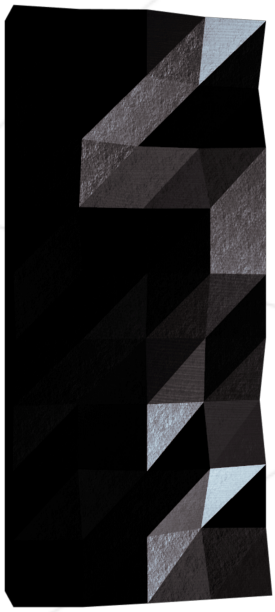

The Codex

Mastering Python's Import Mysteries

Summary: Discover the ancient art of Python imports through alchemical experiments. Master the four sacred mysteries: package initialization, import pathways, absolute vs relative transmutations, and breaking circular dependencies.

Version: 1.0

Contents

I Foreword **2**

II AI Instructions **3**

III Introduction **5**

IV General Instructions **6**
IV.1 Authorized and Forbidden 6
IV.2 Laboratory Structure 7

V Mandatory part **8**
V.1 Part I: The Sacred Scroll 8
V.2 Part II: Import Transmutation 11
V.3 Part III: The Great Pathway Debate 13
V.4 Part IV: Breaking the Circular Curse 15

VI Submission Instructions **17**

Chapter I

Foreword

Welcome to the Alchemist's Laboratory, young apprentice!

You've mastered the basic elements of Python - variables, functions, classes, and data structures. Now it's time to learn the ancient art of code transmutation through Python's import system.

In medieval times, alchemists sought to transform base metals into gold. Today, as a Python alchemist, you'll learn to transform scattered code into organized, reusable magical formulas. Just as ancient alchemists had their grimoires (spell books) organized into chapters and sections, Python code must be organized into modules and packages.

But beware! The path of the alchemist is fraught with mysteries:

- The Sacred Scroll of ___init___.py - the gateway that transforms a simple folder into a magical package
- The Art of Import Transmutation - summoning code from distant modules
- The Great Debate of Pathways - absolute vs relative imports, like choosing between different alchemical formulas
- The Curse of Circular Dependencies - when spells reference each other in an endless loop, threatening to tear apart the fabric of your code!

In this codex, you'll build your own Alchemical Laboratory - a complete Python package that demonstrates these four sacred mysteries. Each experiment will teach you one aspect of Python's import magic.

By the end, you'll understand how to organize code like a master alchemist, avoiding the pitfalls that have trapped many apprentices before you.

Chapter II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Introduction

Welcome to The Alchemist's Codex!

As an apprentice alchemist, you'll discover Python's import system through hands-on magical experiments. Each experiment builds upon the previous one, creating a complete Alchemical Laboratory package that demonstrates professional Python organization.

PREREQUISITES REQUIRED: This activity assumes solid mastery of Python fundamentals including syntax, functions, classes, error handling, lists, dictionaries, and data structures. You should be comfortable writing Python classes, handling exceptions, and working with collections before attempting this import-focused activity. Without these foundations, the import concepts will be difficult to understand.

Focus on understanding **how imports work**, not just making them work. A true alchemist understands every step of the transmutation process!

This activity is about **code organization** and **import mechanisms**. Keep your alchemical formulas simple (basic functions) so you can focus on the import concepts.

ERROR HANDLING: All functions should return strings. When testing imports that might fail (like accessing hidden functions), use try/except blocks and return descriptive error messages instead of letting the program crash.

