Designing Functional Data Pipelines for Reproducibility and Maintainability

By: Chin Hwee Ong (@ongchinhwee)

29 November 2021

PyData Global

About me

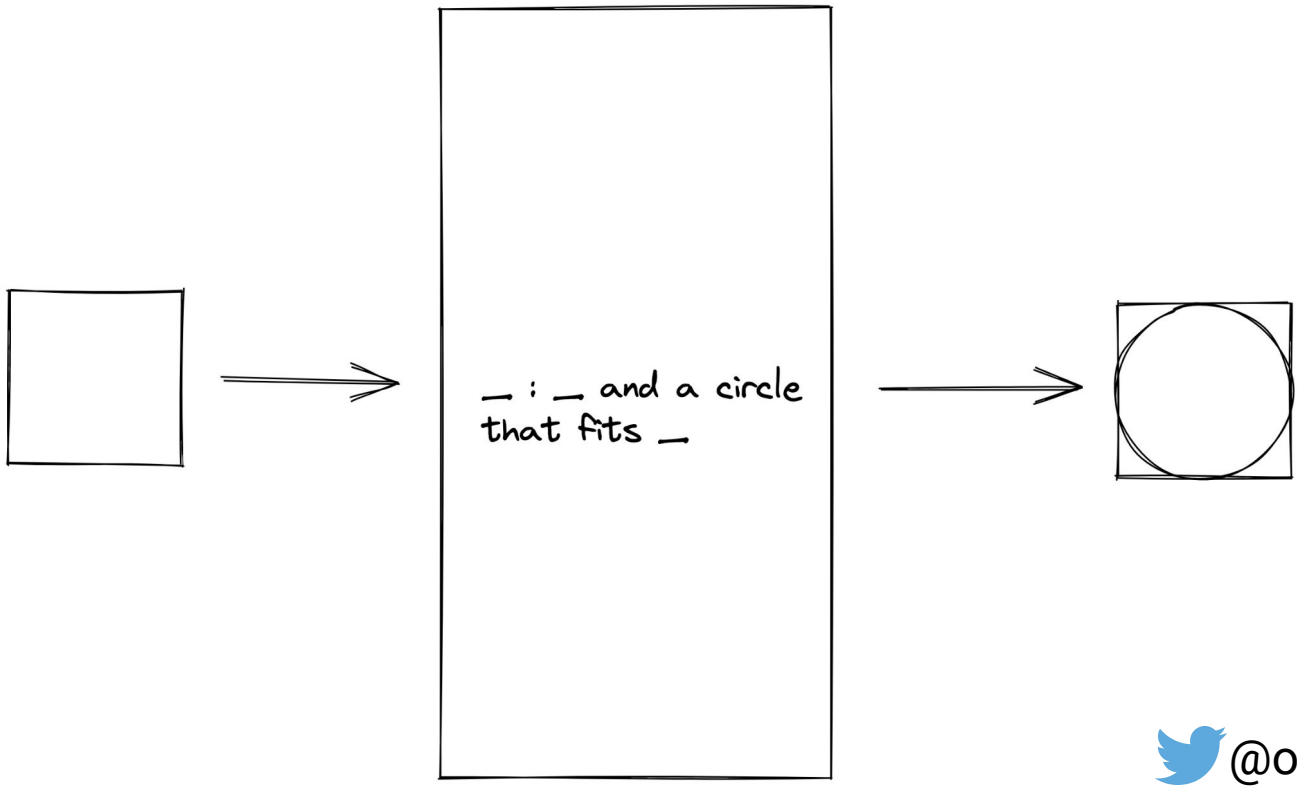
Ong Chin Hwee 王敬惠

- Data Engineer @ DT One
- Aerospace Engineering + Computational Modelling
- Speaker and (occasional) writer on data processing

Slides link:
bit.ly/pg2021-design-fp-data

@ongchinhwee

Basic Design Pattern: Data Pipeline



@ongchinhwee

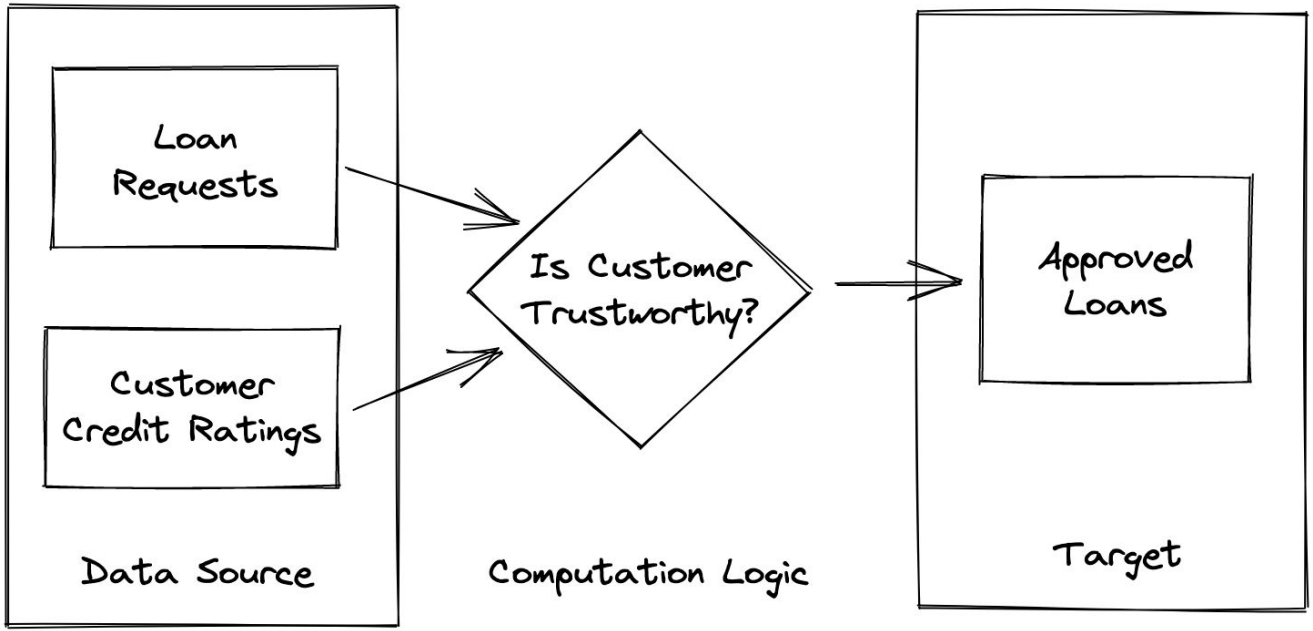
Designing a Data Pipeline at Scale

- **Reliable**
  - Data pipeline must produce the desired output → Reproducibility
- **Scalable**
  - Data pipeline must run independently across multiple nodes → Parallelism
- **Extensible**
  - Able to extend data pipeline with changing business logic → Maintainability

@ongchinhwee

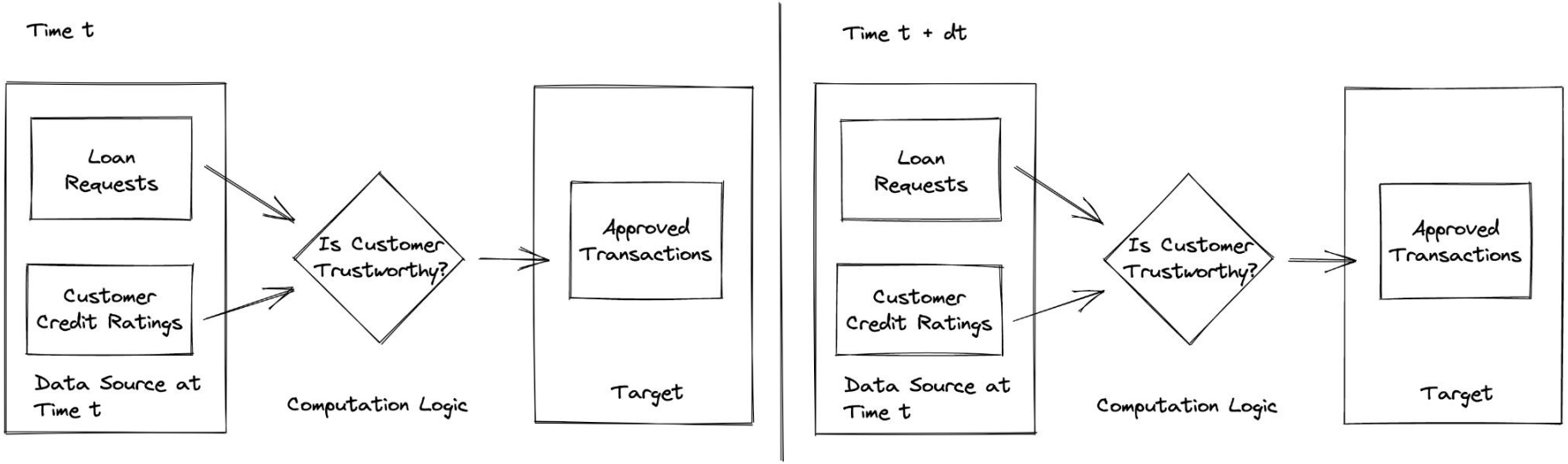
Challenges in Designing Data Pipelines at Scale

- **Reproducibility during Testing**



Challenges in Designing Data Pipelines at Scale

- **Reproducibility during Testing**



Same data source, same computation logic, processed at different times => same output?

@ongchinhwee

Challenges in Designing Data Pipelines at Scale

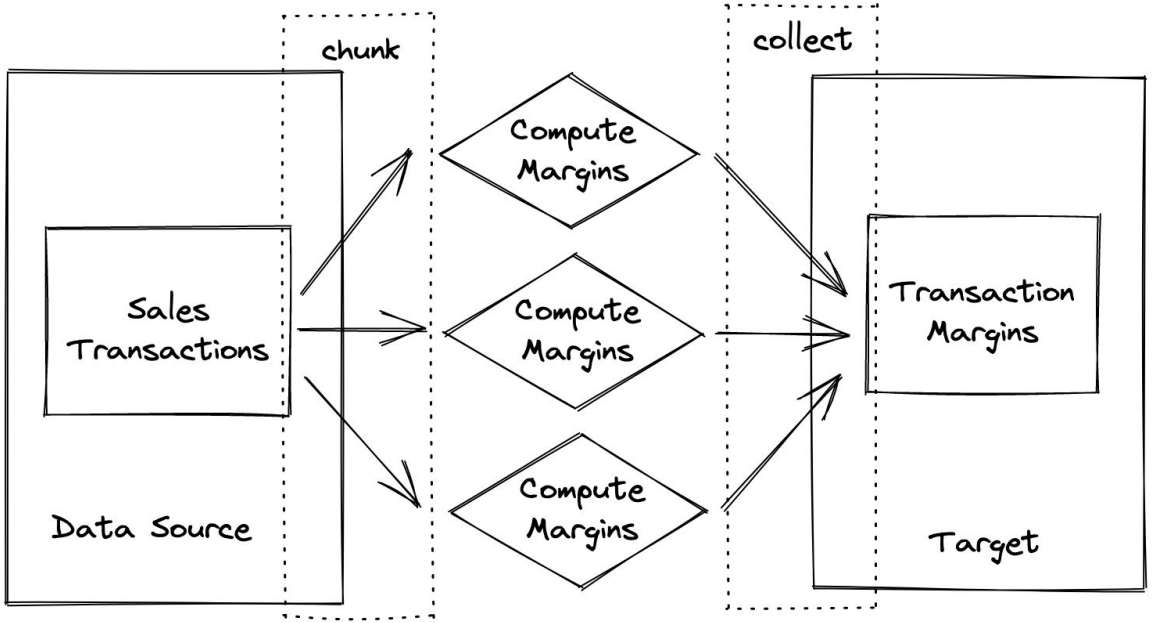
- **Reproducibility during Testing**
  - **Dependencies** in data pipeline design
    - Data source
    - Computation logic

Challenges in Designing Data Pipelines at Scale

- **Reproducibility during Testing**
  - Challenge: Given the **same data source**, how do we ensure that we replicate the **same result** every time we re-run the same process?

