Python: Day 03

Professional Development

Previous Agenda

01

Data Structures

Grouped data

02

Packaging

Handling Python Files

03

String+

Format & Methods

04

File Manage

Persistent Storage

05

Lab Session

Culminating Exercise

Agenda

01

Introduction

Objects and Classes

02

Key Concepts

Concept Organization

03

Maintainability

Development Guidelines

04

Learn: Streamlit

Making Web Applications

05

Lab Session

Culminating Exercise

01

Introduction

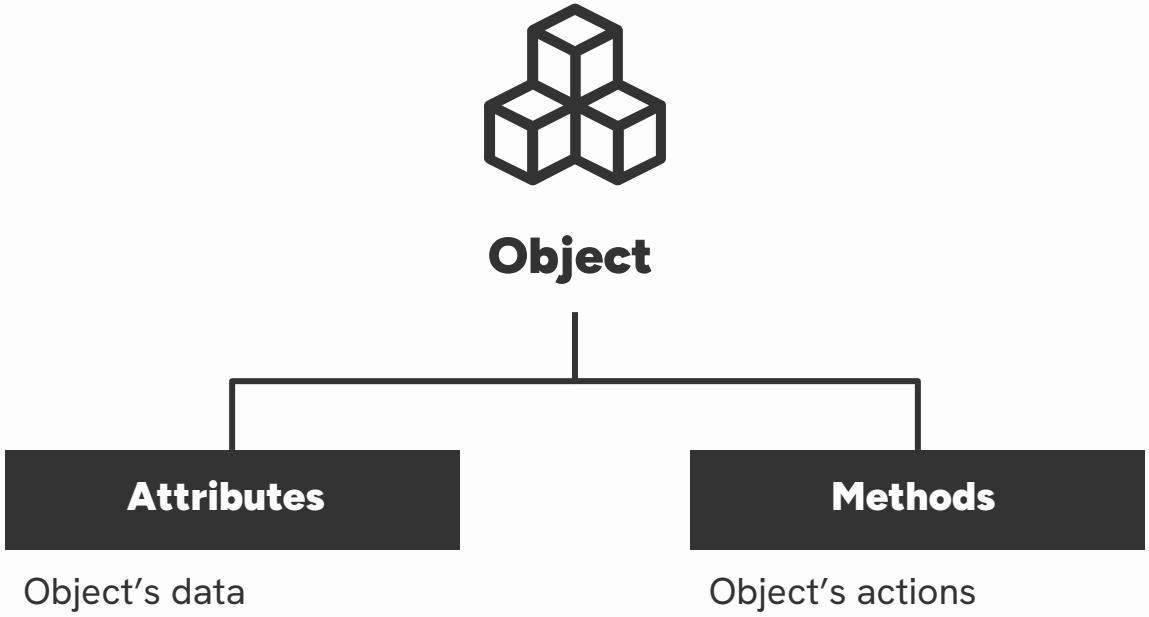
Understanding object and classes

Motivation

Bridging concepts into manageable code blocks

What makes something something ?


PARKER
PARKER



Has → Is

Functional Identity

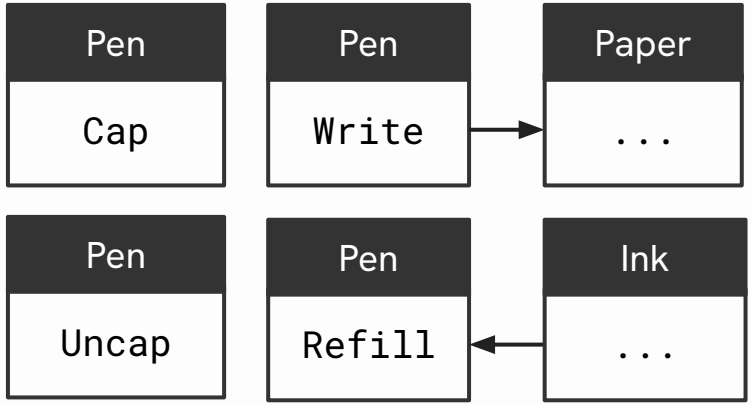
Attributes

- Attributes are unique to one object

Pen	
Brand	Pilot
Color	Black
Capped	False

Methods

- Methods can change itself or others



Object Similarities

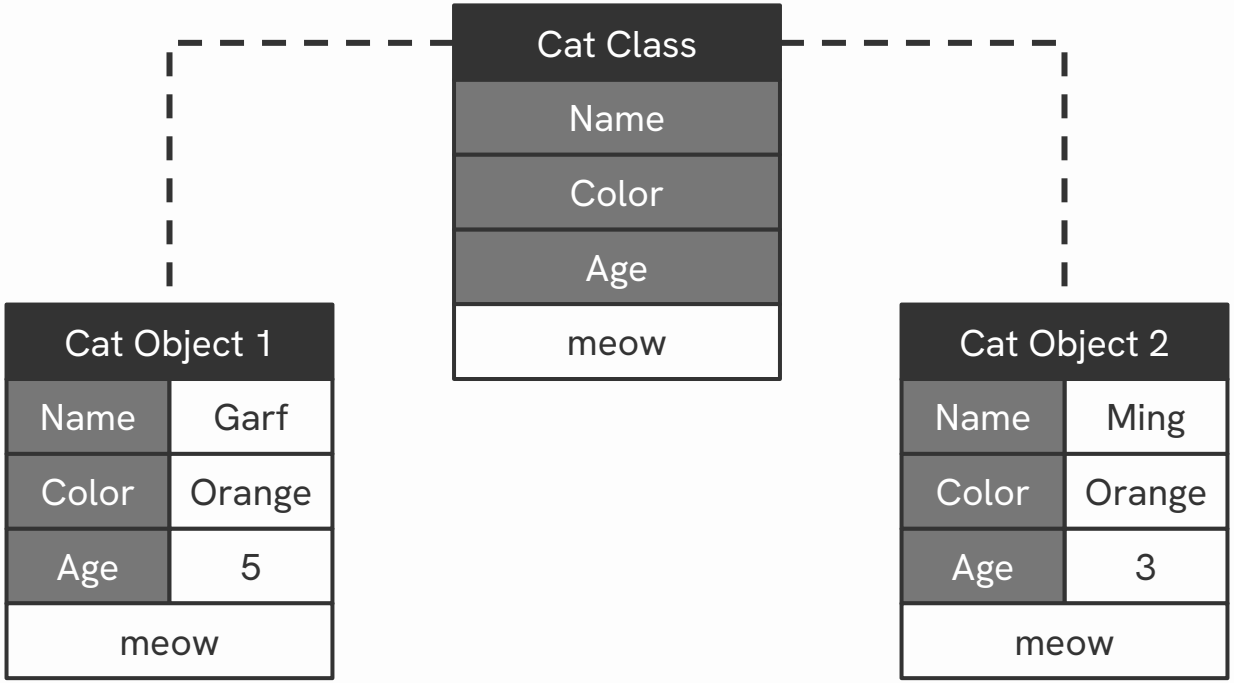
cat1	
Name	Garf
Color	Orange
Age	5
meow	

cat2	
Name	Ming
Color	Orange
Age	3
meow	

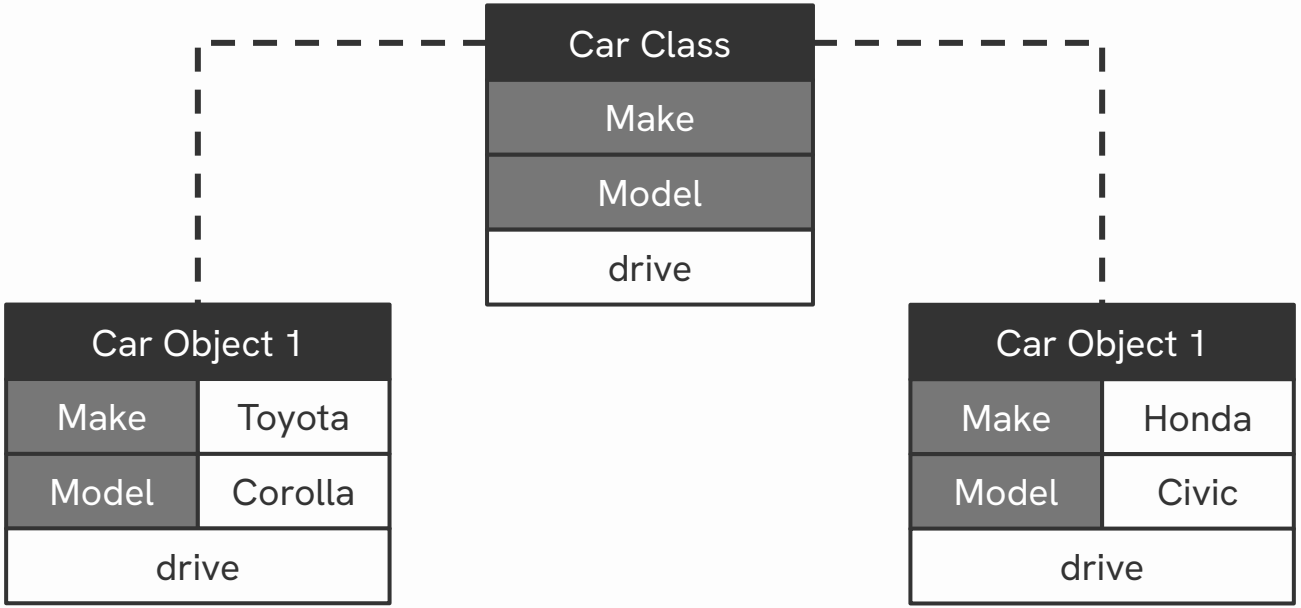
cat3	
Name	Mona
Color	Black
Age	2
meow	

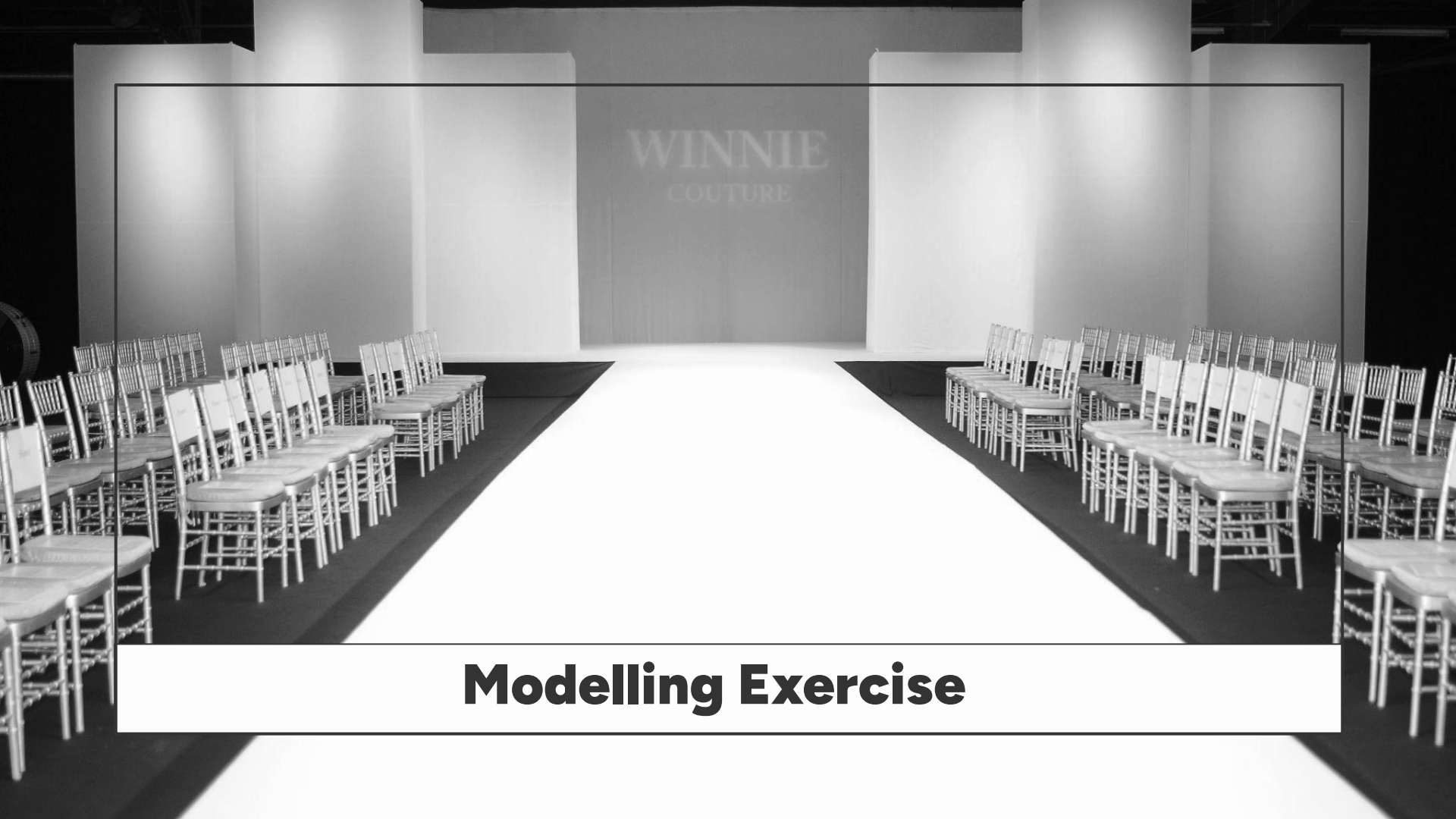
What makes them different/similar?

Classes to Objects

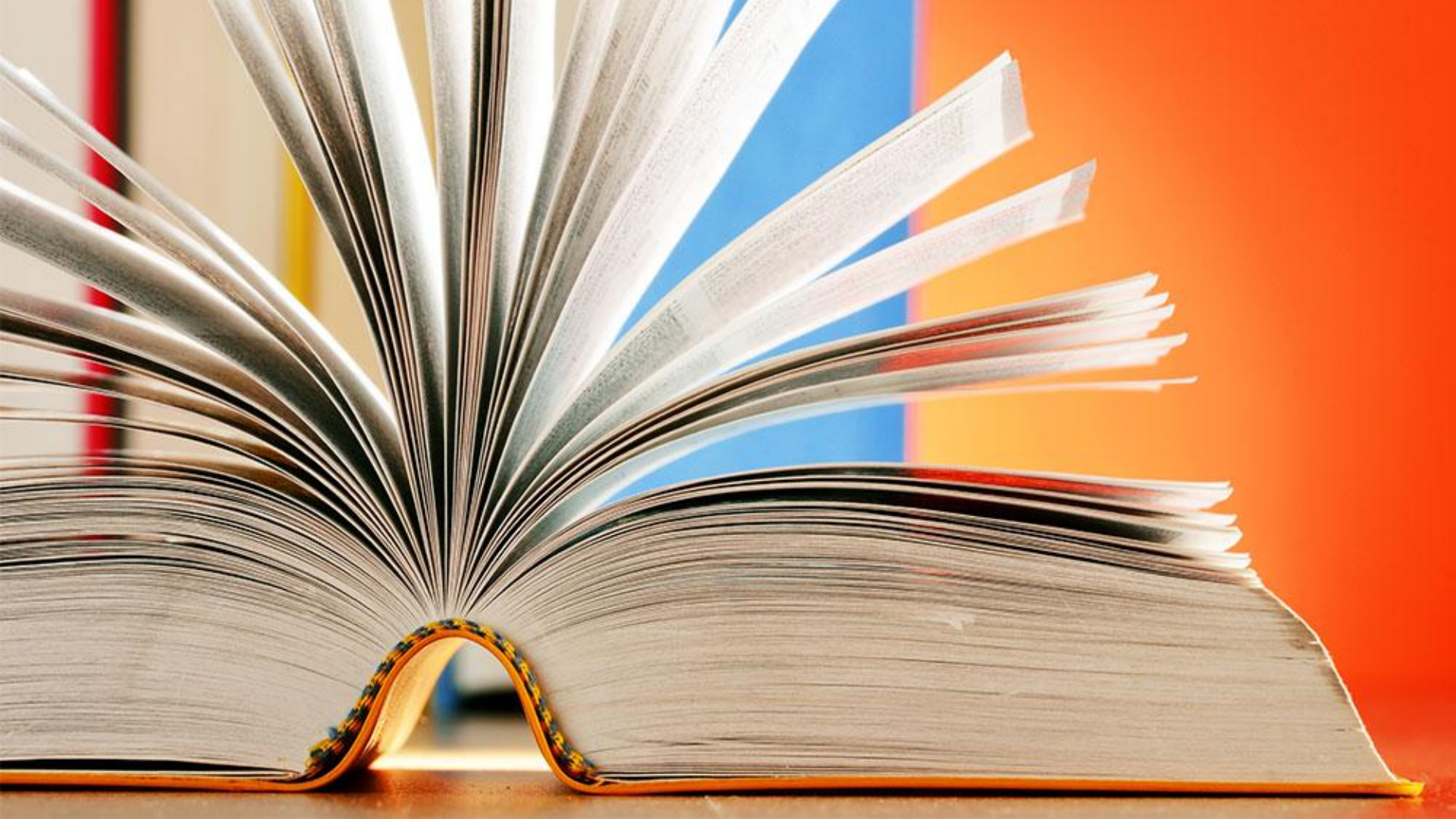


Classes to Objects



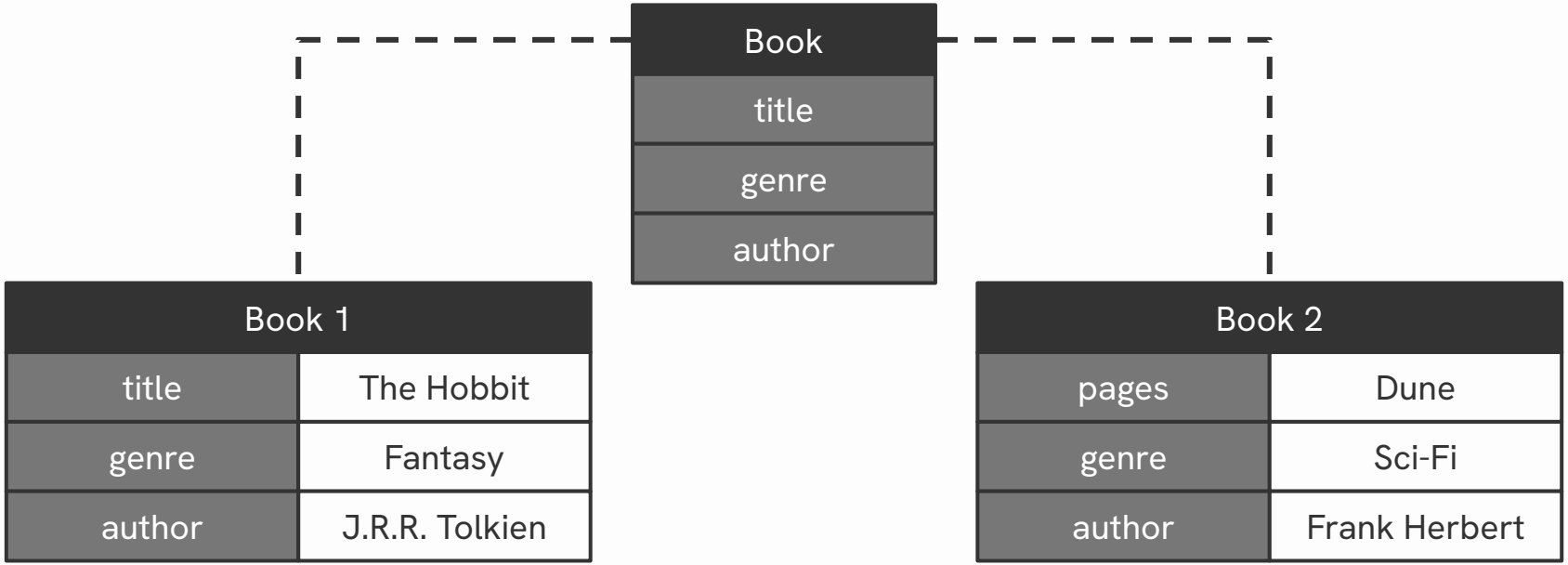


Modelling Exercise

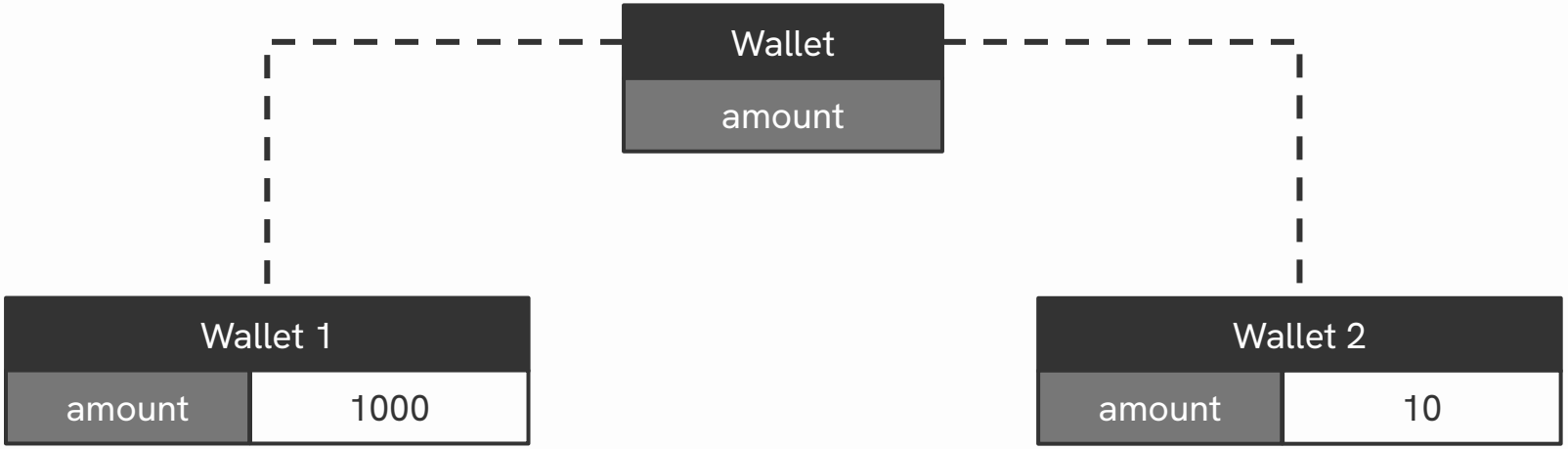



BPI
BPI
Makati Main
OPEN
WELCOME

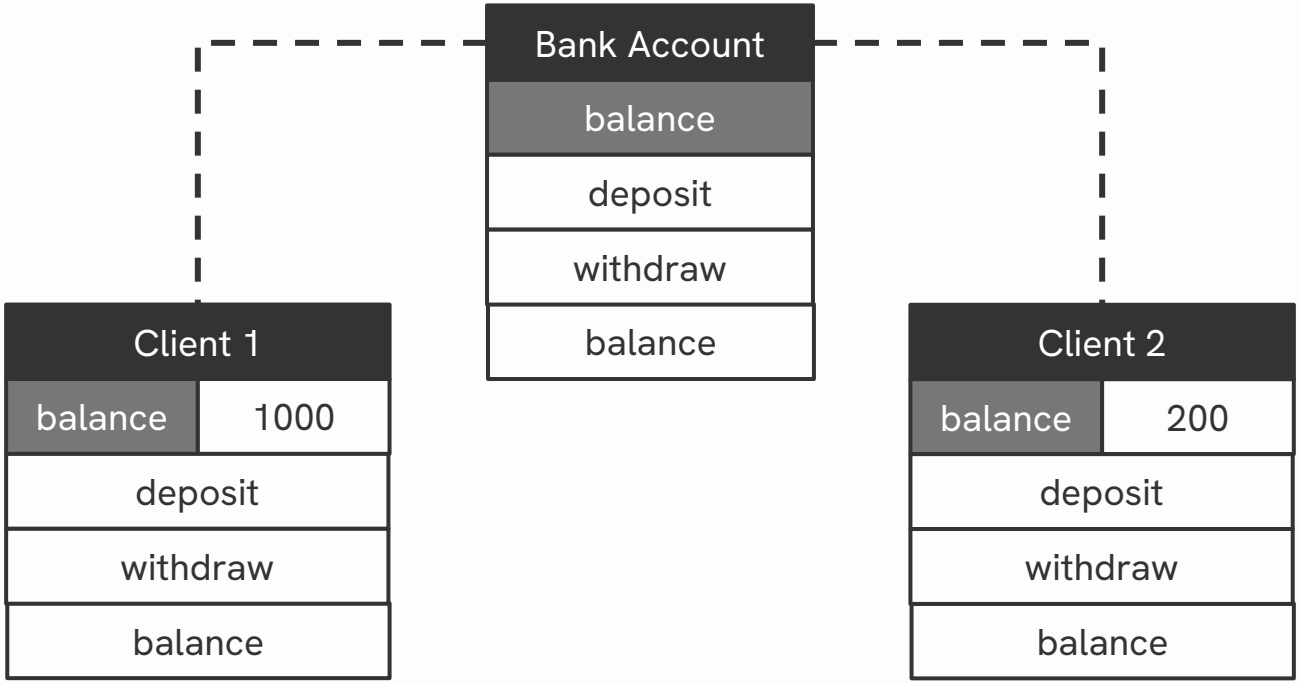
Book


Book
title
genre
author
Book 1
title
The Hobbit
genre
Fantasy
author
J.R.R. Tolkien
Book 2
pages
Dune
genre
Sci-Fi
author
Frank Herbert

