

МIНIСТЕРСТВО ОСВIТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**Лабораторна робота № 1**

з дисципліни “ Компоненти програмної інженерії 1. Основи розроблення ПЗ”

|  |  |  |
| --- | --- | --- |
| Виконали  студенти II курсу  групи КП-71  Атаманюк Олексій Віталійович,  Кравчук Аркадій Андрійович,  Мироненко Андрій Станіславович  (*прізвище, ім’я, по батькові*)  варіант № 2 |  | Перевірив  “\_\_\_\_” “\_\_\_\_\_\_\_\_\_\_\_\_” 20\_\_\_ р.  доцент  Замятін Денис Станіславович  (*прізвище, ім’я, по батькові*) |

Київ 2019

**Мета роботи**

Оволодіти процедурними конструкціями мови Python та основним інструментарієм розробника.

**Постановка завдання**

1. Розробити програмні засоби мовою програмування Python 3 згідно з обраним варіантом завдання. Розроблений програмний код повинен складатися з окремих функцій, кожна з яких має не більше 10 рядків та не більше 3 рівнів вкладеності. Розроблені засоби мають бути інтерактивними та надавати користувачу можливість взаємодії з ними через консоль. Дані для роботи програми повинні зберігатися у файлах в форматі JSON.

2. Виконати декомпозицію розробленого програмного коду на окремі модулі, які відповідають за бізнес-логіку, введення-виведення та управління порядком виконання (контролер).

3. Додати до кожної функції документуючий коментар та набір док-тестів. Покриття програмного коду тестами повинно складати не менше 80%.

4. За допомогою засобів статичного аналізу pep8 та pyflakes забезпечити сумісність програмного коду із стандартами кодування.

5. Використовуючи фреймворк flask-restful додати реалізацію REST-інтерфейсу для розроблених програмних засобів. Написати тести до REST-інтерфейсу, використовуючи бібліотеки pytest та requests.

6. Розмістити розроблені програмні засоби у публічному сховищі програмного коду (github, bitbucket тощо).