Chapter IV

General Instructions

IV.1 Authorized and Forbidden

AUTHORIZED INGREDIENTS:

- All Python standard library modules (datetime, math, os, sys, etc.)
- Creating your own modules and packages
- All import styles: import, from...import, import...as
- Creating ___init___.py files (Sacred Scrolls)
- Absolute and relative imports
- Type hints and annotations

FORBIDDEN DARK MAGIC:

- External libraries (no pip install)
- Using eval() or exec() (dangerous transmutations)
- Modifying sys.path directly
- Using importlib for dynamic imports
- Complex algorithms (keep your spells simple)

IV.2 Laboratory Structure

You'll build your Alchemical Laboratory progressively through four parts:

- **Part I**: The Sacred Scroll (__init__.py mystery)
- **Part II**: Import Transmutation (from...import mastery)
- **Part III**: The Great Pathway Debate (absolute vs relative)
- **Part IV**: Breaking the Circular Curse (dependency resolution)

Final Laboratory Structure:

- `alchemy/` - Your main laboratory package
- `alchemy/elements.py` - Basic elemental spell functions
- `alchemy/potions.py` - Advanced potion recipe functions
- `alchemy/transmutation/` - Transformation spell package
- `alchemy/grimoire/` - Spell documentation package

IMPORTANT: All functions should be simple and return strings. Focus on import concepts, not complex logic. Handle errors by returning descriptive error messages as strings.

Chapter V

Mandatory part

V.1 Part I: The Sacred Scroll

Objective

Discover the power of ___init___.py - the sacred scroll that transforms ordinary folders into magical Python packages.

Files to create for Part I

- `ft_sacred_scroll.py` - Demonstration script (at repository root)
- `alchemy/__init__.py` - The main sacred scroll
- `alchemy/elements.py` - Basic elemental spells

Instructions

Create your first alchemical package and learn how ___init___.py controls what magic is available to other alchemists.

alchemy/elements.py should contain:

- `create_fire()` - Returns "Fire element created"
- `create_water()` - Returns "Water element created"
- `create_earth()` - Returns "Earth element created"
- `create_air()` - Returns "Air element created"

alchemy/___init___.py must contain exactly:

- `__version__ = "1.0.0"`
- `__author__ = "Master Pythonicus"`
- Import and expose ONLY `create_fire` and `create_water` from elements.py
- Do NOT expose `create_earth` and `create_air` (they remain hidden)
- Use: `from .elements import create_fire, create_water`

The __init__.py file controls package interface. Functions imported here become available as alchemy.function_name. Functions not imported remain hidden and require direct module access.

Expected Output Example:
Test your sacred scroll:

```
$> python3 ft_sacred_scroll.py

=== Sacred Scroll Mastery ===

Testing direct module access:
alchemy.elements.create_fire(): Fire element created
alchemy.elements.create_water(): Water element created
alchemy.elements.create_earth(): Earth element created
alchemy.elements.create_air(): Air element created

Testing package-level access (controlled by __init__.py):
alchemy.create_fire(): Fire element created
alchemy.create_water(): Water element created
alchemy.create_earth(): AttributeError - not exposed
alchemy.create_air(): AttributeError - not exposed

Package metadata:
Version: 1.0.0
Author: Master Pythonicus
```

Your ft_sacred_scroll.py should demonstrate both direct module access (alchemy.elements.function) and package-level access (alchemy.function). Handle AttributeError exceptions gracefully by printing error messages.

How does the Sacred Scroll (__init__.py) control which spells are available to other alchemists? What's the difference between what exists in a module and what's exposed by the package?

V.2 Part II: Import Transmutation

Objective

Master the art of from...import - summoning specific spells from distant grimoires without bringing the entire book.

Files to create for Part II

- `ft_import_transmutation.py` - Demonstration script (at repository root)
- `alchemy/potions.py` - Advanced potion recipes

Instructions

Expand your laboratory and learn different ways to summon magical formulas.

alchemy/potions.py must contain exactly:

- `healing_potion()` - Returns "Healing potion brewed with [fire_result] and [water_result]"
- `strength_potion()` - Returns "Strength potion brewed with [earth_result] and [fire_result]"
- `invisibility_potion()` - Returns "Invisibility potion brewed with [air_result] and [water_result]"
- `wisdom_potion()` - Returns "Wisdom potion brewed with all elements: [all_four_results]"

`Each potion function must import and call the required elemental functions, then return a string that includes the elemental results. For example: from .elements import create_fire, create_water`

Your demonstration should show:

