#1

#2

Створіть клас Animal. Також створіть екземпляр класу Animal та привласніть змінній animal. Для класу Animal у конструкторі створіть дві властивості: nickname - кличка тварини та weight - вага тварини. Реалізуйте також метод класу say. При реалізації методу можна використати оператор pass, поки що головне - це визначення, а не конкретна реалізація.

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

animal = Animal("nickname", 3)

#3

Для попереднього завдання реалізуйте в класі Animal метод change\_weight, який має змінювати вагу тварини.

Викличте функцію change\_weight(12) для об'єкта animal та змініть значення початкової ваги з 10 на 12 одиниць.

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

def change\_weight(self, weight):

self.weight = weight

animal = Animal("Simon", 10)

animal.change\_weight(12)

#4

Додамо в клас Animal змінну класу color, значення якої спочатку дорівнює 'white', і метод change\_color, який повинен змінювати значення змінної класу color.

Створіть екземпляри об'єкта: first\_animal та second\_animal

Викличте функцію change\_color("red") для будь-якого екземпляра об'єкту Animal та змініть значення змінної класу color на "red".

class Animal:

color = "white"

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

def change\_weight(self, weight):

self.weight = weight

def change\_color(color):

Animal.color = color

first\_animal = Animal("cat", 5)

second\_animal = Animal("bird", 1)

Animal.change\_color("red")

print(first\_animal.color)

print(second\_animal.color)

#5

Створіть клас Cat, батьківським класом якого є клас Animal. У класі Cat виконайте перевизначення методу say, щоб він повертав рядок "Meow" для екземплярів класу Cat.

Фактично ми виконуємо при цьому поліморфізм. Поліморфізм - це здатність програми вибирати різні реалізації при виклику операцій з однією і тією ж назвою. Тобто при виклику методу say в екземпляра класу Cat викликається нова реалізація, а не успадкована від класу Animal

Створіть також змінну cat, яка буде екземпляром класу Cat. При створенні змінної cat ім'я кота має бути "Simon", а вага - 10 одиниць.

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

def change\_weight(self, weight):

self.weight = weight

class Cat(Animal):

def say(self):

return f"Meow"

cat = Cat("Simon", 10)

#6

Створіть клас Dog, батьківським класом якого є клас Animal. У класі Dog виконайте перевизначення методу say, щоб він повертав рядок "Woof" для екземплярів класу Dog.

У конструкторі класу Dog введіть нову властивість breed - порода, при цьому повинні залишитись всі властивості, успадковані від класу Animal.

Створіть у коді наступний екземпляр класу Dog.

dog = Dog("Barbos", 23, "labrador")

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

def change\_weight(self, weight):

self.weight = weight

class Dog(Animal):

def \_\_init\_\_(self, nickname, weight, breed):

super().\_\_init\_\_(nickname, weight)

self.breed = breed

def say(self):

return "Woof"

dog = Dog("Barbos", 23, "labrador")

#7

Для минулого завдання додамо клас Owner — власника собаки. У класу є три атрибути: ім'я — name, вік — age та адреса — address. Також необхідно реалізувати метод info, який повертає словник з ключами 'name', 'age' і 'address', та значення яких дорівнюють відповідним властивостям екземпляра класу.

Реалізувати для класу Dog атрибут owner, який буде екземпляром класу Owner. Додати до класу Dog метод who\_is\_owner, який повертає результат виклику методу info екземпляра класу Owner, тобто це словник з ключами name, age та address власника.