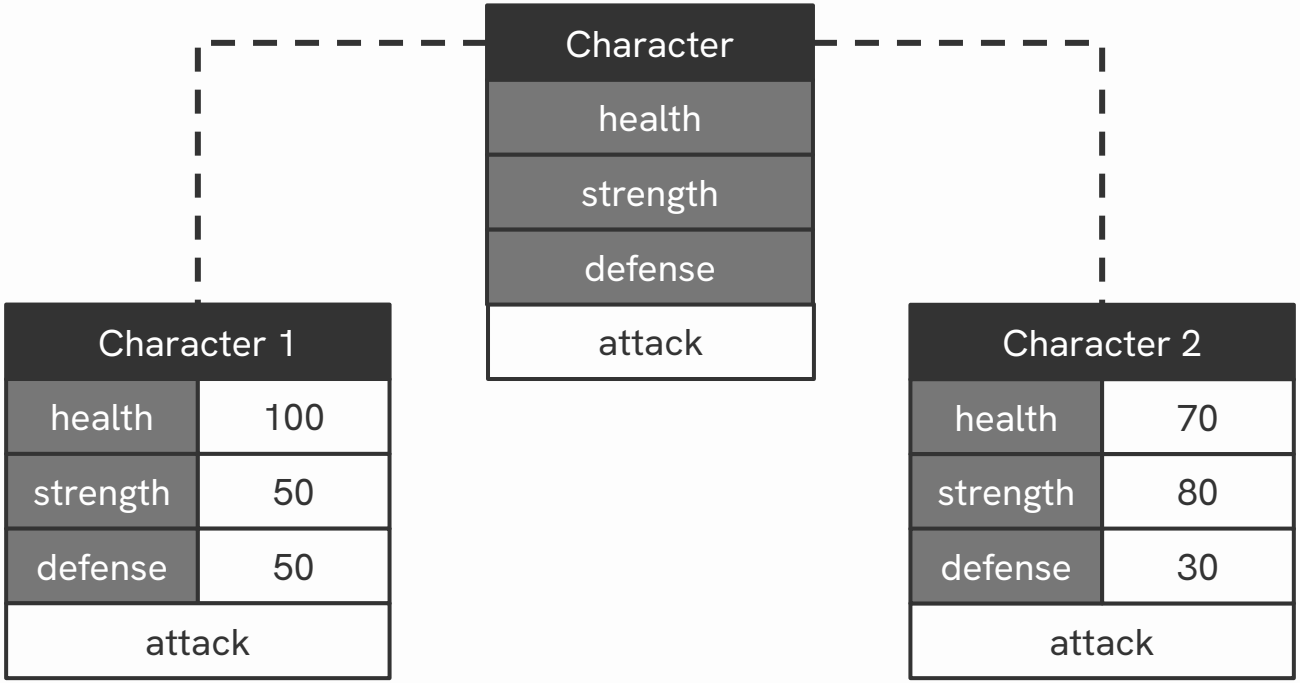
Wallet

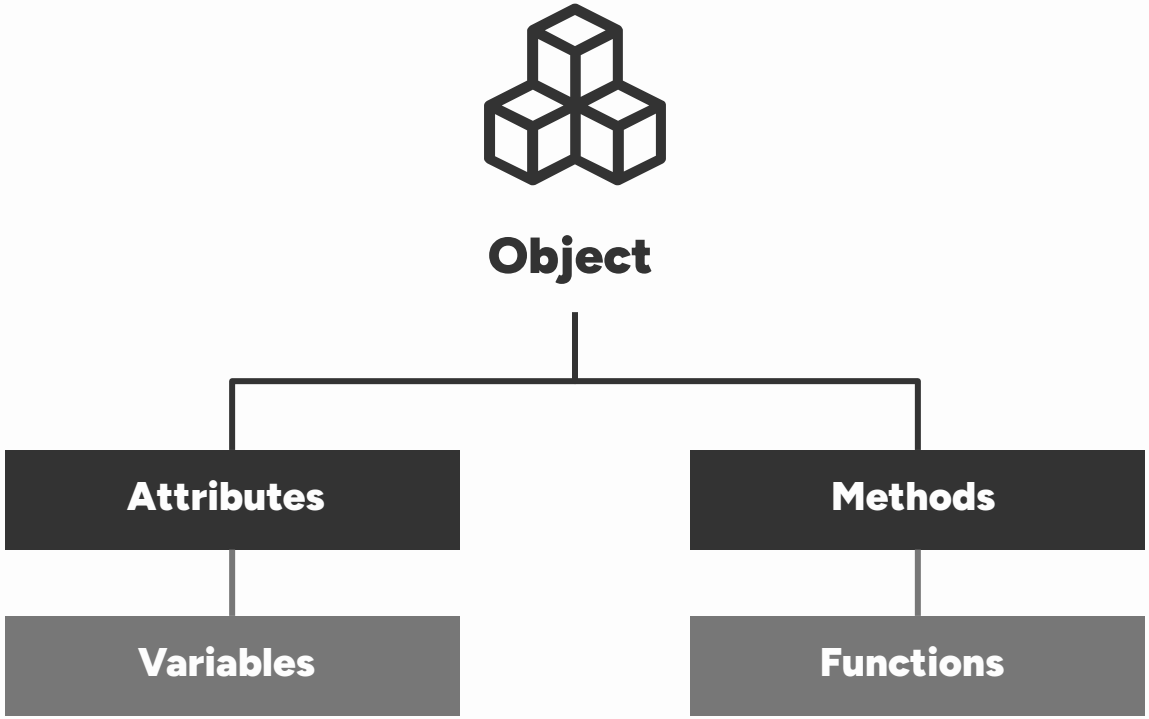


Bank Account



Game Character




Object
Attributes
Methods
Variables
Functions


Classes: Piece by Piece

Example Class

```
1  class Employee:
2      """Class representation for employee data"""
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Object Creation

```
1  class Employee:
2      """Class representation for employee data"""
3
4  employee1 = Employee()
5
6
7
8
9
10
11
12
13
14
15
```

Multiple Object Creation

```
1  class Employee:
2      """Class representation for employee data"""
3
4  employee1 = Employee()
5  employee2 = Employee()
6
7
8
9
10
11
12
13
14
15
```

Class Constructor/Initialization

```
1  class Employee:
2      """Class representation for employee data"""
3
4      def __init__(self, name, id):
5          print(f"Employee {name} assigned ID {id}")
6
7  employee1 = Employee("Richard", "1234")
8  employee2 = Employee("Jelly", "9876")
9
10
11
12
13
14
15
```

Object Attributes

```
1  class Employee:
2      """Class representation for employee data"""
3
4      def __init__(self, name, id):
5          self.name = name
6          self.id = id
7
8          print(f"Employee {self.name} assigned ID {self.id}")
9
10 employee1 = Employee("Richard", "1234")
11 employee2 = Employee("Jelly", "9876")
12
13 print("Employee 1 Name:", employee1.name)
14 print("Employee 2 Name:", employee2.name)
15
```

Object Attributes

self .name

employee1 .name

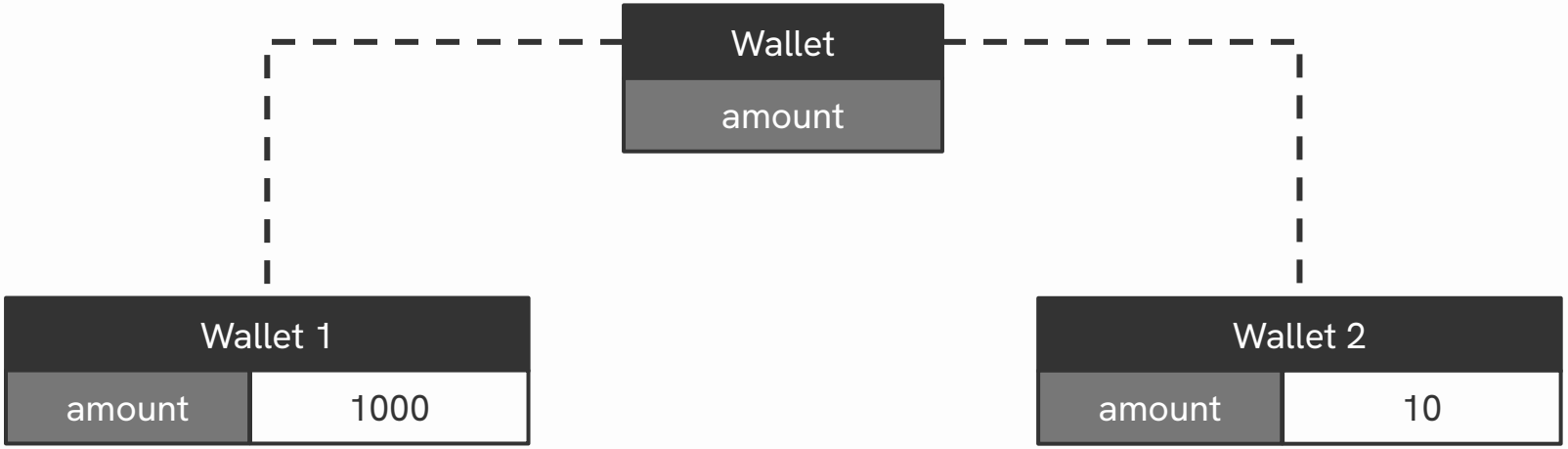
Object Methods

```
1  class Employee:
2        """Class representation for employee data"""
3
4        def __init__(self, name, id):
5              self.name = name
6              self.id = id
7              self.tasks = []
8
9              print(f"Employee {self.name} assigned ID {self.id}")
10
11       def add_work(self, task):
12             return self.tasks.append(task)
13
14 employee = Employee("Richard", "1234")
15 employee2 = Employee("Jelly", "9876")
16
17 print("Employee 1 Name:", employee1.name)
18 print("Employee 2 Name:", employee2.name)
19
20 employee.add_work("create charts")
21 print(employee.tasks)
```

Object Methods

```
employee.add_work(task)
add_work(employee, task)
```

Wallet

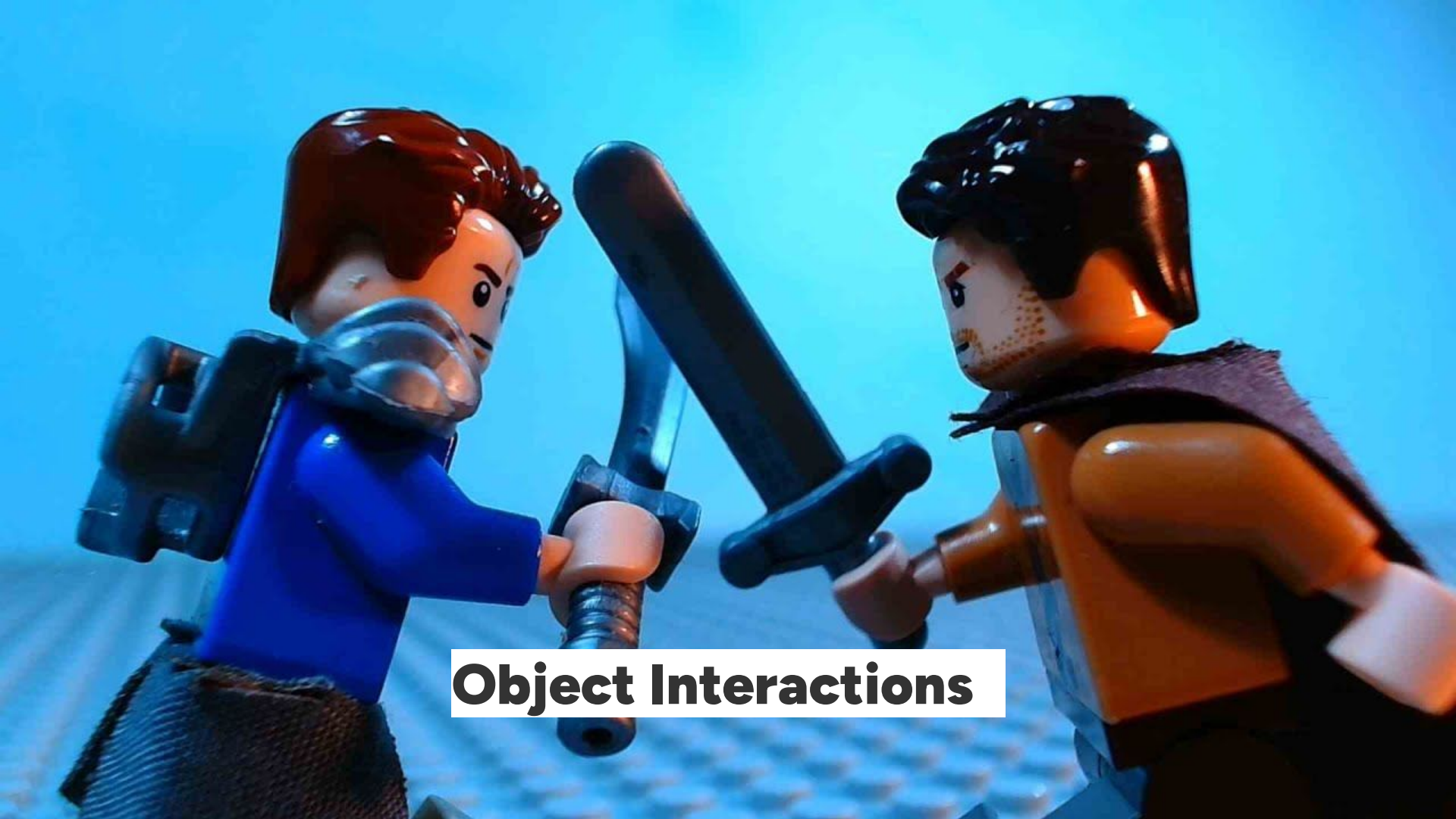


Implement : Wallet

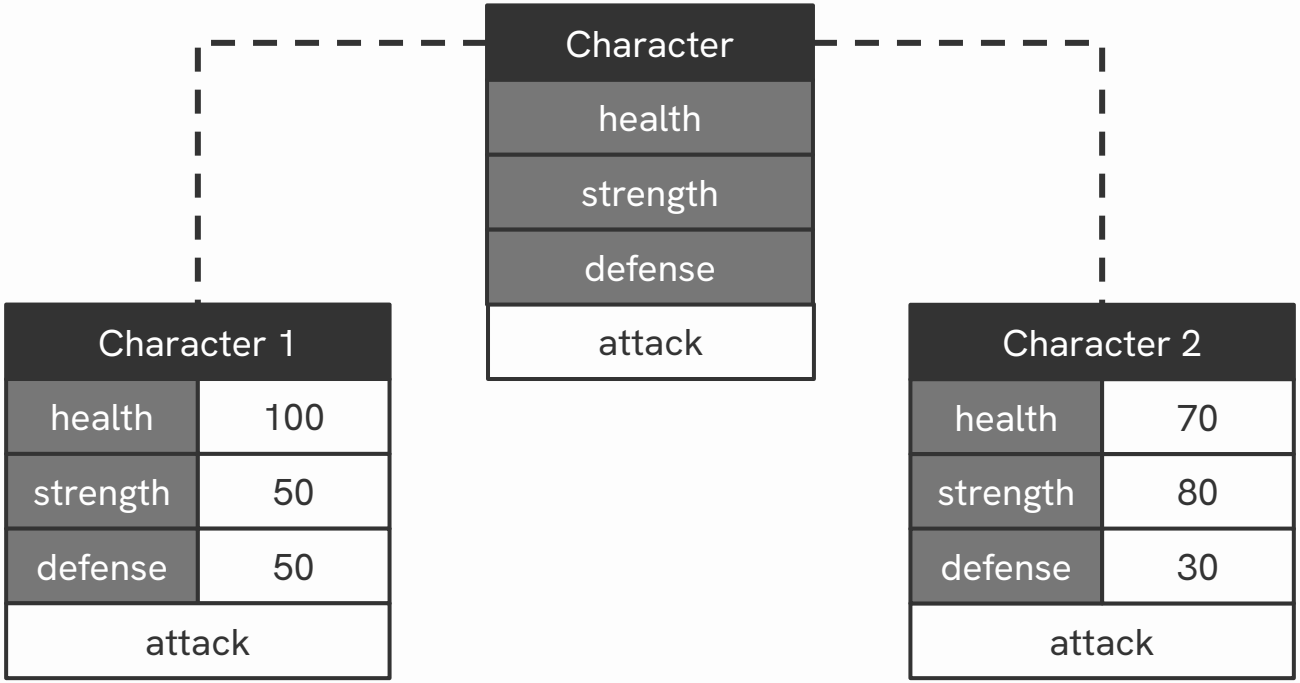
```
1  class Wallet:
2      def __init__(self, initial_amount=0):
3          self.amount = initial_amount
4
5  food_wallet = Wallet(250)
6  transport_wallet = Wallet(1000)
7
8  print("Food Budget: ", food_wallet.amount)
9  print("Transport Budget: ", transport_wallet.amount)
```

Implement : Student

```
1  class Student:
2      def __init__(self, name, level):
3          self.name = name
4          self.level = level
5
6      def introduction(self):
7          return f“I’m {self.name}! I’m a {self.level} student.”
8
9  student1 = Student(“Juan”, ‘3rd Year College’)
10 print(student1.introduction())
11
12 student2 = Student(“Maria”, ‘Grade Three’)
13 print(student2.introduction())
```


Object Interactions

Game Character



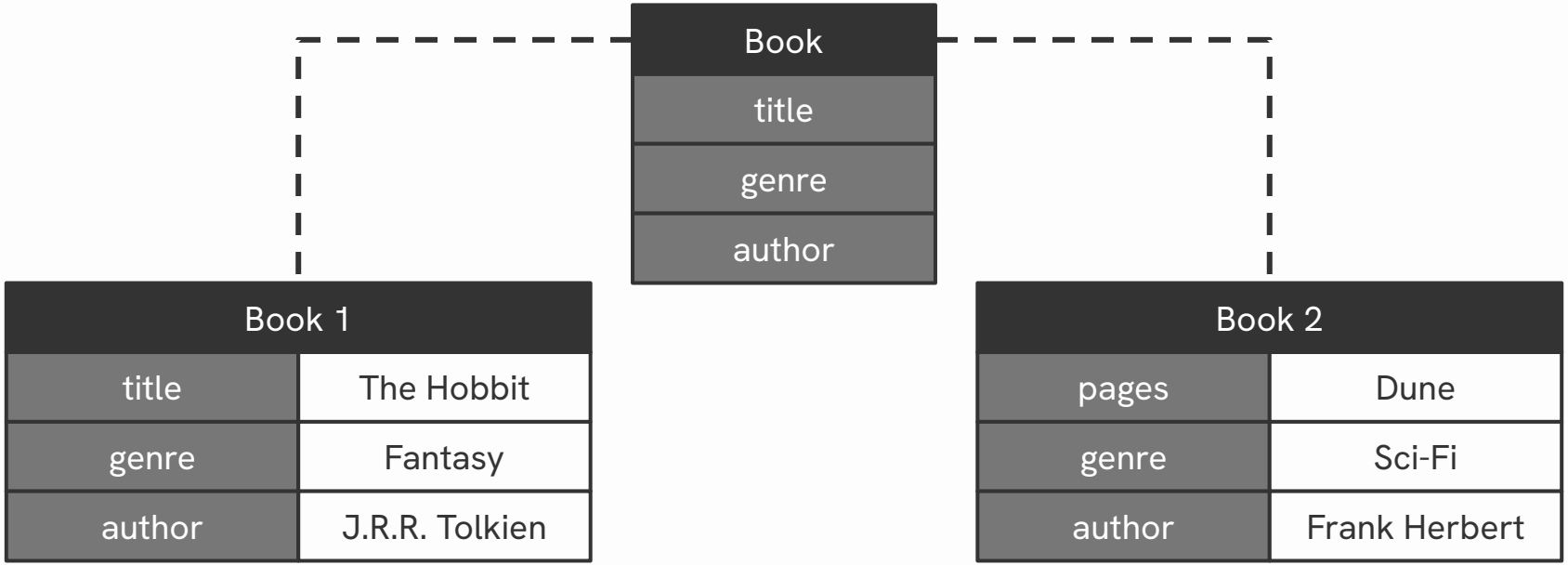
Implement: Character

