Functional Patterns & Pipelines

SECT.10.2026

- Introduce some key concepts of *functional programming* & explore functional design patterns.
- Explore some functional patterns: **memoization**, **command pattern**, **decorators**.

Paradigms

Procedural / Imperative

- Series of ordered steps ("procedures")
- Mutable variables
- Conditionals
- Iteration & Loops (`while` , `for`)

Functional

- Composition of pure functions (`g(f(x)) → y`)
- Immutable values & comprehensions
- First-class & higher-order functions (e.g. `filter` , `map` , `reduce`)
- Recursion

Others

- Object-Oriented (*up next*)
- Declarative (HTML/CSS, SQL)
- Event-driven (JS)

Key Functional Ideas

- **Immutability** - Data should not be modified in place, instead functions should return modified copies. This avoids ambiguity & bugs caused by in-place modification.
- **Pure functions** - Functions should be simple, and given the same inputs return the same outputs with no side-effects on the rest of the program. This aids in testability & composability.
- **First-class functions** - Functions themselves can be used as variables, including arguments to other functions.

Function Composition

A key idea within functional programming is *composition of functions*, which often takes the form of passing functions as parameters to other functions.

A function that takes another function is a **higher order** function.

A programming language that allows you to treat a function like any other kind of variable is said to have **first-class functions**.

```
data = [
    "Al Zebra",
    "Betty Yggdrasil",
    "Cyril X",
    "Dawn Waters",
]

sorted(data, key=lambda name: name.rsplit(" ", 1)[1])
```

```
['Dawn Waters', 'Cyril X', 'Betty Yggdrasil', 'Al Zebra']
```

(Functional) **Strategy Pattern**: Inject a behavior (last-name extraction) into an algorithm.

First-Class Functions

In Python functions are just another *type* a variable can be, can be used anywhere:

```
file_reader = {
  "json": json.load,
  "yaml": yaml.load,
}

extension = filename.split(".")[-1]
file_reader[extension](open(filename))
```

`map(func, *iterables)`

Makes an iterable that returns `func` applied to arguments from each of the iterables. (For shortest iterable.)

```
iterA = "ABCDEFG"
iterB = [1, 3, 5]
mult = lambda a, b: a * b
list(map(mult, iterA, iterB))
```

```
mult("A", 1)
mult("B", 3)
mult("C", 5)
```

```
["A", "BBB", "CCCCC"]
```

(Number of parameters == Number of iterables)

`filter(pred, iterable)`

Returns a new iterable of all values from `iterable` where `pred(val)` is truth.

```
list(filter(
  lambda v: "aeiou" in v,
  ["cow", "lynx", "spy", "cherry"]
))
```

```
["cow", "cherry"]
```

Procedural vs. Functional

```python
# procedural
def sum_of_squares_of_evens(numbers):
    total = 0
    for num in numbers:
        if num % 2 == 0:
            total += num ** 2
    return total
```

```python
# functional
def sum_of_squares_of_evens(numbers):
    return sum(
        map(
            lambda x: x ** 2,
            filter(lambda x: x % 2 == 0, numbers)
        )
    )
```

How would you write this?

```python
# comprehensions are functional too!
def sum_of_squares_of_evens(numbers):
    return sum(num ** 2 for num in numbers if num % 2 == 0)
```

Pure Functions

When writing functional code, we aspire to keep our functions **pure**:

- **Deterministic**: same inputs -> same outputs
- **No Side Effects**: No "hidden state" changed

Idempotent - Can be applied multiple times without changing result.