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

def change\_weight(self, weight):

self.weight = weight

class Owner:

def \_\_init\_\_(self, name, age, address):

self.name = name

self.age = age

self.address = address

def info(self):

return {"name": self.name, "age": self.age, "address": self.address}

class Dog(Animal):

def \_\_init\_\_(self, nickname, weight, breed, owner):

self.breed = breed

self.owner = owner

super().\_\_init\_\_(nickname, weight)

def say(self):

return "Woof"

def who\_is\_owner(self):

return self.owner.info()

ben = Owner("Ben", 25, "Kiyv")

#dog = Dog("Barbos", 23, "labrador", ben)

print(ben.info())

#print(ben.age)

#print(ben.address)

#8

Створіть два класи: CatDog та DogCat. Ці класи повинні наслідуватись від двох класів відразу: Cat та Dog. Після успадкування в екземпляра класу CatDog, батьківський метод say повинен повертати "Meow", а у класу DogCat — "Woof". Для обох зазначених класів реалізуйте метод info, який повертає рядок у наступному форматі f"{self.nickname}-{self.weight}".

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

class Cat(Animal):

def say(self):

return "Meow"

class Dog(Animal):

def say(self):

return "Woof"

class CatDog(Cat, Dog):

def info(self):

return f"{self.nickname}-{self.weight}"

class DogCat(Dog, Cat):

def info(self):

return f"{self.nickname}-{self.weight}"

#9

У четвертому модулі ми реалізували функцію lookup\_key для пошуку всіх ключів за значенням у словнику. Першим параметром у функцію ми передавали словник, а другим – значення, яке хотіли знайти. Результатом був список ключів або порожній список, якщо ми нічого не знаходили.

def lookup\_key(data, value):

keys = []

for key in data:

if data[key] == value:

keys.append(key)

return keys

Створіть клас LookUpKeyDict, батьком якого буде клас UserDict. Зробіть функцію lookup\_key методом класу LookUpKeyDict.

from collections import UserDict

class LookUpKeyDict(UserDict):

def lookup\_key(self, value):

keys = []

[keys.append(key) for key in self if self[key] == value]

return keys

#10

Перепишемо завдання розрахунку заборгованостей з комунальних послуг за допомогою класу UserList.

payment = [1, -3, 4]

def amount\_payment(payment):

sum = 0

for value in payment:

if value > 0:

sum = sum + value

return sum

Нагадаємо умову. У нас є список показань заборгованостей з комунальних послуг наприкінці місяця, список payment. Заборгованості можуть бути від'ємними — у нас переплата, або додатними, якщо потрібно сплатити за рахунками.

Створіть клас AmountPaymentList, успадковуйте його від класу UserList. Зробіть функцію amount\_payment методом класу AmountPaymentList.

from collections import UserList

class AmountPaymentList(UserList):

def amount\_payment(self):

sum = 0

for value in self:

if value > 0:

sum = sum + value

return sum

#11

Створіть клас NumberString, успадкуйте його від класу UserString, визначте для нього метод number\_count(self), який буде рахувати кількість цифр у рядку.

from collections import UserString

class NumberString(UserString):

def number\_count(self):

count = 0

for el in self:

if el.isdigit():

count += 1

return count

#12

Створіть клас IDException, який успадковуватиме клас Exception.

Також реалізуйте функцію add\_id(id\_list, employee\_id), яка додає до списку id\_list ідентифікатор користувача employee\_id та повертає вказаний оновлений список id\_list.

Функція add\_id буде викликати власне виключення IDException, якщо employee\_id не починається з '01', інакше employee\_id буде додано до списку id\_list.

class IDException(Exception):

pass

def add\_id(id\_list, employee\_id):

if employee\_id[0:2] != "01":

raise IDException

else:

id\_list.append(employee\_id)

return id\_list

#13

Як ми вже говорили, поліморфізм - це здатність програми вибирати різні реалізації при виклику операцій з однією і тією ж назвою.

Але поліморфізм - це також здатність об'єктів прикидатись чимось іншим. У наведеному вище прикладі Chupakabra прикидалася собакою та кішкою.