```
1  class Character:
2      def __init__(self, health=10, strength=10, defense=10):
3          self.health = health
4          self.strength = strength
5          self.defense = defense
6
7      def attack(self, other):
8          damage = self.strength - other.defense
9          other.health -= damage
10
11 player = Character(strength=100)
12 enemy = Character()
13
14 player.attack(enemy)
15 print(enemy.health)
```

Implement: Pen and Paper

```
1  class Paper:
2      def __init__(self):
3          self.content = ""
4  class Pen:
5      def __init__(self, ink_level):
6          self.ink_level = ink_level
7
8      def write(self, paper, text):
9          if self.ink_level > 0:
10             paper.content += text
11
12 pen = Pen(100)
13 paper = Paper()
14 pen.write(paper)
15 print(paper.content)
```

Book


Book
title
genre
author
Book 1
title
The Hobbit
genre
Fantasy
author
J.R.R. Tolkien
Book 2
pages
Dune
genre
Sci-Fi
author
Frank Herbert

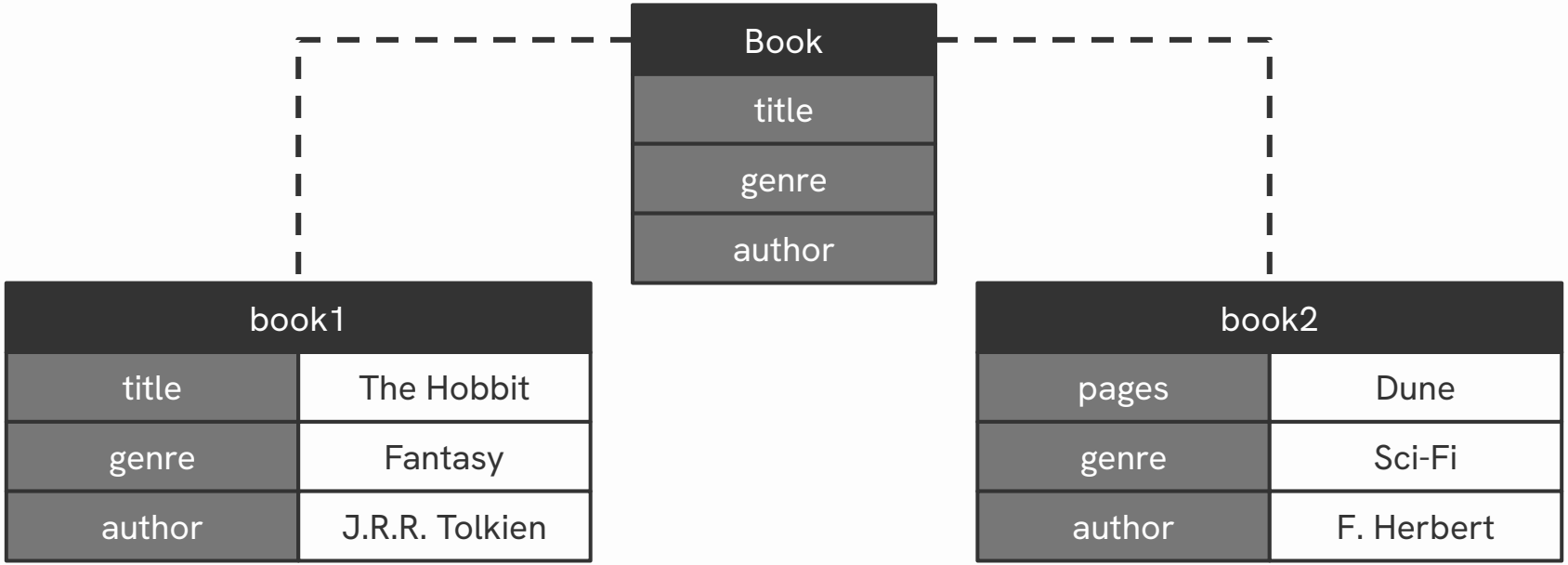

Magic Methods

Magic/Dunder Methods

Dunder methods are special, built-in methods that start and end with dunders (double underscores). Using these methods change or add custom behaviors to classes.

Method Name	Input(s)	Output(s)	Note
__init__	*	None	Sets behavior when creating objects
__str__	None	String	Used in **`str()`** and **`print()`**
__eq__	Any	Boolean	Sets behavior for **==** operations
__add__	Any	Any	Sets behavior for **+** operations
__len__	None	Integer	Sets behavior when used in **`len()`**

Book


Book
title
genre
author
book1
title
The Hobbit
genre
Fantasy
author
J.R.R. Tolkien
book2
pages
Dune
genre
Sci-Fi
author
F. Herbert

Implement: Book

```
1  class Book:
2      def __init__(self, title=None, genre=None, author=None):
3          self.title = title
4          self.genre = genre
5          self.author = author
6
7  book1 = Book("The Hobbit", "Fantasy", "Tolkien")
8  book2 = Book("Dune", "Sci-Fi", "Herbert")
9  print(book1)
```

```
<__main__.Book object at 0x0000019FE4F27BC0>
```

Magic Method __str__

The **__str__** dunder method defines what is used if the object is printed or used in **str()**

```
1  class Book:
2      def __init__(self, title=None, genre=None, author=None):
3          self.title = title
4          self.genre = genre
5          self.author = author
6
7      def __repr__(self):
8          return f"{self.title} [{self.genre}] - {self.author}"
9
   book1 = Book("The Hobbit", "Fantasy", "Tolkien")
   book2 = Book("Dune", "Sci-Fi", "Herbert")
   print(book1)
```

```
The Hobbit [Fantasy] - Tolkien
```

Magic Method __add__

The **__add__** dunder method defines the result when an **+** operation is used with the object

```
1  class Wallet:
2      def __init__(self, initial_amount=0):
3          self.amount = initial_amount
4
5      def __add__(self, other):
6          new_amount = self.amount + other.amount
7          return Wallet(new_amount )
8
9  food_wallet = Wallet(250)
10 transport_wallet = Wallet(1000)
11 total_wallet = food_wallet + transport_wallet
12
13 print("Food Budget: ", food_wallet.amount)
14 print("Transport Budget: ", transport_wallet.amount)
15 print("Total Budget: ", total_wallet.amount)
```

Object Identity

Python uses the memory location of an object to check for equality

```
1  class Candy:
2      def __init__(self, flavor):
3          self.flavor = flavor
4
5  choco1 = Candy("chocolate")
6  choco2 = Candy("chocolate")
7  milk = Candy("milk")
8
9  print(choco1 == milk)
10 print(choco1 == choco2)
```

Magic Method __eq__

The **__eq__** dunder method defines whether two objects are equal (or not)

```
1  class Candy:
2      def __init__(self, flavor):
3          self.flavor = flavor
4
5      def __eq__(self, other):
6          return self.flavor == other.flavor
7
8  choco1 = Candy("chocolate")
9  choco2 = Candy("chocolate")
10 milk = Candy("milk")
11
12 print(choco1 == milk)
13 print(choco1 == choco2)
```

H1

Ticket System

Using the concepts in intermediate Python with classes

Implement: Ticket System

Ticket
name
`__str__(self)`

TicketManager
queue
`issue(self, ticket)`
`cancel(self, name)`
`process(self)`
`__str__(self)`

```
def ticket_app():
    queue = TicketManager()
    user_choice = input("command: ")

    while user_choice != "exit":

        if user_choice == "add":
            ticket_name = input("Enter name: ")
            ticket = Ticket(ticket_name)
            queue.issue(ticket)

        ...
        user_choice = input("command: ")

ticket_app()
```

02

OOP Concepts

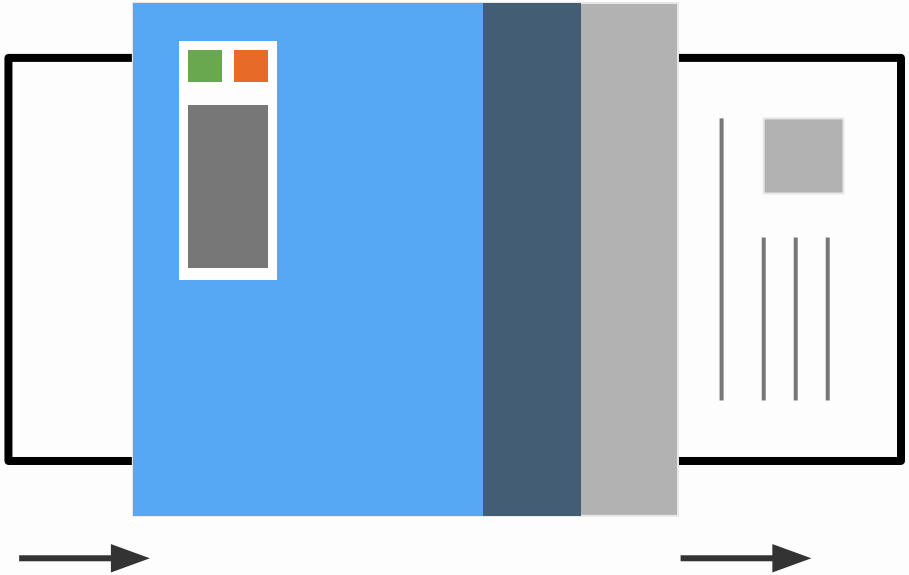
Intermediate code organization techniques

Data Hiding

Not everything needs to be seen

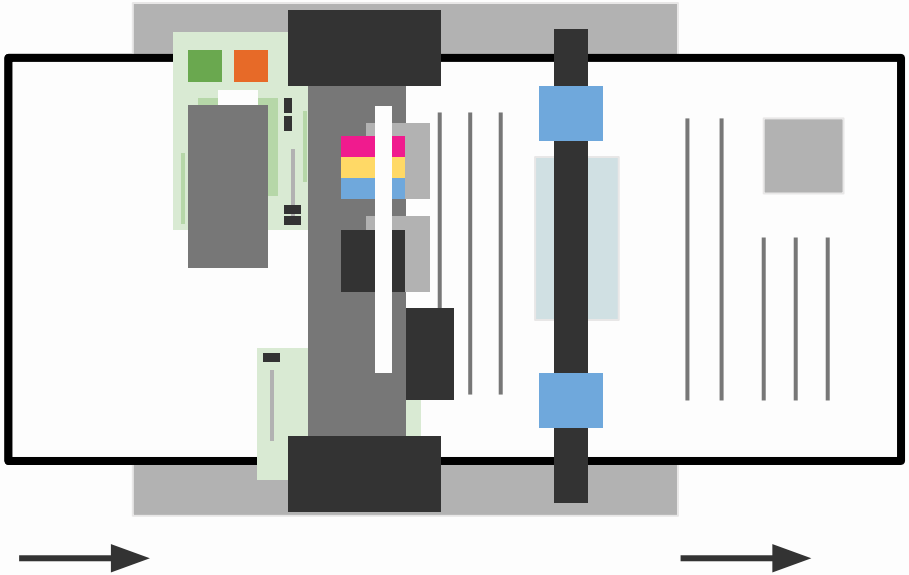
Strategic Data Hiding

Manage which variables are accessible to the public



Strategic Data Hiding

Manage which variables are accessible to the public



Why not show the parts of a printer?

Reasons to Hide Data

Encapsulation: Security

Prevent unauthorized read or write operations to sensitive data and processes within the code

Abstraction: Hide Complexity

Not every detail of a process needs to be known. Classes can set up their own logic to handle changes

Maintainability

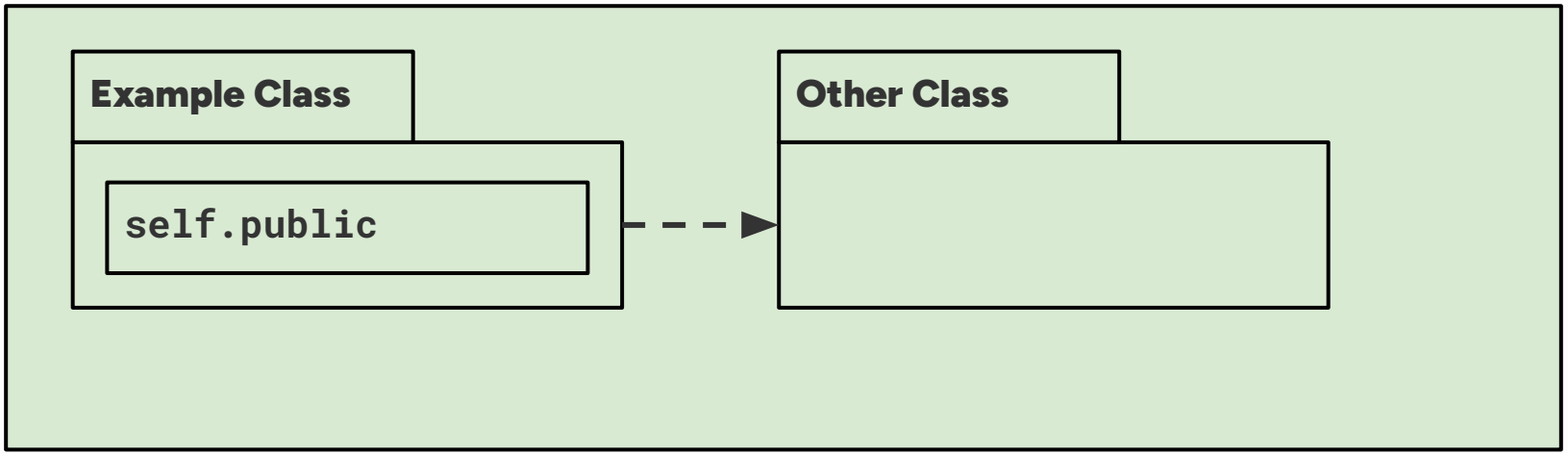
Less access to data means less suspects when debugging problems or issues when developing

Implement: Ecosystem

```python
import math

class Ecosystem:
    def __init__(self, initial, birth_rate, death_rate, env_factor):
        self.initial = initial
        self.birth_rate = birth_rate
        self.death_rate = death_rate
        self.env_factor = env_factor

    def simulate_growth(self, years):
        growth_factor = math.exp(self.birth_rate - self.death_rate)
        env_impact = (self.env_factor ** 0.5)
        final = self.initial * (growth_factor ** years) * env_impact
        return final

eco = Ecosystem(1000, 0.02, 0.01, 1.1)
print(f"Population after 10 years: {eco.simulate_growth(10):,.2f}")
```

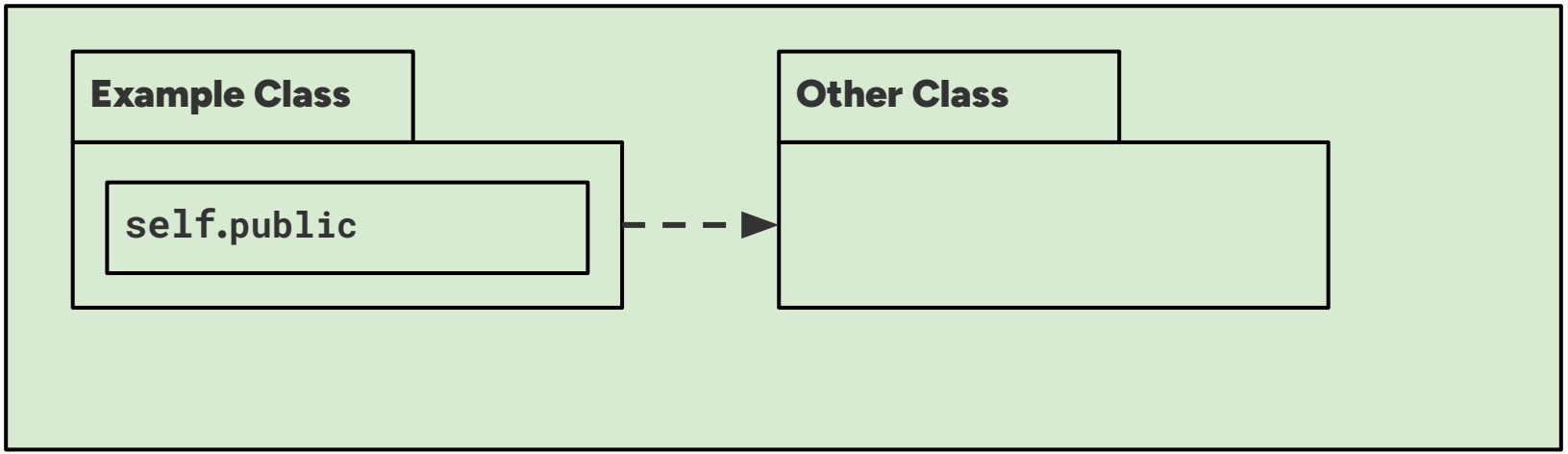
Public Attribute

```
1  class Example:
2      def __init__(self):
3          self.public = 1
```



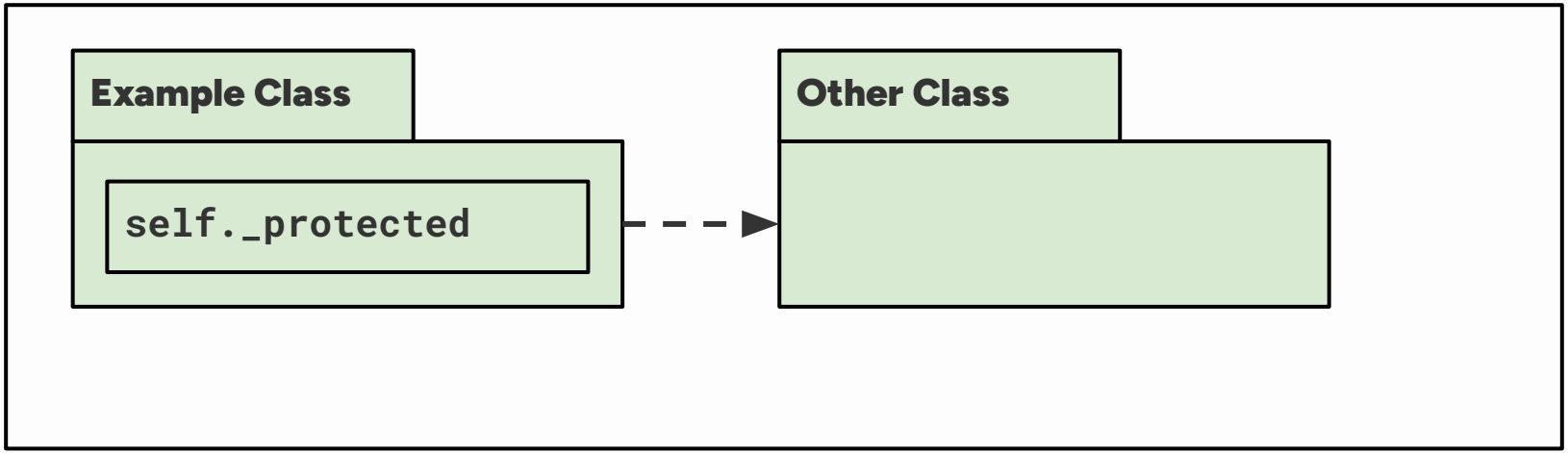
Public Method

```
1 class Example:
2     def public(self):
3         print(1)
```



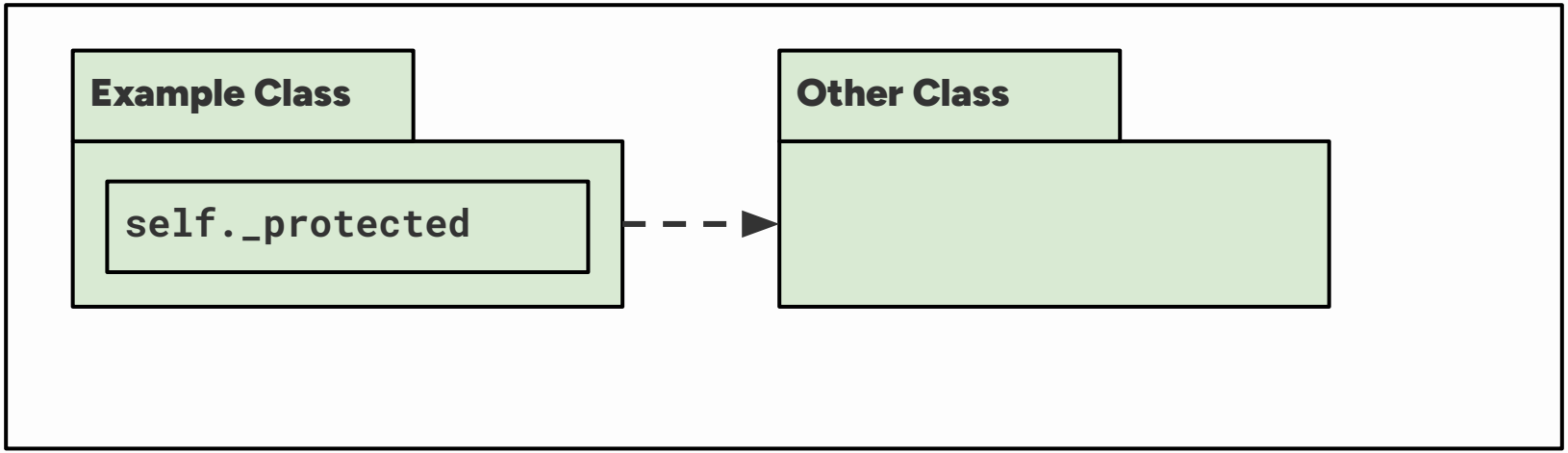
Protected Attribute

