# LambdaCat Beginner’s Manual (v2)

Welcome to **LambdaCat**, a Python toolkit for categories, functors, and functional programming patterns.
This manual is written for **Python developers, AI researchers, data/game modelers, and agent designers**.
No prior category theory knowledge is needed — we’ll build intuition step by step with examples, analogies, and runnable code.

---

## 0. The Big Idea

Think of a **category** as a *Lego set for programs*.
- **Objects** are Lego pieces (types, states).
- **Morphisms (arrows)** are the ways pieces fit together (functions, transitions).
- **Laws** make sure the pieces always snap together consistently.

Why does this matter?
- If you’re building a **data pipeline**, you want each stage to compose.
- If you’re designing an **AI agent**, you want decisions to follow consistent rules.
- If you’re modeling a **game**, you want state transitions to behave predictably.

LambdaCat gives you tools to **build, check, and play** with these structures in Python.

---

## 1. Functions as Arrows

In Python, a function is just an arrow from one type to another.

```
def f(x: int) -> str:
    return str(x)

def g(s: str) -> float:
    return float(len(s))

h = lambda x: g(f(x))
print(h(42))  # 2.0
```

This is **composition**: `h = g ■ f`.

■■ If you try to compose mismatched arrows (e.g. `f: int→str`, `k: bool→float`), it fails.
Categories make sure you **only compose arrows with matching types**.

---

## 2. Building Your First Category

A category has:
- Objects (types, states).
- Morphisms (arrows/functions).
- Composition rules.
- Identity arrows.

Example: Numbers → Strings → Floats

```
from lambdacat.core.category import Cat

objects = ["int","str","float"]
morphisms = {
    ("int","int"): ["id_int"],
    ("str","str"): ["id_str"],
    ("float","float"): ["id_float"],
    ("int","str"): ["f"],
    ("str","float"): ["g"],
    ("int","float"): ["gf"],
}
composition = {("f","g"): "gf"}
identities = {"int":"id_int","str":"id_str","float":"id_float"}

C = Cat(objects, morphisms, composition, identities)

print("Objects:", C.objects)
print("Morphisms from int to str:", C.morphisms[("int","str")])
```

Output:
```
Objects: ['int','str','float']
Morphisms from int to str: ['f']
```

---

## 3. Why Laws Matter

### *Identity Law*

Every object must have an identity arrow that does nothing.
Analogy: “Skip” button in Spotify should not change your playlist.

### *Associativity Law*

Composition must be associative: `(f■g)■h = f■(g■h)`.
Analogy: In math, `(2+3)+4 = 2+(3+4)`.

### *Checking Laws*

```
from lambdacat.lawsuite import CATEGORY_SUITE
CATEGORY_SUITE.run_suite(C)
```

Output:
```
[✓] Identity laws hold
[✓] Associativity holds
```

### ***Broken Category Example***

```
C_bad = Cat(["A"], {("A","A"): []}, {}, {})
CATEGORY_SUITE.run_suite(C_bad)
```

Output:
```
[✗] Identity law failed: object 'A' has no identity
```

---

## 4. Drawing and Chasing Diagrams

Diagrams are pictures of arrows.

Example: Currency conversion
- USD→EUR (`f`)
- EUR→JPY (`g`)
- USD→JPY (`h`)

```
from lambdacat.render import mermaid

diagram = {
    "objects": ["USD","EUR","JPY"],
    "morphisms": [
        ("USD","EUR","f"),
        ("EUR","JPY","g"),
        ("USD","JPY","h"),
    ]
}
print(mermaid(diagram))
```

Check if diagram commutes:

```
from lambdacat.lawsuite import check_commutativity
paths = [["f","g"], ["h"]]
print(check_commutativity(C, "USD","JPY", paths))
```