Remember: HTTP GET requests are assumed to be idempotent. What advantages did that offer?

```python
# side-effects
def add_like(post: Post, user: User) -> None:
    if post.id not in user.liked_posts:
        # modify objects & save to DB
        post.likes += 1
        post.save()
```

```python
# no side-effects
def add_like(post: Post, user: User) -> Post:
    if post.id not in user.liked_posts:
         # construct new objects (not saved to db yet)
         return Post( ... , likes=post.likes+1)
```

Advantage: Easier to Test

```
def test_func():
    assert func(input) == output
```

vs.

```
def test_func():
    # lots more to get wrong!
    with mock_db() as db:
        assert func(input)
    db.query( ... ) == expected
```

Advantage: memoization/caching

An essential functional pattern, avoiding redundant calls to pure functions.

```
@functools.cache
def pure_func(param1, param2, param3):
    print("performing expensive calculation...")
    ...

# decorator replaces the function! same as:
pure_func = functools.cache(pure_func)
```

Only safe to cache calls to `pure_func` if it has no side effects!

Often essential for efficient recursion.

@decorator

Python's decorator is a hybrid of the traditional Proxy/Decorator patterns.

Modifies a function, wrapping its interface with its own.

Anytime you want to ensure X is always done any time Y is called. (auth, logging, setup/teardown, etc.)

```
def login_required(view_func):
    def new_view_func(request):
        if not request.user.is_authorized():
            raise Http403()
        return view_func(request)
    # return new function which will replace original
    return new_view_func
```

```
@login_required
def profile_view(request: HttpRequest) → HttpResponse:
   ...
```

(In practice: `django.contrib.auth.decorators` provides these)

https://docs.djangoproject.com/en/5.0/topics/auth/default/#the-login-required-decorator)

Cache Decorator

```
def cache(orig_func):
    _seen_calls = {}

    # decorators define a new function
    def new_func(*args):
        # only call function if we've never seen this call
        if args not in _seen_calls:
            # populate cache
            _seen_calls[args] = orig_func(*args)

        # return from cache
        return _seen_calls[args]

    return new_func
```

Bug?

```python
def get_lowest(numbers: list[int]) -> int:
    # sorted returns a copy, leaving original intact
    return sorted(numbers)[0]
```

Reminder: `list`, `dict`, `set`, `objects` all mutable – need to be careful with copies/parameters

Functional Pattern: Immutable Objects

Return a mutated copy instead of modifying parameters.

Important for avoiding accidental side-effects/purity violations.

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: float
    y: float

p = Point(1, 2)
p.x = 3  # raises FrozenInstanceError

# instead create a new instance
p2 = dataclasses.replace(p, x=3)
```

Enforces **no side-effects**.

More Advantages of Functional Programming

- Simplified Debugging & Testing
  - Step-debugger (`pdb`) less important if we don't need to watch for side-effects. Can embrace `print` -style debugging at beginning/end of function.
- Much easier to test functions in isolation, reduced mocking.
- Often easier to document, fewer caveats.
- Makes reasoning about parallel/concurrent code easier.

Challenges

- Real programs need I/O which is a side-effect and often non-determinism creeps in as well.
- Can incur performance overhead, esp. in hybrid languages including Python. (recursion, immutability, etc.)
  - Function calls more expensive than iteration, lots of copies of immutable data.

(Functional) Strategy Pattern

Interchangeable family of algorithms.

Each strategy function has same type signature & purpose.

```
generate_invoice(items, address, shipping_func)
```

`shipping_cost_func(order: Order) → float`

```python
# shipping.py
def flat_rate(order) -> float:
    return 5

def weight_based(order) -> float:
    weight = sum(item.weight for item in order.items.all())

def bulk_discount(order) -> float:
    ...
```

(Functional) Command Pattern

Converts operations into objects for delayed execution.

(ex. `QuerySet` object shown last week)

```
def clean_phone(obj: dict) -> dict:
def format_address(obj: dict) -> dict:
def geocode(obj: dict) -> dict:
   ...

