Міністерство освіти і науки України

Державний університет “Житомирська політехніка”

Кафедра інженерії програмного забезпечення

Група: ВТ-21-1[1]

Програмування мовою Python

Лабораторна робота № 9

«КЛАСИ. Ч. 3»

Виконав: Бабушко А. С.

Прийняв: Морозов Д. С.

***Мета роботи:*** ознайомитися з ООП, множинним наслідуванням, міксинами в мові Python

***Хід роботи:***

***Завдання на лабораторну роботу:***

1. Створіть клас Alphabet. Його метод \_\_init \_\_ (), буде мати визначені два параметри: lang - мова і letters - список букв. Значення змінних lang і letters будуть визначенні за замовчуванням і міститимуться у вигляді статичних атрибутів для української мови. Клас матиме метод метод print\_alphabet(), який виведе в консоль літери україхнського алфавіту. Метод letters\_num(), повертатиме кількість букв в алфавіті. Метод is\_ua\_lang() прийматиме довільний текст і визначатиме чи відноситься він до української мови (незалежно від регістру). Створіть клас EngAlphabet шляхом успадкування від класу Alphabet. Для його методу \_\_init \_\_(), всередині якого буде викликатися батьківський метод \_\_init \_\_(), в якості параметрів будуть передаватися позначення мови (наприклад, 'En') і рядок, що складається з усіх букв алфавіту. Додайте приватний статичний атрибут \_\_en\_letters\_num, який буде зберігати кількість букв в алфавіті. Створіть метод is\_en\_letter(), який буде приймати строку в якості параметра і визначати, чи відноситься ця строка до англійського алфавіту. Перевизначити метод letters\_num() - нехай в поточному класі класі він буде повертати значення властивості \_\_en\_letters\_num. 6. Створіть статичний метод example(), який буде повертати приклад тексту англійською мовою.

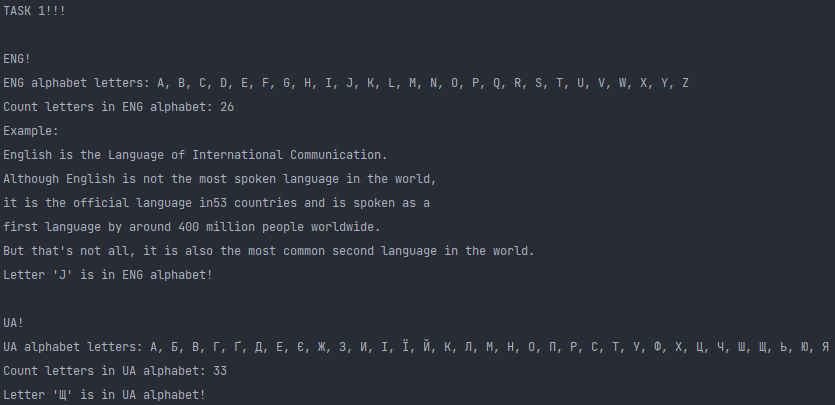
Тести до модуля:

* Створіть об'єкт класу EngAlphabet
* Надрукуйте літери алфавіту для цього об'єкту
* Виведіть кількість букв в алфавіті
* Перевірте, чи відноситься буква J до англійського алфавіту.
* Перевірте, чи відноситься буква Щ до українського алфавіту
* Виведіть приклад тексту англійською мовою

***Лістинг програми:***

*""" Lab 9. Python. Andrii Babushko. Repository: https://github.com/AndriiBabushko/Python """  
from* \_\_future\_\_ *import* annotations  
*from* typing *import* TextIO  
*from* matplotlib *import* pyplot *as* plt  
*import* csv  
*import* os.path  
*import* string  
  
*# task 1*  
*class* Alphabet:  
 *def \_\_init\_\_*(*self*, lang: str = 'UA', letters=*None*):  
 *if* letters *is None*:  
 letters = ['А', 'Б', 'В', 'Г', 'Ґ', 'Д', 'Е', 'Є', 'Ж', 'З', 'И', 'І', 'Ї', 'Й', 'К', 'Л', 'М', 'Н', 'О', 'П', 'Р', 'С', 'Т', 'У', 'Ф', 'Х', 'Ц', 'Ч', 'Ш',  
 'Щ', 'Ь', 'Ю', 'Я']  
 *self*.language: str = lang  
 *self*.letters: list = letters  
  
 *def* print\_alphabet(*self*) -> *None*:  
 alphabet\_letters = ', '.join(*self*.letters)  
 print(f'{*self*.language} alphabet letters: {alphabet\_letters}')  
  
 *def* letters\_num(*self*) -> int:  
 *return* len(*self*.letters)  
  
