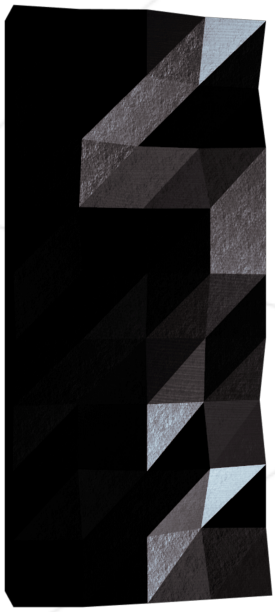

FuncMage

Master the Ancient Arts of Functional Programming

Summary: Welcome to FuncMage Chronicles! In the year 2142, you are a Function Mage learning the ancient arts of higher-order functions, decorators, and lambda spells. Master these powerful techniques to restore balance to the digital realm through inclusive adventures that welcome all aspiring mages.

Version: 2.0

Contents

Chapter I

Foreword

Welcome, aspiring Function Mage, to the year 2142!

In this cyberpunk future, the digital realm faces chaos. Ancient programming techniques have been forgotten, and only the **Function Mages** possess the knowledge to restore balance. You are one of these chosen few, regardless of your background, identity, or experience level—every mage brings unique strengths to our diverse guild.

The **Lambda Codex**, a legendary artifact containing the secrets of **higher-order functions**, **decorators**, and **lexical scoping**, has been scattered across the digital dimensions. As a Function Mage, you must master these ancient arts to reunite the fragments and save both the virtual and physical worlds.

Your journey will take you through five mystical realms, each teaching you a fundamental aspect of functional programming. You'll meet fellow mages from all walks of life—Alex the Lambda Specialist, Jordan the Decorator Master, Riley the Scope Guardian, and many others who will guide you on this inclusive adventure.

Remember, young mage: **functions are first-class citizens** in Python, meaning they can be passed around, stored in variables, and transformed just like any other data. This power, when wielded correctly, can create elegant, reusable, and powerful spells that adapt to any situation.

Your quest begins now. The digital realm awaits your unique contribution to the Function Mage legacy!

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 FuncMage Chronicles: Master the Ancient Arts of Functional Programming!

You've mastered Python's fundamentals, conquered object-oriented programming, and learned to handle exceptions with grace. Now it's time to discover the most elegant and powerful programming paradigms: **the techniques of functional programming** that will transform how you think about code!

In this epic adventure, you'll explore five mystical realms, each revealing ancient secrets:

- **Exercise 0**: Lambda Sanctum - Master anonymous functions and lambda expressions
- **Exercise 1**: Higher Realm - Discover the power of higher-order functions
- **Exercise 2**: Memory Depths - Understand lexical scoping and closures
- **Exercise 3**: Ancient Library - Explore the functools module's treasures
- **Exercise 4**: Master's Tower - Create powerful decorators and class methods

Each realm presents challenges that will expand your understanding of Python's functional capabilities while telling an engaging story of digital adventure and discovery.

PREREQUISITES: This activity requires solid mastery of Python functions, classes, exception handling, and basic data structures. You should be comfortable with Python syntax and object-oriented concepts before embarking on this functional programming journey.

Focus on understanding **why** functional programming patterns are powerful, not just **how** to implement them. A great Function Mage understands the philosophy behind the magic.

HELPER TOOL AVAILABLE: This activity includes a helper tool to assist your learning journey! Check the tools/ directory for:

- data_generator.py - Generate realistic test data for all exercises with fantasy-themed mages, artifacts, and spells

Use python3 tools/data_generator.py to generate test data for your implementations.

Chapter IV

Common Instructions

IV.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- Your functions should handle exceptions gracefully to avoid crashes.
- Use **type hints** for all function signatures and return types.
- Focus on demonstrating functional programming patterns clearly.
- Each exercise should be in its own file with the specified name.
- Write clean, readable code that demonstrates your understanding.