def execute(item, commands):
    for command in commands:
        item = command(item)

commands = [clean_phone, format_address, geocode]

for item in load_data():
    execute(item, commands)
    save_data(item)
```

Related Pattern: Currying (`partial` function application)

Allow partial application of functions for delayed eval/execution.

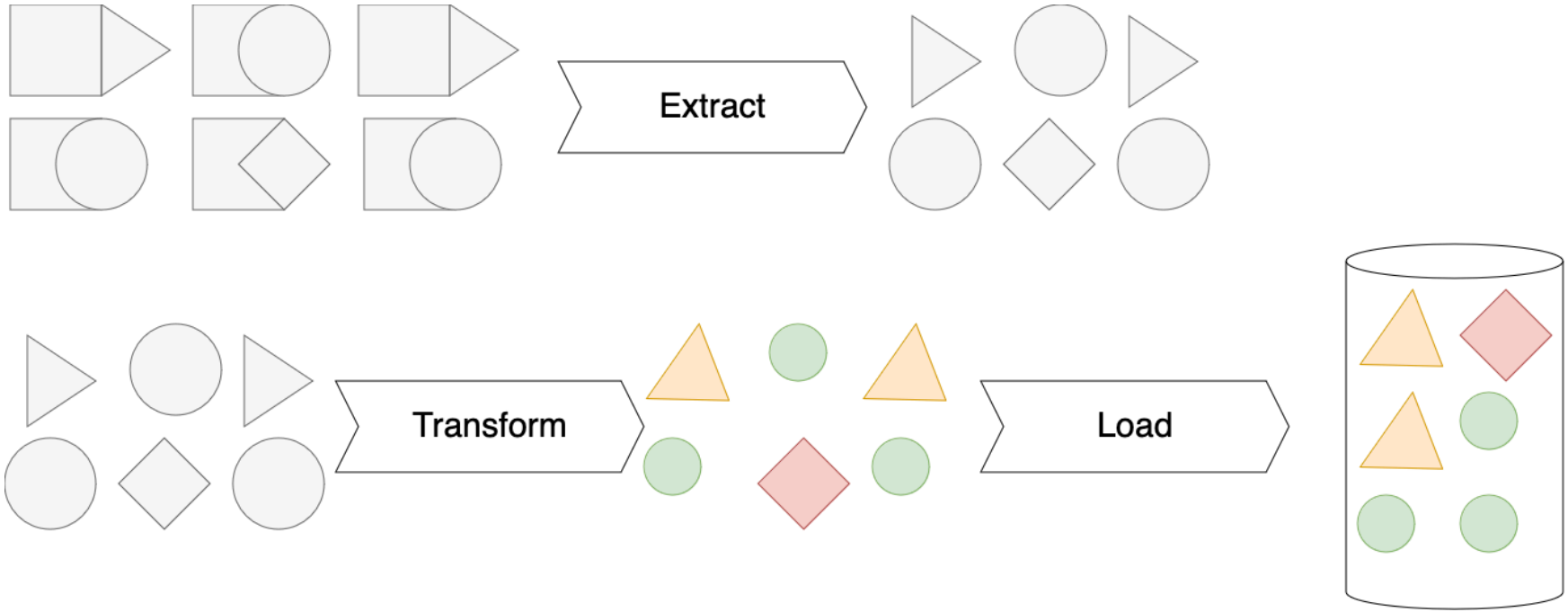
```
request_with_timeout = partial(
  httpx.get,
  timeout=10
)

# creates a new function
request_with_timeout("https://example.com")
# calls
httpx.get("https://example.com", timeout=10)
```

named after mathematician Haskell Curry

Bonus Section: Pipelines

(mostly covered in 30122 Pipelines: https://notes.jpt.sh/mpd/pipelines/)


Extract
Transform
Load

Extract

Scraping, API, DB Query, etc.

Transform

Cleaning, Merging, Validation, Normalization

Load

Reconciliation & Import

Extract can provide common interface:

```
load_xls(Path) → list[Expenditure]
load_xlsx(Path) → list[Expenditure]
load_from_sql(Path) → list[Expenditure]
```

Transforms can be **pure functions**.