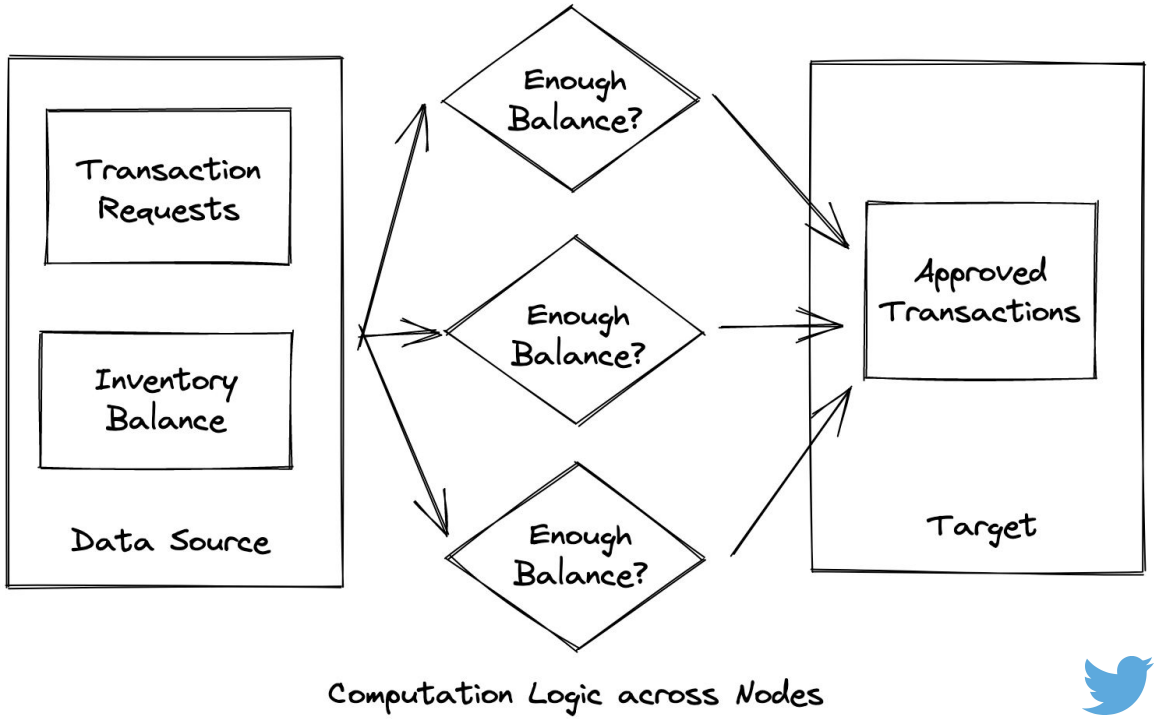
Challenges in Designing Data Pipelines at Scale

- **Reproducibility in Production**



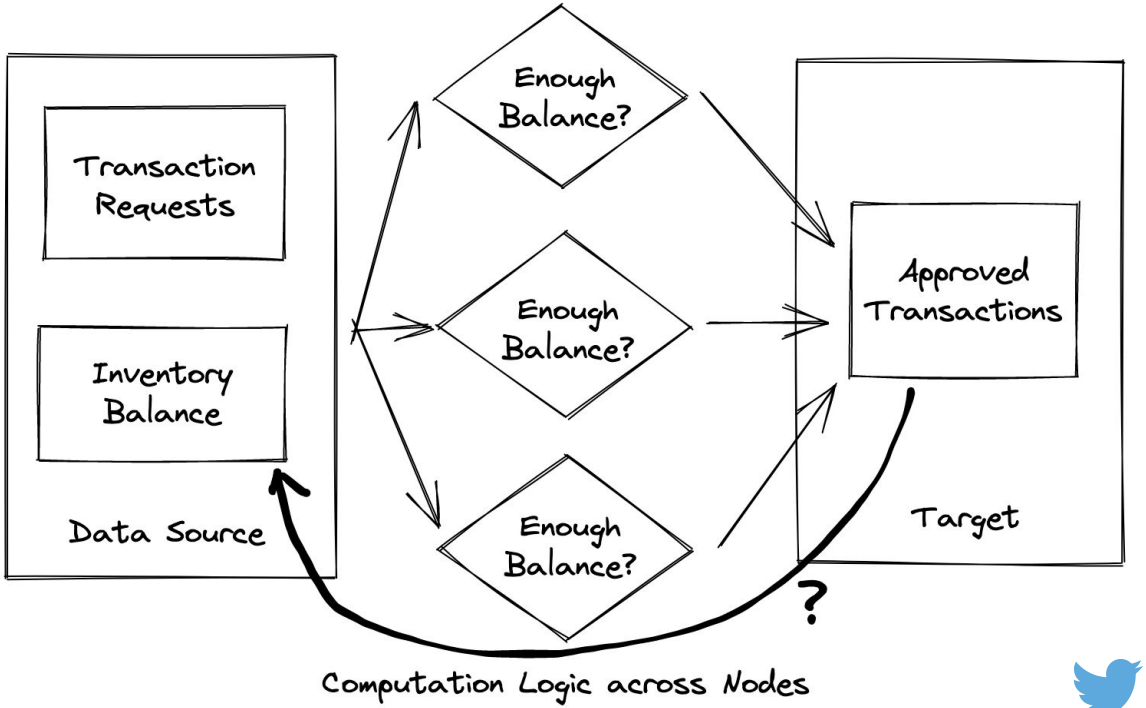
Challenges in Designing Data Pipelines at Scale

- **Reproducibility in Production**



Challenges in Designing Data Pipelines at Scale

- **Reproducibility in Production**



@ongchinhwee

Challenges in Designing Data Pipelines at Scale

- **Reproducibility in Production**
  - Debugging parallel/concurrent code at runtime due to **shared states**
    - E.g. What is the **current state** of the data source?

Challenges in Designing Data Pipelines at Scale

- **Reproducibility in Production**
  - Challenge: How do we design data pipelines that run the **same computation logic** across **multiple nodes** and **reproduce predictable results every time**?

Challenges in Designing Data Pipelines at Scale

- **Maintainability during Debugging**
  - “Works in testing, **breaks in production**” 😔
    - **Edge cases** and **inefficiencies not detected in test cases** causing performance issues and/or failures in production
    - Complexities in **debugging and logging** for **parallelism**

Challenges in Designing Data Pipelines at Scale

- **Maintainability during Debugging**
  - Challenge: How do we design data pipelines that are **readable and maintainable at its core** to reduce inefficiencies in production debugging at scale?

Challenges in Designing Data Pipelines at Scale

- **Maintainability when Adding New Features**
  - Adding new features to an **evolving (growing) codebase**
    - **Code reasoning** becomes more challenging with **increasing code complexity**
    - Risk of introducing **unintended behaviour** due to **dependencies**

Challenges in Designing Data Pipelines at Scale

- **Maintainability when Adding New Features**
  - Challenge: How do we design data pipelines that **adapts well to changing business and technical requirements** and ensures developer productivity?

Data Pipelines as Functions

@ongchinhwee

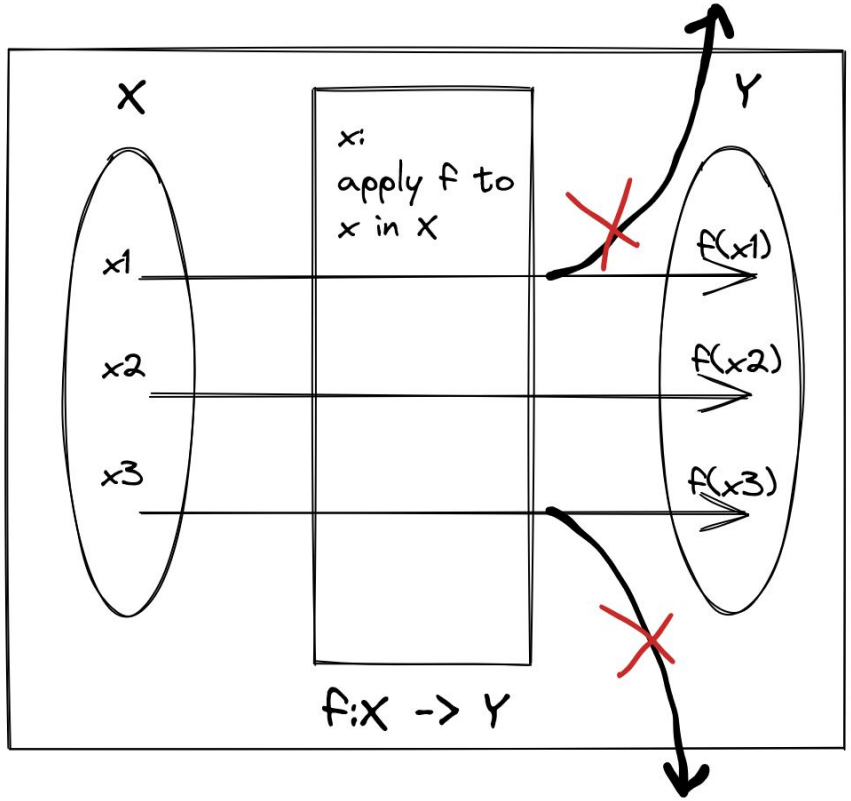
What is Functional Programming?

- **Declarative style of programming** that emphasizes writing software using only:
  - **Pure functions**; and
  - **Immutable values**.

3 Key Principles of Functional Programming

- **Pure functions** and avoid side effects
- **Immutability**
- **Referential transparency**

Pure Function and Avoid Side Effects



Pure function

1. output depends on
- input
- internal algorithm
- ...nothing else.

2. no side effects

@ongchinhwee

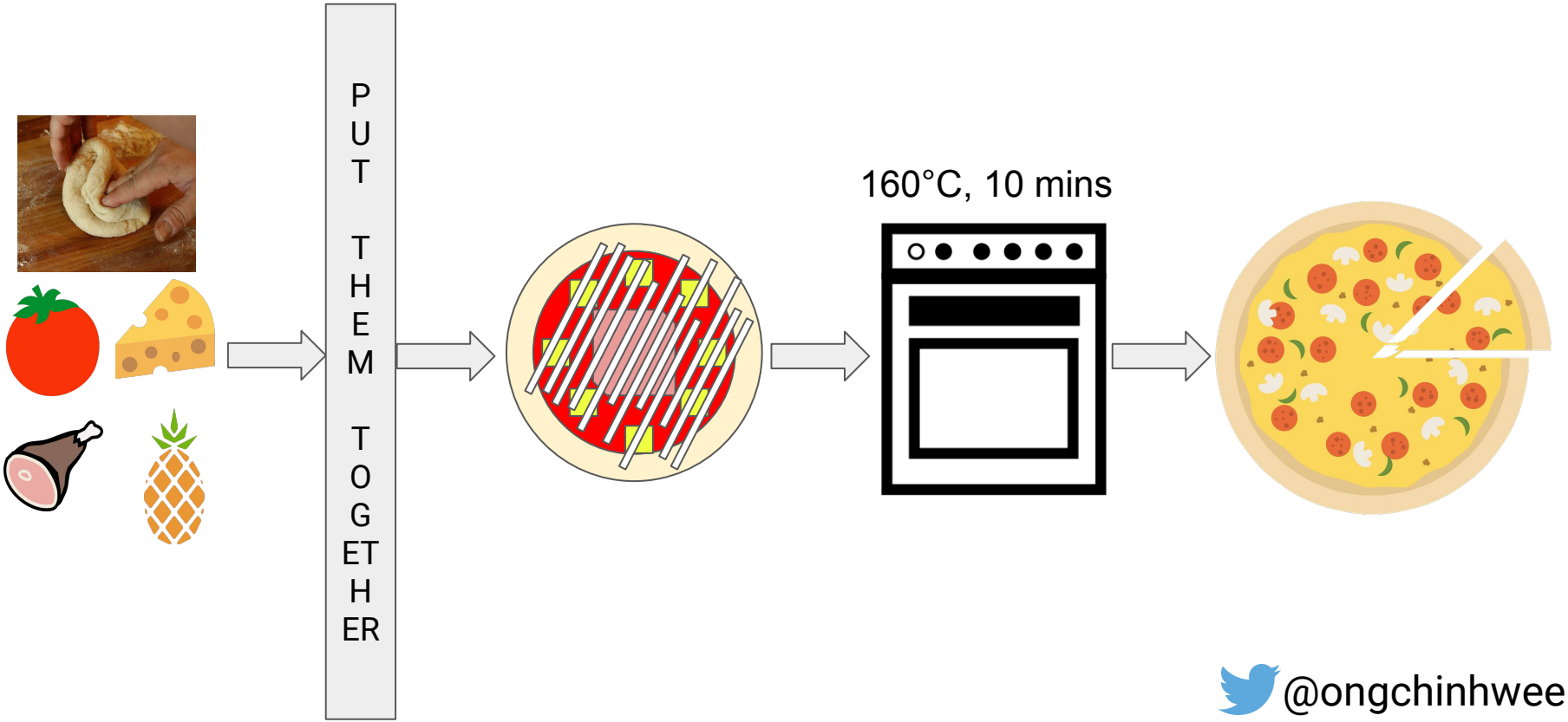
The concept of a “pure function”

- **Pure function**
  - Output depends only on its **input parameters** and its **internal algorithm**
  - **No side effects**