 *def* is\_ua\_lang(*self*, text: str) -> bool:  
 text = text.upper()  
 text\_list: list = list(text)  
  
 *if* text\_list[0] *in self*.letters:  
 *return True  
  
 return False  
  
  
class* EngAlphabet(Alphabet):  
 \_\_en\_letters\_num: int = 26  
  
 *def \_\_init\_\_*(*self*, lang: str, letters: list):  
 super().\_\_init\_\_(lang, letters)  
  
 *def* is\_en\_letter(*self*, eng\_text: str) -> bool:  
 eng\_text = eng\_text.upper()  
 eng\_text\_list: list = list(eng\_text)  
  
 *if* eng\_text\_list[0] *in self*.letters:  
 *return True  
  
 return False  
  
 def* letters\_num(*self*) -> int:  
 *return self*.\_\_en\_letters\_num  
  
 @staticmethod  
 *def* example() -> str:  
 *return* "Example:\nEnglish is the Language of International Communication.\nAlthough English is not the most spoken language in the world,\nit is the official language in" \  
 "53 countries and is spoken as a\nfirst language by around 400 million people worldwide.\nBut that's not all, it is also the most common second language in the world."  
  
  
print('TASK 1!!!')  
print('ENG!')  
task\_1\_eng\_alphabet: EngAlphabet = EngAlphabet('ENG', list(string.ascii\_uppercase))  
task\_1\_eng\_alphabet.print\_alphabet()  
print(f'Count letters in {task\_1\_eng\_alphabet.language} alphabet: {task\_1\_eng\_alphabet.letters\_num()}')  
print(task\_1\_eng\_alphabet.example())  
*if* task\_1\_eng\_alphabet.is\_en\_letter('J'):  
 print('Letter \'J\' is in ENG alphabet!')  
*else*:  
 print('Letter \'J\' isn\'t in ENG alphabet!')  
  
print('\nUA!')  
task\_1\_alphabet: Alphabet = Alphabet()  
task\_1\_alphabet.print\_alphabet()  
print(f'Count letters in {task\_1\_alphabet.language} alphabet: {task\_1\_alphabet.letters\_num()}')  
*if* task\_1\_alphabet.is\_ua\_lang('Щ'):  
 print('Letter \'Щ\' is in UA alphabet!')  
*else*:  
 print('Letter \'Щ\' isn\'t in UA alphabet!')

***Результат програми:***

******

1. Створіть клас Human. Визначте для нього два статичних атрибути: default\_name і default\_age. Його метод \_\_init \_\_(), який крім self приймає ще два публічних параметри(name і age) і два приватних (money і house). Параметр money визначатиме кількість грошей, а house – посилання на об’єкт класу House. Метод info(), має виводити поля name, age, house і money. Реалізуйте довідковий статичний метод default\_info(), який буде виводити статичні поля default\_name і default\_age. Реалізуйте приватний метод make\_deal(), який буде відповідати за технічну реалізацію покупки будинку: зменшувати кількість грошей на рахунку і привласнювати посилання на тільки що куплений будинок. В якості аргументів даний метод приймає об'єкт будинку та його ціну. Реалізуйте метод earn\_money(), що збільшує значення поля money. Реалізуйте метод buy\_house(), який буде перевіряти, що у людини достатньо грошей для покупки, і здійснювати операцію. Якщо грошей занадто мало - потрібно вивести попередження в консоль. Параметри методу: посилання на будинок і розмір знижки (за замовчуванням 10%). Створіть клас House. Його метод \_\_init \_\_() містить два динамічних параметри: \_area і \_price, що мають значення за замовчуваннями. Створіть метод final\_price(), який приймає як параметр розмір знижки і повертає ціну з урахуванням даної знижки. Створіть клас SmallHouse, успадкувавши його функціонал від класу House. Всередині класу SmallHouse перевизначите метод \_\_init \_\_() так, щоб він створював об'єкт з площею 40м2

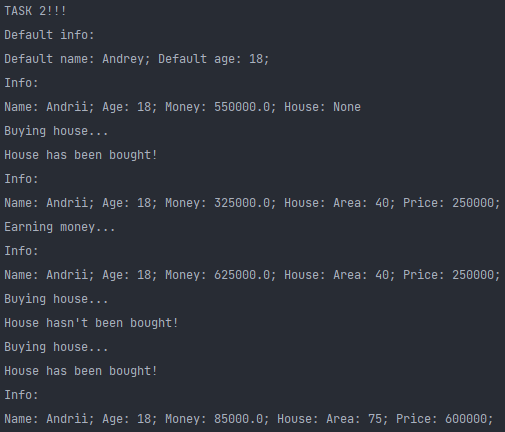
Тести до модуля:

* Викличте довідковий метод default\_info() для класу Human
* Створіть об'єкт класу Human
* Виведіть довідкову інформацію про створений об'єкт (викличте метод info ()).
* Створіть об'єкт класу SmallHouse
* Спробуйте купити створений будинок, переконайтеся в отриманні попередження.
* Виправте фінансове становище об'єкта - викличте метод earn\_money()
* Знову спробуйте купити будинок
* Подивіться, як змінився стан об'єкта класу Human.