IV.2 Authorized Imports

- **functools** module - Essential for this project
- **typing** module - For advanced type hints
- **operator** module - For functional operations
- **itertools** module - For advanced iteration patterns
- Standard library modules as needed for the exercises
- No external libraries (no pip install)

IV.3 Forbidden

- External libraries (no pip install)
- File I/O operations (focus on in-memory processing)
- Complex algorithms (keep focus on functional patterns)
- Using eval() or exec()
- Global variables (embrace functional purity when possible)

Chapter V

Exercise 0: Lambda Sanctum

	Exercise0
	ex0
Directory: *ex*0/	
Files to Submit: `lambda_spells.py`	
Authorized: `map, filter, sorted, print()`	

`For this exercise, you must use lambda expressions for all transformations. Do not use the 'def' keyword to create named functions for simple operations. The goal is to master anonymous functions and functional programming patterns.`

The Lambda Sanctum: Mastering Anonymous Functions

Welcome to the Lambda Sanctum, young mage! Here, the ancient art of **anonymous functions** is taught. Lambda expressions are like quick incantations—short, powerful spells that can transform data without the ceremony of full function definitions.

Your Mission: `Master the art of lambda expressions by helping the Sanctum's guardian, Sage Lambda, organize magical artifacts using anonymous functions. Learn to create functions on-the-fly for data transformation.`

The Challenge

Create a file `lambda_spells.py` that contains functions demonstrating lambda mastery:
Function Signatures:

```
def artifact_sorter(artifacts: list[dict]) -> list[dict]
def power_filter(mages: list[dict], min_power: int) -> list[dict]
def spell_transformer(spells: list[str]) -> list[str]
def mage_stats(mages: list[dict]) -> dict
```

Implementation Requirements

artifact_sorter(artifacts) - Sort magical artifacts:

- Use `sorted()` with a lambda to sort by 'power' level (descending)
- Each artifact is a dict: `{'name': str, 'power': int, 'type': str}`
- Return the sorted list

power_filter(mages, min_power) - Filter mages by power:

- Use `filter()` with a lambda to find mages with power >= min_power
- Each mage is a dict: `{'name': str, 'power': int, 'element': str}`
- Return a list of filtered mages

spell_transformer(spells) - Transform spell names:

- Use `map()` with a lambda to add "* " prefix and " *" suffix
- Input: list of spell names (strings)
- Return a list of transformed spell names

mage_stats(mages) - Calculate statistics:

- Use lambdas with `max()`, `min()`, and `sum()` to find:
- Most powerful mage's power level
- Least powerful mage's power level
- Average power level (rounded to 2 decimals)
- Return dict: `{'max_power': int, 'min_power': int, 'avg_power': float}`

Expected Output Example:

```
$> python3 lambda_spells.py

Testing artifact sorter...
Fire Staff (92 power) comes before Crystal Orb (85 power)

Testing spell transformer...
* fireball * * heal * * shield *
```

How do **lambda expressions** make code more concise? When should you use lambda vs. regular function definitions?

Chapter VI

Exercise 1: Higher Realm

	Exercise1
ex1	
Directory: *ex*1/	
Files to Submit: `higher_magic.py`	
Authorized: `callable(), print()`	

The Higher Realm: Functions Operating on Functions

Welcome to the Higher Realm, where functions become the subjects of other functions! Here, you'll learn that functions are **first-class citizens**—they can be passed as arguments, returned from other functions, and stored in data structures.

Your Mission: Help the Realm's guardian, Mage Functional, create a spell-crafting system where functions can modify, combine, and enhance other functions. Master the art of higher-order functions!

The Challenge

Create a file `higher_magic.py` that demonstrates higher-order function mastery:
Function Signatures:

```
def spell_combiner(spell1: callable, spell2: callable) -> callable
def power_amplifier(base_spell: callable, multiplier: int) -> callable
def conditional_caster(condition: callable, spell: callable) -> callable
def spell_sequence(spells: list[callable]) -> callable
```

Implementation Requirements

spell_combiner(spell1, spell2) - Combine two spells:

- Return a new function that calls both spells with the same arguments
- The combined spell should return a tuple of both results
- Example: `combined = spell_combiner(fireball, heal)`

power_amplifier(base_spell, multiplier) - Amplify spell power:

- Return a new function that multiplies the base spell's result by multiplier
- Assume base spell returns a number (damage, healing, etc.)
- Example: `mega_fireball = power_amplifier(fireball, 3)`

conditional_caster(condition, spell) - Cast spell conditionally:

- Return a function that only casts the spell if condition returns True
- If condition fails, return "Spell fizzled"
- Both condition and spell receive the same arguments

spell_sequence(spells) - Create spell sequence:

- Return a function that casts all spells in order
- Each spell receives the same arguments
- Return a list of all spell results

Expected Output Example:

```
$> python3 higher_magic.py

Testing spell combiner...
Combined spell result: Fireball hits Dragon, Heals Dragon

Testing power amplifier...
Original: 10, Amplified: 30
```

How do **higher-order functions** enable code reuse and composition?
What makes functions "first-class citizens" in Python?

Chapter VII

Exercise 2: Memory Depths

	Exercise2
ex2	
Directory: *ex2/*	
Files to Submit: `scope_mysteries.py`	
Authorized: `nonlocal, print()`	

The Memory Depths: Lexical Scoping and Closures

Deep in the Memory Depths, the ancient secrets of **lexical scoping** are guarded. Here, functions remember the environment where they were created, capturing variables in mystical **closures** that persist beyond their original scope.

Your Mission: Help the Memory Keeper, Sage Closure, understand how functions can "remember" variables from their creation environment. Master closures and lexical scoping to create persistent magical effects!

The Challenge

Create a file `scope_mysteries.py` that demonstrates lexical scoping mastery: Function Signatures:

```
def mage_counter() -> callable
def spell_accumulator(initial_power: int) -> callable
def enchantment_factory(enchantment_type: str) -> callable
def memory_vault() -> dict[str, callable]
```

Implementation Requirements

mage_counter() - Create a counting closure:

- Return a function that counts how many times it's been called
- Each call should return the current count (starting from 1)
- The counter should persist between calls
- Use closure to maintain state without global variables

spell_accumulator(initial_power) - Create power accumulator:

- Return a function that accumulates power over time
- Each call adds the given amount to the total power
- Return the new total power after each addition
- Start with initial_power as the base

enchantment_factory(enchantment_type) - Create enchantment functions:

- Return a function that applies the specified enchantment
- The returned function takes an item name and returns enchanted description
- Format: "enchantment_type item_name" (e.g., "Flaming Sword")
- Each factory creates functions with different enchantment types

memory_vault() - Create a memory management system:

- Return a dict with 'store' and 'recall' functions
- 'store' function: takes (key, value) and stores the memory
- 'recall' function: takes (key) and returns stored value or "Memory not found"
- Use closure to maintain private memory storage

Expected Output Example:

```
$> python3 scope_mysteries.py

Testing mage counter...
Call 1: 1
Call 2: 2
Call 3: 3

Testing enchantment factory...
Flaming Sword
Frozen Shield
```

How do **closures** enable functions to "remember" their creation environment? What are the benefits of lexical scoping in functional programming?

Chapter VIII

Exercise 3: Ancient Library

	Exercise3
ex3	
Directory: *ex*3/	
Files to Submit: `functools_artifacts.py`	
Authorized: `functools, operator, print()`	

The Ancient Library: Functools Treasures

In the Ancient Library, the most powerful functional programming artifacts are stored. The **functools** module contains legendary tools: `reduce`, `partial`, `wraps`, and more. These artifacts can transform how you approach complex problems.

Your Mission: Help the Library's curator, Archivist Functools, catalog and demonstrate the power of these ancient artifacts. Learn to wield reduce, partial, and other functools treasures!

The Challenge

Create a file `functools_artifacts.py` that demonstrates functools mastery: Function Signatures:

```
def spell_reducer(spells: list[int], operation: str) -> int
def partial_enchanter(base_enchantment: callable) -> dict[str, callable]
def memoized_fibonacci(n: int) -> int
def spell_dispatcher() -> callable
```

Implementation Requirements

spell_reducer(spells, operation) - Reduce spell powers:

- Use `functools.reduce` to combine all spell powers
- Support operations: "add", "multiply", "max", "min"
- Use `operator` module functions (add, mul, etc.)
- Return the final reduced value

partial_enchanter(base_enchantment) - Create partial applications:

- Take a base enchantment function that needs (power, element, target)
- Use `functools.partial` to create specialized versions
- Return dict with keys: 'fire_enchant', 'ice_enchant', 'lightning_enchant'
- Each should be a partial with power=50 and the respective element

memoized_fibonacci(n) - Cached fibonacci:

- Use `functools.lru_cache` decorator for memoization
- Implement fibonacci sequence calculation
- The cache should improve performance for repeated calls
- Return the nth fibonacci number

spell_dispatcher() - Create single dispatch system:

- Use `functools.singledispatch` to create a spell system
- Handle different types: int (damage spell), str (enchantment), list (multi-cast)
- Return the dispatcher function
- Each type should have appropriate spell behavior

Expected Output Example:

```
$> python3 functools_artifacts.py

Testing spell reducer...
Sum: 100
Product: 240000
Max: 40

Testing memoized fibonacci...
Fib(10): 55
Fib(15): 610
```

How does **functools.reduce** enable powerful data aggregation? What are the performance benefits of **memoization** with lru_cache?

Chapter IX

Exercise 4: Master's Tower

	Exercise4
ex4	
Directory: *ex4/*	
Files to Submit: `decorator_mastery.py`	
Authorized: `functools.wraps, staticmethod, print()`	

The Master's Tower: Decorator Mastery and Class Methods

At the pinnacle of your journey lies the Master's Tower, where the most advanced Function Mages learn to create **decorators**—magical wrappers that can transform any function's behavior. Here, you'll also master `@staticmethod` and understand how decorators work with classes.

Your Mission: Prove your mastery to the Tower's guardian, Grandmaster Decorator, by creating powerful decorators that can enhance any spell or method. This is your final test as a Function Mage!

The Challenge

Create a file `decorator_mastery.py` that demonstrates decorator mastery: Function Signatures:

```
def spell_timer(func: callable) -> callable
def power_validator(min_power: int) -> callable
def retry_spell(max_attempts: int) -> callable
class MageGuild:
@staticmethod
def validate_mage_name(name: str) -> bool
def cast_spell(self, spell_name: str, power: int) -> str
```

Implementation Requirements

spell_timer(func) - Time execution decorator:

- Create a decorator that measures function execution time
- Print "Casting function_name..." before execution
- Print "Spell completed in time seconds" after execution
- Use `functools.wraps` to preserve original function metadata
- Return the original function's result

power_validator(min_power) - Parameterized validation decorator:

- Create a decorator factory that validates power levels
- Check if the first argument (power) is >= min_power
- If valid, execute the function normally
- If invalid, return "Insufficient power for this spell"
- Use `functools.wraps` properly

retry_spell(max_attempts) - Retry decorator:

- Create a decorator that retries failed spells
- If function raises an exception, retry up to max_attempts times
- Print "Spell failed, retrying... (attempt n/max_attempts)"
- If all attempts fail, return "Spell casting failed after max_attempts attempts"
- If successful, return the result normally

MageGuild class - Demonstrate staticmethod:

- `validate_mage_name(name)` - Static method that checks if name is valid
- Name is valid if it's at least 3 characters and contains only letters/spaces
- `cast_spell(self, spell_name, power)` - Instance method
- Should use the power_validator decorator with min_power=10
- Return "Successfully cast spell_name with power power"

Expected Output Example:

```
$> python3 decorator_mastery.py

Testing spell timer...
Casting fireball...
Spell completed in 0.101 seconds
Result: Fireball cast!

Testing MageGuild...
True
False
Successfully cast Lightning with 15 power
Insufficient power for this spell
```

How do **decorators** enable separation of concerns? What's the difference between @staticmethod and regular instance methods?

Chapter X

Submission and Peer-Review

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the peer-review. Don't hesitate to double-check the names of your files to ensure they are correct.

During peer-review, you may be asked to explain functional programming concepts, demonstrate how closures work, or show how decorators transform functions. Focus on understanding the concepts, not just the implementation.

Keep your implementations focused on demonstrating the functional programming concepts clearly. Avoid over-engineering-the goal is to show mastery of higher-order functions, closures, and decorators.

Congratulations, Function Mage! You've mastered the ancient arts of functional programming. These techniques will make your code more elegant, reusable, and powerful. Use them wisely in your future adventures!