⇒ same function f, same input parameter x ⟶ **same result y**

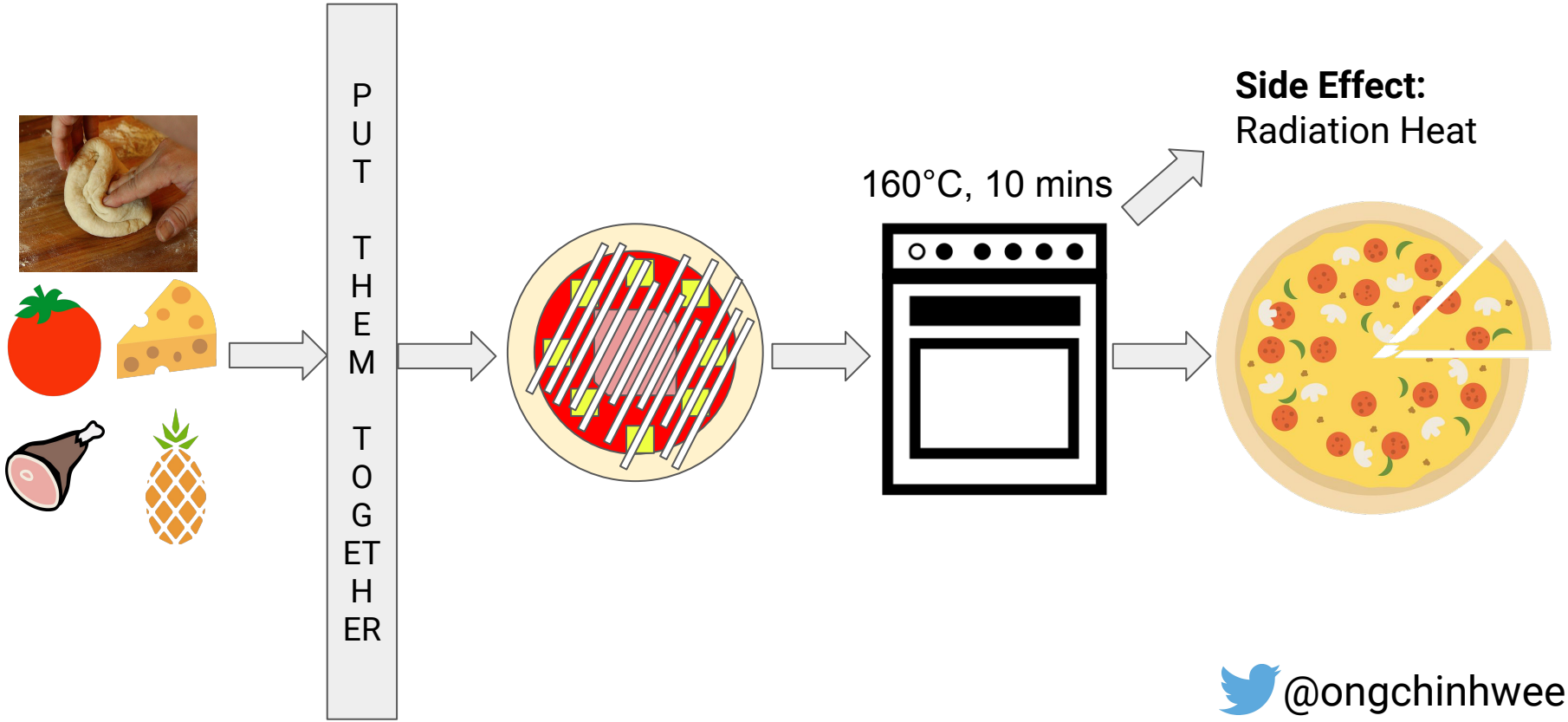
<u>regardless of number of invocations</u>

Pure Function: Making Pizza



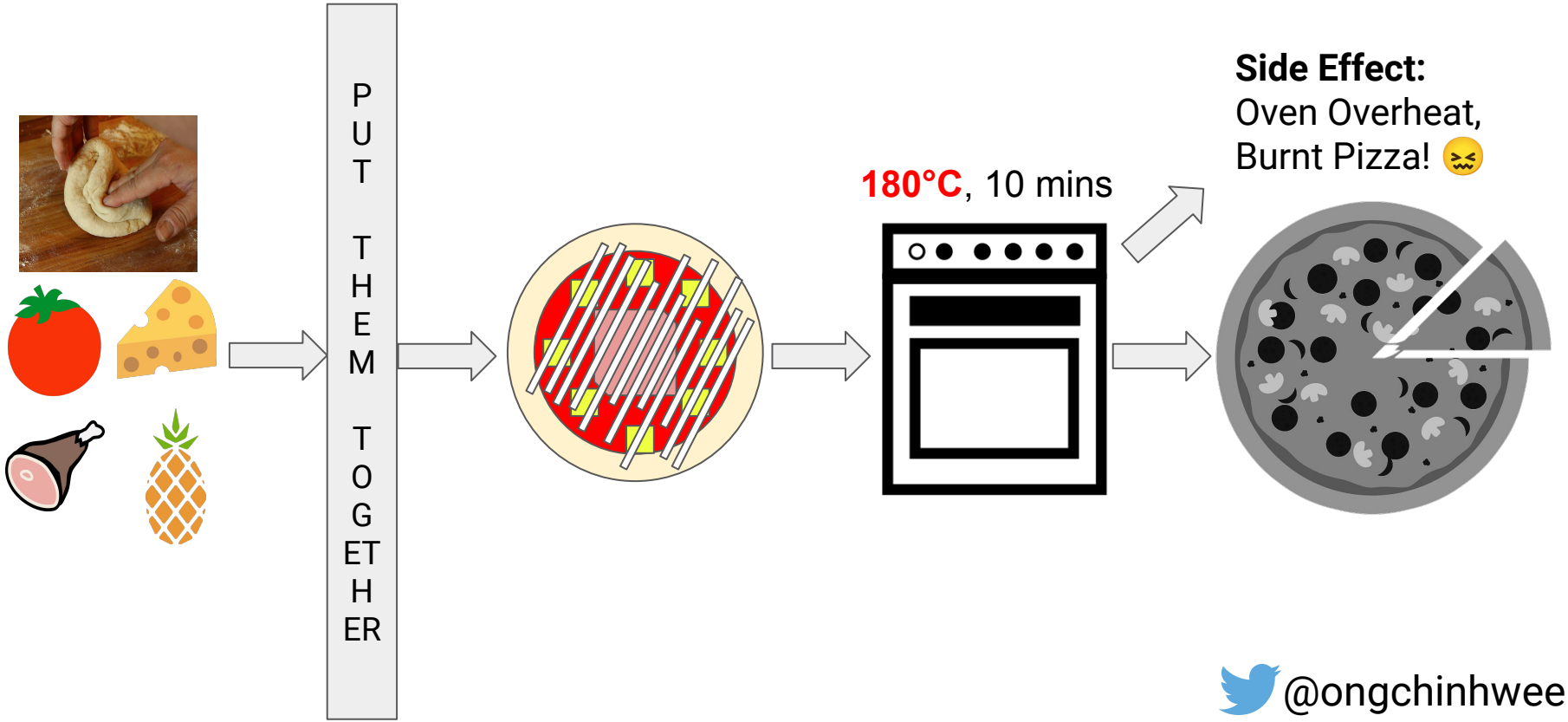
@ongchinhwee

“Impure” Function: Making Pizza with Side Effects



@ongchinhwee

“Impure” Function: Making Pizza with Side Effects

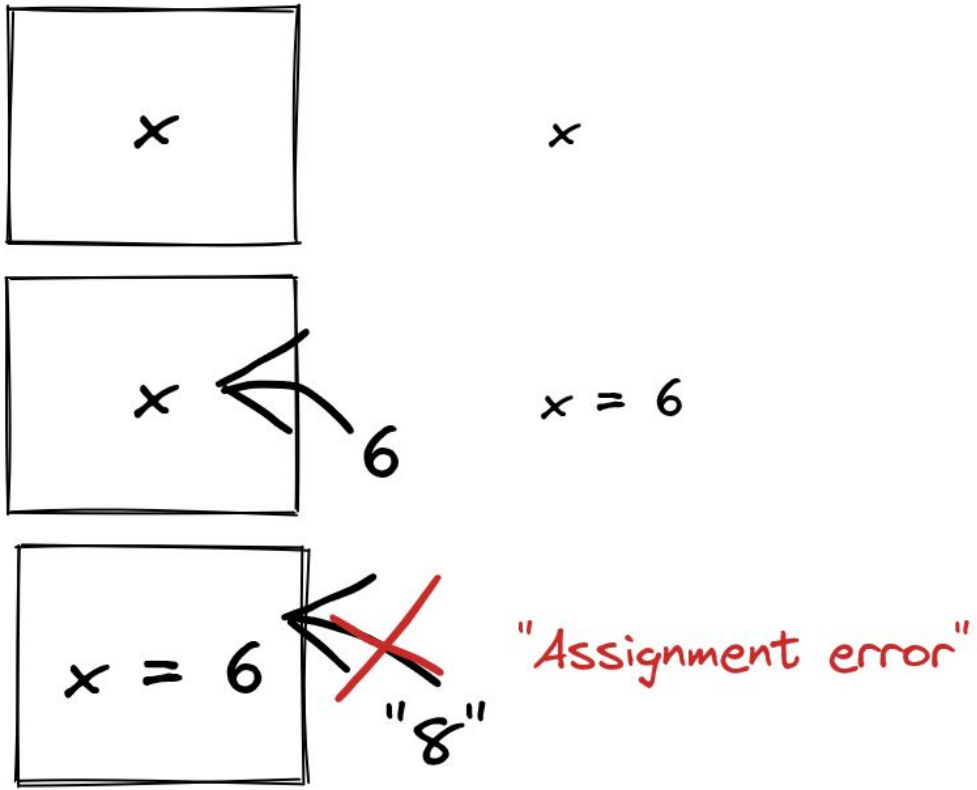


@ongchinhwee

What is a side effect?

- A function with **side effects** changes state outside the local function scope
  - Examples:
    - modifying a variable or data structure in place
    - modifying a global state
    - performing any I/O operation
    - throwing an exception with an error

The concept of Immutability



@ongchinhwee

The concept of Immutability

- **Immutability of an assigned variable**
  - Once a value is assigned to a variable, the **state of the variable cannot be changed**.

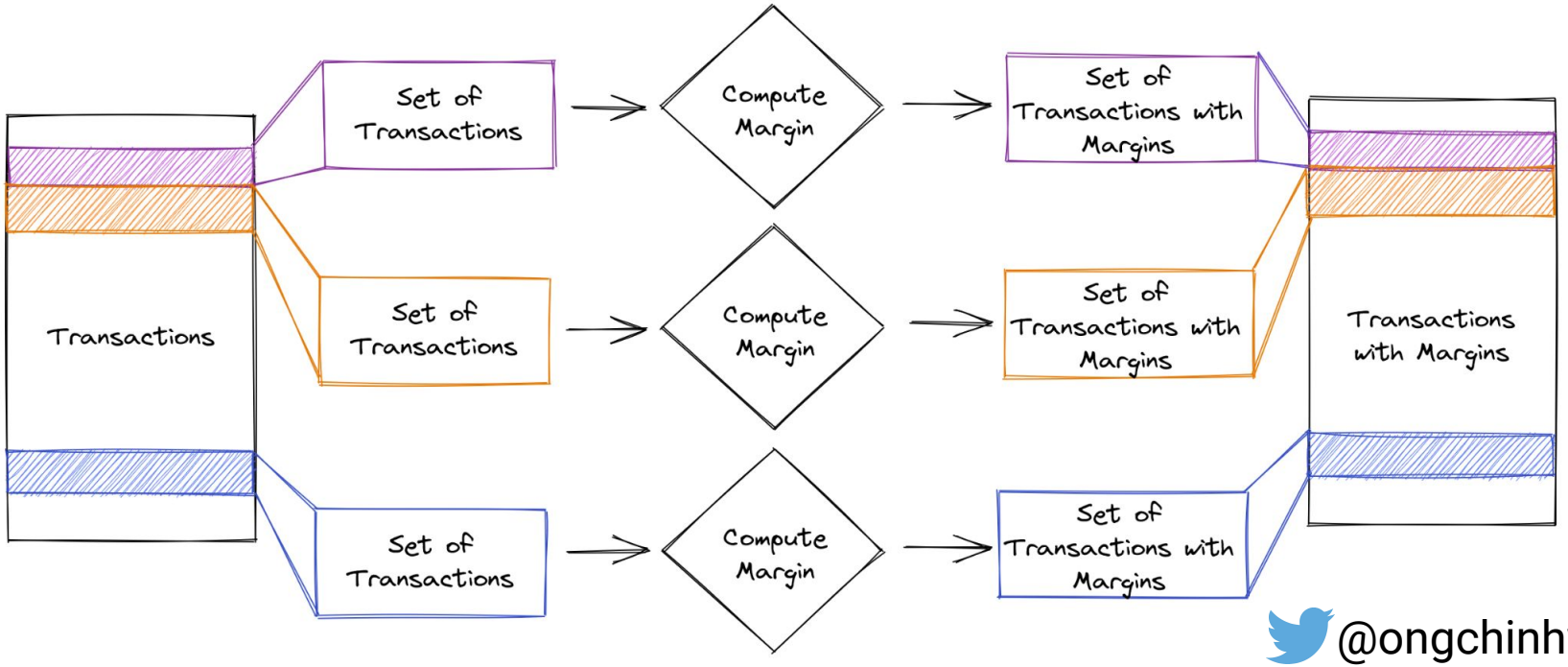
⇒ Disciplined **state management**

⇒ **Prevents side effect** resulting from state change → “pure function”