***Лістинг програми:***

*# task 2*  
  
*class* House:  
 *def \_\_init\_\_*(*self*, area: int = 50, price: float = 250000) -> *None*:  
 *self*.\_area: int = area  
 *self*.\_price: float = price  
  
 *def* final\_price(*self*, discount=0) -> float:  
 *return self*.\_price - (*self*.\_price \* discount / 100)  
  
 *def \_\_str\_\_*(*self*) -> str:  
 *return* f'Area: {*self*.\_area}; Price: {*self*.\_price};'  
  
 *def* get\_price(*self*) -> float:  
 *return self*.\_price  
  
  
*class* SmallHouse(House):  
 *def \_\_init\_\_*(*self*) -> *None*:  
 *self*.area = 40  
 super().\_\_init\_\_(*self*.area)  
  
  
*class* Human:  
 default\_name: str = 'Andrey'  
 default\_age: int = 18  
  
 *def \_\_init\_\_*(*self*, name: str, age: int, money: float = 0, house: House = *None*) -> *None*:  
 *self*.name: str = name  
 *self*.age: int = age  
 *self*.\_\_money: float = money  
 *self*.\_\_house: House = house  
  
 *def* info(*self*) -> str:  
 *return* f'Info:\nName: {*self*.name}; Age: {*self*.age}; Money: {*self*.\_\_money}; House: {*self*.\_\_house}'  
  
 @staticmethod  
 *def* default\_info() -> str:  
 *return* f'Default info:\nDefault name: {Human.default\_name}; Default age: {Human.default\_age};'  
  
 *def* \_\_make\_deal(*self*, house: House, price: float = 0) -> tuple:  
 *self*.\_\_money -= price  
 *self*.\_\_house = house  
 *return self*.\_\_money, *self*.\_\_house  
  
 *def* earn\_money(*self*, salary: float) -> float:  
 print('Earning money...')  
 *self*.\_\_money += salary  
 *return self*.\_\_money  
  
 *def* buy\_house(*self*, house, price: float, discount: int = 10) -> tuple:  
 print('Buying house...')  
 price: float = price - (price \* discount / 100)  
 *if* price <= *self*.\_\_money:  
 print('House has been bought!')  
 *self*.\_\_house = house  
 *self*.\_\_money -= price  
 *return self*.\_\_money, *self*.\_\_house  
 *else*:  
 print("House hasn't been bought!")  
 *return self*.\_\_money, *self*.\_\_house  
  
  
print('\nTASK 2!!!')  
  
print(Human.default\_info())  
task\_2\_human: Human = Human('Andrii', 18, 550000.0)  
print(task\_2\_human.info())  
  
task\_2\_small\_house: SmallHouse = SmallHouse()  
task\_2\_human.buy\_house(task\_2\_small\_house, task\_2\_small\_house.get\_price())  
print(task\_2\_human.info())  
  
task\_2\_human.earn\_money(300000.0)  
print(task\_2\_human.info())  
  
task\_2\_house: House = House(100, 800000)  
task\_2\_human.buy\_house(task\_2\_house, task\_2\_house.get\_price())  
task\_2\_house: House = House(75, 600000)  
task\_2\_human.buy\_house(task\_2\_house, task\_2\_house.get\_price())  
print(task\_2\_human.info())

***Результат програми:***

******

1. Створіть клас Apple. Його статичний атрибут states, яке буде містити всі стадії дозрівання яблука («Відсутнє», «Цвітіння», «Зелене», «Червоне»). Метод \_\_init \_\_(), всередині якого будуть визначені два динамічних protected атрибути: \_index (номер яблука) і \_state (приймає перше значення зі словника states). Створіть метод grow(), який буде переводити яблуко на наступну стадію дозрівання Створіть метод is\_ripe(), який буде перевіряти, що яблуко дозріло (досягло останньої стадії дозрівання). Створіть клас AppleTree. Визначте метод \_\_init \_\_(), який буде приймати як параметр кількість яблук і на його основі буде створювати список об'єктів класу Apple. Даний список буде зберігатися всередині динамічного атрибуту apples. Створіть метод grow\_all(), який буде переводити всі об'єкти зі списку яблук на наступний етап дозрівання. Створіть метод all\_are\_ripe(), який буде повертати True, якщо все яблука зі списку стали стиглими. Створіть метод give\_away\_all(), який буде чистити список яблук після збору врожаю Створіть клас Gardener. Його метод \_\_init \_\_(), міститиме два динамічних атриути: name (ім’я садівника, публічний атрибут) і \_tree (приймає об’єкт класу AppleTree). Створіть метод work(), який змушує садівника працювати, що дозволяє яблукам ставати більш стиглими. Створіть метод harvest(), який перевіряє, чи всі плоди дозріли. Якщо всі - садівник збирає урожай. Якщо і - метод друкує попередження. Створіть статичний метод apple\_base(), який виведе в консоль довідку з кількості яблук і ступені їх стиглості.