```
1 class Example:
2     def __init__(self):
3         self._protected = 2
```



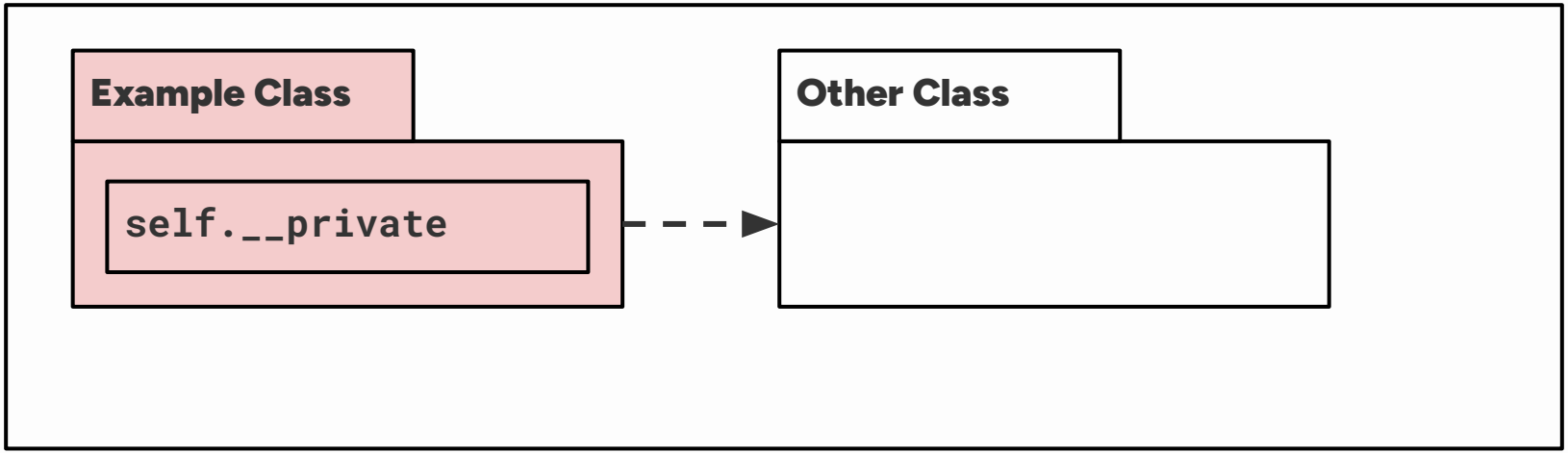
Protected Method

```
1 class Example:
2     def _protected(self):
3         print(2)
```



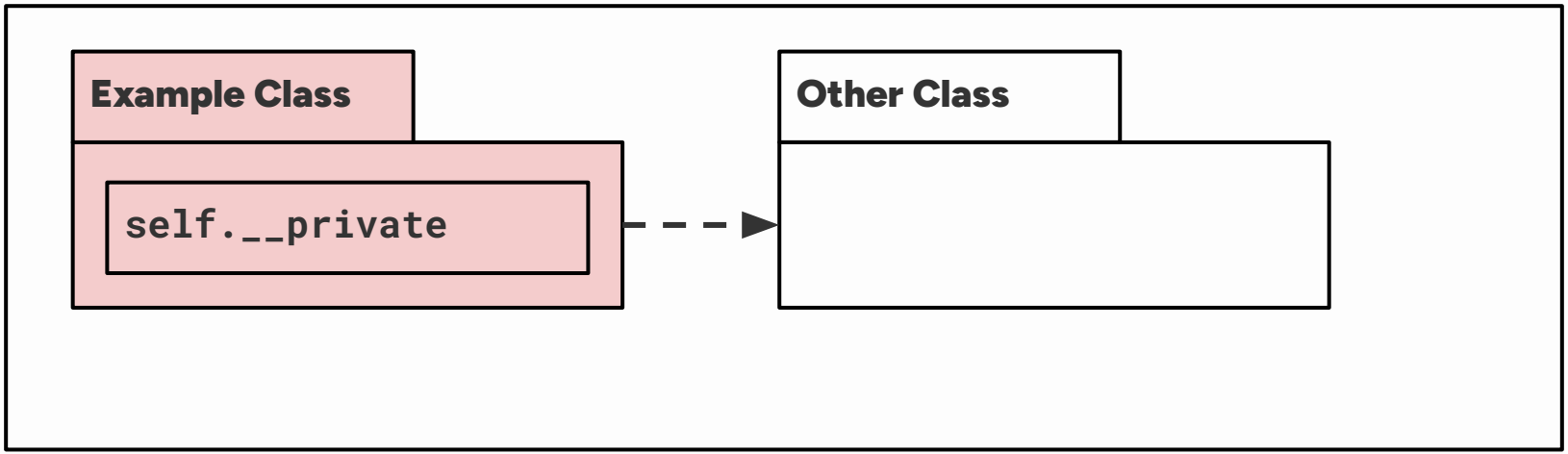
Forced Private Attribute

```
1  class Example:
2      def __init__(self):
3          self.__private = 3
```

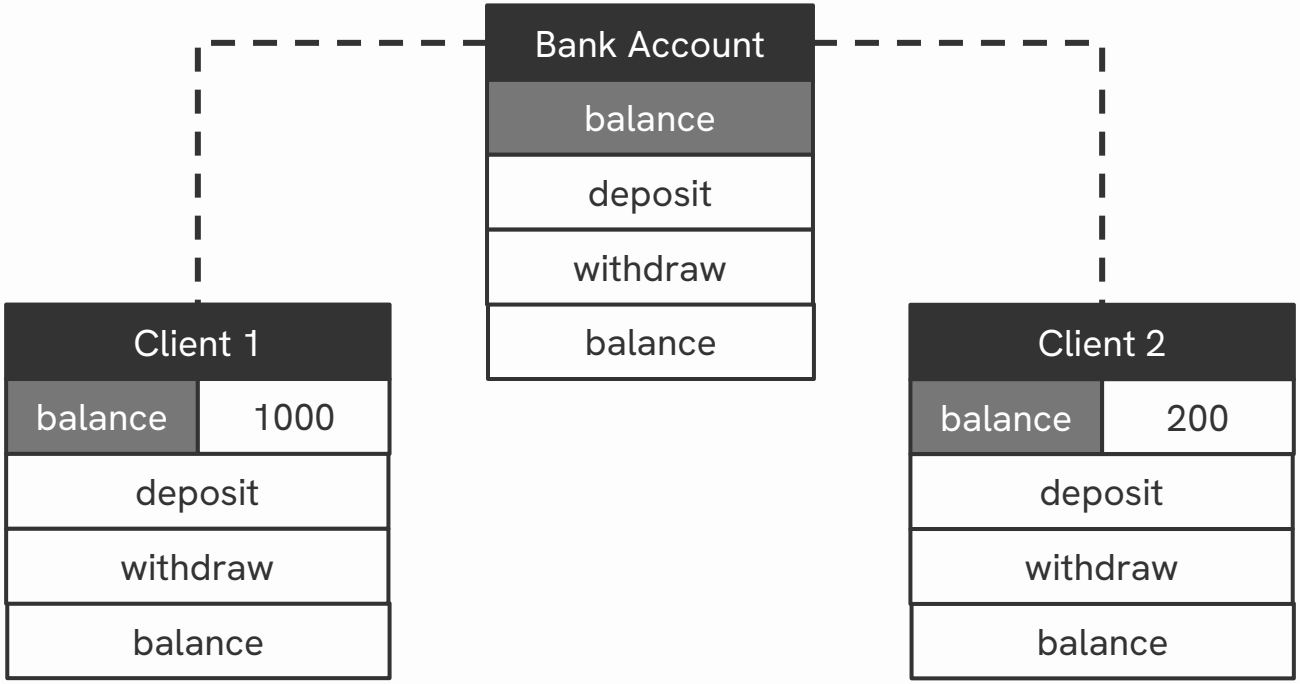


Forced Private Attribute

```
1  class Example:
2      def __private(self):
3          print(3)
```



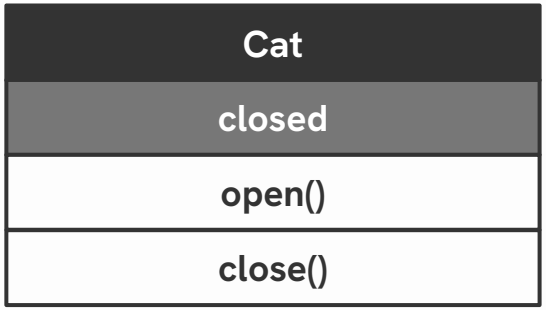
Bank Account



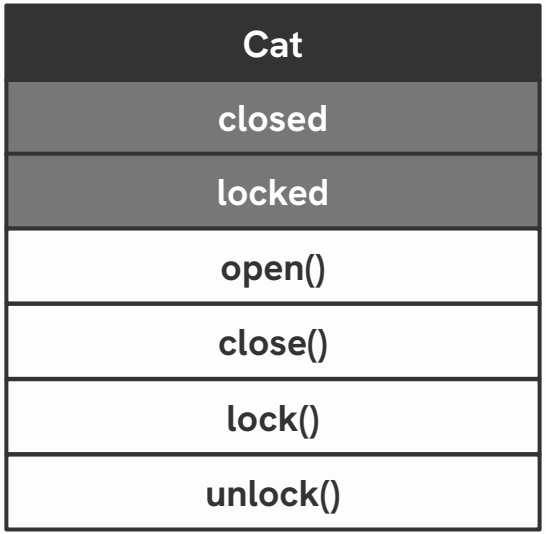
Implement: Bank Account

```
1  class BankAccount:
2      def __init__(self, initial_balance=0):
3          self._balance = initial_balance
4
5      def deposit(self, amount):
6          if self.amount > 0:
7              self._balance += amount
8
9      def withdraw(self, amount):
10         if 0 <= amount <= self._balance:
11             self._balance -= amount
12
13 account = BankAccount(100)
14 print(account.deposit(50))
15 print(account.withdraw(30))
```

Implement: Door



Challenge: Locked Door



Inheritance

Reduce code redundancy among classes

Code Redundancy

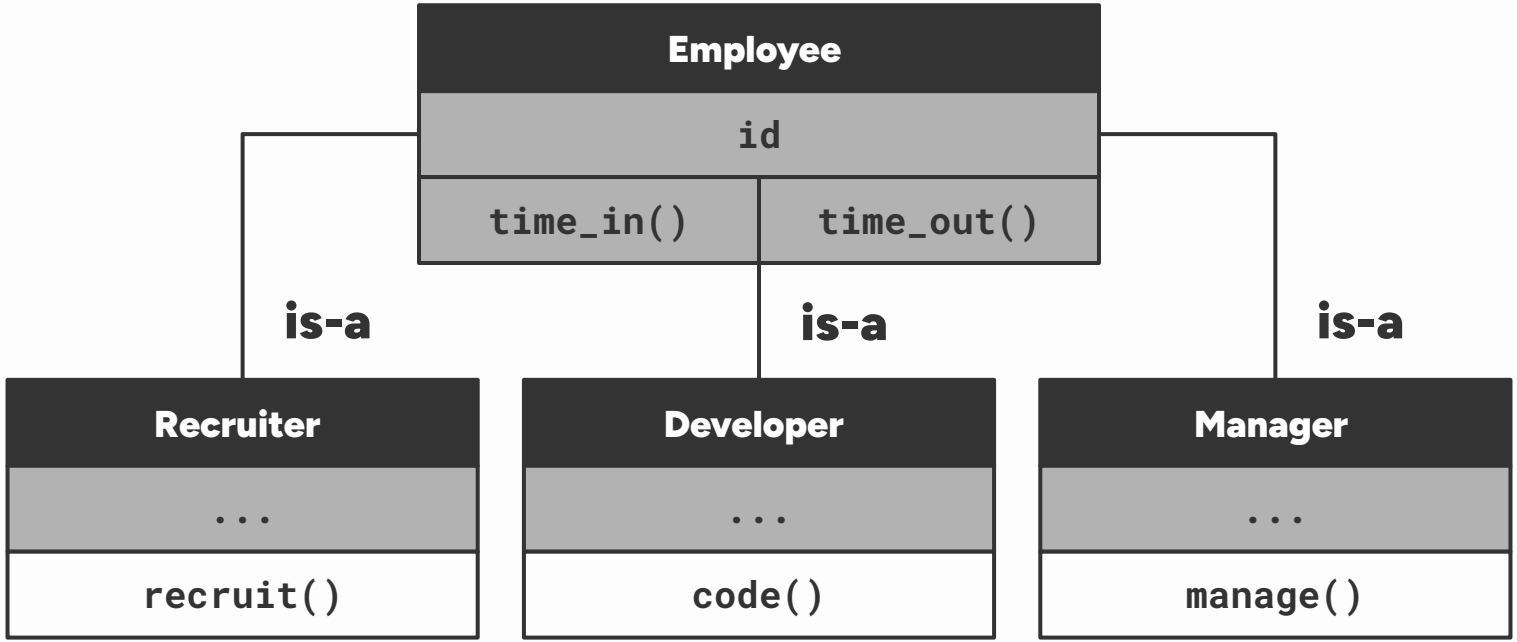
```
class Recruiter:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def recruit(self)
```

```
class Designer:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def design(self)
```

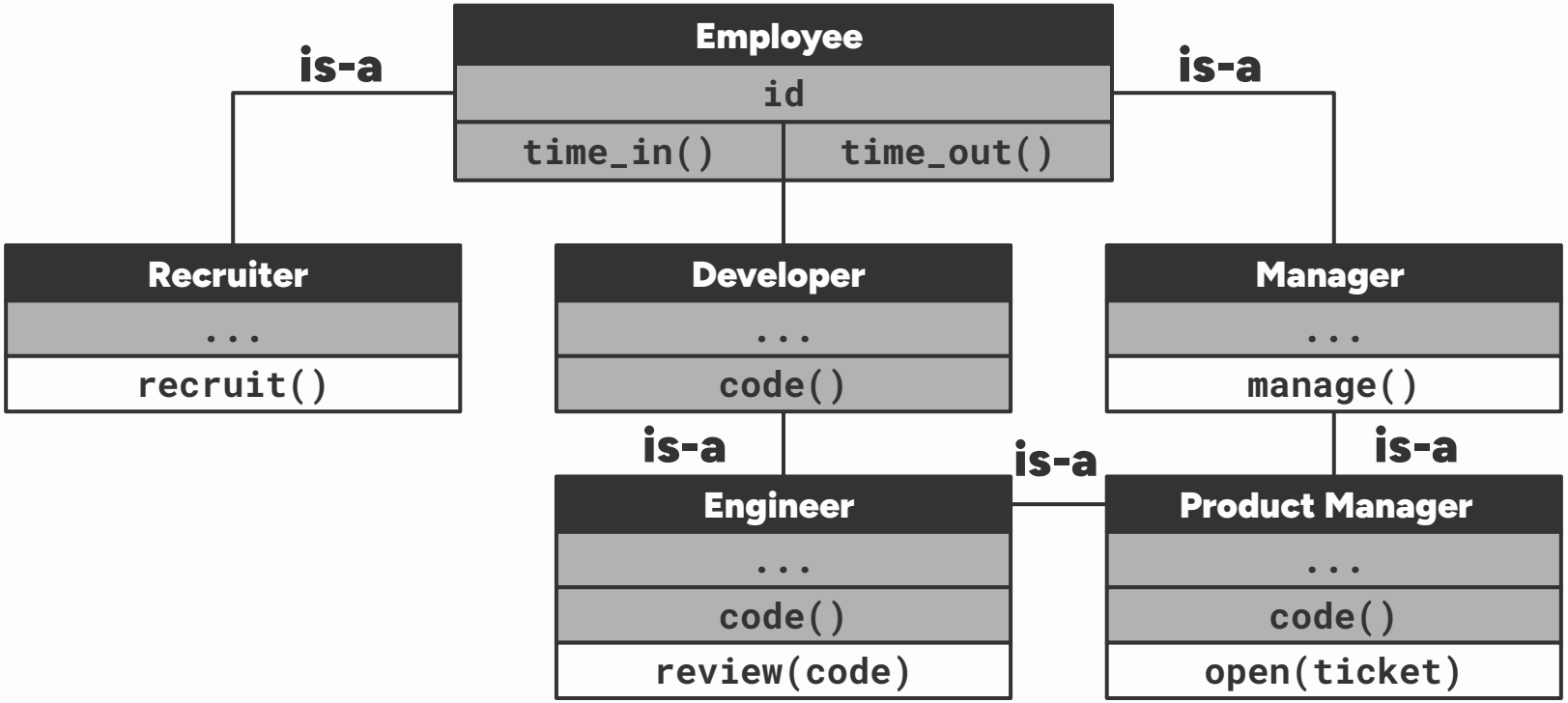
```
class Developer:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def code(self)
```

```
class Manager:
    def __init__(self, id)
    def time_in(self)
    def time_out(self)

    def manage(self)
```

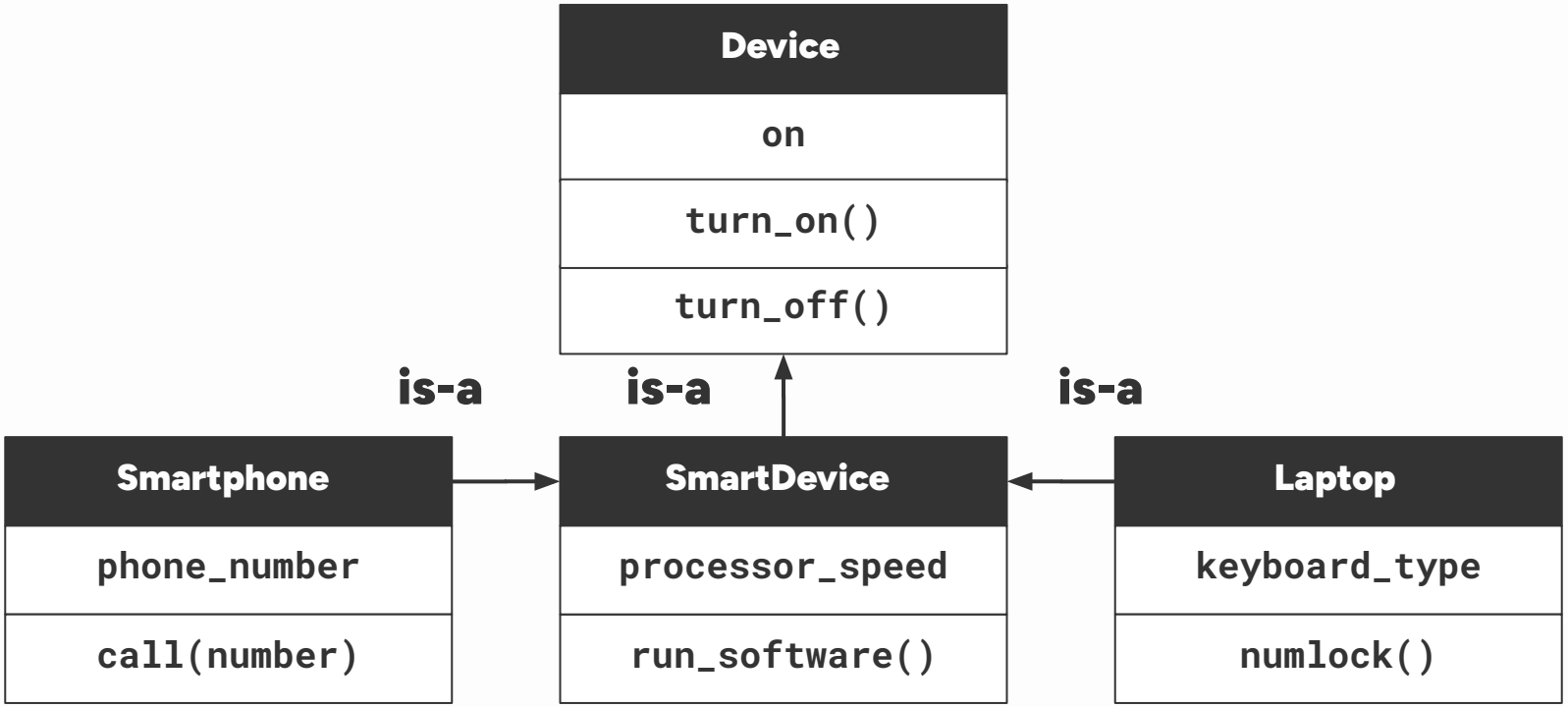
Hierarchy


Employee
id
time_in()
time_out()
is-a
is-a
is-a
Recruiter
...
recruit()
Developer
...
code()
Manager
...
manage()

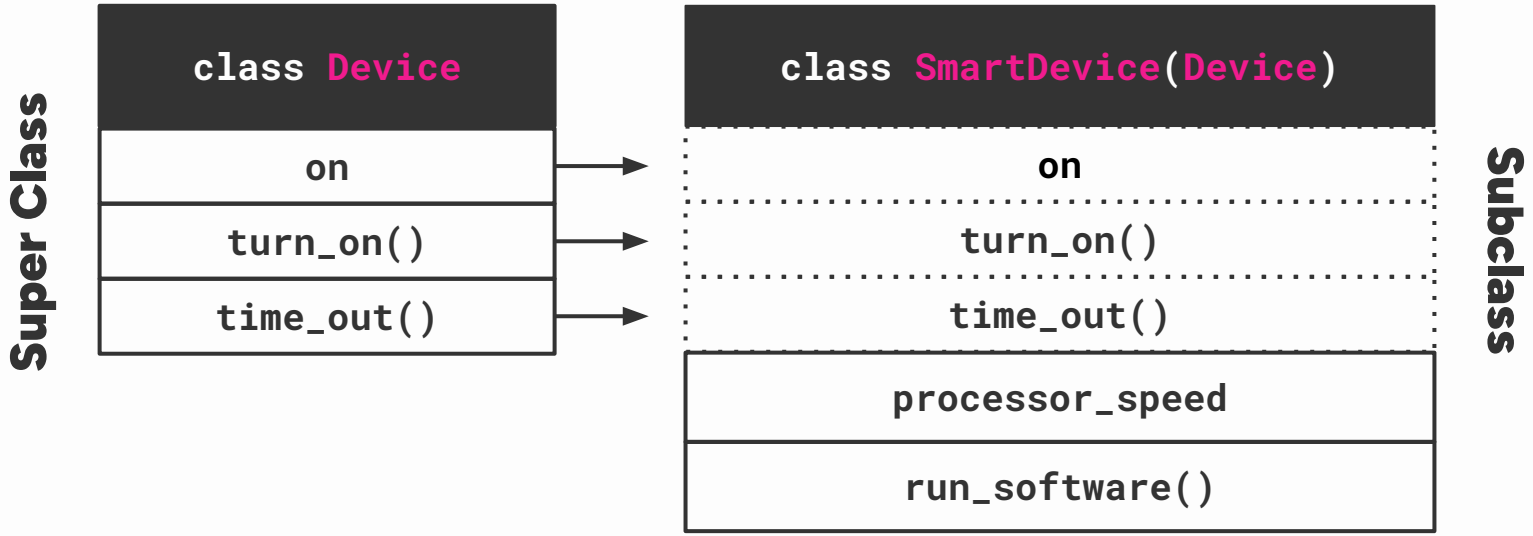
Hierarchy (Complex)



Hierarchy


Device
on
turn_on()
turn_off()
is-a
is-a
is-a
Smartphone
phone_number
call(number)
SmartDevice
processor_speed
run_software()
Laptop
keyboard_type
numlock()

Class Inheritance



Inheritance Format