* **Варіант 2:**

*Планувальник задач за датами.*

* **Посилання на репозиторій:**

<https://github.com/100-gram/task-scheduler>

1. **Приклад сесії взаємодії з розробленими програмними засобами**

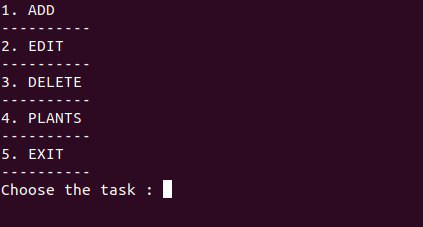


Рис. 1 Головне Меню

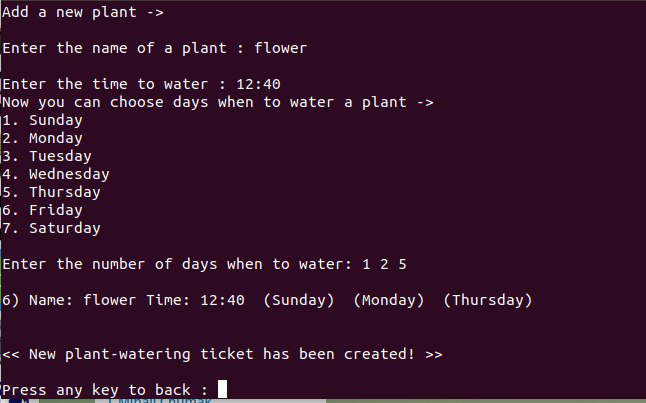


Рис. 2 Меню Додавання

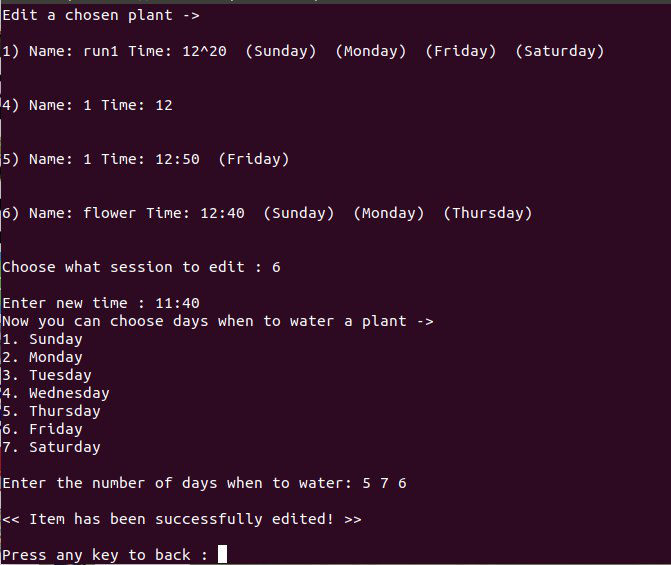


Рис. 3 Меню Редагування

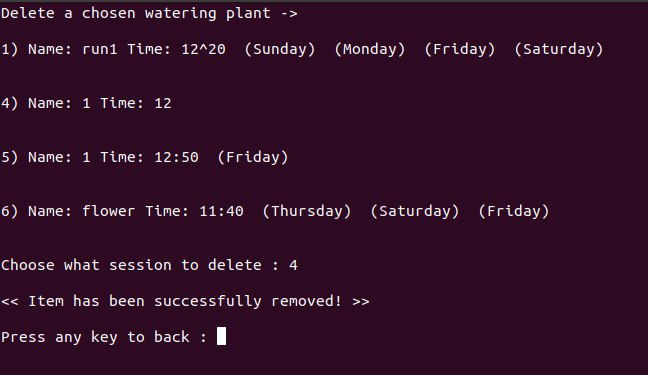


Рис. 4 Меню Видалення

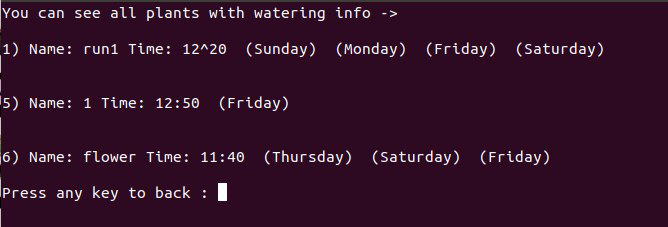


Рис. 5 Меню Перегляду Інформації

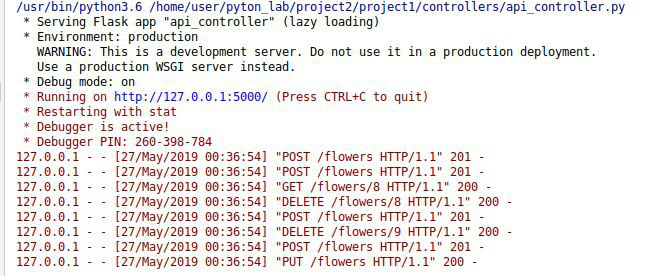


Рис. 6 Робота з REST-інтерфейсом

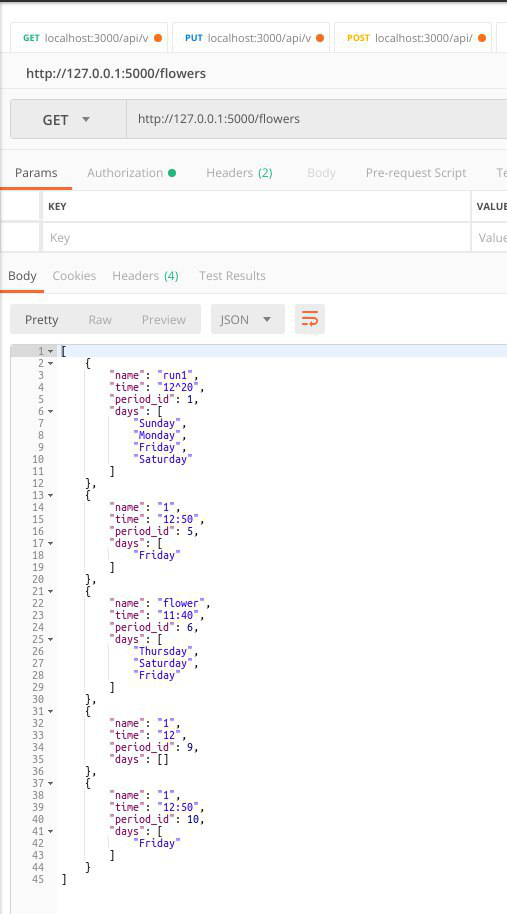


Рис. 7 Відповідь на GET-запит

1. Згенерована документація

|  |
| --- |
| **""" This controller for json storage of Task entities  Provide all methods to manage with Task entities """   class** DataManager:  **def** \_\_init\_\_(self, next\_id: int, tasks: [Task], file\_path=**None**):  *"""  Constructor for DataManager object* **:param** *next\_id: id of task, that will be created next* **:param** *tasks: list of Task entity;  """* self.next\_id = next\_id  self.tasks = tasks  self.storage\_path = file\_path **if** file\_path **is** str **else** storage\_path   **def** \_\_json\_\_(self):  *"""  JSON serialization for DataManager object* **:return***: dict with all inner fields  >>> file\_path = "./../tests"  >>> a = DataManager.load\_from\_file(file\_path + "/test\_suit1.json")  >>> a.\_\_json\_\_().\_\_str\_\_() == open(file\_path + "/test\_suit1.txt", 'r').read()  True  """* **return** {  **'next\_id'**: self.next\_id,  **'tasks'**: DataManager.tasks\_to\_json(self.tasks),  }   for\_json = \_\_json\_\_ *# supported by simplejson* @classmethod  **def** from\_json(cls, json\_obj):  *"""  Creating DataManager object from JSON object* **:param** *json\_obj: JSON object (dict)* **:return***: instance of DataManager  >>> a = DataManager.load\_from\_file("./../tests/test\_suit1.json")  >>> b = DataManager.from\_json(a.\_\_json\_\_())  >>> b.next\_id  2  >>> b.tasks.\_\_len\_\_()  1  """* **return** cls(json\_obj[**'next\_id'**], DataManager.tasks\_from\_json(json\_obj[**'tasks'**]))   @staticmethod  **def** tasks\_from\_json(storage\_obj):  *"""  Deserialize Tasks from JSON object* **:param** *storage\_obj: JSON serialized Tasks* **:return***: list of Task entities  >>> file\_path = "./../tests"  >>> data\_str = open(file\_path + "/test\_suit1.json", 'r').read()  >>> storage\_obj = simplejson.loads(data\_str)['tasks']  >>> a = DataManager.tasks\_from\_json(storage\_obj)[0].