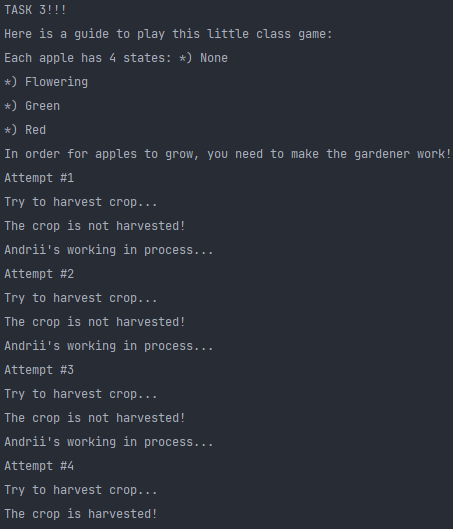
Тести до модуля:

* Створіть декілька об’єктів класу Apple.
* Викличте довідку по всім наявним яблукам
* Створіть об'єкти класів AppleTree і Gardener
* Використовуючи об'єкт класу Gardener, попрацювати над яблучним деревом.
* Спробуйте зібрати урожай
* Якщо яблука ще не дозріли, продовжуйте доглядати за деревом
* Зберіть урожай.

***Лістинг програми:***

*# task 3*  
  
*class* Apple:  
 states: dict[int, str] = {0: 'Відсутнє', 1: 'Цвітіння', 2: 'Зелене', 3: 'Червоне'}  
  
 *def \_\_init\_\_*(*self*, index: int, state: int = 0) -> *None*:  
 *self*.\_index: int = index  
 *self*.\_state: int = state  
  
 *def* grow(*self*) -> *None*:  
 *if self*.\_state < 3:  
 *self*.\_state += 1  
  
 *def* is\_ripe(*self*) -> bool:  
 *if self*.\_state == 3:  
 *return True  
 return False  
  
 def* info(*self*) -> str:  
 *return* f'Apple number: {*self*.\_index}; Apple state: {*self*.\_state}'  
  
  
*class* AppleTree:  
 *def \_\_init\_\_*(*self*, apples\_count: int) -> *None*:  
 *self*.apples: list[Apple] = [Apple(apple\_number) *for* apple\_number *in* range(0, apples\_count)]  
  
 *def* grow\_all(*self*) -> *None*:  
 *for* apple *in self*.apples:  
 apple.grow()  
  
 *def* all\_are\_ripe(*self*) -> bool:  
 *for* apple *in self*.apples:  
 *if not* apple.is\_ripe():  
 *return False  
 return True  
  
 def* give\_away\_all(*self*) -> bool:  
 *if self*.all\_are\_ripe():  
 *self*.apples = []  
 *return True  
 return False  
  
  
class* Gardener:  
 *def \_\_init\_\_*(*self*, name: str, apple\_tree: AppleTree) -> *None*:  
 *self*.name: str = name  
 *self*.\_tree: AppleTree = apple\_tree  
  
 *def* work(*self*) -> *None*:  
 print(f'{*self*.name}\'s working in process...')  
 *self*.\_tree.grow\_all()  
  
 *def* harvest(*self*) -> bool:  
 print('Try to harvest crop...')  
 *if self*.\_tree.all\_are\_ripe():  
 print('The crop is harvested!')  
 *self*.\_tree.give\_away\_all()  
 *return True  
 else*:  
 print('The crop is not harvested!')  
 *return False* @staticmethod  
 *def* apple\_base() -> *None*:  
 print('Here is a guide to play this little class game:\n'  
 'Each apple has 4 states: '  
 '\*) None\n'  
 '\*) Flowering\n'  
 '\*) Green\n'  
 '\*) Red\n'  
 'In order for apples to grow, you need to make the gardener work!')  
  
  
print('\nTASK 3!!!')  
task\_3\_apple\_1: Apple = Apple(1)  
task\_3\_apple\_1.info()  
task\_3\_apple\_2: Apple = Apple(2)  
task\_3\_apple\_2.info()  
task\_3\_apple\_3: Apple = Apple(3)  
task\_3\_apple\_3.info()  
  
