1. Employee Salary with Bonus Logic

```
class Employee:
    def __init__(self, name, base_salary):
        self.__name = name
        self.__salary = base_salary

    def add_bonus(self, bonus):
        if bonus < 0:
            raise ValueError("Bonus cannot be negative.")
        self.__salary += bonus

    def get_details(self):
        return f"Employee: {self.__name}, Salary: {self.__salary}"

emp = Employee("Alice", 50000)
emp.add_bonus(5000)
print(emp.get_details())
```

Key Points :

1. The Employee class encapsulates name and salary as private attributes.
2. A method add_bonus() is used to add bonus to the salary.
3. add_bonus() method includes validation to avoid negative bonus values.
4. The get_details() method returns the final salary along with the name.
5. Later we print the employee's details with name and salary.

2. Validated Bank Account with Deposit and Withdraw

```
class BankAccount:
    def __init__(self, owner, balance):
        self.__owner = owner
        self.__balance = balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Invalid deposit amount.")
        self.__balance += amount

    def withdraw(self, amount):
        if amount > self.__balance:
            raise ValueError("Insufficient funds.")
        self.__balance -= amount

    def get_balance(self):
        return self.__balance

acc = BankAccount("John", 1000)
acc.deposit(500)
acc.withdraw(200)
print("Balance:", acc.get_balance())
```

Key Points :

1. __balance and __owner are private attributes.
2. deposit() ensures only positive amounts are added.
3. withdraw() checks if the balance is sufficient before deducting.
4. withdraw() prevents invalid operations like over-withdrawing.
5. get_balance() helps in securely accessing account details.
6. Later we print the Balance of the account.

3. Encapsulation with Password Protection

```
class User:
    def __init__(self, username, password):
        self.__username = username
        self.__password = password

    def authenticate(self, input_password):
        return self.__password == input_password

    def get_username(self):
        return self.__username

user = User("admin", "12345")
print(user.authenticate("12345"))
print(user.authenticate("abc"))
```

Key Points :

1. username and password are stored privately.
2. The authenticate() method checks if the entered password is correct.
3. It returns True for valid input and False otherwise.
4. Username can be accessed using get_username().

4. Encapsulated Stock Portfolio Tracker

```
class StockPortfolio:
    def __init__(self):
        self.__stocks = {}

    def add_stock(self, symbol, quantity):
        if quantity <= 0:
            raise ValueError("Invalid quantity.")
        self.__stocks[symbol] = self.__stocks.get(symbol, 0) + quantity

    def remove_stock(self, symbol, quantity):
        if symbol not in self.__stocks or self.__stocks[symbol] <
quantity:
            raise ValueError("Not enough stock to remove.")
        self.__stocks[symbol] -= quantity

    def get_holdings(self):
        return self.__stocks

portfolio = StockPortfolio()
portfolio.add_stock("AAPL", 10)
portfolio.add_stock("TSLA", 5)
portfolio.remove_stock("AAPL", 5)
print(portfolio.get_holdings())
```

Key Points :

1. Stocks are stored privately in a dictionary.
2. add_stock() increases quantity or adds new symbols.
3. remove_stock() decreases quantity with validation.
4. Invalid stock operations raise errors.
5. get_holdings() returns the current stock portfolio.

5. Student Grades with Private Data

```
class Student:
    def __init__(self, name):
        self.__name = name
        self.__grades = []

    def add_grade(self, grade):
        if not (0 <= grade <= 100):
            raise ValueError("Invalid grade.")
        self.__grades.append(grade)

    def get_average(self):
        return sum(self.__grades) / len(self.__grades)

student = Student("Emma")
student.add_grade(90)
student.add_grade(80)
print(f"Average: {student.get_average()}")
```

Key Points :

1. Grades are stored privately using a list.
2. Grades between 0 and 100 are valid.
3. All grades go into a list.
4. The average of all grades is calculated using get_average().
5. Later we print the Average.