\_\_json\_\_()  >>> a['name']  'Name'  >>> a['date\_start']  '1999-08-28T21:03:05'  """* **return** list(map(**lambda** x: Task.from\_json(x), storage\_obj))   @staticmethod  **def** tasks\_to\_json(tasks):  *"""  Deserialize Tasks from JSON object* **:param** *tasks: JSON serialized Tasks* **:return***: list of Task entities  >>> arr = [Task(1, "Name", "nothing", parser.parse("1999-08-28T21:03:05"), \  parser.parse("1999-08-28T05:55:23"), True)]  >>> a = DataManager.tasks\_to\_json(arr)[0]  >>> a['name']  'Name'  >>> a['date\_start']  '1999-08-28T21:03:05'  >>> a['id']  1  """* **return** list(map(**lambda** x: x.\_\_json\_\_(), tasks))   @classmethod  **def** load\_from\_file(cls, file\_path: str):  *"""  Create DataManager instance from file with JSON data* **:param** *file\_path: set explicitly file path of data storage* **:return***:instance of DataManager  >>> DataManager.load\_from\_file("./../tests/test\_suit1.json")  """* data\_str = open(file\_path, **'r'**).read()  storage\_obj = simplejson.loads(data\_str)  **return** DataManager.from\_json(storage\_obj)   **def** set\_storage\_path(self, file\_path: str):  *"""  Set storage path for data* **:param** *file\_path: file path to json file* **:return***: None  """* self.storage\_path = file\_path   **def** save\_to\_file(self, file\_path: str):  *"""  Save changes(tasks ans next\_id) in data file* **:param** *file\_path: set explicitly file path of data storage* **:return***: None  """* **with** open(file\_path, **'w'**) **as** outfile:  simplejson.dump(self, outfile, indent=4, for\_json=**True**)   **def** update\_from\_file(self, file\_path: str):  *"""  Get changes( tasks ans next\_id) from data file* **:param** *file\_path: set explicitly file path of data storage* **:return***: None  """* data\_str = open(file\_path, **'r'**).read()  storage\_obj = simplejson.loads(data\_str)  self.next\_id = storage\_obj[**'next\_id'**]  self.tasks = DataManager.tasks\_from\_json(storage\_obj[**'tasks'**])   **def** get\_all(self, offset=0, limit=**None**, query=**None**):  *"""  Get all filtered and paginated tasks* **:param** *offset: count of skipping tasks* **:param** *limit: count of tasks in page* **:param** *query: search parameter* **:return***: list of Task entities  """* self.update\_from\_file(self.storage\_path)  **return** DataManager.\_\_paginate\_and\_search\_\_(self.tasks, offset, limit, query)   **def** get\_with\_status(self, status: TaskStatus, offset=0, limit=**None**, query=**None**):  *"""  Get all filtered and paginated tasks with selected status* **:param** *status: selected status of Task* **:param** *offset: count of skipping tasks* **:param** *limit: count of tasks in page* **:param** *query: search parameter* **:return***: list of Task entities  """* self.update\_from\_file(self.storage\_path)  filter\_function = {  TaskStatus.COMPLETED: **lambda** x: x.completed(),  TaskStatus.UNCOMPLETED: **lambda** x: **not** x.completed(),  TaskStatus.PLANNED: **lambda** x: **not** x.is\_finished() **and not** x.is\_running(),  TaskStatus.RUNNING: **lambda** x: x.is\_running(),  TaskStatus.FINISHED: **lambda** x: x.is\_finished(),  }[status]  result = list(filter(filter\_function, self.tasks))  **return** DataManager.