Gardener.apple\_base()  
task\_3\_apple\_tree: AppleTree = AppleTree(10)  
task\_3\_gardener: Gardener = Gardener('Andrii', task\_3\_apple\_tree)  
attempt: int = 1  
*while True*:  
 print(f'Attempt #{attempt}')  
 *if* task\_3\_gardener.harvest():  
 *break  
 else*:  
 task\_3\_gardener.work()  
 attempt += 1

***Результат програми:***

******

1. Створіть клас KmrCsv, який має два атрибути класу за замовчуванням: ref (посилання на CSV файл з оцінками) і num (номер КМР), та методи для встановлення і, відповідно, визначення посилання на файл з оцінками, встановлення номеру КМР, читання файлу з оцінками та виведення інформації про файл (номер КМР і кількість студентів, що її виконали).

Створіть клас Statistic, що містить наступні методи:

- avg\_stat() визначає відсотки правильних відповідей на кожне питання серед усіх студентів і повертає результат у вигляді кортежу чисел;

- метод marks\_stat() визначає яку оцінку набрала відповідна кількість студентів і повертає результат у викляді словника формату {оцінка: кількість студентів};

- метод marks\_per\_time() визначає який середній бал за хвилину набирав студент за під час виконання КМР і повертає результат у вигляді словника формату {id студента (це перша колонка csv файлу): середній бал за хвилину} ;

- метод best\_marks\_per\_time(), який приймає два аргументи bottom\_margin і top\_margin (нижня і верхня межа вибірки підсумкових балів за КМР), та формує для цієї вибірки п’ять найкращих результатів середніх балів за хвилину у вигляді кортежу формату (id студента, підсумкова оцінка, середній бал за хвилину).

Створіть клас Plots, що містить наступні методи:

* set\_cat() – встановлює каталог в який зберігатимуться отримані графіки;
* avg\_plot() – приймає кортеж з відсотками правильних відповідей на кожне окреме питання, формує гістограму на його основі і зберігає отриманий графік;
* marks\_plot() – приймає словник з оцінками і кількістю студентів, що їх набрали, формує на його основі гістрограму і зберігає її
* best\_marks\_plot() – формує для п’яти найкращих результатів середніх балів за хвилину гістрограму і зберігає її.

Створіть клас KmrWork, що успадковує класи CsvKmr, Statistic і Plots. В якості аргументів екземпляр класу приймає посилання на csv файл та номер КМР.

Клас KmrWork містить наступні статичні атрибути

* kmrs - в ньому зберігається словник формату {номер КМР: адреса відповідного csv файла}
* cat – каталог для збереження результатів роботи

Крім успадкованих, клас KmrWork містить наступні методи:

* compare\_csv() – виводить на екран і зберігає в txt файл результат порівняння статистики двох КМР (кількість виконаних КМР, середній бал за КМР, середній час виконання КМР);
* compare\_avg\_plots() – виводить на екран і зберігає дві гістограми з відсотками правильних відповідей на кожні окремі питання.

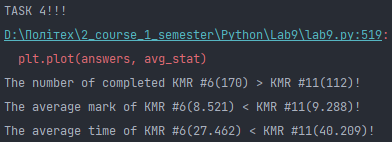
Тести до модуля:

* Створіть об’єкти kmr1 і kmr2 класу KmrWork.
* Використайте для об’єкту kmr2 методи avg\_plot() і marks\_plot()
* Для класу KmrWork використайте методи compare\_csv() і compare\_avg\_plots().

***Лістинг програми:***

*# task 4*  
  
*class* KmrCsv:  
 *import* os  
  
 *def \_\_init\_\_*(*self*, ref: str, num: int):  
 *if self*.os.path.isfile(ref):  
 *self*.ref: str = ref  
 *self*.number: int = num  
 *else*:  
 print('File does not exist!')  
 *self*.ref = *None  
 self*.number = *None  
  
 def* \_get\_read\_lines(*self*) -> list:  
 *import* io  
 *try*:  
 *if* os.path.isfile(*self*.ref):  
 *with* io.open(rf'{*self*.ref}', 'rt', encoding='utf-8') *as* krm\_csv:  
 marks\_csv = csv.reader(krm\_csv)  
 *return* [line *for* line *in* marks\_csv]  
 *else*:  
 *raise* Exception('Can\'t find file!')  
 *except* Exception *as* exc:  
 print('Caught this error: ' + repr(exc))  
  
 *def* \_get\_students\_id(*self*) -> list:  
 students: list = *self*.\_get\_read\_lines()  
  
 students\_id: list = []  
 *for* student *in* students:  
 students\_id.append(student[0])  
  