@ongchinhwee

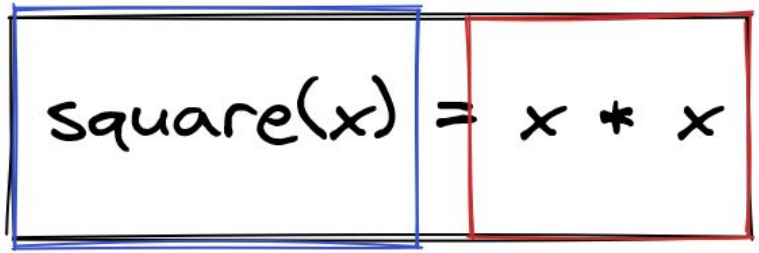
The concept of Immutability: Key Implication

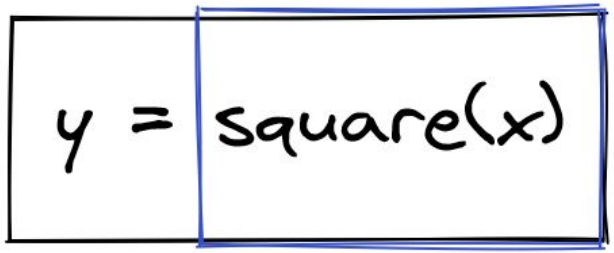
- **Key implication: Ease of writing parallel/concurrent programs**

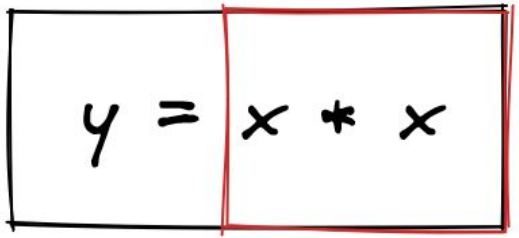


@ongchinhwee

Referential Transparency



=> 

equivalent to 

@ongchinhwee

The concept of Referential Transparency

A function is **referentially transparent** when an expression can be substituted by its equivalent algorithm without affecting the program logic for all programs

@ongchinhwee

Conditions for Referential Transparency

- **Pure function**
- **Deterministic**
  - Expression **always returns** the **same output** given the same input

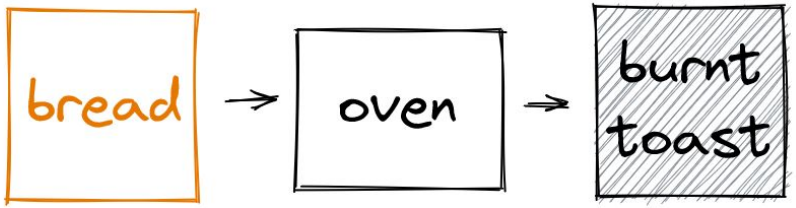
Deterministic vs Non-Deterministic

for all time t:

deterministic

at time t=T:

at time t=T+1:



non-deterministic

@ongchinhwee

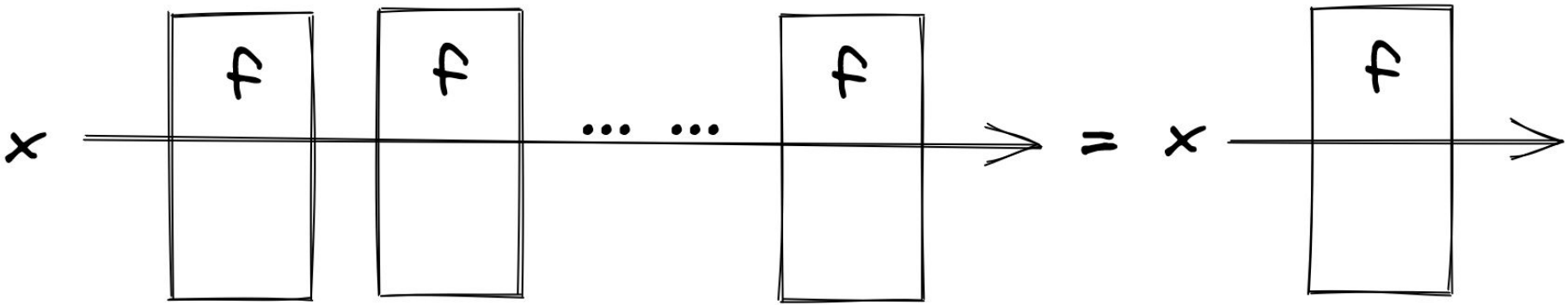
Conditions for Referential Transparency

- **Pure function**
- **Deterministic**
  - Expression returns the **same output** given the **same input**
- **Idempotent**
  - Expression can be applied **multiple times without changing the result beyond its initial application**

@ongchinhwee

Idempotence

For pure functions:

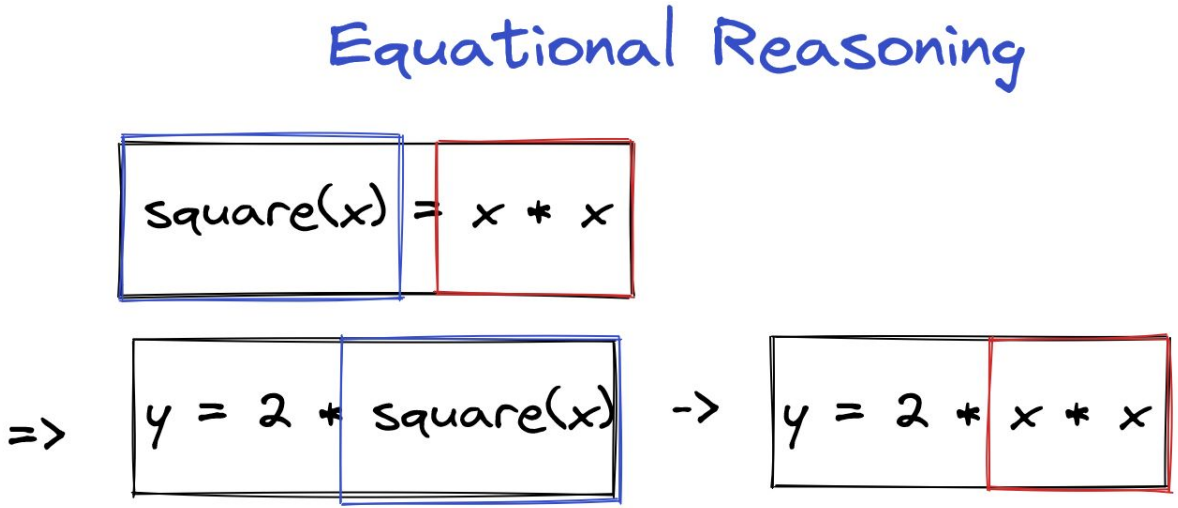


e.g. abs(abs(x)) = abs(x)

@ongchinhwee

Equational Reasoning

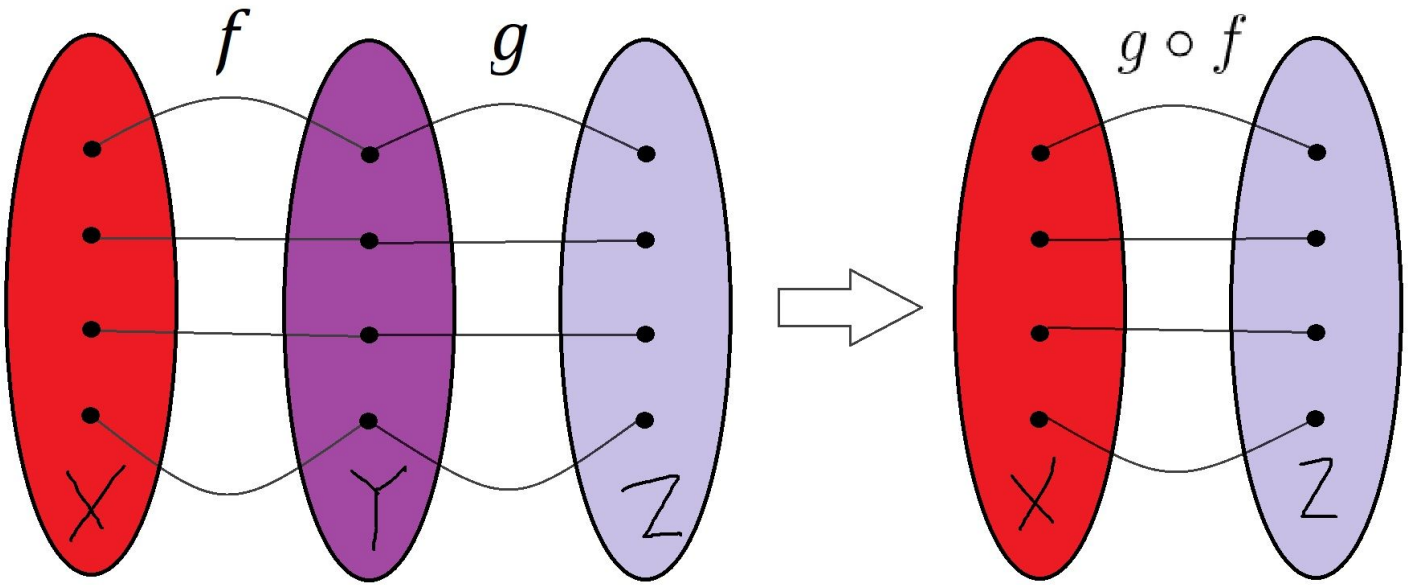
- A key consequent of referential transparency
  - Expression can be **replaced with its <u>equivalent result</u>**



Functional Control Flow

@ongchinhwee

Function Composition



@ongchinhwee

Functions are Values

- In Python, functions are **first-class objects.**
- A function can be:
  - assigned to a variable
  - passed as a parameter to other functions
  - returned as a value from other functions

Higher-order Functions

- Key consequent of **first-class functions**
- A higher-order function has at least one of these properties:
  - Accepts functions as parameters
  - Returns a function as a value

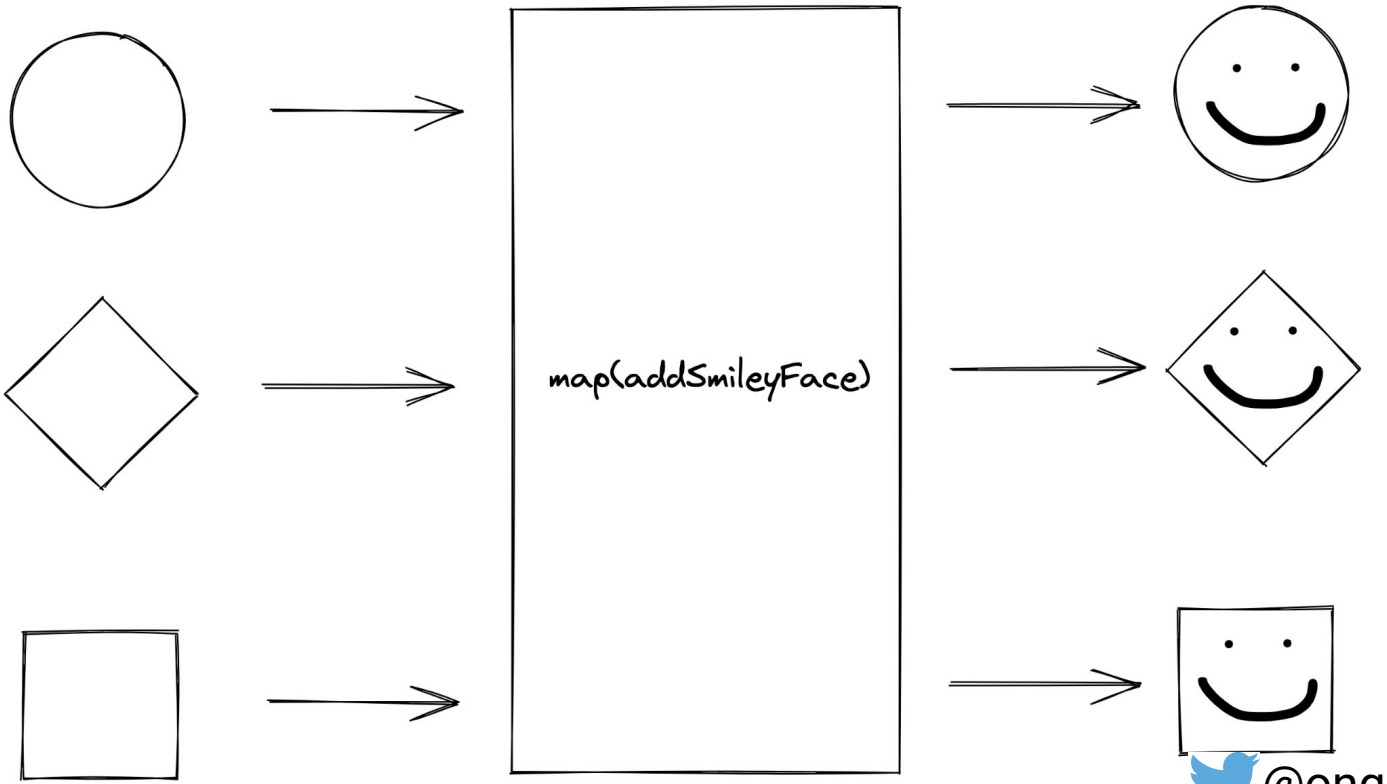
Anonymous Functions

- Also known as **“lambda expressions”** in Python
- Using function as input without defining named function object

```
def main(args):

    collection = [1,2,3,4,5]
    squared = map(lambda x: x * x, collection)
    print(squared)
```