```
class Person:
     def __init__(self, first_name, last_name):
          self.first_name = first_name
          self.last_name = last_name

     def introduction(self):
          return f"I'm {self.first_name} {self.last_name}!"
```

inherits

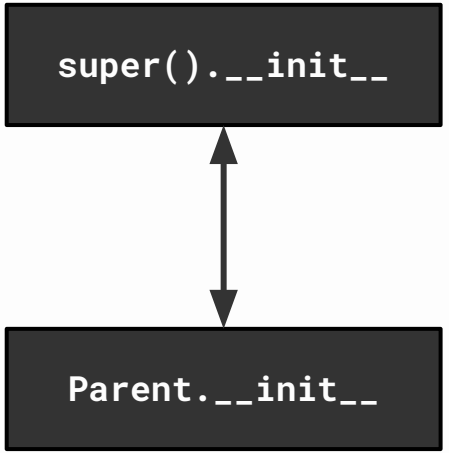
```
class Student(Person):
     def __init__(self, first_name, last_name, level):
          super().__init__(first_name, last_name)
          self.level = level

     def introduction(self):
          return super().introduction() + f" I'm a {self.level} student."
```

Inheritance Format

```
class Person:
     def __init__(self, first_name, last_name):
          self.first_name = first_name
          self.last_name = last_name
```

```
class Student(Person):
     def __init__(self, first_name, last_name, level):
          super().__init__(first_name, last_name)
          self.level = level
```

```
class Student(Parent):
     def __init__(self, likes, toys):
          self.first_name = first_name
          self.last_name = last_name
          self.level = level
```



Implement: Person-Student Class

```
class Person:
     def __init__(self, first_name, last_name):
          self.first_name = first_name
          self.last_name = last_name

     def introduction(self):
          return f“I'm {self.first_name} {self.last_name}!”

     def sleep():
          print(“I will sleep for eight hours”)

class Student(Person):
     def __init__(self, first_name, last_name, level):
          super().__init__(first_name, last_name)
          self.level = level

     def introduction(self):
          return super().introduction() + f“ I'm a {self.level} student.”
```

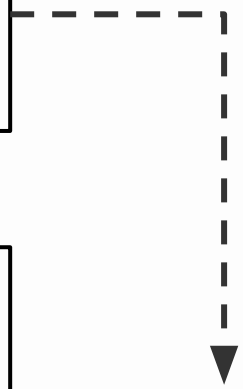
Silent Inheritance

```
class Person:
     def sleep(self):
          print(“I will sleep for eight hours”)
```

```
class Student(Parent):
     def sleep(self):
          print(“I will sleep for eight hours”)
```

Overriding

```
class Person:
     def sleep(self):
          print(“I will sleep for eight hours”)
```



```
class Child(Parent):
     def sleep(self):
          print(“I will sleep for six hours”)
```

Polymorphism

Loose connection via attributes and methods

Type Checking

You can directly check the type of a class using the `isinstance` function

```
1  class Parent:
2      pass
3
4  class Child(Parent):
5      pass
6
7  child = Child()
8  print(isinstance(child, Parent))
```

```
True
```

Multiple Type Checking

The `isinstance` function can be used to check for multiple types at once

```
1  class Unrelated:
2      pass
3
4  class Parent:
5      pass
6
7  class Child(Parent):
8      pass
9
10 child = Child()
11 type_check = isinstance(child,(Unrelated, Parent))
12 print(type_check)
```

Use Case: Input Checking

Type checking with `isinstance` is often done to apply different logic depending on the input

```
1  def add_item(items, item):
2      if isinstance(items, list):
3          items.append(item)
4      elif isinstance(items, set):
5          items.add(item)
6      elif isinstance(items, dict):
7          items[item] = item
8      else:
9          raise NotImplementedError()
```

Quick Exercise: General Add

Implement the following add function

```
def add(first, second):
```

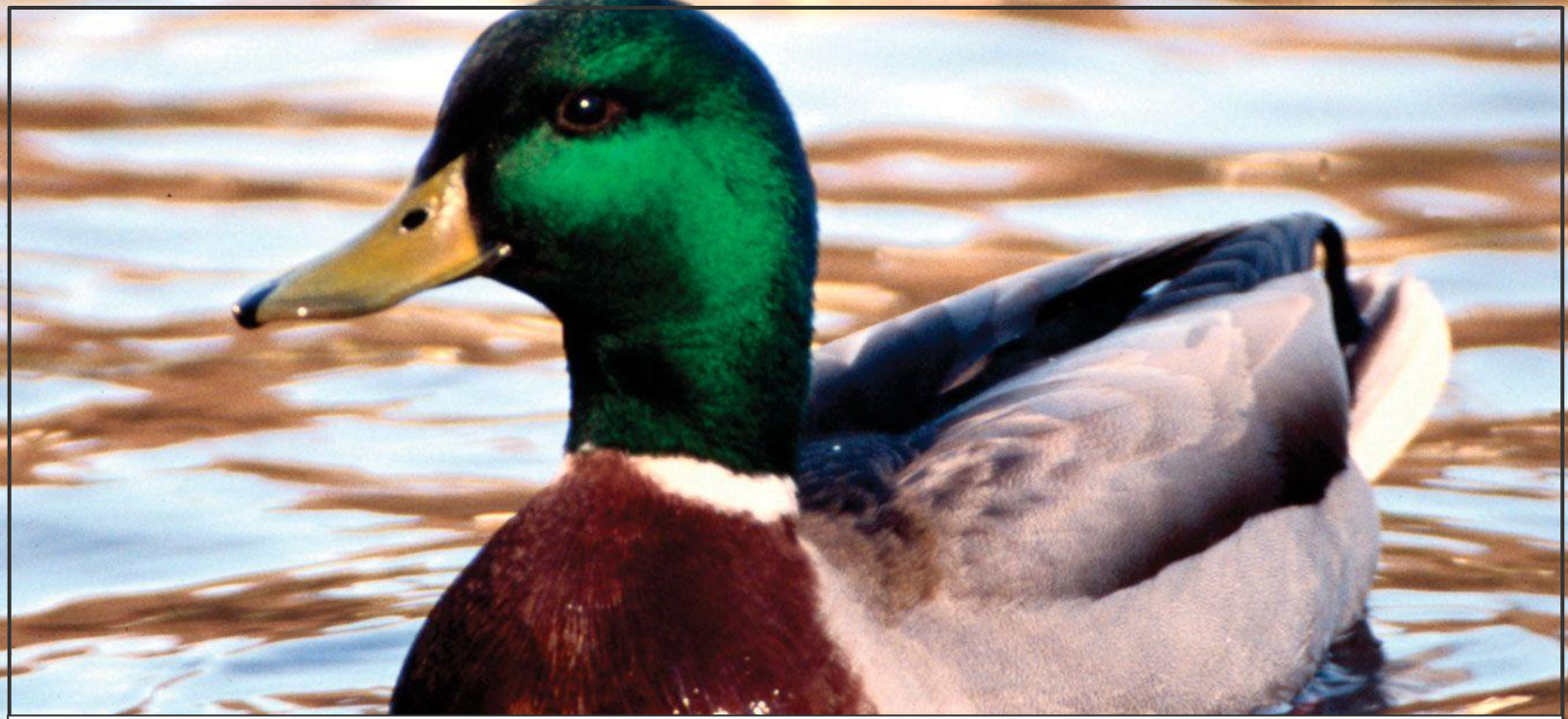
Use the following mechanisms

both int, float, or bool	→	return their sum
both list, set, dict, str	→	return items combined
Anything else	→	raise error

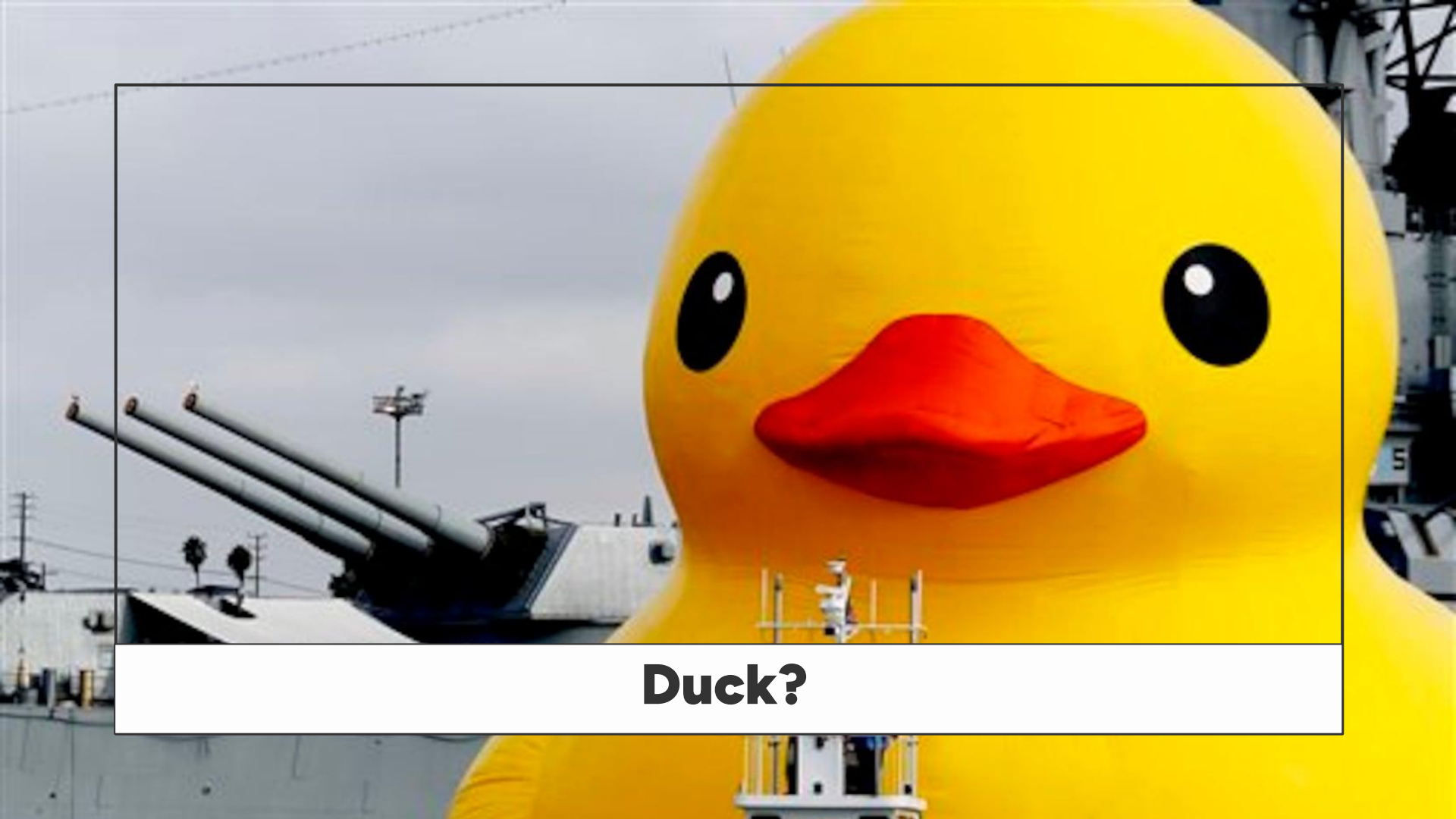
“”If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.””

—Duck Typing

Has → Is



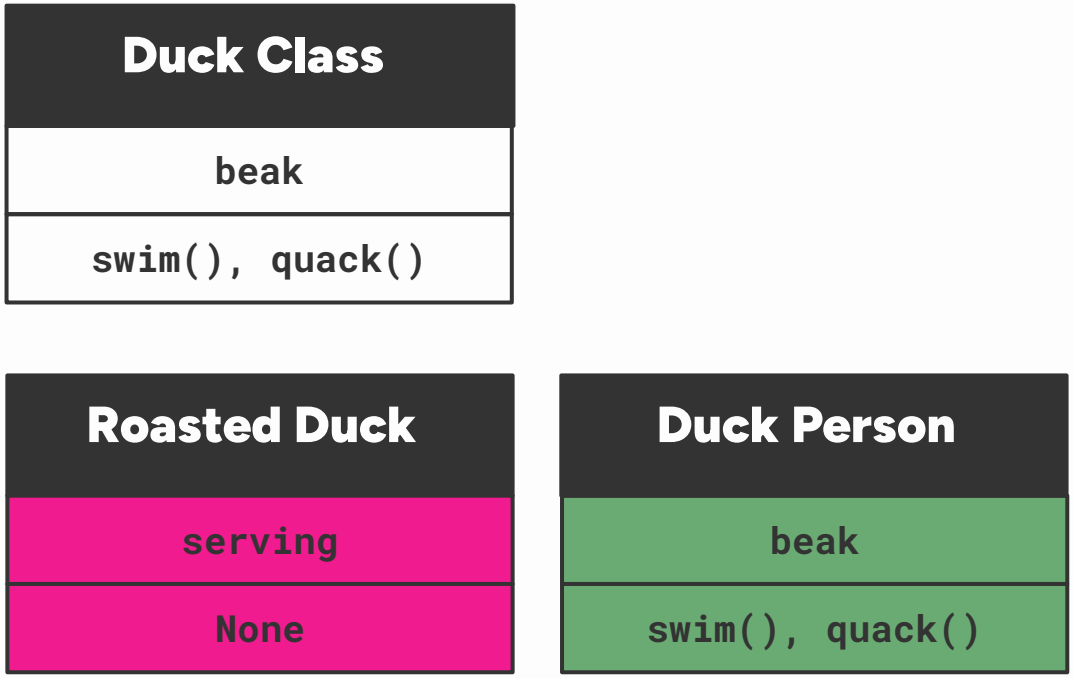
Duck?


Duck?

Duck?

Duck?

Duck Typing


Duck Class
beak
swim(), quack()
Rubber Duck
beak
swim(), quack()
Roasted Duck
serving
None
Duck Person
beak
swim(), quack()

Implement: “Duck” Classes

```
class Duck:
    def __init__(self, beak):
        self.beak = beak
    def swim(self):
        print(“Swimming”)
    def quack(self):
        print(“Quack”)
```

```
class RubberDuck:
    def __init__(self, beak):
        self.beak = beak
    def swim(self):
        print(“Splish Splosh”)
    def quack(self):
        print(“Squeak Quack”)
```

```
class DuckPerson:
    def __init__(self, beak):
        self.beak = beak
    def swim(self):
        print(“Swim hehe!”)
    def quack(self):
        print(“Quack hehe”)
```

```
class RoastedDuck:
    def __init__(self, serving):
        self.serving = serving
```

Attribute Checking

Similar to `isinstance` the function `hasattr` can be used to check if an instance has an **attribute** or **method** with the **same name**

```
1  dish = RoastedDuck(1)
2  if hasattr(dish, 'beak') and hasattr(dish, 'quack'):
3      dish.quack()
4  else:
5      print("Dinner is served")
```

```
Dinner is served
```

Attribute Checking

Similar to `isinstance` the function `hasattr` can be used to check if an instance has an **attribute** or **method** with the **same name**

```
1  duck = DuckPerson(“plastic”)
2  if hasattr(dish, ‘beak’) and hasattr(dish, ‘quack’):
3      dish.quack()
4  else:
5      print(“Dinner is served”)
```

```
Quack hehe
```

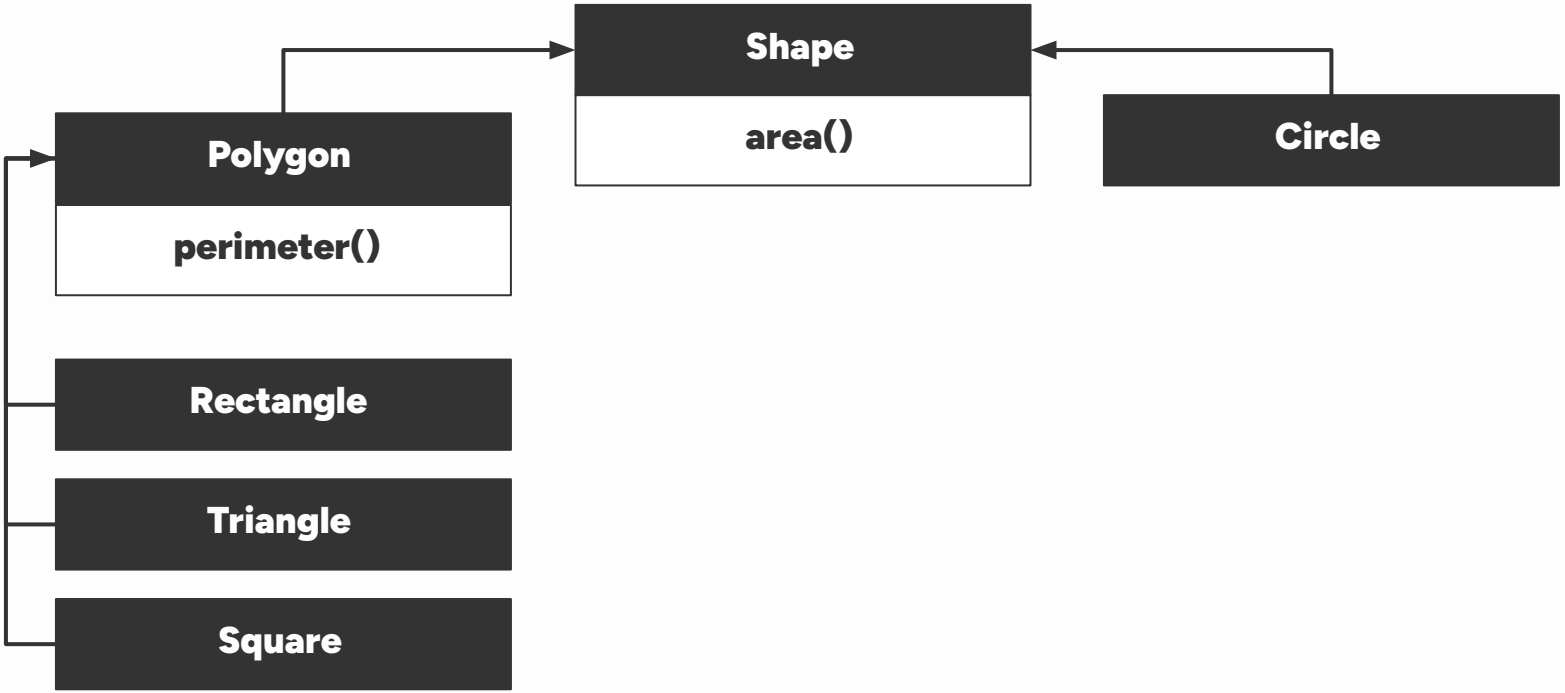
Implement: Knight

```
1  class Knight:
2      def __init__(self, health=10, defense=10):
3          self.health = 10
4          self.defense = defense
5
6      def attack(self, other):
7          damage = self.defense - other.defense
8          other.health -= damage
9
10 player = Knight(defense=20)
11 enemy = Character()
12
13 player.attack(enemy)
14 print(enemy.health)
```

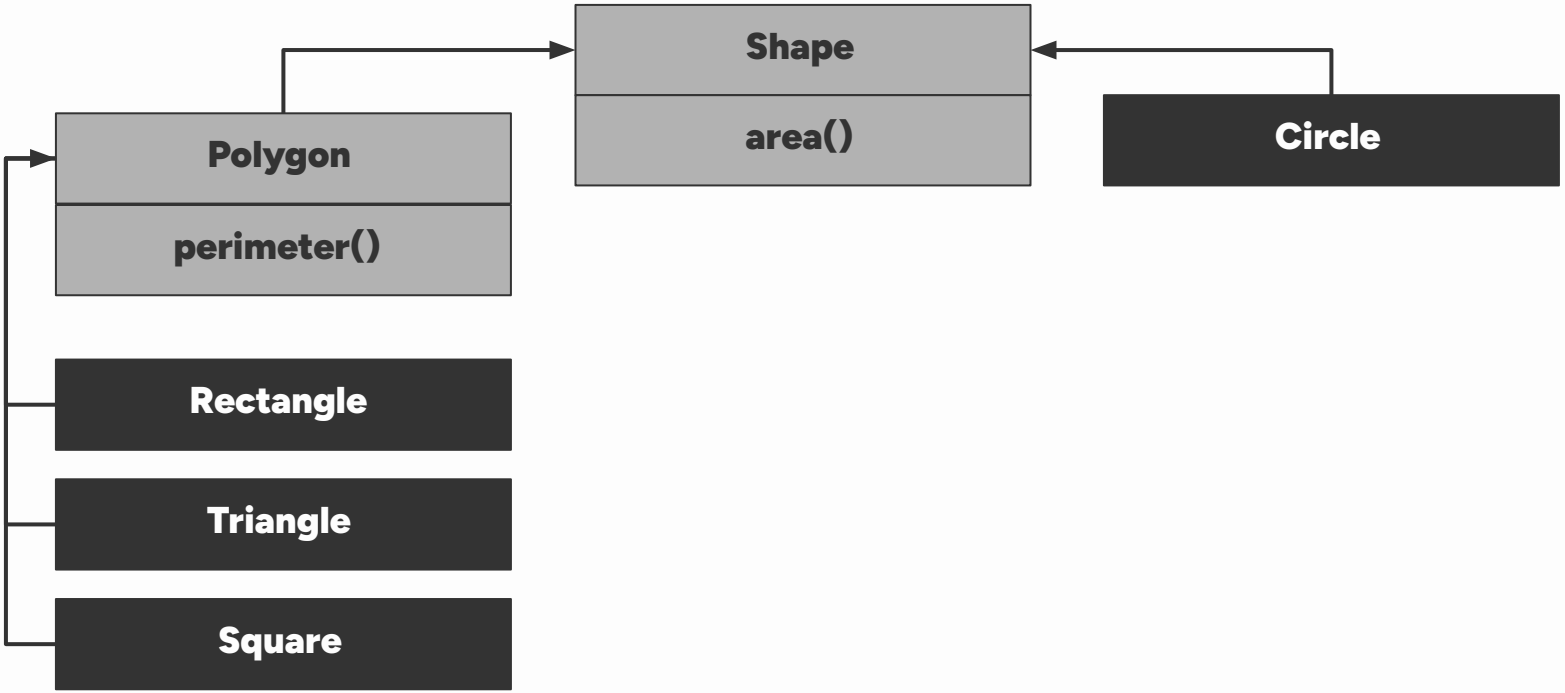
Abstract Classes

Enforced relationships via contracts

Shape Hierarchy



Shape Hierarchy



Method Implementation