```
clean_purpose(Expenditure) → Expenditure
match_department(Expenditure) → Expenditure
get_department_with_subdepartment(Expenditure) → Expenditure
```

Load functions can often be made (mostly) generic:

```
save_to_db(SqlModel)
```

Generic behavior provided on base `SqlModel` class, **strategy** pattern to override specifics if needed.

ETL Functional Benefits

- **Testablity**
- **Reusability** - small functions that can be applied in many contexts
- **Clarity** - clearly defined list of operations being performed
- **Scalability** - will aid in parallelism
- **Divisibility** - often want to (re)run subset of the pipeline

Classic Data Pipeline

```
# extract
data1 = load_data1()
data2 = load_data2()
output = []

# transform
output = clean_dataset_1(data1)
output += clean_dataset_2(data2)

# load
save_output_to_file(output)
```

what do `clean_dataset_*` look like?

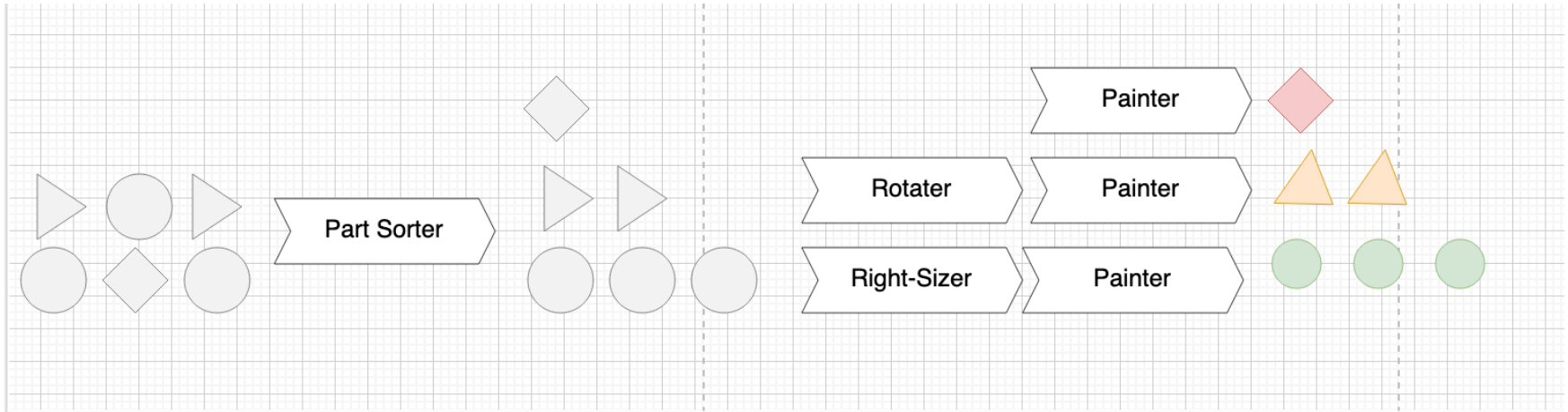
```
from .utils import clean_phone, clean_name, geocode

def clean_dataset_1(items: list) -> None:
    for item in items:
        item["phone"] = clean_phone(item["phone"])
        item["lat_lon"] = geocode(item["address"])

def clean_dataset_2(items: list) -> None:
    for item in items:
        item["name"] = clean_name(item["name"])
        item["lat_lon"] = geocode(item["address"])
```

Already extracted some helper functions, but can/should we go further?

Functional Approach



- From: `clean_fix_and_color(parts)`
- To: `clean(fix(color(part))) for part in parts`

Functional Refactor

```
# extract
data1 = load_data1()
data2 = load_data2()

# transform & load (incrementally)
for item in data1:
    save(clean(item, item1_steps))
for item in data2:
     save(clean(item, item2_steps))
```

Functional Refactor

```
# refactored to have uniform interface (item→item)
from .new_utils import clean_phone, clean_name, geocode


def clean(item, steps):
    for step_f in steps:
        item = step_f(item)
    return item

item1_steps = [clean_phone, geocode]
item2_steps = [clean_name, geocode]
```

Functional ETL Signatures

```
def load( ... ) → Iterable[Item]
""" raw data to list(?) of items """

def clean_step(item: Item) → Item
""" Item to modified-Item """

def save(Item) → None
""" Side-effect function: writes to DB or similar. """
```

Better yet, less opinionated about where save occurs.