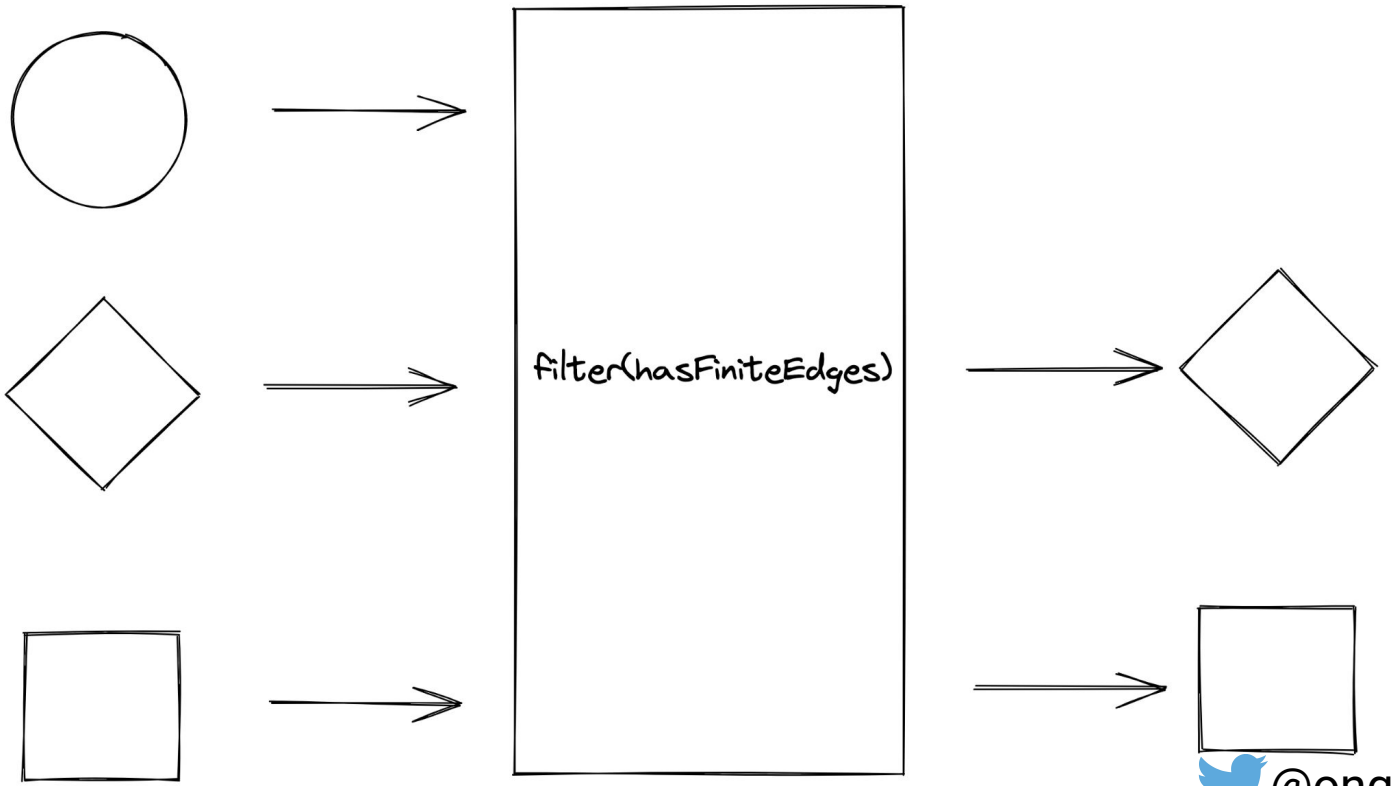
@ongchinhwee

map



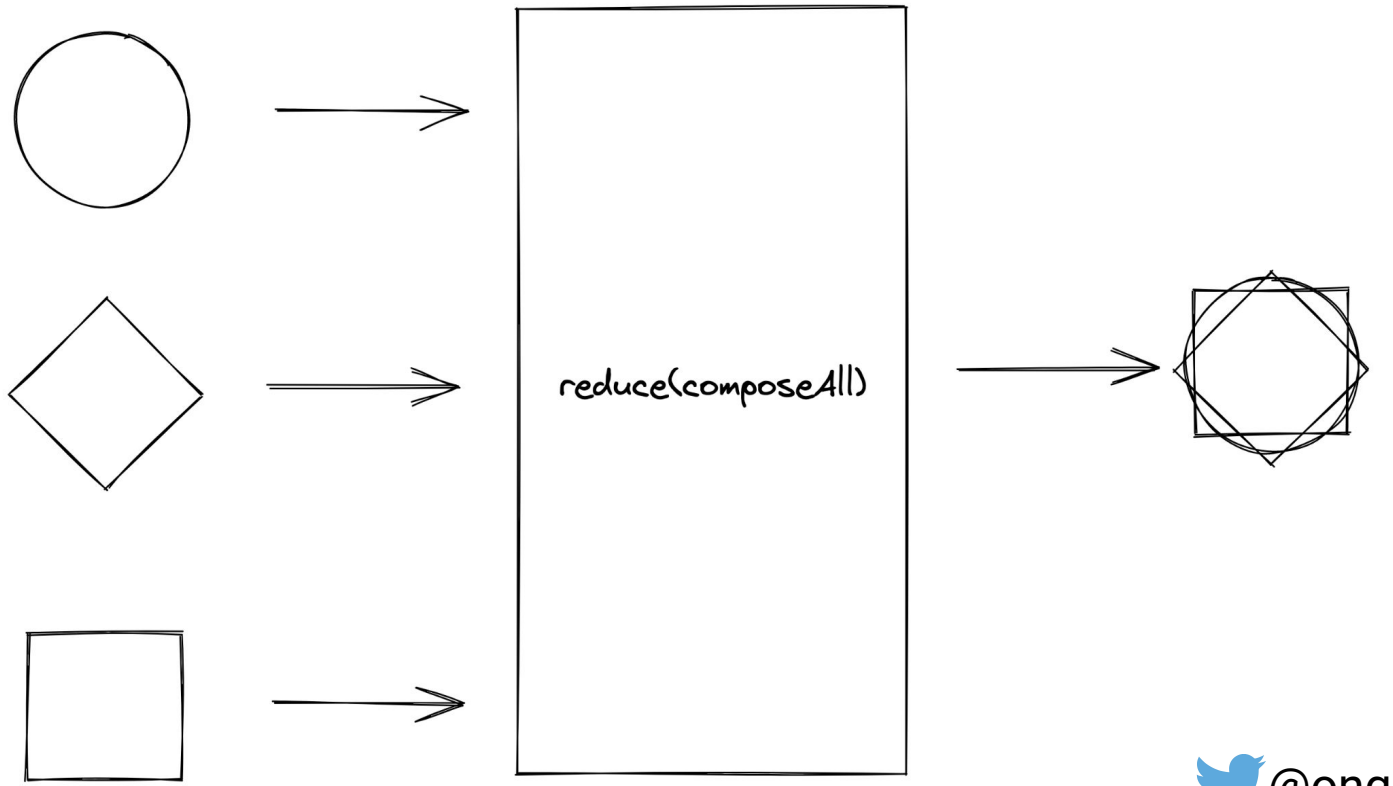
@ongchinhwee

filter



@ongchinhwee

reduce



@ongchinhwee

map/filter/reduce vs for-loops

```python
def square(x):
      return x * x

  def main(args):

      collection = [1,2,3,4,5]
      # initialize list to hold results
      squared_collection = []
      # loop till the end of the collection
      for num in collection:
          # square the current number
          squared = square(num)
          # add the result to list
          squared_collection.append(squared)

      print(squared_collection)
```

```python
def square(x):
    return x * x

def main(args):

    collection = [1,2,3,4,5]
    squared = list(map(square, collection))
    print(squared)
```

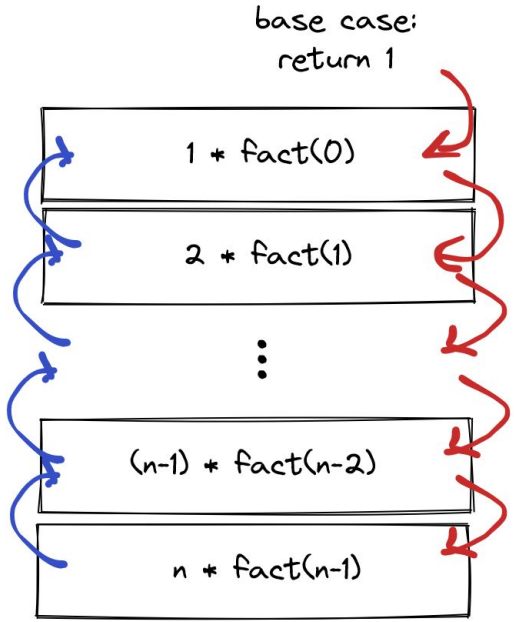
Managing **state changes of mutable variables** in a for-loop

@ongchinhwee

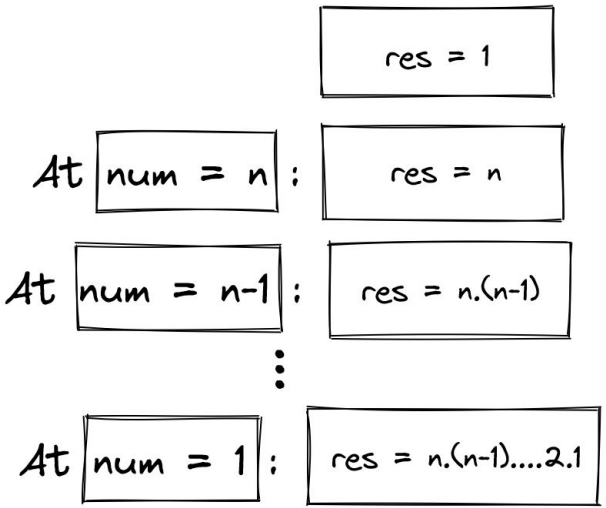
Recursion as a form of “functional iteration”

- Recursion is a form of self-referential **function composition**
  - Takes the **results of itself** as **inputs into another instance of itself**
  - To prevent infinite recursive loop, **base case** required as **terminating condition**

Recursion as a form of “functional iteration”



recursive call stack



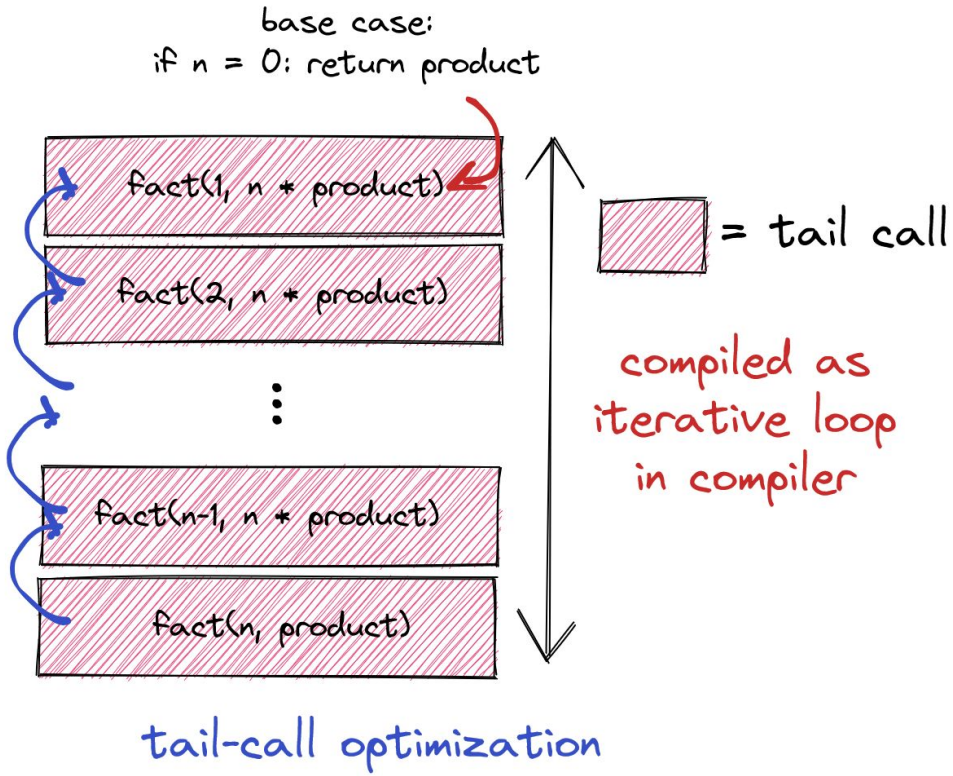
iterative loop

@ongchinhwee

Recursion as a form of “functional iteration”