Output:
```
[✓] Paths f■g and h agree from USD→JPY
```

---

## 5. Functors: Mapping Categories

Functors translate one category into another, preserving structure.
Analogy: Different **views** of the same data — one in memory, one in a database.

```
from lambdacat.core.functor import FunctorBuilder
from lambdacat.lawsuite import FUNCTOR_SUITE

F = FunctorBuilder(C, C)
F.add_object_mapping("int","int")
F.add_morphism_mapping("f","f")
functor = F.build()

FUNCTOR_SUITE.run_suite(F)
```

Output:

```
[✓] Preserves identities
[✓] Preserves composition
```

---

## 6. Natural Transformations

Given two functors F, G: C → D, a **natural transformation** is a consistent way to turn F(X) into G(X).
Analogy: Two camera lenses — one wide, one zoom — both produce images that align.

```
from lambdacat.core.natural import check_naturality
check_naturality(F, G, eta)
```

If all squares commute, the transformation is natural.

---

## 7. From Functors to Monads

Monads are functors with extra powers.

### *Functor: map over a box*

```
from lambdacat.data import Option
print(Option.some(3).map(lambda x: x+1))  # Some(4)
```

### *Applicative: apply wrapped functions*

```
add = Option.pure(lambda x,y: x+y)
print(add.ap(Option.some(2)).ap(Option.some(3)))  # Some(5)
```

### *Monad: chain dependent computations*

```
from lambdacat.data import Result

safe_div = lambda x,y: Result.err("div by zero") if y==0 else Result.ok(x/y)
print(Result.ok((10,2)).bind(lambda xy: safe_div(*xy)))  # Ok(5.0)
```

```
print(Result.ok((10,0)).bind(lambda xy: safe_div(*xy)))  # Err("div by zero")
```

Analogy:
- Functor = “buff character stats in a game.”
- Applicative = “combine two independent buffs.”
- Monad = “sequence actions where the next depends on the last.”

---

## 8. State and Kleisli Categories

Monads model state transitions.
Analogy: **Game state machine**.

```
from lambdacat.monads.instances import State, Kleisli

def get(): return State(lambda s: (s,s))
def put(ns): return State(lambda _: (None, ns))

inc = Kleisli(lambda _: get().bind(lambda n: put(n+1)), State)
prog = inc.then(inc).then(inc)

_, s_final = prog.run(None).run(0)
print(s_final)  # 3
```

---

## 9. Optics: Lenses

Lenses let you “zoom in” on nested data.
Analogy: camera lens focusing on a subfield.

```
from lambdacat.optics import lens

user = {"name":"Ada","address":{"city":"London"}}
city_lens = lens["address"]["city"]

print(city_lens.get(user))          # London
print(city_lens.set(user,"Paris"))  # {'name':'Ada','address':{'city':'Paris'}}
```

Laws:
- Get-Set: get after set = new value.
- Set-Get: set after get = no change.

---

## 10. Agents and Plans

Agents are programs built from **plans**.

```
from lambdacat.agents.plan import Sequence, Plan
```

```
from lambdacat.agents.runtime import strong_monoidal_functor

plan = Sequence([
    Plan.primitive("greet", lambda s: s+["hi"]),
    Plan.primitive("ask", lambda s: s+["how are you?"]),
])
agent = strong_monoidal_functor(plan)

print(agent([]))  # ['hi','how are you?']
```

Analogy: workflow engine — steps are arrows, the agent is their composition.

---

## 11. Realistic Examples

### *Data Pipeline*

Objects = stages, morphisms = transforms.
Check diagram commutativity ensures consistent flow.

### *Game Model*

Objects = player states, morphisms = moves.
Category encodes all possible transitions.

### *AI Agent*

- Writer monad for logs.
- State monad for memory.
- Result monad for failures.
- Kleisli category composes decisions.

---

## 12. Where to Go Next

- Use LambdaCat for **small models, teaching, FP workflows, agent prototypes**.
- Use HyperCat for **higher categories, homotopy, serious research proofs**.

---

LambdaCat is your **entry point**: it makes category theory tangible in Python, with laws and diagrams you can run and check.