\_\_paginate\_and\_search\_\_(result, offset, limit, query, status)   **def** get\_by\_id(self, task\_id: int):  *"""  Get Task entity with selected id* **:param** *task\_id: id of searched Task* **:return***: Task entity if task with selected id exists else False  """* self.update\_from\_file(self.storage\_path)  **for** i, task **in** enumerate(self.tasks):  **if** task.task\_id == task\_id:  **return** self.tasks[i]  **return False   def** update\_task(self, task\_id: int, new\_task: Task):  *"""  Update Task by id* **:param** *task\_id: id of updating Task* **:param** *new\_task: Task entity with updating info* **:return***: True if entity was updated else False  """* self.update\_from\_file(self.storage\_path)  new\_task.task\_id = task\_id  **for** i, task **in** enumerate(self.tasks):  **if** task.task\_id == task\_id:  self.tasks[i] = new\_task  self.save\_to\_file(self.storage\_path)  **return True  return False   def** change\_task\_status(self, task\_id: int, is\_completed: bool):  *"""  Change Task status* **:param** *task\_id: id of updating Task* **:param** *is\_completed: Task status* **:return***:True if entity was updated else False  """* self.update\_from\_file(self.storage\_path)  **for** i, task **in** enumerate(self.tasks):  **if** task.task\_id == task\_id:  self.tasks[i].is\_completed = is\_completed  self.save\_to\_file(self.storage\_path)  **return True  return False   def** update\_task\_info(self, task\_id: int, name: str, description: str, date\_start: str, duration: int):  *"""  Update Task info* **:param** *task\_id: id of updating Task* **:param** *name: updated name of Task* **:param** *description: updated description of Task* **:param** *date\_start: updated start date of Task* **:param** *duration: updated duration of Task* **:return***: True if entity was updated else False  """* self.update\_from\_file(self.storage\_path)  **for** i, task **in** enumerate(self.tasks):  **if** task.task\_id == task\_id:  self.tasks[i].name = name  self.tasks[i].description = description  self.tasks[i].date\_start = parser.parse(date\_start)  self.tasks[i].date\_end = parser.parse(date\_start) + timedelta(seconds=duration)  self.save\_to\_file(self.storage\_path)  **return True  return False   def** delete\_task(self, task\_id: int):  *"""  Delete Task* **:param** *task\_id: id of deleted Task* **:return***: True if entity was deleted else False  """* self.update\_from\_file(self.storage\_path)  length = self.tasks.