```
1  class Shape:
2      def area(self):
3          pass
```

How do we define the area of a “Shape?”

Method Implementation

```
class Circle(Shape):
    def area(self):
        return 3.14 * self.radius
```

```
class Square(Shape):
    def area(self):
        return self.side_length * self.side_length
```

The area of a “Shape” is defined by its implementation

Abstraction: General to Specific

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
class Circle(Shape):
    pass
```

Abstraction: General to Specific

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
class Circle(Shape):
    def area(self):
        return 3.14 * self.radius ** 2
```

Extended Abstract

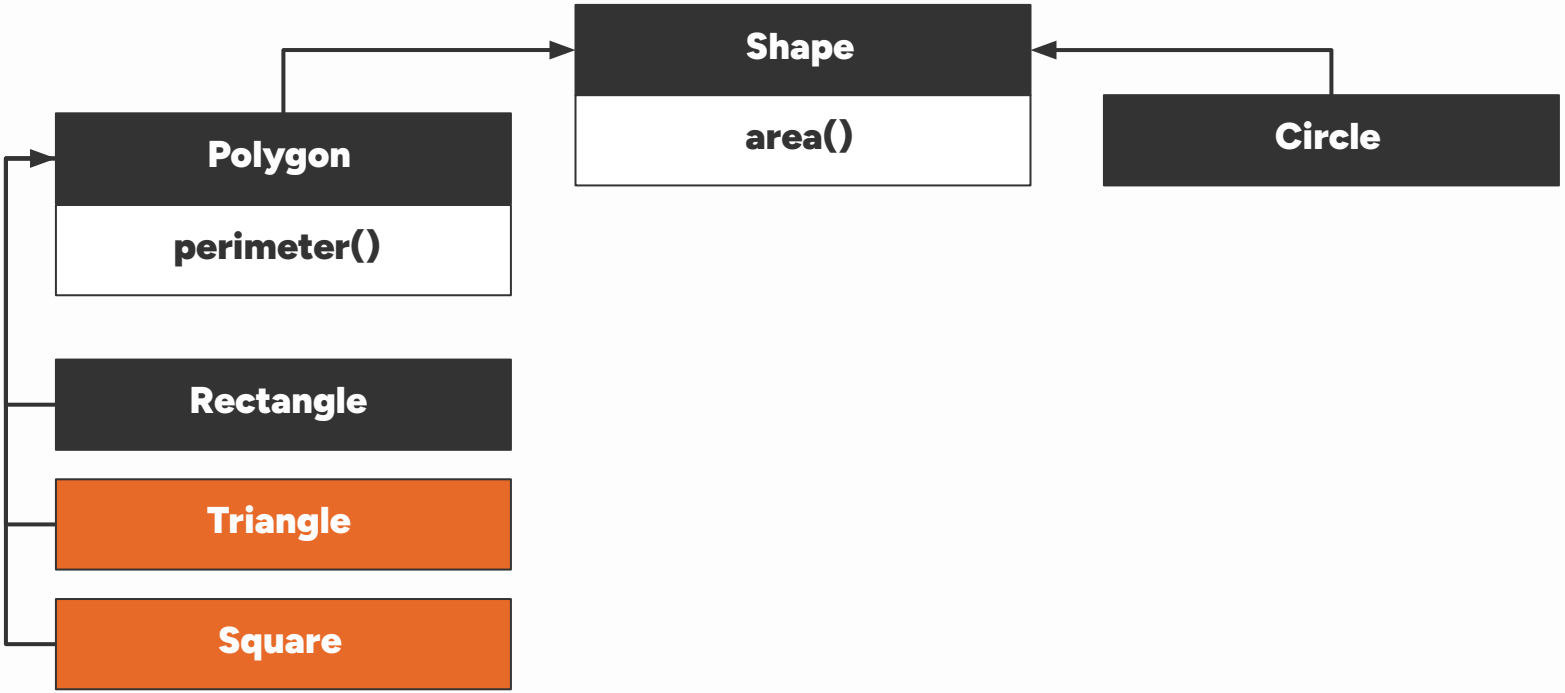
```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
class Polygon(Shape):
    @abstractmethod
    def perimeter(self):
        pass
```

Concrete Child

```
class Rectangle(Polygon):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def perimeter(self):
        return (self.width * 2) + (self.height * 2)

    def area(self):
        return self.width * self.height
```

Implement: Other Shapes



SOLID Principle

Conceptual Discussion on Design Principles

Single Responsibility Rule

A class should have only one reason to change. It should only have one job or responsibility.

```
class User:
    def __init__(self, name):
        self.name = name

    def save(self):
        print(f"Saving {self.name} to database")

    def send_email(self):
        print(f"Sending email to {self.name}")
```

Single Responsibility Rule

A class should have only one reason to change. It should only have one job or responsibility.

```
class User:
    def __init__(self, name):
        self.name = name

class UserRepository:
    def save(self, user):
        print(f"Saving {user.name} to database")

class EmailService:
    def send_email(self, user):
        print(f"Sending email to {user.name}")
```

Open/Closed Principle

Classes (even functions and modules) should be open for extension but closed for modification

```
class AreaCalculator:
    def calculate_area(self, shape):
        if isinstance(shape, Rectangle):
            return shape.width * shape.height
        elif isinstance(shape, Circle):
            return 3.14 * shape.radius ** 2
```

Open/Closed Principle

Classes (even functions and modules) should be open for extension but closed for modification

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2

class AreaCalculator:
    def calculate_area(self, shape):
        return shape.area()
```

```
class Shape:
    def area(self):
        pass
```

Liskov Substitution Principle

Subclasses must be able to substitute their superclass without issues

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def set_width(self, width):
        self.width = width

    def set_height(self, height):
        self.height = height

    def get_area(self):
        return self.width * self.height
```

```
class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)

    def set_width(self, width):
        self.width = width
        self.height = width

    def set_height(self, height):
        self.height = height
        self.width = height
```

Liskov Substitution Principle

Subclasses must be able to substitute their superclass without issues

```
class Shape:
    def get_area(self):
        pass
```

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height
```

```
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def get_area(self):
        return self.side * self.side
```

Interface Segregation Principle

Subclasses should not be forced to implement methods it doesn’t need

```
class CoffeeMachine:
    def make_espresso(self): pass
    def make_latte(self): pass
    def make_hot_chocolate(self): pass

class EspressoMachine(CoffeeMachine):
    def make_espresso(self):
        print("Espresso ready!")
    def make_latte(self):
        raise Exception("This machine can't make latte")
    def make_hot_chocolate(self):
        raise Exception("This machine can't make hot chocolate")
```

Interface Segregation Principle

Subclasses should not be forced to implement methods it doesn't need

```
class FancyMachine(
    EspressoMaker,
    LatteMaker,
    HotChocoMaker
):
    def make_espresso(self):
        print("Espresso ready!")
    def make_latte(self):
        print("Latte ready!")
    def make_hot_chocolate(self):
        print("Hot choco ready!")
```

```
class EspressoMaker:
    def make_espresso(self):
        Pass

class LatteMaker:
    def make_latte(self):
        pass

class TeaMaker:
    def make_tea(self):
        pass
```

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Rely on abstractions

```
class LightBulb:
    def turn_on(self):
        print("Light on")

    def turn_off(self):
        print("Light off")

class LightSwitch:
    def __init__(self, bulb):
        self.bulb = bulb

    def operate(self):
        self.bulb.turn_on()
```

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Rely on abstractions

```
class LightSwitch:
    def __init__(self, device):
        self.device = device

    def operate(self):
        self.device.turn_on()
```

```
class Switchable:
    def turn_on(self):
        pass

    def turn_off(self):
        pass

class LightBulb(Switchable):
    def turn_on(self):
        print("Light on")

    def turn_off(self):
        print("Light off")
```

F2

Employee System

High-level to code-level for a common scenario

Interface Segregation Principle

```
1  from abc import ABC, abstractmethod
2
3
4  class Worker(ABC):
5      @abstractmethod
6      def work(self):
7          pass
```

Dependency Inversion Layer

```
10  class WorkExecutor(ABC):
11      def __init__(self, worker):
12          self.worker = worker
13
14      def perform_work(self):
15          return self.worker.work()
```

Single Responsibility, Open/Close

```
18 class Employee:
19     def __init__(self, name, salary):
20         self.name = name
21         self.base_salary = salary
22 
23     def __str__(self):
24         return f"{self.name} - ${self.base_salary}"
```

Single Responsibility, Liskov Substitution

```
27 class Manager(Employee, Worker):
28     def __init__(self, name, salary, department):
29         super().__init__(name, salary)
30         self.department = department
31 
32     def work(self):
33         return f"{self.name} manages {self.department}."
```

Single Responsibility, Liskov Substitution

```
36 class Engineer(Employee, Worker):
37     def __init__(self, name, salary, tech_language):
38         super().__init__(name, salary)
39         self.tech_language = tech_language
40 
41     def work(self):
42         return f"{self.name} codes in {self.tech_language}."
```

Final Use Case

```
45 manager = Manager("Alice", 100000, "Engineering")
46 developer = Developer("Bob", 80000, "Python")
47
48 manager_worker = WorkExecutor(manager)
49 developer_worker = WorkExecutor(developer)
50
51 print(manager_worker.perform_work())
52 print(developer_worker.perform_work())
```

H2

Payroll System

Larger system design experience

Payroll System

Given the following interface, create subclasses for every type of employee

```
class SalaryCalculator(ABC):
    @abstractmethod
    def calculate_salary(self, employee):
        pass
```

Payroll System

Fill-in the details of the following class to complete the Payroll System

```
class PayrollSystem:
    def add_worker(self, worker, salary_calculator):
        """Add code to record new worker and salary calculator"""

    def process_payroll(self):
        total_payroll = 0
        """Add code to print total payroll (challenge: employee)"""
        print(f"Total Payroll: ${total_payroll:,.2f}")
```

Payroll System

Use the following to test your payroll system

```
manager = Manager("Alice", 100000, "Engineering")
developer = Developer("Bob", 80000, "Python")

manager_salary_calculator = ...
dev_salary_calculator = ...

payroll_system = PayrollSystem()

payroll_system.add_worker(manager, manager_salary_calculator)
payroll_system.add_worker(developer, dev_salary_calculator)

payroll_system.process_payroll()
```

```
Total Payroll: $194,000.00
```

03

Maintainability

Software Engineering Guidelines

Pythonic Code

Programming Language for humans

Example Code No. 1

```
1  def function(s):
2      ws = s.split()
3
4      vc = 0
5      vs  = "aeiou"
6
7      for w in ws:
8          if any(v in w for v in vs):
9              vc += 1
10
11     return vc
```

Example Code No. 1 (Refactor)

```
1  def count_words_with_vowel(text):
2      words = text.split()
3
4      words_with_vowels_count = 0
5      vowels= "aeiou"
6
7      for word in words:
8          if any(vowel in words for vowel in vowels):
9              words_with_vowels_count += 1
10
11     return words_with_vowels_count
```

Example Code No. 2

```
1  def function(is):
2      ic = {}
3
4      for i in is:
5
6          if i in ic:
7              ic[i] += 1
8          else:
9              ic[i] = 1
10
11     return ic
```

Example Code 2 (Refactor)

```
1  def count_per_item(items):
2      item_count = {}
3
4      for item in items:
5
6          if item in item_count:
7              item_count[item] += 1
8          else:
9              item_count[item] = 1
10
11     return item_count
```

“Code is read much more often
than it is written.”

— Guido van Rossum

Python ≈ English

Programming language

import this

If the implementation is **hard to explain**, it's a **bad idea**

Programming Principles

Don’t Repeat Yourself

Code duplication is a sign to use variables, functions, classes, and loops

Keep it Simple, Silly

Always aim for the simplest approach to the code

Loose Coupling

Minimize dependency of functions and classes with each other

Abstraction

Hide details in classes and functions to make things simpler at a quick glance

Python Enhancement Proposal (PEP) 8

Consistency

Makes it easier to read code quickly out of experience

Maintenance

PEP 8 is built for the purpose of making code easier to debug

Community

PEP 8 reflects the format and conventions that communities use

PEP 8 Quick Notes

Use 4 Spaces

Don’t use tabs and especially don’t mix spaces and tab

Start Private

If you’re not sure, start private as it’s harder to go from public to private

Limit to 79 Chars

Limit lines (72 characters for comments) to make code more readable or digestible

Naming Convention

Use snake_case for variables, functions, and files. Use PascalCase for classes.

PEP 8 Long Statements

For long operations, place the operator at the front

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

PEP 8 Extra Whitespaces

Avoid extra spaces as it is unnecessary

```
spam(ham[1], {eggs: 2})
```

```
spam( ham[ 1 ], { eggs: 2 } )
```

```
dct['key'] = lst[index]
```

```
dct ['key'] = lst [index]
```

```
x             = 1
y             = 2
long_variable = 3
```

PEP 8 Implicit Boolean Checks

If your variable is a Boolean, don’t use an equality check (remember, it auto-uses `bool()`)

```
if greeting == True:
```

```
if greeting is True:
```

```
if greeting:
```

Documentation

Notes for your colleagues and future self

Documentation

Provide Some Context

Note all of the prerequisites or key insights needed to understand a process. Mainly, explain why you are doing it

Enhance Readability

If a process is really hard to understand, explain it in alternative ways of phrasing

Summarize Immediately

One line can summarize paragraphs or entire documents depending on the use case

Hallmarks of a Good Comment/Name

Clear

Very specific and relevant

Updated

Outdated code is a severe liability

Not Redundant

Provide information not yet revealed

Proper Grammar

Keep it professional

Simple

A New Developer should follow it

References

Provide links to related or source of truth

Descriptive Variables

The variable name should be enough

```
x = 10
y = [1, 2]
data = “yes”
```

```
total_items = 10
list_of_attendees_per_day = [1, 2]
question01_response = “yes”
```

Consistent Variable Names

Do not suddenly shift your themes or word choice in-between code entities

```
customer_name = "John Doe"
client_age = 30 customer
shopper_order = ["apple", "banana", "orange"]
```

```
customer_name = "John Doe"
customer_age = 30 customer
customer_order = ["apple", "banana", "orange"]
```

Avoid Abbreviations

Make it very clear from the get-go

```
hrb = 5000
```

```
human_resources_budget = 5000
```

Inline Comments

Inline comments can be used to make quick notes or one-off explanations on why

```
# Convert temperature from Celsius to Fahrenheit
temperature_f = (temperature_c * 9/5) + 32
```

```
# This is a variable
x = 10

# This prints x
print(x)
```

Docstrings (Review)

Adding a multiline string after a function definition serves as a guide called docstring

```
def helpful_function():
    """
    Adding a multiline string after a function definition
    creates a guide when calling the help function
    """
    return 0

help(helpful_function)
```

```
help(helpful_function)
```

Docstring: Example Format

```
def calculate_circle_area(radius):
    """Calculates the area of a circle with the given radius.

    Args:
        radius (float): Circle's radius. Must be non-negative.

    Returns:
        float: Area of the circle.

    Raises:
        ValueError: If radius is negative.
    """
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius ** 2
```

Docstrings

Docstrings can still be used for simple functions. In this case, they span for a single line

```
def greet():
    """Print a simple greeting message."""
    print("Hello, welcome!")
```

Docstrings

Docstrings can also be used for classes.

```
class VideoPlayer:
    """Contains auxiliary functions for playing video files"""

    def __init__(self, video):
        self.video = video
```

Type Hinting

Saving yourself future debugging headaches

Type Hinting (Input)

You can provide a hint on what data type you're expecting for function parameters

```
def add(number1: int, number2: int):
    """Returns the mathematical summation of the two numbers.

    Args:
        number1 (int): First addend in summation
        number2 (int): Second addend in summation

    Returns:
        int: Addition of the two numbers
    """
    return number1 + number2
```

Type Hinting (Output)

You can provide a hint on what data type you're expecting for function outputs

```
def add(number1: int, number2: int) -> int:
    """Returns the mathematical summation of the two numbers.

    Args:
        number1 (int): First addend in summation
        number2 (int): Second addend in summation

    Returns:
        int: Addition of the two numbers
    """
    return number1 + number2
```

Type Hinting (Complete)

You can support more than one type of hinting

```
def add(number1: int|float, number2: int|float) -> int|float:
    """Returns the mathematical summation of the two numbers.

    Args:
        number1 (int|float): First addend in summation
        number2 (int|float): Second addend in summation

    Returns:
        int|float: Addition of the two numbers
    """
    return number1 + number2
```

Type Hinting Examples

There are a lot of built-in type hints for the standard data types and for nested data types

```
variable1: int = 1
```

```
variable2: list[int] = [1, 2, 3]
```

```
variable3: dict[str, int] = {“a”: 123, “b”: 456, “c”: 890}
```

```
variable4: dict[str, list[int]] = {“num1”: [1, 2, 3], “num2”: [4]}
```

```
variable5: tuple[int, int] = (0, 1)
```

```
variable6: list[tuple[int, int]] = [(9, 1), (2, 3), (5, 2)]
```

Variable Type Hinting

Type hints also work for regular variables. Here is an example of the syntax for data structures

```
total_tasks: int = 81

points: list[int] = [1, 2, 3]
priority: tuple[str, str, str] = [“low”, “medium”, “urgent”]

employees: dict[int, str] = dict()
employees.update({9823: “Jay”, 1821: “Caroline”})

downtime_logs: list[ dict[str, str] ] = [
    {“Engineering”: “Lunch”, “Finance”: “Team Building”},
    {“Security”: “Maintenance”},
    {“Hiring”: “Tax Filing”, “Engineering”: “System Update”},
]
```

Complex Type Hinting

For type hinting that is hard to read due to nesting, type hints can be stored in variables

```
UserData = dict[str, str|int|float]

users: list[UserData] = [
    {"name": "Alice", "email": "alice@example.com"},
    {"name": "Bob", "email": "bob@example.com"},
]
```

Typing Module

The typing module has additional typing and syntax for convenience

```
from typing import Literal, Iterable

priority = Literal[“low”, “medium”, “urgent”]
priorities: list[priority] = [“medium”, “urgent”, “urgent”, “low”]

def urgent_points(items: Iterable) -> int:
    urgent_point: int= 10
    return sum(urgent_point for item in items if item == “urgent”)
```

Class Typing: Pen and Paper

```
1  class Paper:
2      def __init__(self):
3          self.content = ""
4  class Pen:
5      def __init__(self, ink_level: int):
6          self.ink_level = ink_level
7
8      def write(self, paper: Paper, text: str):
9          if self.ink_level > 0:
10             paper.content += text
11
12 pen = Pen(100)
13 paper = Paper()
14 pen.write(paper)
15 print(paper.content)
```

Testing

Notes for your colleagues and future self

Common Types of Testing

Unit

Testing individual parts or functions in isolation

Integration

Testing if different components work together correctly

Regression

Testing if changes in the code doesn’t accidentally break anything

Unit Test

Testing individual components or functions in isolation from other parts

```
1  def square(x):
2      return x * x
3
4  def test_square():
5      assert square(2) == 4
6      assert square(-3) == 9
7      assert square(0) == 0
8      print("All unit tests passed!")
9
10 test_square()
```

Integration Test

Testing if different components work as intended when combined together

```
1  def add(a, b):
2      return a + b
3
4  def square(x):
5      return x * x
6
7  def multiply(a, b):
8      return a * b
9
```

Integration Test

Testing if different components work as intended when combined together

```
10  def calculate_expression(x, y):
11      return add(square(x), multiply(y, 2))
12
13  def test_calculate_expression():
14      assert calculate_expression(2, 3) == 10
15      assert calculate_expression(0, 5) == 10
16
17      print("All integration tests passed!")
18
19  test_calculate_expression()
```

Regression Test

Check if changes in the code have not affected existing functionality

```
10  def calculate_expression(x, y, z=0):
11      return add(square(x), multiply(y, 2)) - z
12
13  def test_calculate_expression():
14      assert calculate_expression(2, 3) == 10
15      assert calculate_expression(0, 5) == 10
16      assert calculate_expression(2, 3, 2) == 10
17      print("All integration tests passed!")
18
19  test_calculate_expression()
```

Pytest Framework

The **pytest** framework is one of the most common testing frameworks, known for its simplicity, scalability, and powerful features.

```
$ pip install pytest
```

For as long as the function has **test** at the start of its name, it will be detected as a test.

```
def test_sanity():
    assert len([99, 98, 97]) == 3
```

```
$ pytest
```

Pytest Classes

Tests can be grouped into classes for further organization

```
1  class TestClass:
2      def test_one(self):
3          word = "this"
4          assert "h" in word
5
6      def test_two(self):
7          word = "hello"
8          assert hasattr(word, "check")
```

Standard Packaging Format (Review)

Most Python projects follow this project structure:

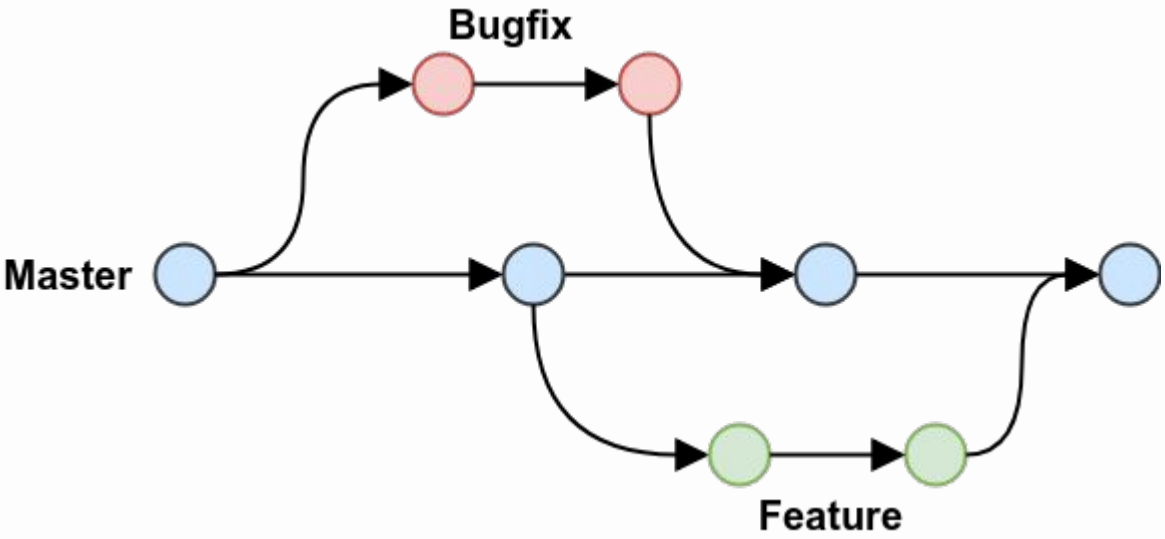
```
.
└── project_name/
    ├── ...
    └── src/
        ├── example_package_1/
        ├── example_package_2/
        └── tests
            ├── example_package_1/
            │   └── test_package_1.py
            └── example_package_2/
                └── test_package_.py
```

Version Control

Taught in the context of git

Git

`Git` is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.