- Different import styles: `import alchemy.elements`
- Specific imports: `from alchemy.elements import create_fire`
- Aliased imports: `from alchemy.potions import healing_potion as heal`
- Multiple imports: `from alchemy.elements import create_fire, create_water`
- How each style affects your code differently

Expected Output Example:
Test your transmutation methods:

```
$> python3 ft_import_transmutation.py

=== Import Transmutation Mastery ===

Method 1 - Full module import:
alchemy.elements.create_fire(): Fire element created

Method 2 - Specific function import:
create_water(): Water element created

Method 3 - Aliased import:
heal(): Healing potion brewed with Fire element created and Water element created

Method 4 - Multiple imports:
create_earth(): Earth element created
create_fire(): Fire element created
strength_potion(): Strength potion brewed with Earth element created and Fire element created

All import transmutation methods mastered!
```

Your ft_import_transmutation.py should demonstrate all four import styles clearly. Each method should show the import statement used and the function call result.

What are the advantages and disadvantages of each import transmutation method? When would you use import module vs from module import function?

V.3 Part III: The Great Pathway Debate

Objective

Understand the ancient debate between absolute and relative imports - two different paths to reach the same magical formula.

Files to create for Part III

- `ft_pathway_debate.py` - Demonstration script (at repository root)
- `alchemy/transmutation/__init__.py` - Transmutation package initializer
- `alchemy/transmutation/basic.py` - Basic transmutations
- `alchemy/transmutation/advanced.py` - Advanced transmutations

Instructions

Create a complex laboratory structure and learn when to use each pathway type.

alchemy/transmutation/basic.py must contain exactly:

- Absolute import: `from alchemy.elements import create_fire, create_earth`
- `lead_to_gold()` - Returns "Lead transmuted to gold using [fire_result]"
- `stone_to_gem()` - Returns "Stone transmuted to gem using [earth_result]"

alchemy/transmutation/advanced.py must contain exactly:

- Relative import: `from .basic import lead_to_gold`
- Relative import: `from ..potions import healing_potion`
- `philosophers_stone()` - Returns "Philosopher's stone created using [lead_to_gold_result] and [healing_potion_result]"
- `elixir_of_life()` - Returns "Elixir of life: eternal youth achieved!"

alchemy/transmutation/___init___.py must contain:

- `from .basic import lead_to_gold, stone_to_gem`
- `from .advanced import philosophers_stone, elixir_of_life`

The transmutation package __init__.py exposes all transmutation functions for easy access. This demonstrates package-level organization.

Expected Output Example:
Test your pathway methods:

```
$> python3 ft_pathway_debate.py

=== Pathway Debate Mastery ===

Testing Absolute Imports (from basic.py):
lead_to_gold(): Lead transmuted to gold using Fire element created
stone_to_gem(): Stone transmuted to gem using Earth element created

Testing Relative Imports (from advanced.py):
philosophers_stone(): Philosopher's stone created using Lead transmuted to gold using Fire element
created and Healing potion brewed with Fire element created and Water element created
elixir_of_life(): Elixir of life: eternal youth achieved!

Testing Package Access:
alchemy.transmutation.lead_to_gold(): Lead transmuted to gold using Fire element created
alchemy.transmutation.philosophers_stone(): [same as above]

Both pathways work! Absolute: clear, Relative: concise
```

Your ft_pathway_debate.py should demonstrate the difference between absolute imports (full path) and relative imports (dots). Show how both methods access the same functions.

When should an alchemist use absolute pathways vs relative pathways? What are the trade-offs between clarity and conciseness?

V.4 Part IV: Breaking the Circular Curse

Objective

Learn to identify and break the dreaded Circular Dependency Curse - when spells try to summon each other in an endless loop, threatening to destroy your laboratory!