\_\_len\_\_()  self.tasks = list(filter(**lambda** task: task.task\_id != task\_id, self.tasks))  self.save\_to\_file(self.storage\_path)  **return** length != self.tasks.\_\_len\_\_()   **def** add\_task(self, task: Task):  *"""  Add new Task to storage* **:param** *task: new Task entity* **:return***: added Task entity  """* self.update\_from\_file(self.storage\_path)  task.task\_id = self.next\_id  self.tasks.append(task)  self.next\_id += 1  self.save\_to\_file(self.storage\_path)  **return** task   **def** create\_task(self, name: str, description: str, date\_start: str, duration: int):  *"""  Create new Task* **:param** *name: name of new Task* **:param** *description: description of new Task* **:param** *date\_start: start date of new Task* **:param** *duration: duration of new Task* **:return***: created Task entity  """* self.update\_from\_file(self.storage\_path)  task = Task.create(self.next\_id, name, description, date\_start, duration)  self.tasks.append(task)  self.next\_id += 1  self.save\_to\_file(self.storage\_path)  **return** task   @staticmethod  **def** \_\_paginate\_array\_\_(array, offset=0, limit=**None**):  *"""  Paginate list of Tasks* **:param** *array: list of Tasks* **:param** *offset: count of skipping tasks* **:param** *limit: count of Tasks in page* **:return***: paginated list of Tasks  """* **if** isinstance(offset, int) **and** offset >= 0 **and** ((isinstance(limit, int) **and** limit > 0) **or** limit **is None**):  **return** array[offset:(limit + offset **if** limit **is not None else None**)]  **return** array   @staticmethod  **def** \_\_search\_filter\_\_(array: [Task], query=**None**):  *"""  Search Tasks in list* **:param** *array: list of Tasks* **:param** *query: searched parameter* **:return***: list of Tasks that consist searched parameter  """* length = array.\_\_len\_\_()  **if not** isinstance(query, str) **or** query.\_\_len\_\_() == 0:  **return** array, length  **return** list(filter(**lambda** x: re.search(query, x.name + x.description, re.IGNORECASE), array)), length   @staticmethod  **def** \_\_paginate\_and\_search\_\_(array, offset=0, limit=**None**, query=**None**, status\_filter=**None**):  *"""  Paginate list of Tasks and Search Tasks in it* **:param** *array: list of Tasks* **:param** *offset: count of skipping tasks* **:param** *limit: count of Tasks in page* **:param** *query: searched parameter* **:param** *status\_filter: searched Task status* **:return***: paginated list of Tasks that consist searched parameter  """* filtered, length = DataManager.\_\_search\_filter\_\_(array, query)  tasks\_result = DataManager.\_\_paginate\_array\_\_(filtered, offset, limit)  **return** Response(tasks\_result, offset, limit, query, length, status\_filter) |