```
def save(Item) → Item:
    pass
```

What type should `Item` be?

- `dict`
- `dataclasses.dataclass`
- `typing.NamedTuple`
- `pydantic.BaseModel`

```
from typing import NamedTuple

class Delivery(NamedTuple):
    id: int
    timestamp: datetime
    address: str
    city: str
```

Example Refactor

```
def geocode(address: str) -> Point:
    return Geocoder().call(address)

item["point"] = geocode(item["address"])
item2["point"] = geocode(item2["mailing_address"])
```

```
# item -> item?
def geocode2(item: dict) -> dict:
    item["point"] = Geocoder().call(item["address"])
    return item

item = geocode(item)
```

By making this a generic function, we've made our code *less* reusable.

Example Refactor (fixed)

```
def geocoder(item, from_field, to_field="point"):
    item[to_field] = Geocoder().call(item[from_field])
    return item
```

No longer item->item, but we can use `partial` to get that back (or just write thin adapters if easier).

```
from functools import partial
geocode_type1 = partial(geocoder, from_field="address")
geocode_type2 = partial(geocoder, from_field="mailing_address")

# after binding parameters, can be called with just one param again
item = geocode_type1(item)
item2 = geocode_type2(item2)
```

What would it take to continue to refactor this from:

```
# transform & load (incrementally due to amount of data)
data1 = load_data1()
for item in data1:
    save(clean(item, source1_steps))

data2 = load_data2()
for item in data2:
    save(clean(item, source2_steps))
```

to:

```
for item in itertools.chain(load_data1(), load_data2()):
    save(clean(item, steps))
```

What'd be required?

Two More Improvements

Streaming Data

What if data is bigger than memory? (With full data loaded we often use 2-3X as much memory as our data takes on disk.) *Almost always a good optimization.*

Unified Pipeline

Can’t be pure composition, will need *some* logic. *May or may not be desirable.*

Streaming Data

```
load_data() → list[Item]
```

Requires all items to be in memory at once.

With functional pipeline, we only operate on one at a time, this is **a lot** of wasted memory.

```
# note: most of the time second two parameters to Generator type will be None
def load_items() → Generator[Item, None, None]
    r = httpx.get("/list-page")
    for item in r.json():
        full_item = httpx.get(f"/item/{item['id']}")
        yield full_item
```

One Pipeline or Many?