 *return* students\_id  
  
 *def* \_get\_student\_answers(*self*) -> list:  
 student\_answers: list = []  
 marks: list = *self*.\_get\_read\_lines()  
  
 *for* answers *in* marks:  
 student\_answers.append(answers[5:])  
  
 *return* student\_answers  
  
 *def* \_get\_student\_marks(*self*) -> list:  
 kmr\_marks: list = []  
 marks: list = *self*.\_get\_read\_lines()  
  
 *for* i *in* range(len(marks)):  
 mark\_string = marks[i][4].split(',')  
 mark\_number = float(mark\_string[0]) + float(int(mark\_string[1]) / 100)  
 kmr\_marks.append(mark\_number)  
  
 *return* kmr\_marks  
  
 *def* \_get\_student\_count(*self*) -> int:  
 *return* len(*self*.\_get\_read\_lines())  
  
 *def* file\_info(*self*):  
 *if not* os.path.isdir(r'./task4'):  
 os.mkdir(r'./task4')  
  
 marks = *self*.\_get\_read\_lines()  
  
 print(f'Kmr number: {*self*.number}; Count of student who pass KMR: {len(marks)}')  
  
  
*class* Statistics(KmrCsv):  
 *def \_\_init\_\_*(*self*, ref: str, num: int) -> *None*:  
 super().\_\_init\_\_(ref, num)  
  
 @staticmethod  
 *def* \_\_remove\_dict\_duplicates(dictionary: dict) -> dict:  
 result: dict = {}  
  
 *for* key, value *in* dictionary.items():  
 *if* value *not in* result.values():  
 result[key] = value  
  
 *return* result  
  
 *def* avg\_stat(*self*) -> tuple:  
 numbers\_list: list = []  
 student\_answers: list = list(*self*.\_get\_student\_answers())  
  
 answer\_counter: int = 0  
 *while* answer\_counter < len(student\_answers[0]):  
 correct\_answers: int = 0  
 *for* answer *in* student\_answers:  
 *if* answer[answer\_counter] == '0,50':  
 correct\_answers += 1  
  
 answer\_counter += 1  
 numbers\_list.append(round((correct\_answers / len(student\_answers)) \* 100))  
  
 *return* tuple(numbers\_list)  
  
 *def* marks\_stat(*self*) -> dict:  
 certain\_marks\_dict: dict = {}  
  
 kmr\_marks: list = *self*.\_get\_student\_marks()  
 student\_counter: int = 0  
  
 *for* mark *in* kmr\_marks:  
 *for* mark\_check *in* kmr\_marks:  
 *if* mark\_check == mark:  
 student\_counter += 1  
 certain\_marks\_dict[mark] = student\_counter  
 student\_counter = 0  
  
 certain\_marks\_dict: dict = *self*.\_\_remove\_dict\_duplicates(certain\_marks\_dict)  
  
 *return* certain\_marks\_dict  
  
 *def* marks\_per\_time(*self*) -> dict:  
 students\_time: list = []  
 marks: list = *self*.\_get\_read\_lines()  
 students\_marks: list = *self*.\_get\_student\_marks()  
  
 *for* i *in* range(*self*.\_get\_student\_count()):  
 time: str = marks[i][3].split(' ')  
 *if* len(time) > 2:  
 minutes = int(time[0])  
 seconds = int(time[2]) + minutes \* 60  
 *else*:  
 minutes = int(time[0])  
 seconds = minutes \* 60  
 students\_time.append(seconds)  
  
 students\_mark\_per\_min: dict = {}  
 students\_id: list = *self*.\_get\_students\_id()  
  
 *for* i *in* range(0, *self*.\_get\_student\_count()):  
 students\_mark\_per\_min[students\_id[i]] = (round((students\_marks[i] / students\_time[i]) \* 60, 2))  
  
 *return* students\_mark\_per\_min  
  
 *def* best\_marks\_per\_time(*self*, bottom\_margin: float = 0, top\_margin: float = 10) -> dict:  
 students\_code: list = list(*self*.marks\_per\_time().keys())  
 student\_marks: list = *self*.\_get\_student\_marks()  
 marks\_per\_time: list = list(*self*.marks\_per\_time().values())  
  
 interval\_students\_code: list = []  
 interval\_student\_marks: list = []  
 interval\_marks\_per\_time: list = []  
 *for* i *in* range(0, len(marks\_per\_time)):  
 *if* bottom\_margin <= student\_marks[i] <= top\_margin:  
 interval\_students\_code.append(students\_code[i])  
 interval\_student\_marks.append(student\_marks[i])  
 interval\_marks\_per\_time.append(marks\_per\_time[i])  
  
 top\_5\_results: dict = {}  
 *for* i *in* range(0, 5):  
 max\_average\_mark = max(interval\_marks\_per\_time)  
 index\_of\_student = interval\_marks\_per\_time.index(max\_average\_mark)  
 top\_5\_results[interval\_students\_code[index\_of\_student]] = f'{interval\_student\_marks[index\_of\_student]}, {interval\_marks\_per\_time[index\_of\_student]}'  
 interval\_marks\_per\_time.pop(index\_of\_student)  
 *# print(f'Top {i + 1}! {index\_of\_student + 1} student has {top\_5\_results[i]}/min mark.')  
 return* top\_5\_results  
  