Git Installation Summary

Here are the key option selections for each prompt during the git installation:

Default Editor	Keep as is
Initial Branch Name	Keep as is
Path Setup	Keep as is
SSH Executable	Keep as is
HTTPS Transport Backend	Keep as is
Checkout	Keep as is
Terminal Emulator	**Use Windows Default Console Window**
Git Pull	Keep as is
Credential Helper	Keep as is
Extra Options	Keep as is

Git Project Setup

Run the following on your chosen terminal to setup commits and remote connections

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your@email.com"
```

For every new project, open the project terminal in the terminal and run this

```
$ git init
```

Git Clone

To create a local copy of an online repository, run this command. This doesn't need **`git init`**

```
$ git clone source
```

Here is an example of an existing repository from Github

```
$ git clone https://github.com/Ayumu098/quotes.git
```

Git Create Branch

To see the list of existing branches, run the following command

```
$ git branch
```

To create a new branch in your repository, run the following command

```
$ git switch -c feature/my-feature
```

Git Stage

To save changes in your local repository, you need to stage or note what files to track.

```
$ git add filename1.py
$ git add filename2.py
$ ...
```

You can determine what files have been modified from last time with this command

```
$ git status
```

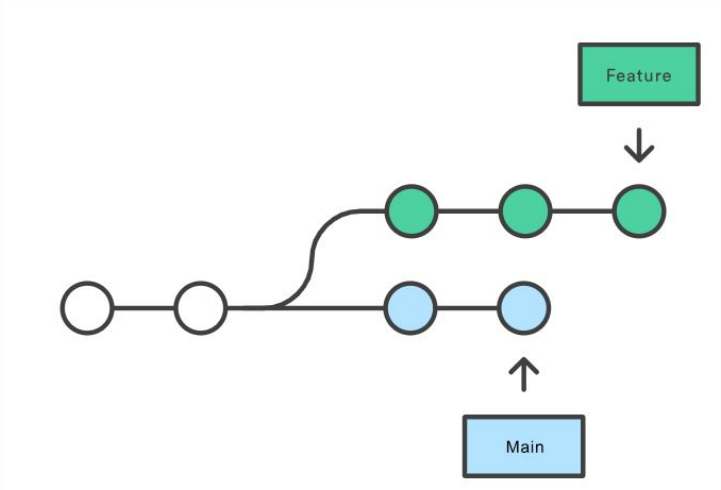
You can also stage all of the changes using this command

```
$ git add .
```

Git Commit

After staging the changes, the last step to saving the changes locally is to commit.

```
$ git commit -m "Describe changes (Verb - Subject - Details)"
```



Git Pull

To ensure the current branch is in sync with the online repository, run the following

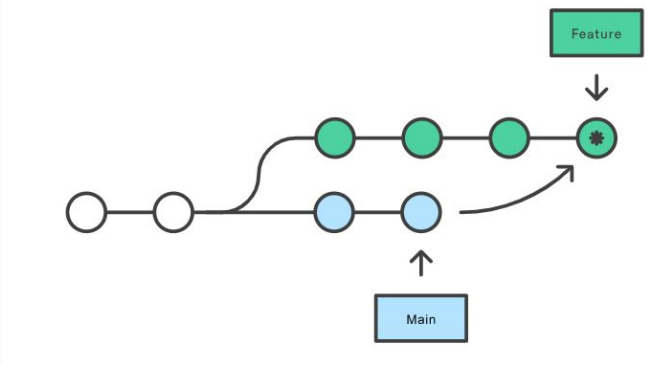
```
$ git switch main
$ git pull --rebase origin main
$
$ git switch feature/my-feature
$ git pull --rebase origin main
```

Git Push and Pull Requests

Finally, reflect the changes in the feature branch to the online repository with this command

```
$ git push origin feature/my-feature
```

To merge the changes in the feature with the develop or main branch, make a pull request on your chosen online repository platform. It can be done in console but this is better for code reviews and tests.



04

Streamlit

Modern web app framework for simple, data-driven use cases

Benefits of Graphical User Interfaces

User Experience

Easier to understand and provides appeal and interactivity

Separation

Clearly Separate Frontend (Design, UI/UX) and Backend (Logic)

Limitations

Limit the possible edge cases and directly get needed data type

Python GUI Libraries

Tkinter

Standard GUI toolkit available in (almost) all Python distributions immediately. Easy to understand and great for building simple applications quickly.

PyQt

Python bindings or implementations for the Qt application framework. It has a lot of flexible components and great for building complex applications.

Kivy

Library built specifically for multi-touch platforms (mobile) but can be used in Desktops as well. Good for complex, cross-platform applications.

Web Frameworks

Flask

- Minimalist and lightweight
- Freedom to choose tools for each part
- **Small and Fast Web Applications**

Django

- Multiple out-of-the-box features
  - Object Relational Mapping
  - Fully functional Admin Panel
  - Security Measures and Authentication
- **Medium to Large Web applications**

A faster way to build and share data apps

Turn your data scripts into shareable web apps in minutes.
All in pure Python. No front-end experience required.

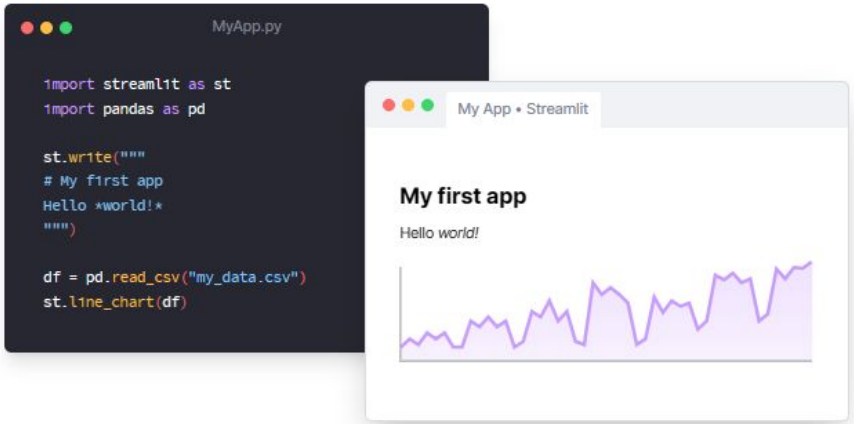
Get started Try the live playground!

On Streamlit.

Learn more with the Streamlit crash course on YouTube

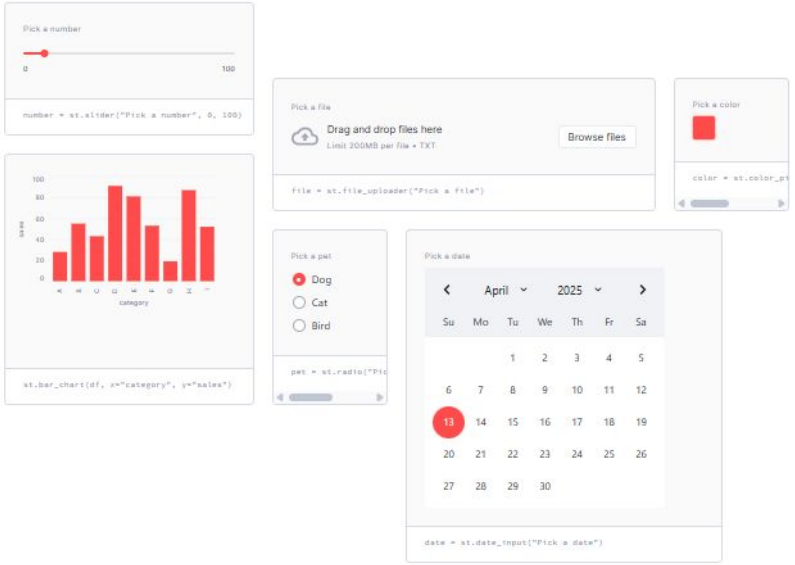
Embrace scripting

Build an app in a few lines of code with our **magically simple API**. Then see it automatically update as you iteratively save the source file.



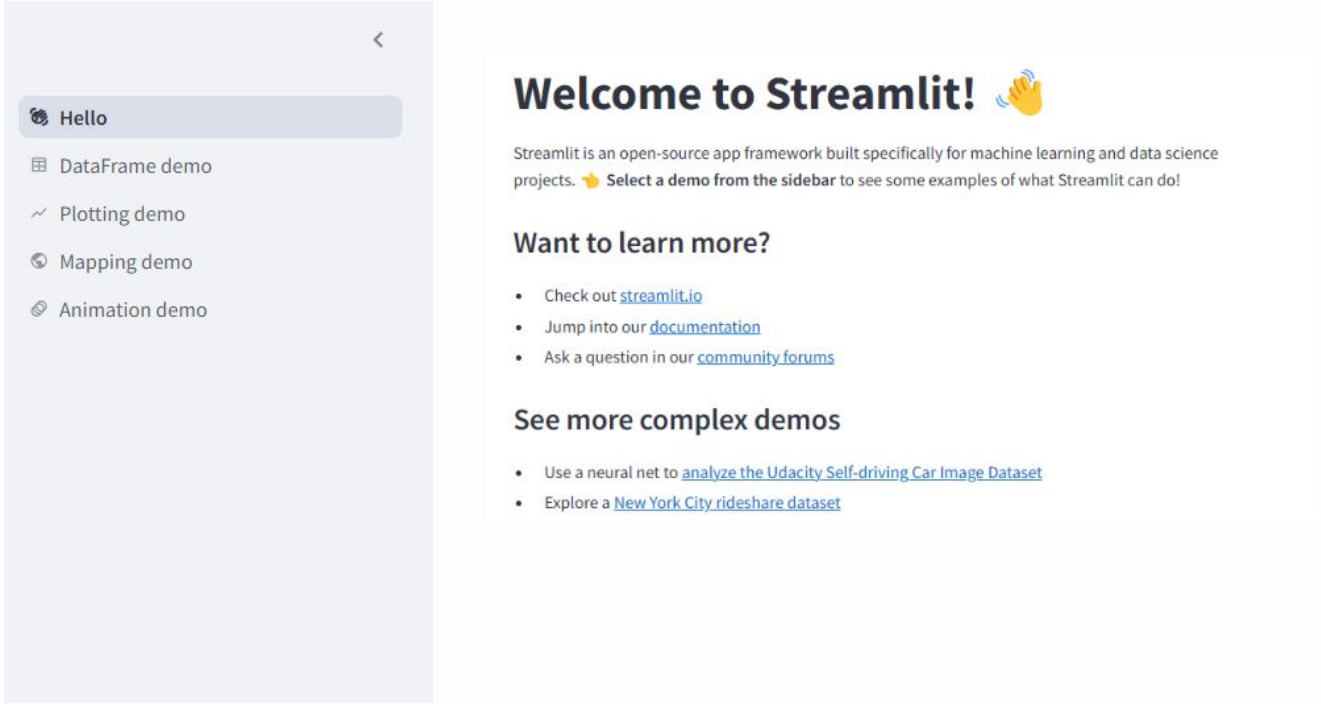
Weave in interaction

Adding a widget is the same as **declaring a variable**. No need to write a backend, define routes, handle HTTP requests, connect a frontend, write HTML, CSS, JavaScript, ...



Get started in under a minute

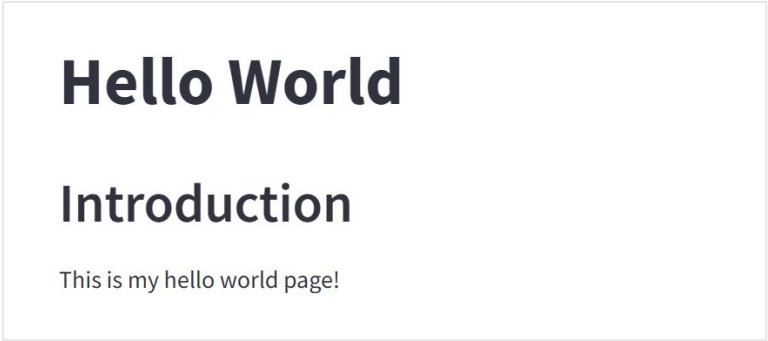
```
$ pip install streamlit
$ streamlit hello
```



Streamlit: Hello World

Make a new file with the following Python code.

```
import streamlit as st

st.title("Hello World")
st.header("Introduction")
st.text("This is my hello world page!")
```



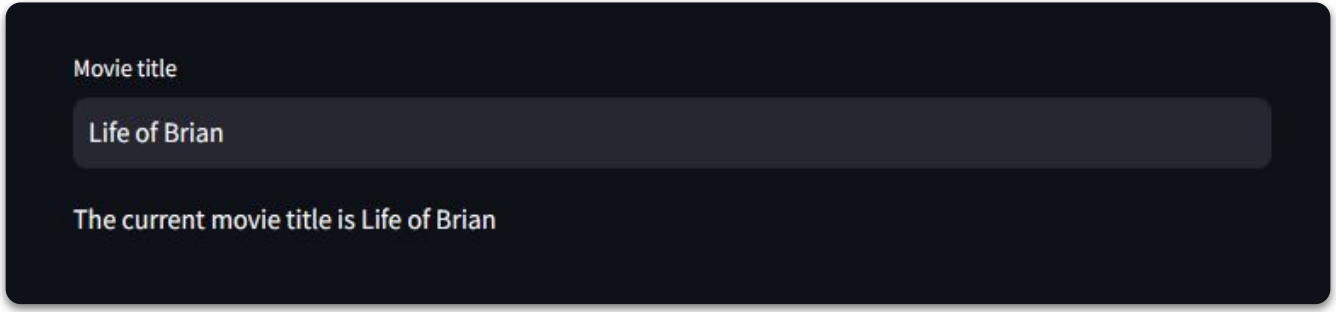
Components

Learn some of the available interactive elements

Text Input

The **`st.text_input`** displays a single-line text input widget.

```
import streamlit as st

title = st.text_input("Movie title", "Life of Brian")
st.write("The current movie title is", title)
```



Radio Buttons

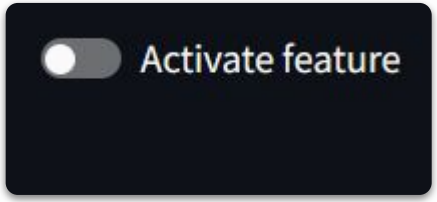
The **`st.radio`** displays a radio button widget

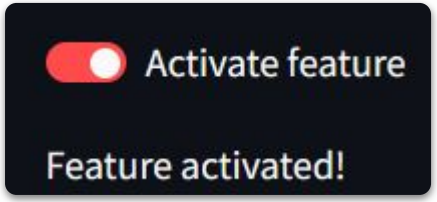
```
import streamlit as st

genre = st.radio(
    "What's your favorite movie genre",
    [":rainbow[Comedy]", "***Drama***", "Documentary :movie_camera:"],
    index=None,
)

st.write("You selected:", genre)
```

Toggle

The **`st.toggle`** displays a slider widget for integers, time, and datetime values

```
import streamlit as st

on = st.toggle("Activate feature")

if on:
    st.write("Feature activated!")
```

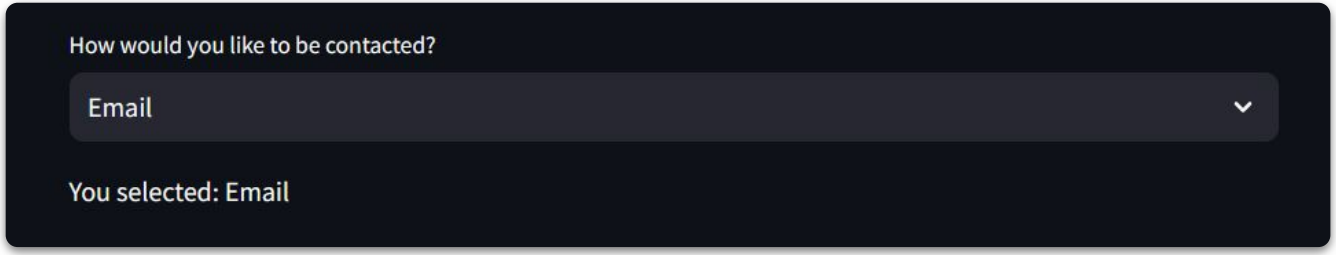




Select Box

The **`st.select_box`** displays a select widget for choosing a single value

```
import streamlit as st

option = st.selectbox(
    "How would you like to be contacted?",
    ("Email", "Home phone", "Mobile phone"),
)

st.write("You selected:", option)
```



Multiselect

The **`st.multiselect`** displays a multiselect widget

```
import streamlit as st

options = st.multiselect(
    "What are your favorite colors",
    ["Green", "Yellow", "Red", "Blue"],
    ["Yellow", "Red"],
)

st.write("You selected:", options)
```

Number Input

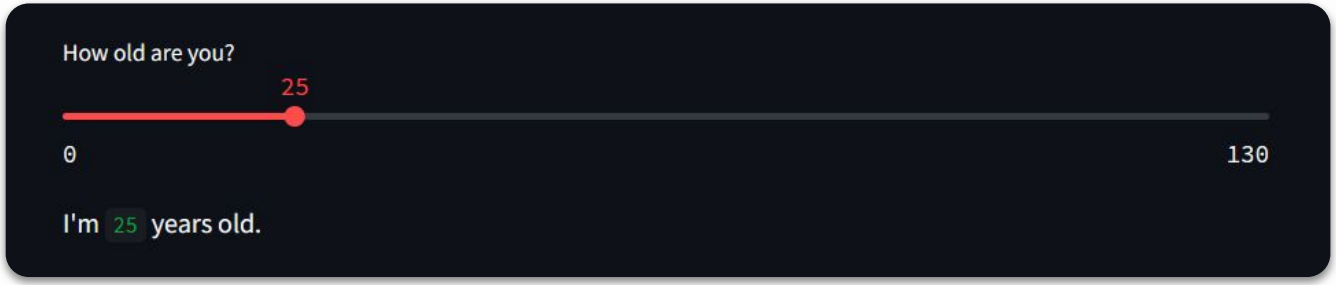
The **`st.number_input`** displays a numeric input widget

```python
import streamlit as st

number = st.number_input(
    "Insert a number", value=None, placeholder="Type a number..."
)
st.write("The current number is ", number)
```

Slider

The **`st.slider`** displays a slider widget for integers, time, and datetime values

```
import streamlit as st

age = st.slider("How old are you?", 0, 130, 25)
st.write("I'm ", age, "years old")
```



Submit Form

The **`st.form`** ensures that every input change doesn't refresh the page every time

```
import streamlit as st

with st.form("my_form"):
   st.write("Inside the form")
   my_number = st.slider('Pick a number', 1, 10)
   my_color = st.selectbox('Pick a color', ['red','orange','green','blue','violet'])
   st.form_submit_button('Submit my picks')

# This is outside the form
st.write(my_number)
st.write(my_color)
```

Data Handling

Process and visualize more data-intensive processes

Upload Files

Run the following on your chosen terminal to setup commits and remote connections

```
import streamlit as st

uploaded_files = st.file_uploader(
    "Choose a CSV file", accept_multiple_files=True
)
for uploaded_file in uploaded_files:
    bytes_data = uploaded_file.read()
    st.write("filename:", uploaded_file.name)
    st.write(bytes_data)
```

Read CSV and Excel File

Run the following on your chosen terminal to setup commits and remote connections

```
import streamlit as st
import pandas as pd

uploaded_file = st.file_uploader(“File:”, type=["csv", "xlsx", "xls"])

if uploaded_file is not None:
    st.write(f"Uploaded file: {uploaded_file.name}")

    if uploaded_file.name.endswith(".csv"):
        df = pd.read_csv(uploaded_file)
    elif uploaded_file.name.endswith((".xlsx", ".xls")):
        df = pd.read_excel(uploaded_file)

    st.write(df)
```

Basic Pandas - Access

Pandas converts tabular data to dataframes that are convenient to read and access