```
# transform & load (incrementally)
for item in load_data1():
    save(clean(item, source1_steps))
for item in load_data2():
    save(clean(item, source2_steps))
```

Option A: OK to stop here.

Avoids logic inside pipeline, and pipelines are easily composable.

Option B: Use of simple conditionals

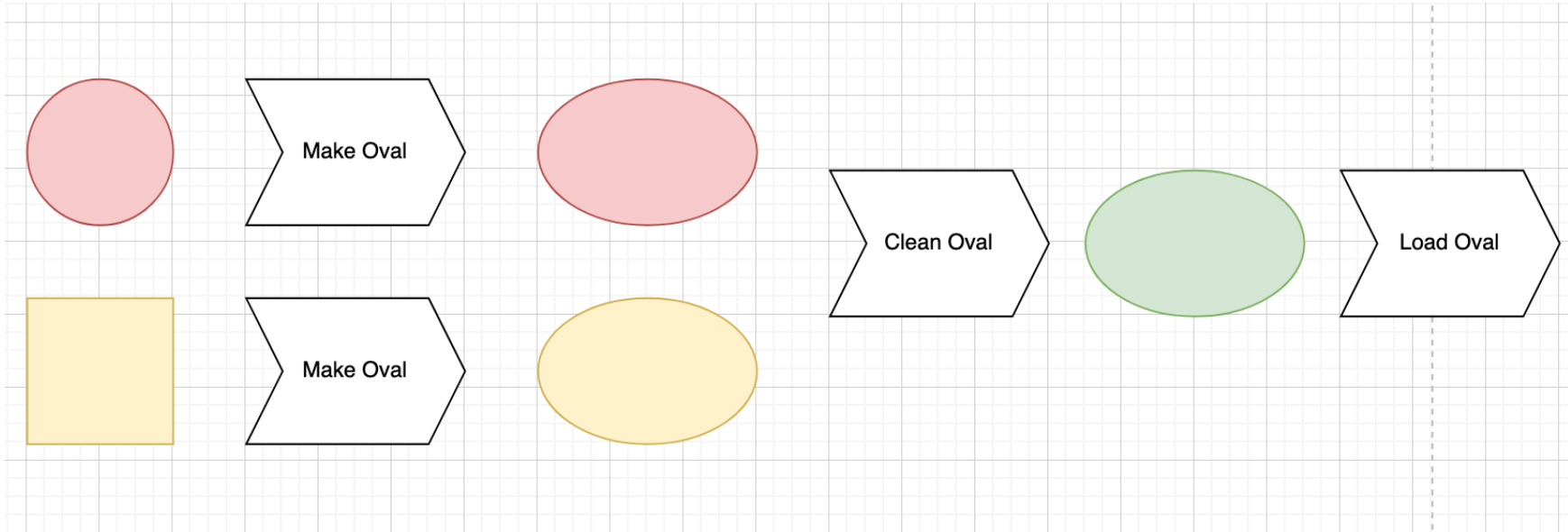
```
def apply_if(item, predicate, func):
    if predicate(item):
        return func(item)
    else:
        return item
```

Can combine with partials:

```
# only call clean_name on data from source A
clean_name_A = partial(
  apply_if,
  predicate=lambda item: item["source"] == "A",
  func=clean_name
)
```

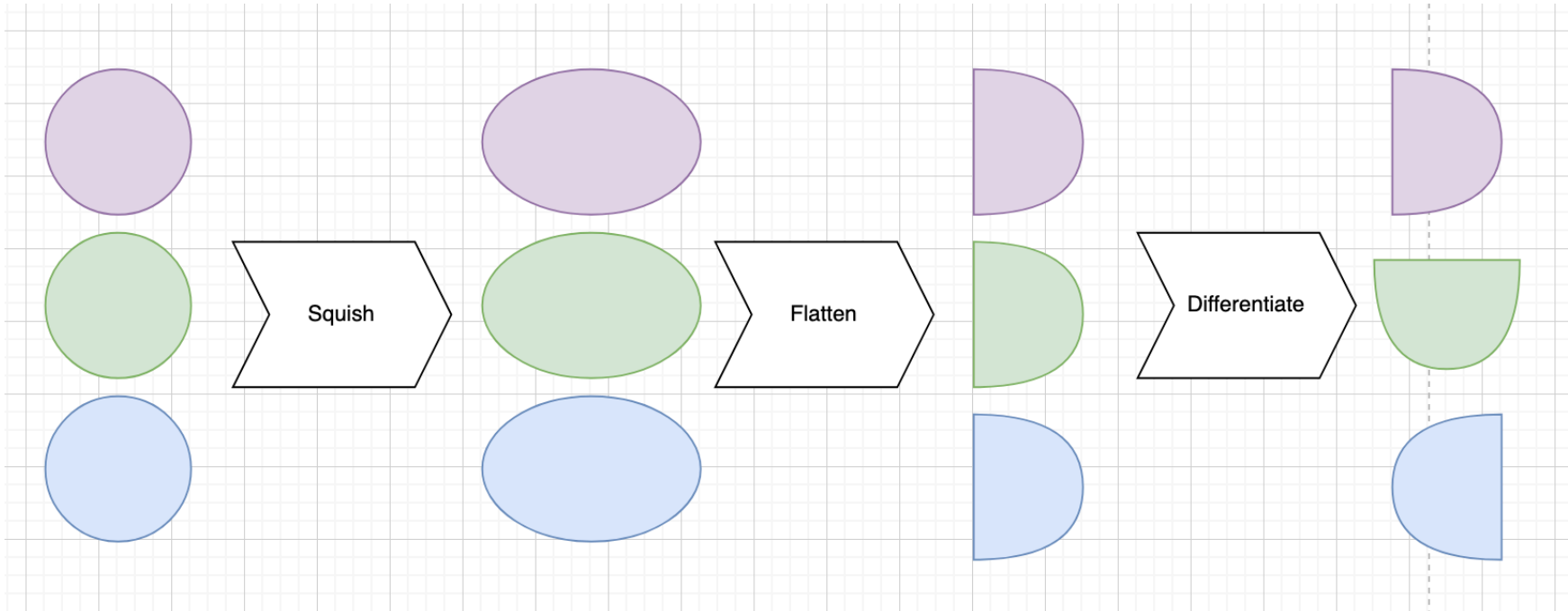
`clean_name_A` is an item->item function!

Option C: Smooth Differences at Start



Useful for when a few fields differ, but differences can be smoothed over with column renames, etc.

Option D: Differentiate at End



Useful for when most processing is shared, and minor differentiation comes at end.

Additional Features

Error Handling

Choice: Trap/handle errors at every level or let them stop pipeline?

Generator-based ETL can be restarted at any point.