  
*class* Plots(Statistics):  
 *def \_\_init\_\_*(*self*, ref: str, num: int, work\_dir: str) -> *None*:  
 super().*\_\_init\_\_*(ref, num)  
 *self*.work\_dir\_ref: str = work\_dir  
  
 *def* set\_cat(*self*, dir\_name: str) -> *None*:  
 *if not* os.path.isdir(rf'./{dir\_name}'):  
 os.mkdir(*self*.work\_dir\_ref)  
  
 *def* avg\_plot(*self*) -> *None*:  
 avg\_stat: tuple = *self*.avg\_stat()  
 answers: tuple = tuple(answer *for* answer *in* range(1, len(avg\_stat) + 1))  
 plt.plot(answers, avg\_stat)  
 plt.ylabel('Average statistics')  
 plt.xlabel('Answers')  
 plt.savefig(rf'{*self*.work\_dir\_ref}/avg\_plot.png')  
 plt.clf()  
  
 *def* marks\_plot(*self*) -> *None*:  
 marks\_stat: dict = *self*.marks\_stat()  
 plt.plot(list(marks\_stat.keys()), list(marks\_stat.values()))  
 plt.ylabel('Total count of marks')  
 plt.xlabel('Marks')  
 plt.savefig(rf'{*self*.work\_dir\_ref}/marks\_plot.png')  
 plt.clf()  
  
 *def* best\_marks\_plot(*self*) -> *None*:  
 best\_marks\_per\_time: dict = *self*.best\_marks\_per\_time(1, 9)  
 marks: list = []  
 avg\_marks: list = []  
 *for* best\_mark *in* best\_marks\_per\_time.values():  
 marks.append(best\_mark.split(', ')[0])  
 avg\_marks.append(best\_mark.split(', ')[1])  
  
 plt.plot(marks, avg\_marks)  
 plt.ylabel('Average marks per time')  
 plt.xlabel('Marks')  
 plt.savefig(rf'{*self*.work\_dir\_ref}/best\_marks\_plot.png')  
 plt.clf()  
  
  
*class* KmrWork(Plots):  
 *import* io  
  
 *def \_\_init\_\_*(*self*, ref: str, num: int, work\_dir: str) -> *None*:  
 super().*\_\_init\_\_*(ref, num, work\_dir)  
  
 @staticmethod  
 *def* \_\_get\_seconds\_from\_str(file\_read\_lines: list[str]) -> int:  
 time\_sec: int = 0  
 *for* line *in* file\_read\_lines:  
 time: list[str] = line[3].split(' ')  
 *if* len(time) > 2:  
 minutes = int(time[0])  
 seconds = int(time[2]) + minutes \* 60  
 time\_sec += minutes \* 60 + seconds  
 *else*:  
 minutes = int(time[0])  
 time\_sec += minutes \* 60  
  
 *return* time\_sec  
  
 @staticmethod  
 *def* \_\_get\_max\_kmr\_time(file\_read\_lines: list[str]) -> int:  
 time\_list: list = []  
 *for* line *in* file\_read\_lines:  
 time: list[str] = line[3].split(' ')  
 *if* len(time) > 2:  
 minutes = int(time[0])  
 seconds = int(time[2]) + minutes \* 60  
 time\_list.append(minutes \* 60 + seconds)  
 *else*:  
 minutes = int(time[0])  
 time\_list.append(minutes \* 60)  
  
 max\_time\_sec: int = max(time\_list)  
 *return* round(max\_time\_sec)  
  
 @staticmethod  
 *def* \_\_check\_kmr\_print(file\_variable: TextIO, first\_kmr: KmrWork, second\_kmr: KmrWork, message: str, first\_check, second\_check):  
 *if* first\_check > second\_check:  
 output\_line: str = f'{message} KMR #{first\_kmr.number}({first\_check}) > KMR #{second\_kmr.number}({second\_check})!'  
 file\_variable.write(output\_line + '\n')  
 print(output\_line)  
 *elif* first\_check < second\_check:  
 output\_line: str = f'{message} KMR #{first\_kmr.number}({first\_check}) < KMR #{second\_kmr.number}({second\_check})!'  
 file\_variable.write(output\_line + '\n')  
 print(output\_line)  
 *else*:  
 output\_line: str = f'{message} KMR #{first\_kmr.number}({first\_check}) == KMR #{second\_kmr.number}({second\_check})!'  
 file\_variable.write(output\_line + '\n')  
 print(output\_line)  
  