6. Property Access with Read/Write Control

```
class Temperature:
    def __init__(self):
        self.__celsius = 0

    @property
    def celsius(self):
        return self.__celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Invalid temperature.")
        self.__celsius = value

temp = Temperature()
temp.celsius = 25
print(temp.celsius)
```

Key Points :

1. The class uses @property for controlled access.
2. @property allows read access to temperature.
3. setter allows us to set it but with a lower limit check.
4. If we try to set a value below -273.15, it gives an error.
5. Later we print the Temperature.

7. Smart Lock Device

```
class SmartLock:
    def __init__(self, pin):
        self.__pin = pin
        self.__locked = True

    def unlock(self, input_pin):
        if input_pin == self.__pin:
            self.__locked = False
        else:
            print("Incorrect PIN")

    def lock(self):
        self.__locked = True

    def is_locked(self):
        return self.__locked

lock = SmartLock("1234")
lock.unlock("1234")
print("Locked?", lock.is_locked())
```

Key Points :

1. PIN is stored privately.
2. unlock() verifies the entered PIN before unlocking.
3. Incorrect PIN does not change the lock status.
4. lock() can relock the system when needed.
5. is_locked() lets us check the current status.

8. Employee Details with Computed Property

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    @property
    def annual_salary(self):
        return self.__salary * 12

    def get_name(self):
        return self.__name

emp = Employee("Sara", 5000)
print(emp.get_name(), emp.annual_salary)
```

Key Points :

1. The class keeps name and salary private.
2. annual_salary is calculated from a private salary.
3. The get_name() method returns the employee's name.
4. Later we print the Employee name and annual salary.

9. Encapsulated Voting System

```
class VotingMachine:
    def __init__(self):
        self.__votes = {}

    def vote(self, candidate):
        self.__votes[candidate] = self.__votes.get(candidate, 0) + 1

    def result(self):
        return sorted(self.__votes.items(), key=lambda x: x[1],
reverse=True)

vm = VotingMachine()
vm.vote("Alice")
vm.vote("Bob")
vm.vote("Alice")
print(vm.result())
```

Key Points :

1. The system stores votes in a private dictionary.
2. Users can only vote via the vote() method.
3. vote() increments vote count for a candidate.
4. result() returns sorted results by highest votes.
5. Later we print the result.

10. Hotel Room Booking with Access Control

```
class HotelRoom:
    def __init__(self, room_no):
        self.__room_no = room_no
        self.__is_booked = False

    def book(self):
        if self.__is_booked:
            raise Exception("Room already booked.")
        self.__is_booked = True

    def status(self):
        return "Booked" if self.__is_booked else "Available"

room = HotelRoom(101)
room.book()
print(room.status())
```

Key Points :

1. Room number and booking status are private.
2. book() prevents double booking by checking status.
3. If already booked, it raises an error.
4. status() clearly tells whether the room is booked or not.
5. Later we print the room status whether its available or it is already booked.

11. Payment Interface using Abstraction

```
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self, amount): pass

class CreditCard(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} using Credit Card")

class UPI(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} using UPI")
```

```
def checkout(method: Payment, amt):
    method.pay(amt)

checkout(CreditCard(), 500)
checkout(UPI(), 200)
```

Key Points :

1. The Payment class is abstract with a pay() method.
2. CreditCard and UPI both give their own version of pay().
3. checkout() accepts any payment type following the interface.
4. We can use the same checkout function for both.
5. Later we print the amount paid using UPI and Credit Card.

12. Abstract Shape Class

```
from abc import ABC, abstractmethod

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

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

sh = Circle(3)
print("Area:", sh.area())
```

Key Points :

1. The abstract Shape class has an abstract area() method.
2. Circle implements the abstract area() method.
3. It uses formula to calculate circle area using radius.
4. Later we print the area of the circle.

13. Abstract Animal Sound Generator

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self): pass

class Dog(Animal):
    def sound(self):
        print("Woof")
```

```
class Cat(Animal):
    def sound(self):
        print("Meow")

animals = [Dog(), Cat()]
for animal in animals:
    animal.sound()
```

Key Points :

1. Animal is an abstract class with a sound() method.
2. Dog and Cat override this method with their own sounds.
3. Demonstrates method overriding and runtime polymorphism.
4. We use a list to call sound on both objects.