Files to create for Part IV

- `ft_circular_curse.py` - Demonstration script (at repository root)
- `alchemy/grimoire/__init__.py` - Grimoire package initializer
- `alchemy/grimoire/spellbook.py` - Records spells and their effects
- `alchemy/grimoire/validator.py` - Validates spell ingredients

Instructions

Create a scenario that would cause circular imports, then learn the ancient techniques to break the curse.

alchemy/grimoire/___init___.py must contain:

- `from .spellbook import record_spell`
- `from .validator import validate_ingredients`

alchemy/grimoire/validator.py must contain exactly:

- `validate_ingredients(ingredients: str) -> str` - Returns "[ingredients] - VALID" or "[ingredients] - INVALID"
- Simple validation: ingredients containing "fire", "water", "earth", or "air" are valid
- Any other ingredients are invalid

alchemy/grimoire/spellbook.py must contain exactly:

- `record_spell(spell_name: str, ingredients: str) -> str` - Records spells after validation
- Must use validator.py to check ingredients before recording (use late import to avoid circular dependency)
- Returns "Spell recorded: [spell_name] ([validation_result])" if valid
- Returns "Spell rejected: [spell_name] ([validation_result])" if invalid

Breaking the circular curse - choose ONE method:

- **Method 1 - Late Import**: Import validator inside the record_spell function
- **Method 2 - Dependency Injection**: Pass validator function as parameter
- **Method 3 - Shared Module**: Create separate validation utilities

DO NOT create actual circular imports in your code! Demonstrate understanding by explaining the problem and implementing one solution method.

Expected Output Example:
Test your curse-breaking techniques:

```
$> python3 ft_circular_curse.py

=== Circular Curse Breaking ===

Testing ingredient validation:
validate_ingredients("fire air"): fire air - VALID
validate_ingredients("dragon scales"): dragon scales - INVALID

Testing spell recording with validation:
record_spell("Fireball", "fire air"): Spell recorded: Fireball (fire air - VALID)
record_spell("Dark Magic", "shadow"): Spell rejected: Dark Magic (shadow - INVALID)

Testing late import technique:
record_spell("Lightning", "air"): Spell recorded: Lightning (air - VALID)

Circular dependency curse avoided using late imports!
All spells processed safely!
```

Your ft_circular_curse.py should demonstrate how to avoid circular imports by using late imports (importing inside functions) or dependency injection. Show both valid and invalid ingredient examples.

What causes the Circular Dependency Curse and why is it dangerous? Which curse-breaking technique is most appropriate for different situations?

Chapter VI

Submission Instructions

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense.

File Organization:
All files and directories must be created at the root of your Git repository:

- `ft_sacred_scroll.py` - at repository root
- `ft_import_transmutation.py` - at repository root
- `ft_pathway_debate.py` - at repository root
- `ft_circular_curse.py` - at repository root
- `alchemy/` - directory at repository root
- `alchemy/__init__.py` - inside alchemy directory
- `alchemy/elements.py` - inside alchemy directory
- `alchemy/potions.py` - inside alchemy directory
- `alchemy/transmutation/` - subdirectory inside alchemy
- `alchemy/transmutation/__init__.py` - inside transmutation directory
- `alchemy/transmutation/basic.py` - inside transmutation directory
- `alchemy/transmutation/advanced.py` - inside transmutation directory
- `alchemy/grimoire/` - subdirectory inside alchemy
- `alchemy/grimoire/__init__.py` - inside grimoire directory
- `alchemy/grimoire/spellbook.py` - inside grimoire directory
- `alchemy/grimoire/validator.py` - inside grimoire directory

During evaluation, you may be asked to explain import mechanisms, demonstrate different import styles, or modify your alchemical laboratory. Make sure you understand the four sacred mysteries, not just the implementation.

Focus on clean, well-organized code that clearly demonstrates Python's import system. The alchemical functions should be simple - the complexity is in mastering the import mysteries.

FINAL REMINDER: This activity requires solid mastery of Python fundamentals. If you struggle with basic Python syntax, functions, classes, lists, dictionaries, or error handling, strengthen these skills first. Import mastery builds upon these foundations - attempting this activit ywithout them will lead to confusion and frustration.

A true Python alchemist understands that good code organization is like a well-organized laboratory - everything has its place, and you can find any spell quickly when you need it!