 *def* compare\_csv(*self*, kmr\_work: KmrWork) -> *None*:  
 first\_kmr\_work\_count: int = *self*.\_get\_student\_count()  
 second\_kmr\_work\_count: int = kmr\_work.\_get\_student\_count()  
  
 first\_student\_marks: list[float] = *self*.\_get\_student\_marks()  
 first\_student\_marks\_sum: float = 0  
 *for* mark *in* first\_student\_marks:  
 first\_student\_marks\_sum += mark  
 first\_students\_avg\_mark: float = round(first\_student\_marks\_sum / len(first\_student\_marks), 3)  
  
 second\_student\_marks: list[float] = kmr\_work.\_get\_student\_marks()  
 second\_student\_marks\_sum: float = 0  
 *for* mark *in* second\_student\_marks:  
 second\_student\_marks\_sum += mark  
 second\_students\_avg\_mark: float = round(second\_student\_marks\_sum / len(second\_student\_marks), 3)  
  
 first\_read\_lines: list = *self*.\_get\_read\_lines()  
 first\_time\_sec: int = *self*.\_\_get\_seconds\_from\_str(first\_read\_lines)  
  
 second\_read\_lines: list = kmr\_work.\_get\_read\_lines()  
 second\_time\_sec: int = *self*.\_\_get\_seconds\_from\_str(second\_read\_lines)  
  
 first\_avg\_time: float = round(float(first\_time\_sec) / first\_kmr\_work\_count / 60, 3)  
 second\_avg\_time: float = round(float(second\_time\_sec) / second\_kmr\_work\_count / 60, 3)  
  
 *if not* os.path.isdir(rf'./{*self*.work\_dir\_ref}'):  
 os.mkdir(rf'./{*self*.work\_dir\_ref}')  
  
 *with self*.io.open(rf'./{*self*.work\_dir\_ref}/compare\_csv\_txt.txt', 'wt', encoding='utf-8') *as* compare\_csv\_txt:  
 KmrWork.\_\_check\_kmr\_print(compare\_csv\_txt, *self*, kmr\_work, 'The number of completed', first\_kmr\_work\_count, second\_kmr\_work\_count)  
 KmrWork.\_\_check\_kmr\_print(compare\_csv\_txt, *self*, kmr\_work, 'The average mark of', first\_students\_avg\_mark, second\_students\_avg\_mark)  
 KmrWork.\_\_check\_kmr\_print(compare\_csv\_txt, *self*, kmr\_work, 'The average time of', first\_avg\_time, second\_avg\_time)  
  
 *def* compare\_avg\_plots(*self*, kmr\_work: KmrWork):  
 first\_avg\_stat: tuple = *self*.avg\_stat()  
 second\_avg\_stat: tuple = kmr\_work.avg\_stat()  
 answers: tuple = tuple(answer *for* answer *in* range(1, len(first\_avg\_stat) + 1))  
  
 plt.plot(answers, first\_avg\_stat)  
 plt.ylabel('Average statistics')  
 plt.xlabel('Answers')  
 plt.savefig(rf'{*self*.work\_dir\_ref}/first\_avg\_plot.png')  
 plt.clf()  
  
 plt.plot(answers, second\_avg\_stat)  
 plt.ylabel('Average statistics')  
 plt.xlabel('Answers')  
 plt.savefig(rf'{*self*.work\_dir\_ref}/second\_avg\_plot.png')  
 plt.clf()  
  
  
print('\nTASK 4!!!')  
task\_4\_kmr1: KmrWork = KmrWork(r'./task4/marks.lab6.csv', 6, 'test')  
task\_4\_kmr2: KmrWork = KmrWork(r'./task4/marks2.lab11.csv', 11, 'test')  
task\_4\_kmr2.avg\_plot()  
task\_4\_kmr2.marks\_plot()  
task\_4\_kmr1.compare\_csv(task\_4\_kmr2)  
task\_4\_kmr1.compare\_avg\_plots(task\_4\_kmr2)

***Результат програми:***

******

***Висновок:*** під час виконання лабораторної роботи було отримано навички створення батьківських та дочірніх класів з використанням наслідування, використання приватних за захищених методів і атрибутів класу, створення статичних та звичайних методів класу, а також побудова графіків в залежності від вхідних даних.