14. Report Generator Template

```
from abc import ABC, abstractmethod

class ReportGenerator(ABC):
    def generate(self):
        self.fetch_data()
        self.format_data()
        self.export()

    @abstractmethod
    def fetch_data(self): pass

    @abstractmethod
    def format_data(self): pass

    def export(self):
        print("Exporting as PDF")

class SalesReport(ReportGenerator):
    def fetch_data(self):
        print("Fetching sales data")

    def format_data(self):
        print("Formatting data")
```

Key Points :

1. ReportGenerator is an abstract class for reports.
2. generate() provides a fixed structure for report generation.
3. Child classes define their own fetch_data() and format_data().
4. export() handles a common export step.

15. Abstract Logger with Subclasses

```
from abc import ABC, abstractmethod

class Logger(ABC):
    @abstractmethod
    def log(self, message): pass

class ConsoleLogger(Logger):
    def log(self, message):
        print("Console:", message)

class FileLogger(Logger):
    def log(self, message):
        print("Writing to file:", message)

logger = ConsoleLogger()
logger.log("App started")
```

Key Points :

1. Logger is an abstract class with log() method.
2. ConsoleLogger and FileLogger show different implementations.
3. Subclasses implement how and where to log (console/file).
4. Main code can work with any type of logger.

16. Interface for Machine Operations

```
from abc import ABC, abstractmethod

class Machine(ABC):
    @abstractmethod
    def start(self): pass

    @abstractmethod
    def stop(self): pass

class Fan(Machine):
    def start(self):
        print("Fan started")

    def stop(self):
        print("Fan stopped")

fan = Fan()
fan.start()
fan.stop()
```

Key Points :

1. Abstract class Machine defines start() and stop() methods.
2. Fan class implements machine operations.
3. We can turn the fan on and off.
4. Same logic can be used for AC, TV, etc.
5. Very useful for home automation or IoT.

17. Plugin Architecture with ABC

```
from abc import ABC, abstractmethod

class Plugin(ABC):
    @abstractmethod
    def execute(self): pass

class SpellCheck(Plugin):
    def execute(self):
        print("Checking spelling")

class GrammarCheck(Plugin):
    def execute(self):
        print("Checking grammar")

for plugin in [SpellCheck(), GrammarCheck()]:
    plugin.execute()
```

Key Points :

1. Plugin is an abstract class with an execute method.
2. SpellCheck and GrammarCheck implement execute().
3. They can run one by one from a list.
4. New plugins can be added without changing the base code.

18. Shape Drawing App

```
from abc import ABC, abstractmethod

class Drawable(ABC):
    @abstractmethod
    def draw(self): pass

class Rectangle(Drawable):
    def draw(self):
        print("Drawing rectangle")

class Triangle(Drawable):
    def draw(self):
        print("Drawing triangle")
```

```
def render(d: Drawable):
    d.draw()

render(Rectangle())
render(Triangle())
```

Key Points :

1. Abstract class Drawable defines a draw() method.
2. Rectangle and Triangle implement drawing logic.
3. The render() function works for any drawable object.

19. Music Player with Interface

```
from abc import ABC, abstractmethod

class MediaPlayer(ABC):
    @abstractmethod
    def play(self): pass

class Mp3Player(MediaPlayer):
    def play(self):
        print("Playing MP3")

class WavPlayer(MediaPlayer):
    def play(self):
        print("Playing WAV")

Mp3Player().play()
WavPlayer().play()
```

Key Points :

1. Abstract class MediaPlayer defines play() method.
2. Mp3Player and WavPlayer handle different formats.
3. Supports multiple file types through one interface.

20. Data Storage Abstraction

```
from abc import ABC, abstractmethod

class Storage(ABC):
    @abstractmethod
    def save(self, data): pass

class Database(Storage):
    def save(self, data):
        print(f"Saving to DB: {data}")

class FileSystem(Storage):
    def save(self, data):
        print(f"Saving to file: {data}")

def store(storage: Storage, data):
    storage.save(data)

store(Database(), "Customer Data")
store(FileSystem(), "Log Data")
```

Key Points :

1. Abstract class Storage defines a save() method.
2. Database and FileSystem implement different storage backends.
3. We can choose where to store the data.
4. store() function works with any storage type.