

---

📚 Required resources:

- *Python Scripts vs. Jupyter Notebooks: Pros and Cons* — **article**, https://www.learnpython.com/blog/python-scripts-vs-jupyter-notebooks, **8 min**
  Strengths/limitations of notebooks vs linear scripts; choose appropriately for tasks.

- *Turing Way: Version Control for Jupyter Notebooks (nbstripout)* — **guide**, https://book.turingway.net/project-design/sdpw/pd-sdp-sensitive-code.html#nbstripout , **~5–10 min**
  Remove notebook outputs/metadata before commits to keep diffs clean.

- *JupyterLab Git Extension (ReviewNB Blog)* — **article**, https://blog.reviewnb.com/jupyterlab-git-extension/, **~10 min**
  Integrate Git with JupyterLab; rich notebook diffs and collaborative reviews.

💡 Optional enrichment:

- *Top 10 Jupyter Notebook Tips* — **article**, https://towardsdatascience.com/top-10-tips-and-tricks-for-jupyter-notebooks-8e5fbf91d2c5, **~7–10 min**
  Keyboard shortcuts, `%debug`, `%timeit`, and productivity tweaks.

- *Google Colab Guide* — **article/notebook**, https://colab.research.google.com/notebooks/intro.ipynb, **N/A**
  Run notebooks in the cloud; mount Drive, enable GPUs, and share easily.

📝 Assignment:

Build a small program twice—once as a script and once as a notebook. It should read from a file or API, process data, and output results. For the notebook, add Markdown narrative and clear outputs; configure `nbstripout` for version control.

🤝 How to use the weekly sync:

Demo both versions of the mini-project; discuss when notebooks helped exploration vs when a script was cleaner. Share favorite Jupyter shortcuts/tips.