- **Tail-call optimization**
  - Objective: reduce **stack frame consumption** in call stack
  - **Tail call**: does nothing other than **returning the value of function call**
  - Identify tail calls and compile them to iterative loops

Recursion as a form of “functional iteration”



@ongchinhwee

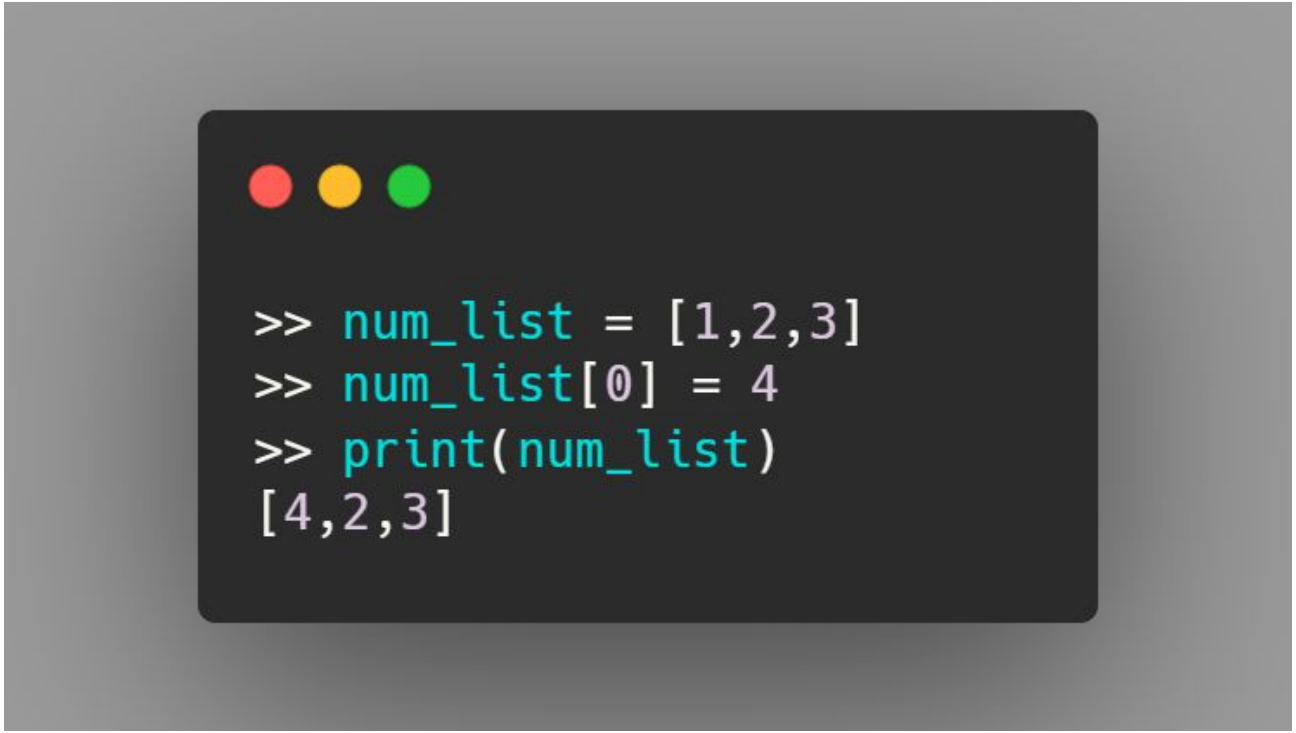
Functional Design Patterns for Data Pipeline Design

@ongchinhwee

Immutable Data Structures

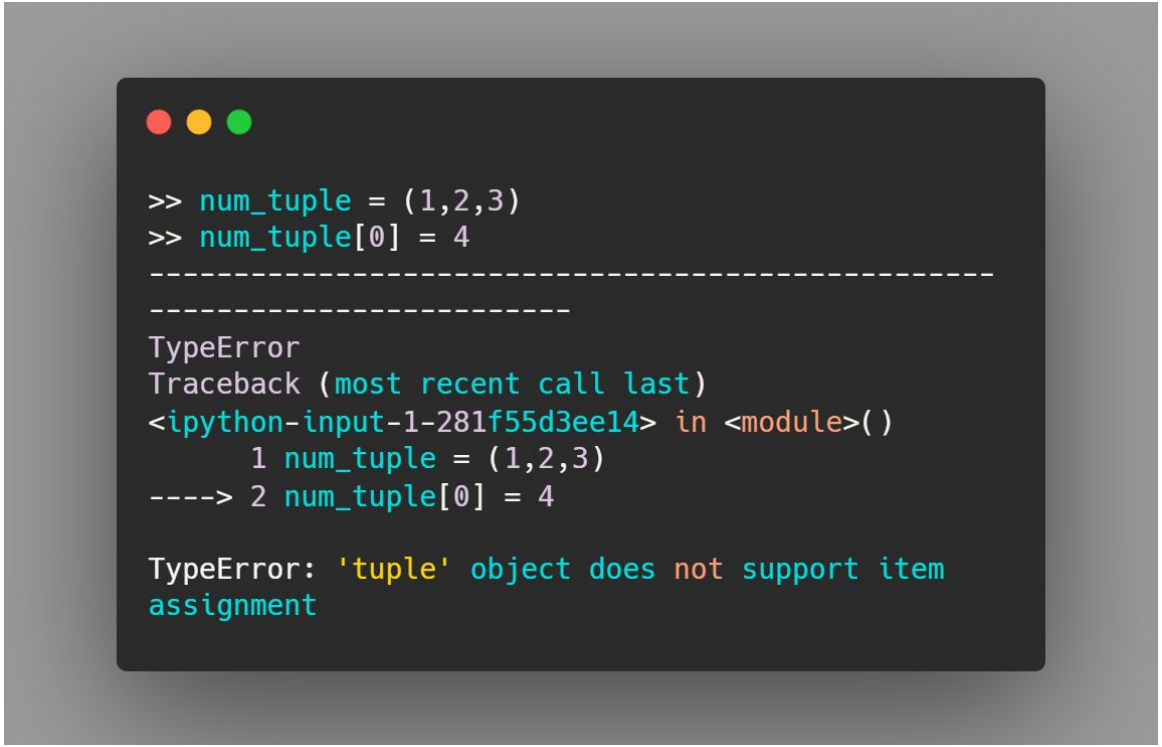
- Once an immutable data structure is created, it **cannot be changed**
- Benefits:
  - Easier to **reason** - “what you see is what you get”
  - Easier to **test** - worry about the logic, not the state
  - **Thread-safe** - easier for parallelism

Tuple vs **List**



@ongchinhwee

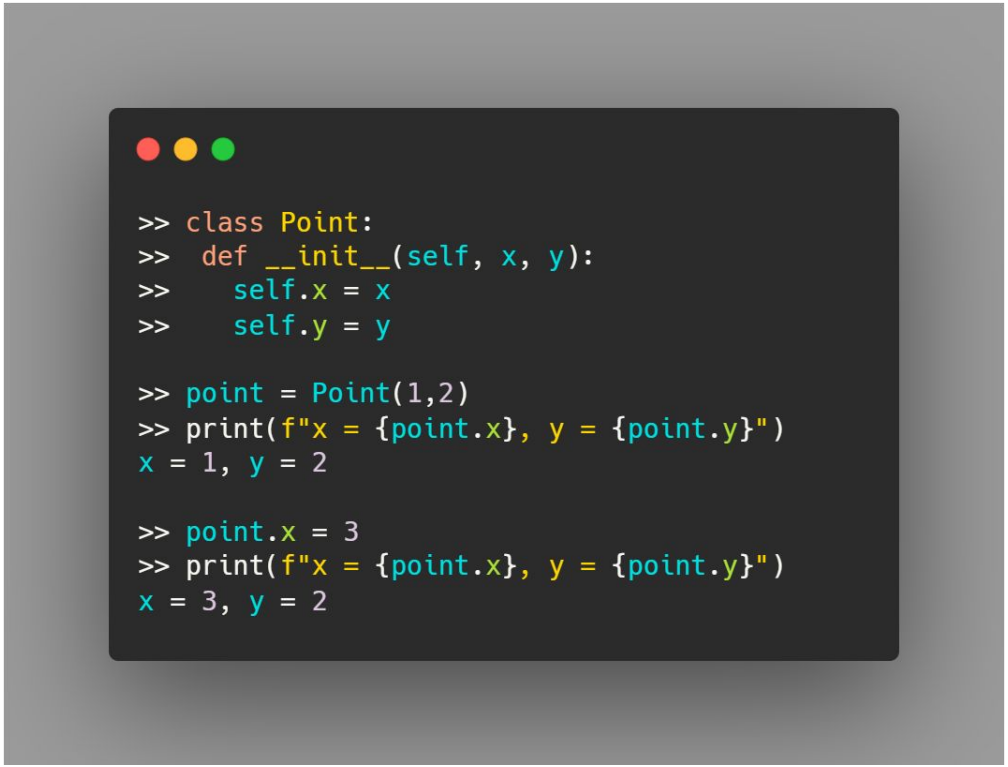
Tuple vs List



```
>> num_tuple = (1,2,3)
>> num_tuple[0] = 4
---------------------------------------------------
------------------------
TypeError
Traceback (most recent call last)
<ipython-input-1-281f55d3ee14> in <module>()
      1 num_tuple = (1,2,3)
----> 2 num_tuple[0] = 4

TypeError: 'tuple' object does not support item
assignment
```

@ongchinhwee

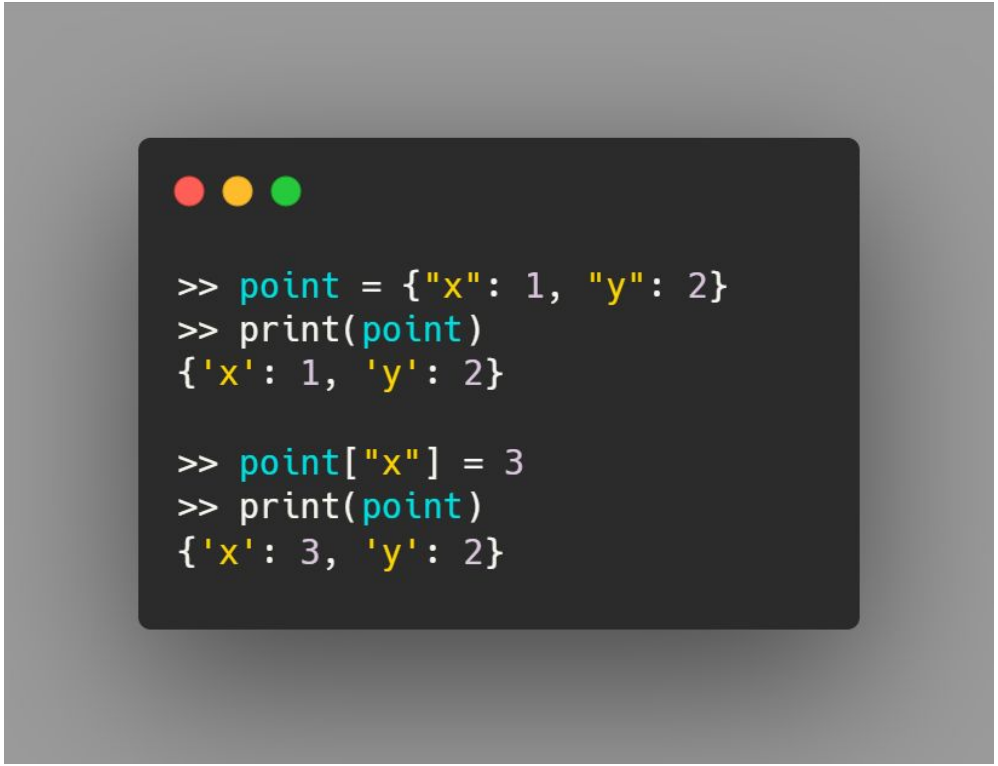
Namedtuple vs **Class** vs Dictionary



```
>> class Point:
>>  def __init__(self, x, y):
>>    self.x = x
>>    self.y = y

>> point = Point(1,2)
>> print(f"x = {point.x}, y = {point.y}")
x = 1, y = 2

>> point.x = 3
>> print(f"x = {point.x}, y = {point.y}")
x = 3, y = 2
```