Для коду із завдання вам необхідно реалізувати клас CatDog, не використовуючи успадкування від класу Animal, але щоб екземпляр класу CatDog поводився як і, як екземпляр класу Cat, тобто. він повинен вдати, що він клас Cat.

class Animal:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

pass

def change\_weight(self, weight):

self.weight = weight

class Cat(Animal):

def say(self):

return "Meow"

class CatDog:

def \_\_init\_\_(self, nickname, weight):

self.nickname = nickname

self.weight = weight

def say(self):

return "Meow"

def change\_weight(self, weight):

self.weight = weight

№14

Реалізуйте клас Contacts, який працюватиме з контактами. На першому етапі ми додамо два методи.

* list\_contacts повертає список контактів це змінна contacts з поточного екземпляра класу
* add\_contacts додає новий контакт до списку, який є змінною об'єкту - contacts

Клас Contacts містить змінну класу current\_id. Ми будемо використовувати її при додаванні нового контакту як унікального ідентифікатора контакту. Коли ми додаємо новий контакт, то передаємо такі аргументи в метод add\_contacts: name, phone, email та favorite. Метод повинен створити словник із зазначеними ключами та значеннями параметрів функції. Також необхідно додати до словника новий ключ id, значенням якого є значення змінної класу current\_id.

Приклад отриманого словника:

{

"id": 1,

"name": "Wylie Pope",

"phone": "(692) 802-2949",

"email": "est@utquamvel.net",

"favorite": True,

}

Вказаний словник ми додаємо до списку contacts. Не забуваймо збільшувати змінну current\_id на одиницю після кожного виклику методу add\_contacts для збереження унікальності ключа id для словника.

Примітка: для правильного проходження тесту не створюйте екземпляр класу в коді.

class Contacts:

current\_id = 1

def \_\_init\_\_(self):

self.contacts = []

def list\_contacts(self):

return self.contacts

def add\_contacts(self, name, phone, email, favorite):

contact ={"id": Contacts.current\_id,

"name": name,

"phone":phone,

"email": email,

"favorite": favorite,}

self.contacts.append(contact)

Contacts.current\_id +=1

print(contact)

return self.contacts

#15

Продовжуємо розширювати функціональність класу Contacts. На цьому етапі ми додамо до класу метод get\_contact\_by\_id. Метод повинен шукати контакт по унікальному id у списку contacts та повертати словник з нього із зазначеним ключем id. Якщо словника із зазначеним id у списку contacts не знайдено, метод повертає None.

Примітка: для правильного проходження тесту не створюйте екземпляр класу в коді.

class Contacts:

current\_id = 1

def \_\_init\_\_(self):

self.contacts = []

def list\_contacts(self):

return self.contacts

def add\_contacts(self, name, phone, email, favorite):

self.contacts.append(

{

"id": Contacts.current\_id,

"name": name,

"phone": phone,

"email": email,

"favorite": favorite,

}

)

Contacts.current\_id += 1

def get\_contact\_by\_id(self, id):

for key in self.contacts:

if key["id"] == id:

return key

else:

return None

#16

Завершуємо функціональність класу Contacts. На цьому етапі ми додамо до класу метод remove\_contacts. Метод винен видаляти контакт по унікальному id у списку contacts. Якщо словника із зазначеним id у списку contacts не знайдено, метод жодних дій над списком contacts не робить.

Примітка: для правильного проходження тесту не створюйте екземпляр класу в коді.

class Contacts:

current\_id = 1

def \_\_init\_\_(self):

self.contacts = []

def list\_contacts(self):

return self.contacts

def add\_contacts(self, name, phone, email, favorite):

self.contacts.append(

{

"id": Contacts.current\_id,

"name": name,

"phone": phone,

"email": email,

"favorite": favorite,

}

)

Contacts.current\_id += 1

def get\_contact\_by\_id(self, id):

result = list(filter(lambda contact: contact.get("id") == id, self.contacts))

return result[0] if len(result) > 0 else None

def remove\_contacts(self, id):

[self.contacts.remove(contact) for contact in self.contacts if contact.get("id") == id]