```
def clean(item, steps):
    for step_f in steps:
        try:
            item = step_f(item)
        except Exception:
            reject_item(item)
            return None
    return item
```

Logging

Easy to implement inside or outside individual cleaning steps.

```
def clean(item, steps):
    for step_f in steps:
        old_item = item
```

Example Logging Decorator

```
def log_t(orig_transform):
    """
    orig_transform is a function with an Item → Item signature.
    """
    def new_transform(item: Item) → Item:
        # actual behavior is pass-through
        new_item = orig_transform(item)
        log(f"{item} ⇒ {new_item} (via {orig_transform.__name__}")
        return new_item

    # return our wrapper function
    return new_transform

@log_t
def fix_address(item: Item) → Item:
    ...

@log_t
def geocode(item: Item) → Item:
    ...
```

ETL Frameworks

Typically *strongly opinionated* about how your code should be written & composed.

That means more overhead, but typically offer easier distributed task running & monitoring.

- Luigi
- Airflow
- Prefect

Defensive Adoption

A lightweight convention that can work well with these tools is to separate your code into two layers:

- **data manipulation** - Focuses on data & does not import *anything* from framework/library.
- **glue layer** - Thin functions that bind your data manipulation layer to the framework/library in question.

Airflow

Directed Graph of functions, can be written in Python or use external tools.

Created at AirBNB.

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime

def fix_address(address, **kwargs):
    fixed_address = address.title()
    # output gets written to parameter ti
    kwargs['ti'].xcom_push(key='fixed_address', value=fixed_address)

def geocode(**kwargs):
    ti = kwargs['ti']
    fixed_address = ti.xcom_pull(key='fixed_address', task_ids='fix_address_task')
    coordinates = {"latitude": 40.7128, "longitude": -74.0060}

with DAG('address_processing_dag') as dag:
    fix_address_task = PythonOperator(
        task_id='fix_address_task',
        python_callable=fix_address,
        op_kwargs={'address': '123 elm street'}
    )
```

Example: Defining `__call__` for higher-level functions defined as classes.

```
class FieldSetter:
  def __init__(self, field_name, value):
    self.field_name = field_name
    self.value = value

  def __call__(self, item: dict) -> dict:
    if self.field_name not in item:
      item[self.field_name] = self.value

pipelines = [
  FieldSetter("address", ""),
  fix_address,
  geocode,
]
```

`__call__` is invoked when the object is called with parens.

```
fs = FieldSetter("address", "")
fs({})
{"address": ""}
```

This can be used to make higher-order functions perfect for data pipelines.