@ongchinhwee

Namedtuple vs Class vs **Dictionary**



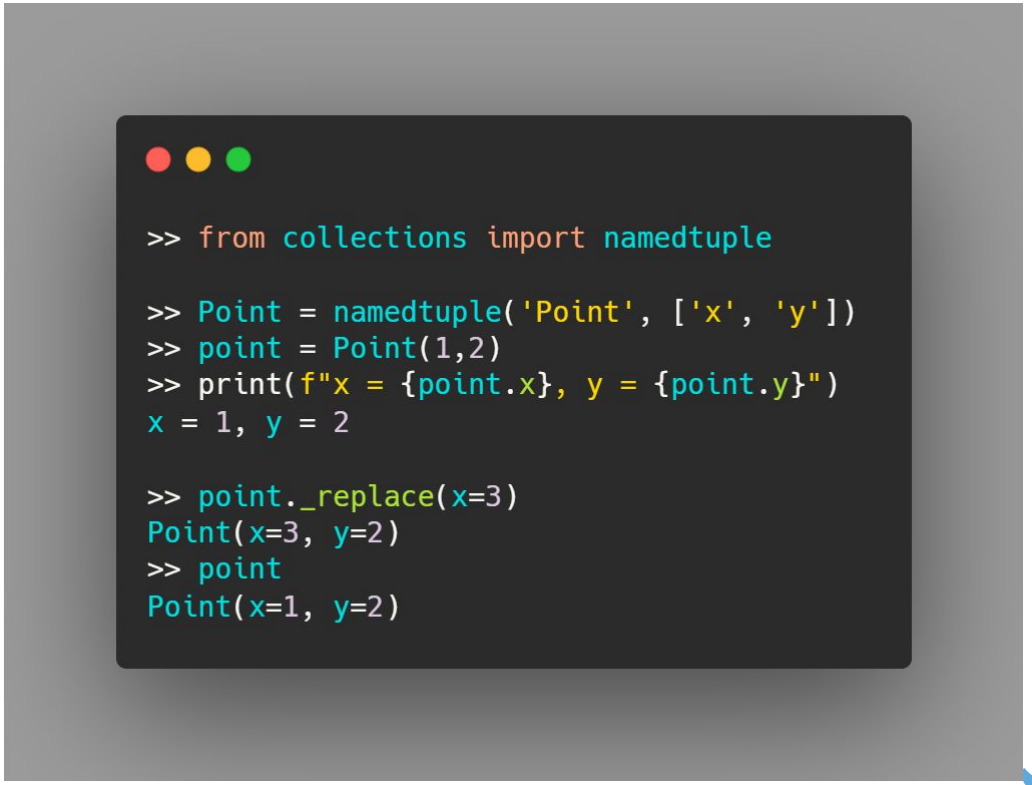
@ongchinhwee

Namedtuple vs Class vs Dictionary

```
>> from collections import namedtuple

>> Point = namedtuple('Point', ['x', 'y'])
>> point = Point(1,2)
>> print(f"x = {point.x}, y = {point.y}")
x = 1, y = 2

>> point.x = 3
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-3587ee6c0406> in <module>()
      4 point = Point(1,2)
      5 print(f"x = {point.x}, y = {point.y}")
----> 6 point.x = 3

AttributeError: can't set attribute
```

@ongchinhwee

Namedtuple vs Class vs Dictionary

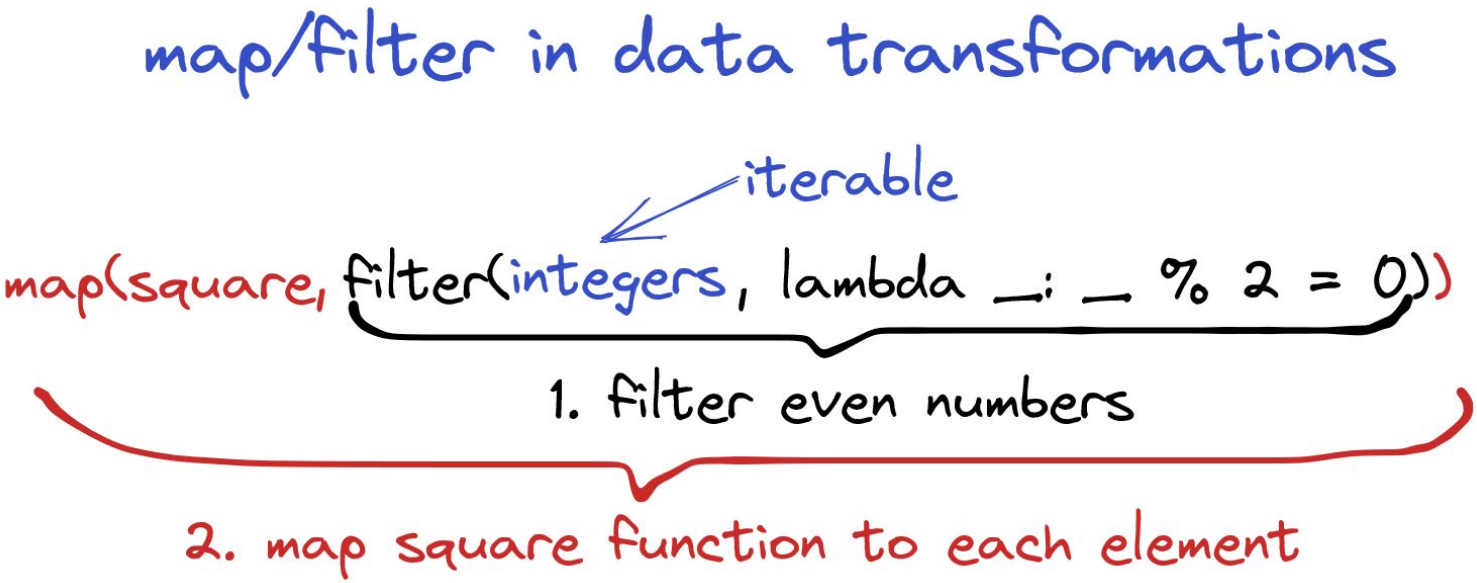


```
>> from collections import namedtuple

>> Point = namedtuple('Point', ['x', 'y'])
>> point = Point(1,2)
>> print(f"x = {point.x}, y = {point.y}")
x = 1, y = 2

>> point._replace(x=3)
Point(x=3, y=2)
>> point
Point(x=1, y=2)
```

@ongchinhwee

Data Transformations

- map/filter in **data transformations**



@ongchinhwee

Data Transformations

- map/filter (and its derivatives) in **data transformations**
  - Keeping **data** and **transformation logic** separate
    - Improved code reusability with better transparency of transformation logic

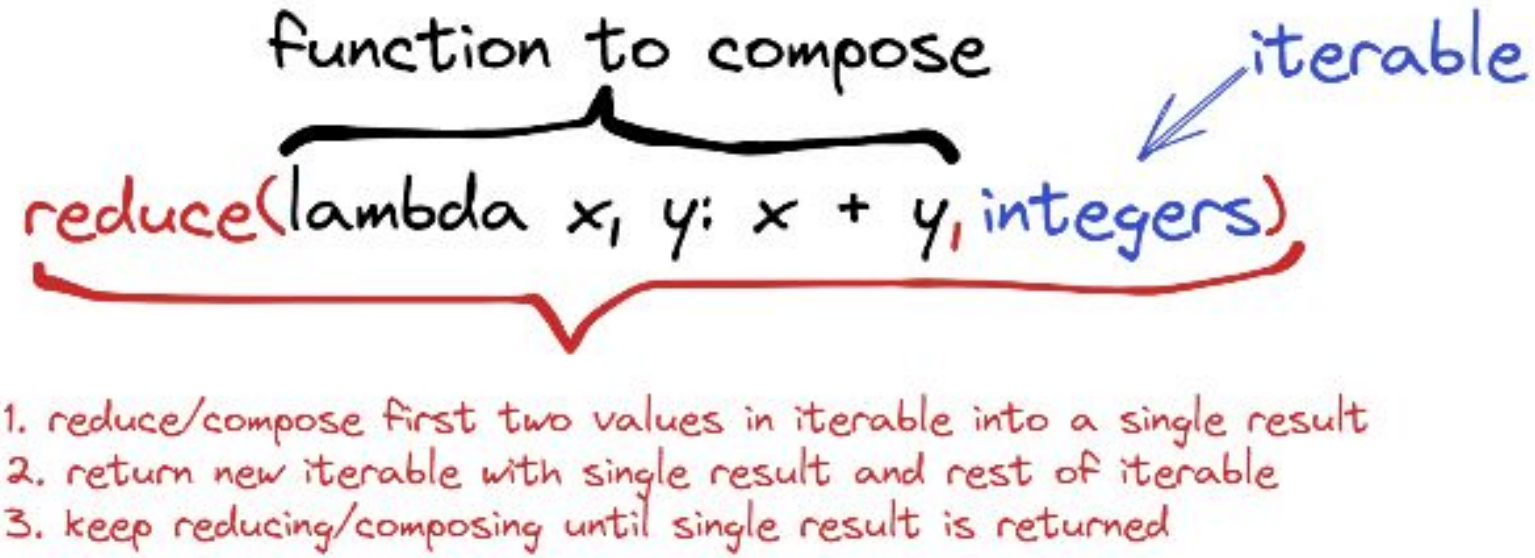
Extending map/filter to **parallel/concurrent programming**

```
from multiprocessing import Pool

def square_even(x):
  if x % 2 == 0: return 0
  return x * x

with Pool(processes=4) as pool:
  # returns iterable of squared even numbers
  res_map = pool.map(square_even, range(1000))
  # filter non-zero results from iterable
  res = [_ for _ in res_map if _ != 0]
```

@ongchinhwee

Data Actions / Aggregations

- reduce in **data actions / aggregations**



Data Actions / Aggregations

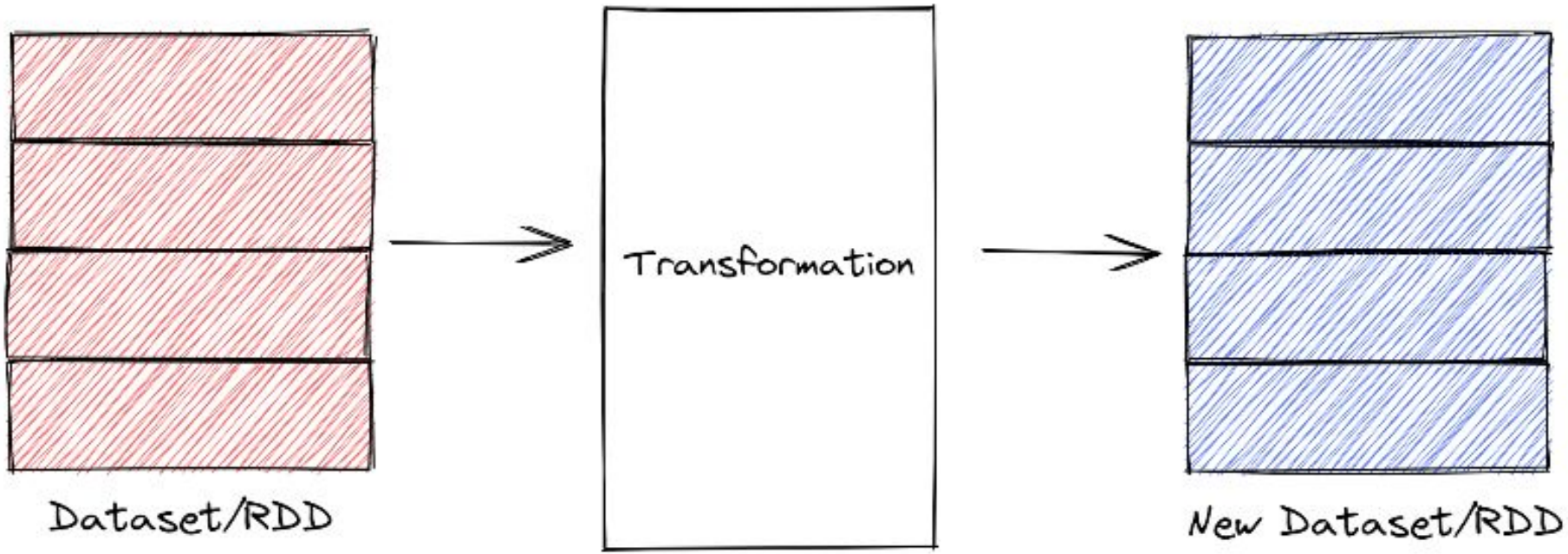
- reduce (and its derivatives) in **data actions / aggregations**
  - **Transformations** first, **actions** last
    - Transformation logic can be applied to each element / partition
    - Actions / aggregations consolidates results from partitions