```
1  import pandas as pd
2
3  df = pd.read_csv("data/sales/accounts.csv")
4  print(df)
```

To access a single column, use the index operation and the column name as key

```
5  print(df[“account”])
6
7  for account in df[“account”]:
8      print(account)
```

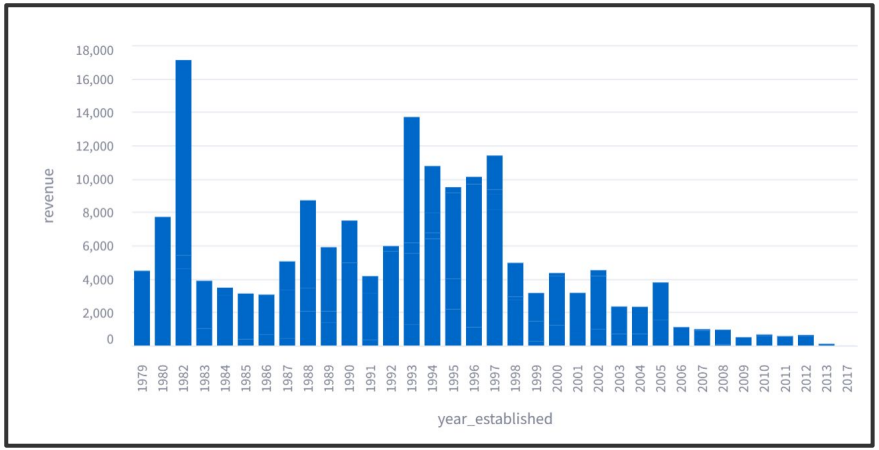
Basic Pandas - Filtering

One of the key benefits to using pandas is the ease of filtering

```
1  import pandas as pd
2
3  df = pd.read_csv("data/sales/accounts.csv")
4
5  quota = 1000
6  quota_fulfilled_filter = df["revenue"] >= quota
7  print(quota_fulfilled_filter )
8
9  quota_fulfilled = df[quota_fulfilled_filter]
10 print(quota_fulfilled)
```

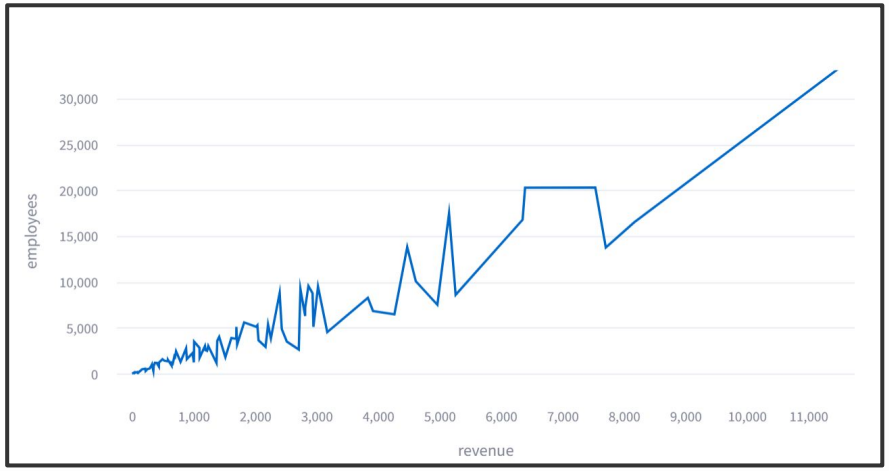
Bar Chart

```
1  import streamlit as st
2  import pandas as pd
3
4  df = pd.read_csv("data/sales/accounts.csv")
5  st.bar_chart(df, x="year_established", y="revenue")
```



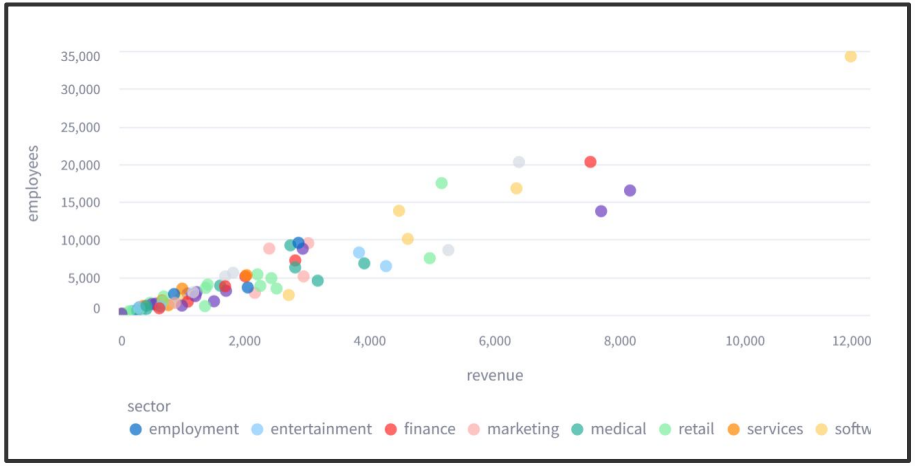
Line Plot

```
1  import streamlit as st
2  import pandas as pd
3
4  df = pd.read_csv("data/sales/accounts.csv")
5  st.line_chart(df, x="revenue", y="employees")
```



Scatter Chart

```python
import streamlit as st
import pandas as pd

df = pd.read_csv("data/sales/accounts.csv")
st.scatter_chart(df, x="revenue", y="employees", color="sector")
```

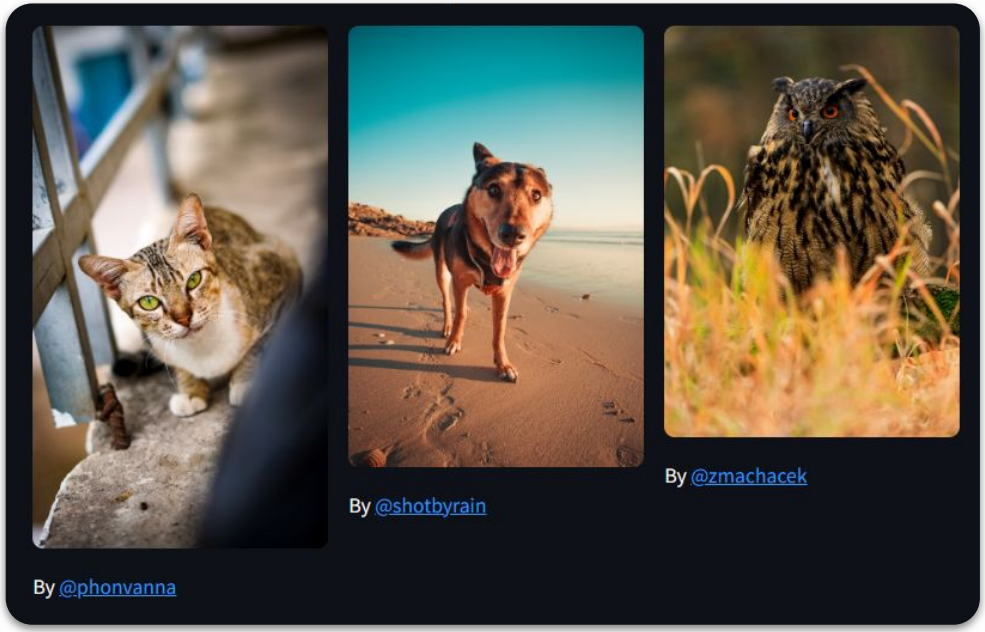


Modularization

High-level Streamlit code organization

Column Layouting

Streamlit supports multi-column layouts



Columns

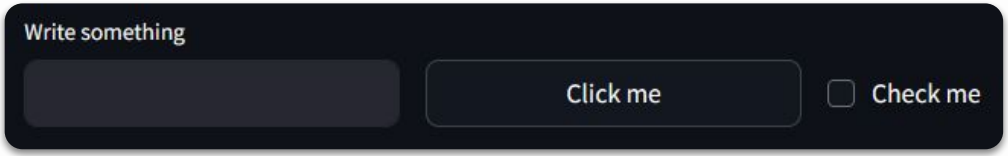
Using the context handler **`with`** syntax, content will be divided into separate columns

```
import streamlit as st

col1, col2, col3 = st.columns(3)

with col1:
    st.header("A cat")
    st.image("https://static.streamlit.io/examples/cat.jpg")

with col2:
    st.header("A dog")
    st.image("https://static.streamlit.io/examples/dog.jpg")

with col3:
    st.header("An owl")
    st.image("https://static.streamlit.io/examples/owl.jpg")
```

Simple Column Layout

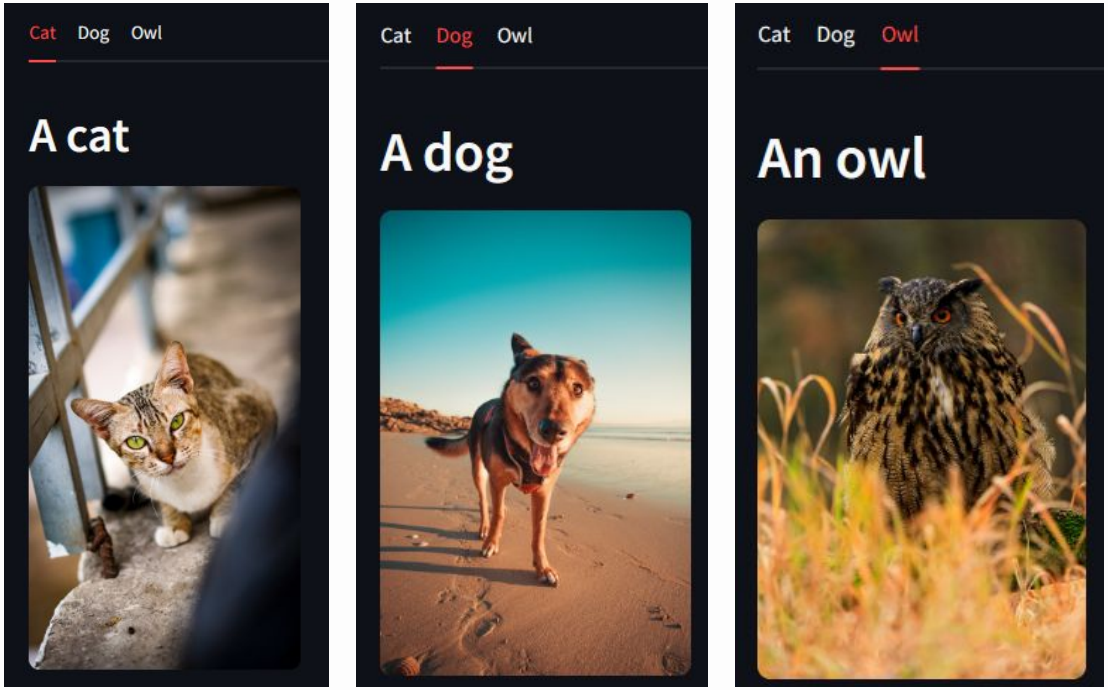
For simple columns, **`st`** can be replaced with the given column name

```
import streamlit as st

left, middle, right = st.columns(3, vertical_alignment="bottom")

left.text_input("Write something")
middle.button("Click me", use_container_width=True)
right.checkbox("Check me")
```



Tabs

Streamlit also supports tab layouts to prevent cluttering the page



Tabs

Using the context handler **`with`** syntax, content will be divided into separate tabs

```
import streamlit as st

tab1, tab2, tab3 = st.tabs(["Cat", "Dog", "Owl"])

with tab1:
    st.header("A cat")
    st.image("https://static.streamlit.io/examples/cat.jpg", width=200)
with tab2:
    st.header("A dog")
    st.image("https://static.streamlit.io/examples/dog.jpg", width=200)
with tab3:
    st.header("An owl")
    st.image("https://static.streamlit.io/examples/owl.jpg", width=200)
```

Multiple Pages

Multiple subpages are easy to implement in Streamlit. Place subpages in the **pages/** folder

```
.
└── project_name/
    ├── ...
    └── src/
        ├── pages/
        │   ├── subpage1.py
        │   ├── subpage2.py
        │   └── subpage3.py
        └── main.py
```

Report Generator

Visual demonstration of Streamlit’s capabilities

Report Generator

Upload CSV

Drag and drop file here

Limit 200MB per file • CSV

Browse files

accounts.csv 4.6KB

Year Established

1979

Data Chart

	account	sector	year_established	revenue	employees	office_location	subsidiary_of
83	Zotware	software	1979	4478.47	13809	United States	None

06

Lab Session

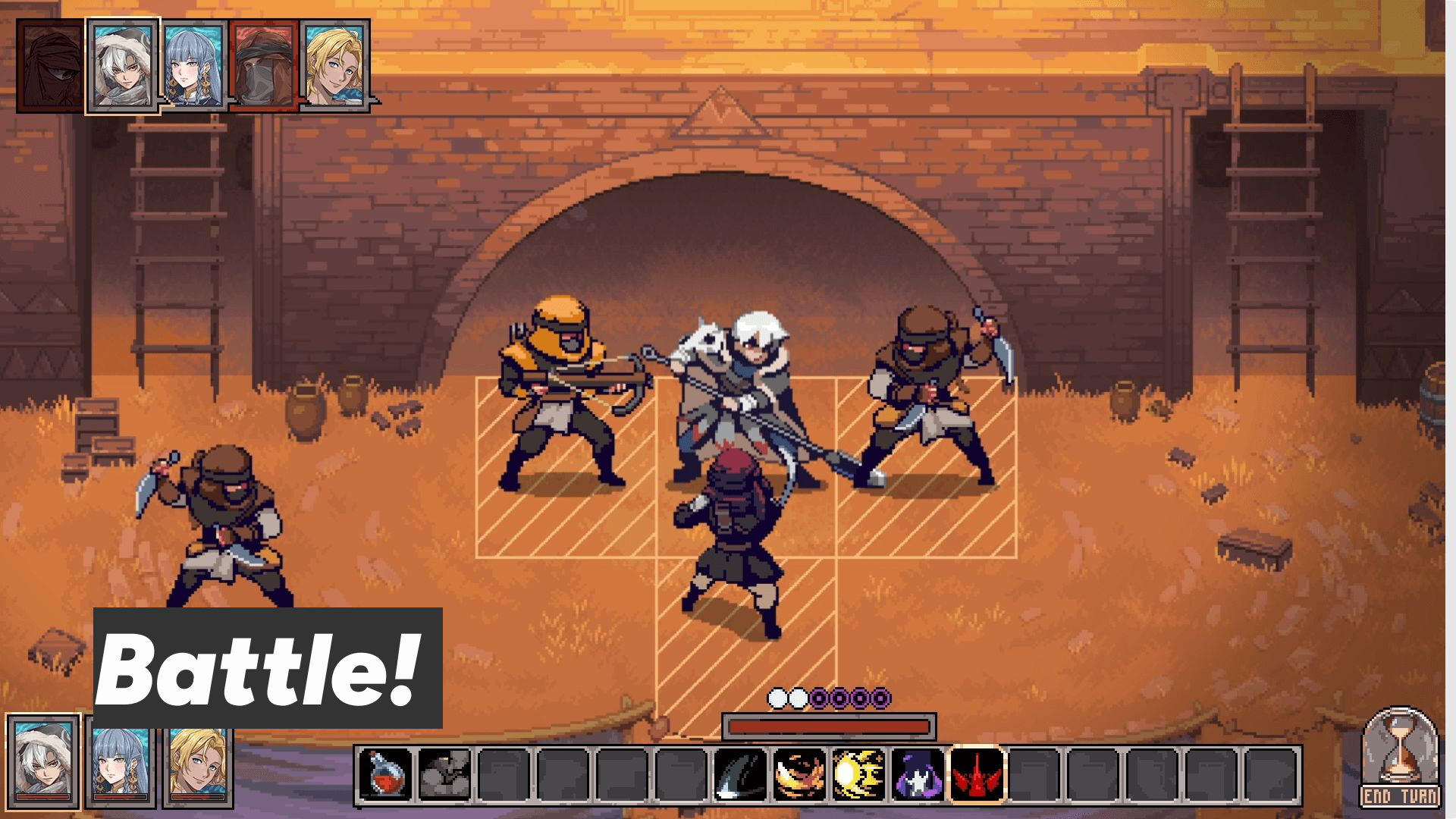
All the Major Features Covered

Business Sim

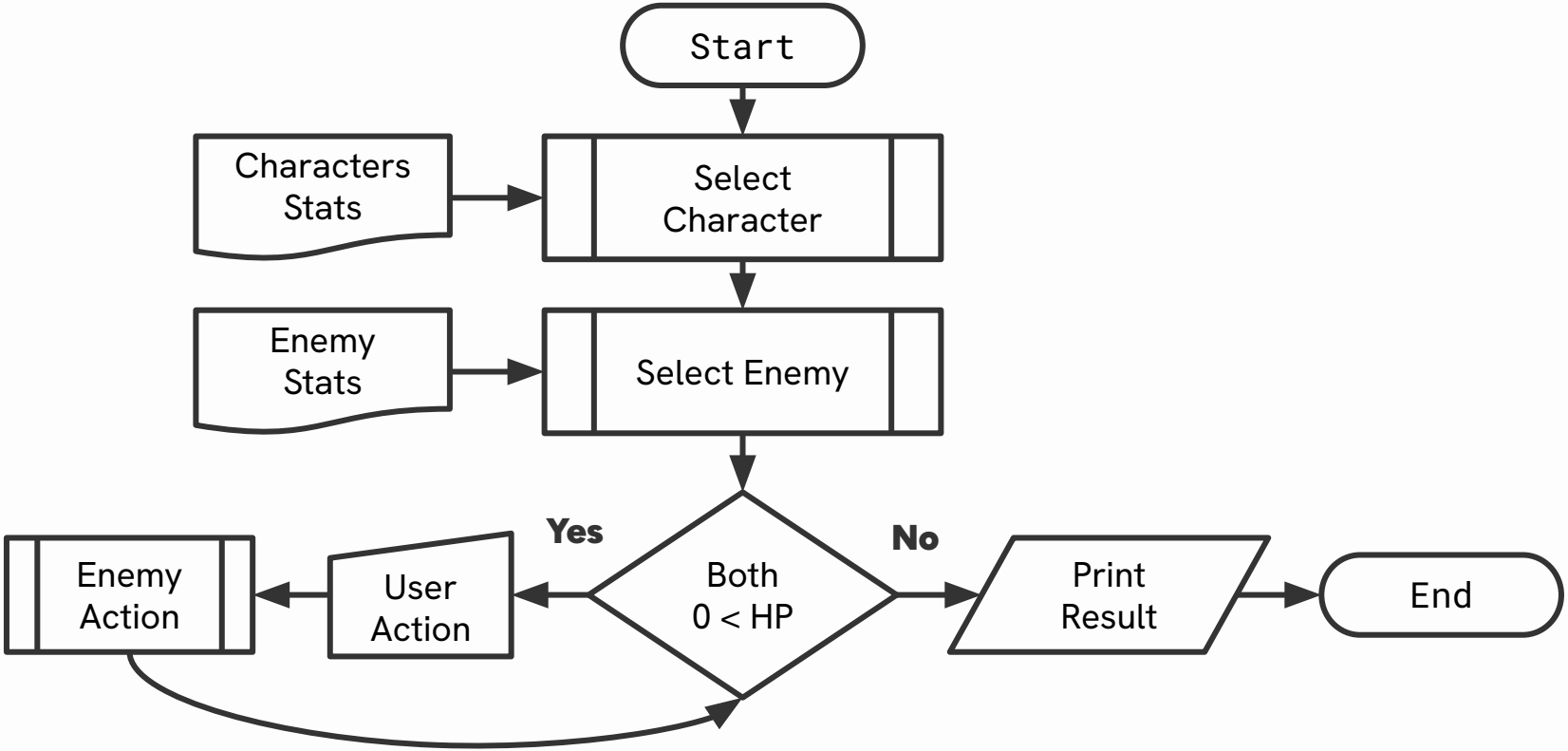
Business Simulation

You will develop a simulation for a medium-sized business with the following specifications:

- Employee System for representing employee information
- Payroll System for calculating salary payments
- Customer System for representing customer information
- Simulator for generating random customers payments and simulating the passage of time (based on user)
- Dashboard for visualizing key details from the systems


Battle!
END TURN

Battle! Game Flow




Card of War

Review: Deck of Cards

```
1  def create_deck() -> list[str]:
2      # Return a list of 52 strings containing a standard deck
3
4  def draw_top(deck: list[str], count: int =1)-> list[str]:
5      # Remove count return count cards from the start from deck
3
4  def draw_bottom(deck: list[str], count: int =1) -> list[str]:
5      # Remove and return count cards from the end of the deck
6
7  def draw_random(deck: list[str], count: int =1) -> list[str]:
8      # Remove and return count random cards from the deck
9
10 def show(deck):
11     # Print all cards in deck
```

Review: Dynamic Adding

```
12  def add_top(deck: list[str], other: list[str])-> list[str]:
13      # Add cards in other to the first parts of deck
14
15  def add_bottom(deck: list[str], other: list[str])-> list[str]:
16      # Add cards in other to the last parts of deck
17
18  def add_random(deck: list[str], other) -> list[str]:
19      # Add cards in other randomly to deck
```

Template: Deck of Cards

```
class Deck:
    def __init__(self)
        self.cards = [...]

    def draw_top(self, count: int = 1)-> list[str]:
    def draw_bottom(self, count: int = 1) -> list[str]:
    def draw_random(self, count: int = 1) -> list[str]:
    def show(self):
    def add_top(self, other: list[str]):
    def add_bottom(self, other: list[str]):
    def add_random(self, other: list[str]):
```

War: Standard Rules

Here is the standard rules of War:

- A standard deck is face-down (not visible), shuffled and evenly split between two players
- The game continues until one player has all of the cards. For every round:
  - Both players flip their top card.
  - The player with the higher card wins both and adds them to their victory deck
  - If the cards are equal, it's WAR:
    - Each player puts down 3 cards face-down and a 4th face-up.
    - The higher face-up card wins all 10 cards.
    - Repeat if tied again
- Player with the most card wins

Sneak Peak

01

Learn: OpenPyXL

Lightweight Excel Handler

02

Learn: XLWings

Dynamic Excel Handler

03

Learn: Pandas

Tables as Dataframes

04

Learn: Win32Com

Windows Control

05

Lab Session

Final Activities

Python: Day 03

Professional Development