@ongchinhwee

Functional Design Patterns in Apache Spark

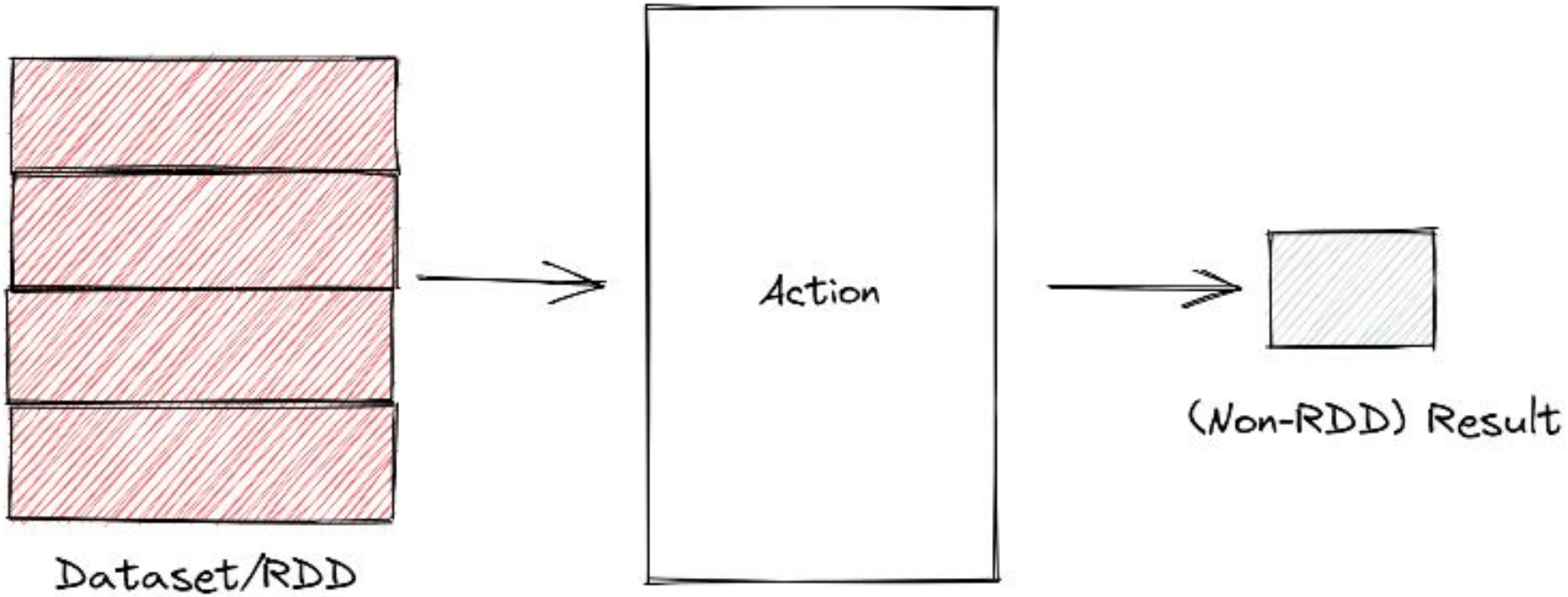
- **Resilient Distributed Datasets (RDDs)**
  - Low-level data abstraction in Apache Spark
  - **Immutable** and read-only
  - Designed for fault-tolerant parallel operations with logical **partitioning** across nodes

Transformations vs Actions in Apache Spark



@ongchinhwee

Transformations vs **Actions** in Apache Spark



@ongchinhwee

Structural Pattern Matching (PEP 634)

- **Python 3.10 feature** inspired by similar syntax with Scala
- Especially useful for **conditional matching of data structure patterns**

```
match Item:
    case Something:
        do_something()
```

@ongchinhwee

Structural Pattern Matching (PEP 634)

- match/case expressions vs if/elif/else

```python
def compute_rebate(revenue, perc):
    match revenue, perc:
        case float(revenue), float(perc):
            return revenue * perc / 100
        case float(revenue), _:
            return revenue * 0.05
        case _, _:
            return None
```

```python
def compute_rebate(revenue, perc):
    if isinstance(revenue, float) and
isinstance(perc, float):
        return revenue * perc / 100
    elif isinstance(revenue, float)
        return revenue * 0.05
    else:
        return None
```

@ongchinhwee

Structural Pattern Matching (PEP 634)

- Pattern matching for **maintainability of data schema**

```python
from dataclasses import dataclass

@dataclass(frozen='true')
class InventoryItem:
  item_name: str
  unit_price: float
  quantity: int = 0

@dataclass(frozen='true')
class SaleItem:
  item_name: str
  unit_price: float
  discount: float = 5.0

def get_greeting(request)
  match request:
    case InventoryItem(item_name, unit_price, quantity):
      return f"Inventory: {quantity} {item_name} at unit cost ${unit_price}"
    case SaleItem(item_name, unit_price, discount):
      return f"Selling {item_name} at {unit_price * discount / 100.0})"
    case _:
      return "Welcome to Uniqlo!"
```

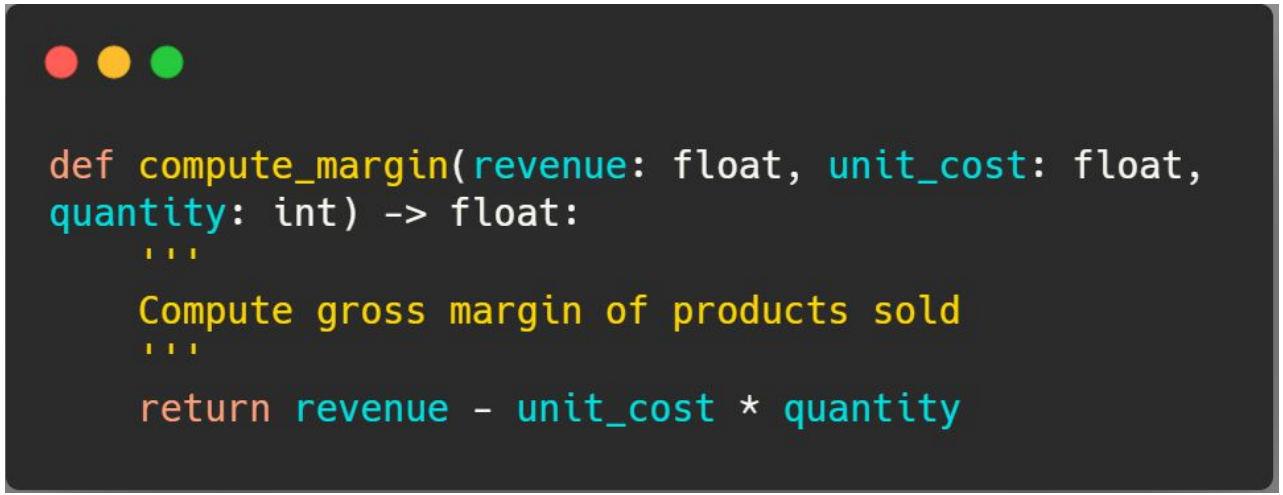
Note: Example based on **case classes and pattern matching syntax in Scala**

Dataclasses used as the Python equivalent of Scala case classes

@ongchinhwee

Type Systems

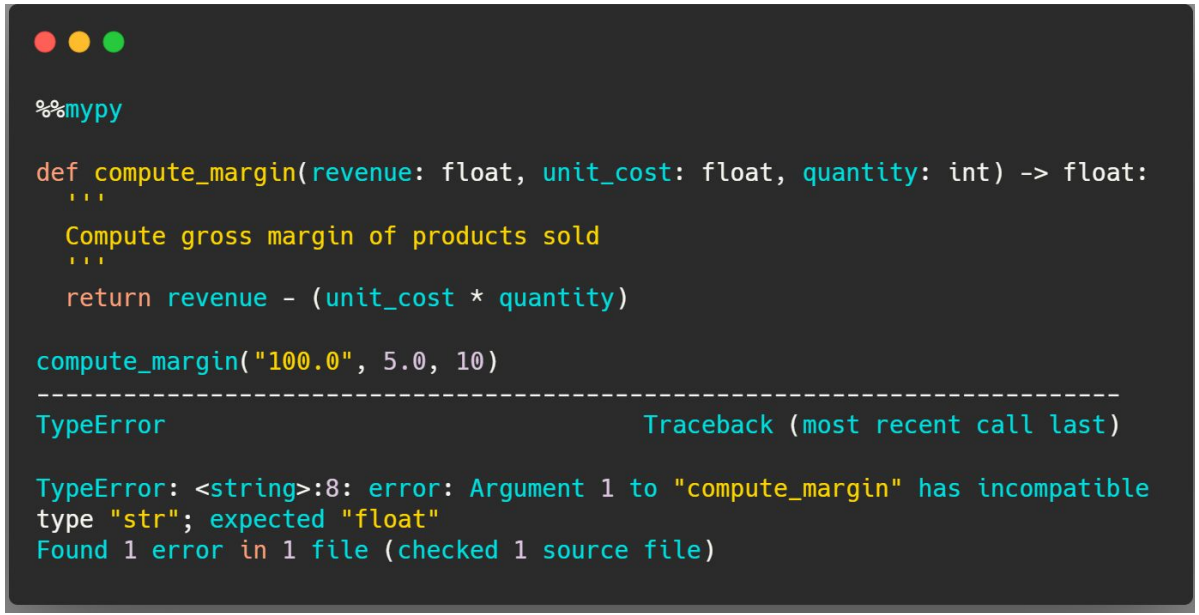
- Python has support for **type hints** (though not enforced in runtime)



```
def compute_margin(revenue: float, unit_cost: float,
quantity: int) -> float:
    '''
    Compute gross margin of products sold
    '''
    return revenue - unit_cost * quantity
```

Type Systems

- **Type checking** with mypy



```
%%mypy

def compute_margin(revenue: float, unit_cost: float, quantity: int) -> float:
  '''
  Compute gross margin of products sold
  '''
  return revenue - (unit_cost * quantity)

compute_margin("100.0", 5.0, 10)
```

```
----------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

TypeError: <string>:8: error: Argument 1 to "compute_margin" has incompatible
type "str"; expected "float"
Found 1 error in 1 file (checked 1 source file)
```

@ongchinhwee

Type Systems

- **Type checking** with mypy
- Preventing bugs at runtime by ensuring **type safety and consistency** across the data pipeline

Can we write a purely functional data pipeline in Python?

@ongchinhwee

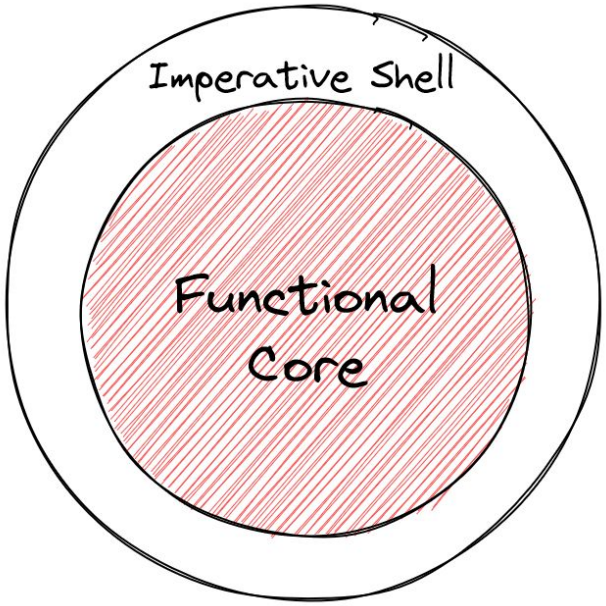
Can we write a purely functional data pipeline in Python?

Short Answer: Not really.

@ongchinhwee

“Functional Core, Imperative Shell”

- **I/O operations** still needed for **reading and writing data** outside of the application domain
- Keeping **core domain logic** and **infrastructure code** separate



Ref: Gary Bernhardt's PyCon 2013 talk on "Boundaries"

@ongchinhwee

“Functional Core, Imperative Shell”

```python
def compute_revenue(sales_item: list) -> float:
  price, qty = sales_item[1], sales_item[2]
  return [sales_item[0], price, qty, price * qty]

def main():
  # import modules
  import pandas as pd
  import numpy as np

  # I/O layer to read data into program
  data = pd.read_csv('data/sales.csv', index_col=0)
  data_cols = data.columns.tolist()

  # Functional layer for computation logic
  data_array = data.to_numpy()
  revenue_array = list(map(compute_revenue, data_array))
  revenue_cols = data_cols + ['revenue']
  revenue_df = pd.DataFrame(revenue_array, columns=revenue_cols)

  # I/O layer to write data outside of program
  revenue_df.to_csv('data/revenue.csv', index=False)
```

@ongchinhwee

Key Takeaways

- Adopt **functional design patterns** when designing data pipelines at scale (parallel and distributed workflows)
  - Reproducible
  - Scalable
  - Maintainable
- **“Functional Core, Imperative Shell”** to manage side effects separately from data pipeline logic

Reach out to me!

 : ongchinhwee

 : @ongchinhwee

 : hweecat

 : https://ongchinhwee.me

And check out my ongoing series on Functional Programming at:

https://ongchinhwee.me/tag/functional-programming

 @ongchinhwee