**2. Звіт програм pep8, pyflakes та об’єм покриття тестами**

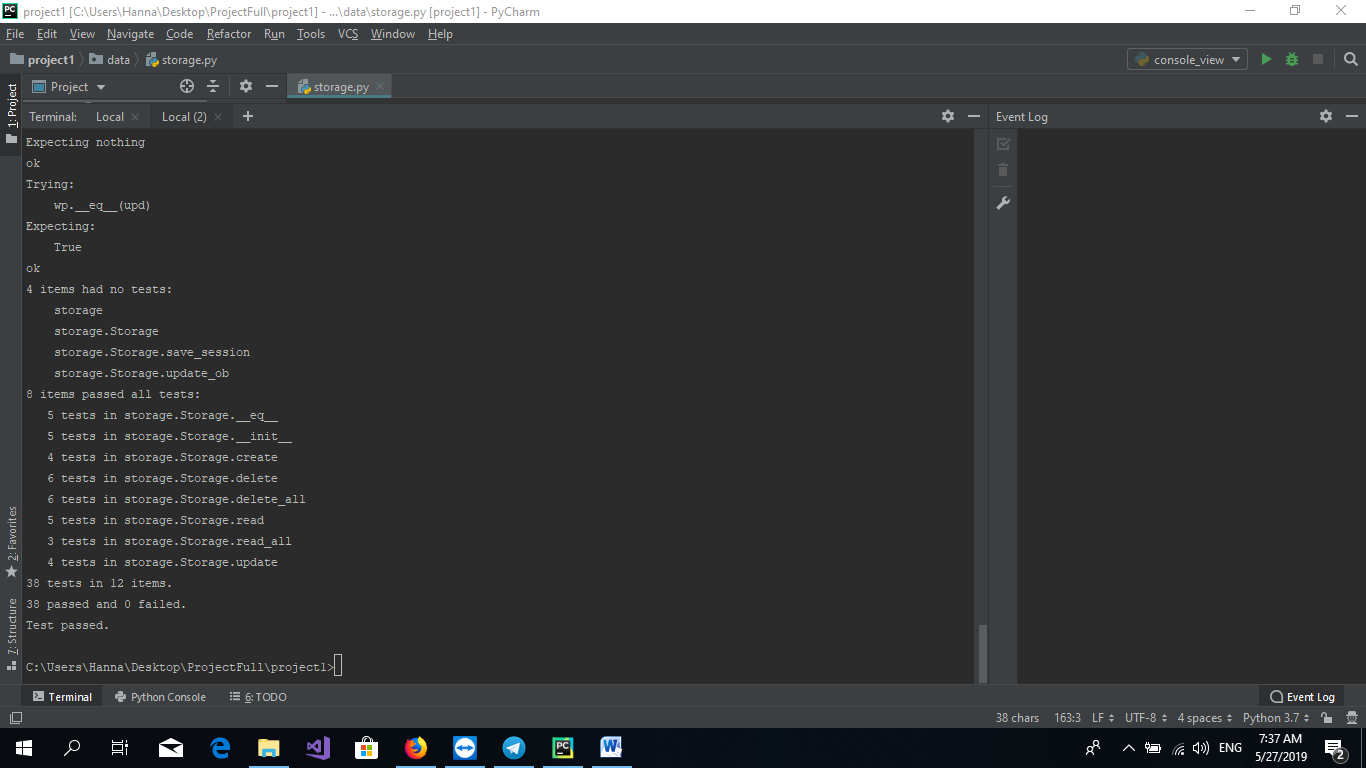


Рис. 8 Покриття тестами модуля storage

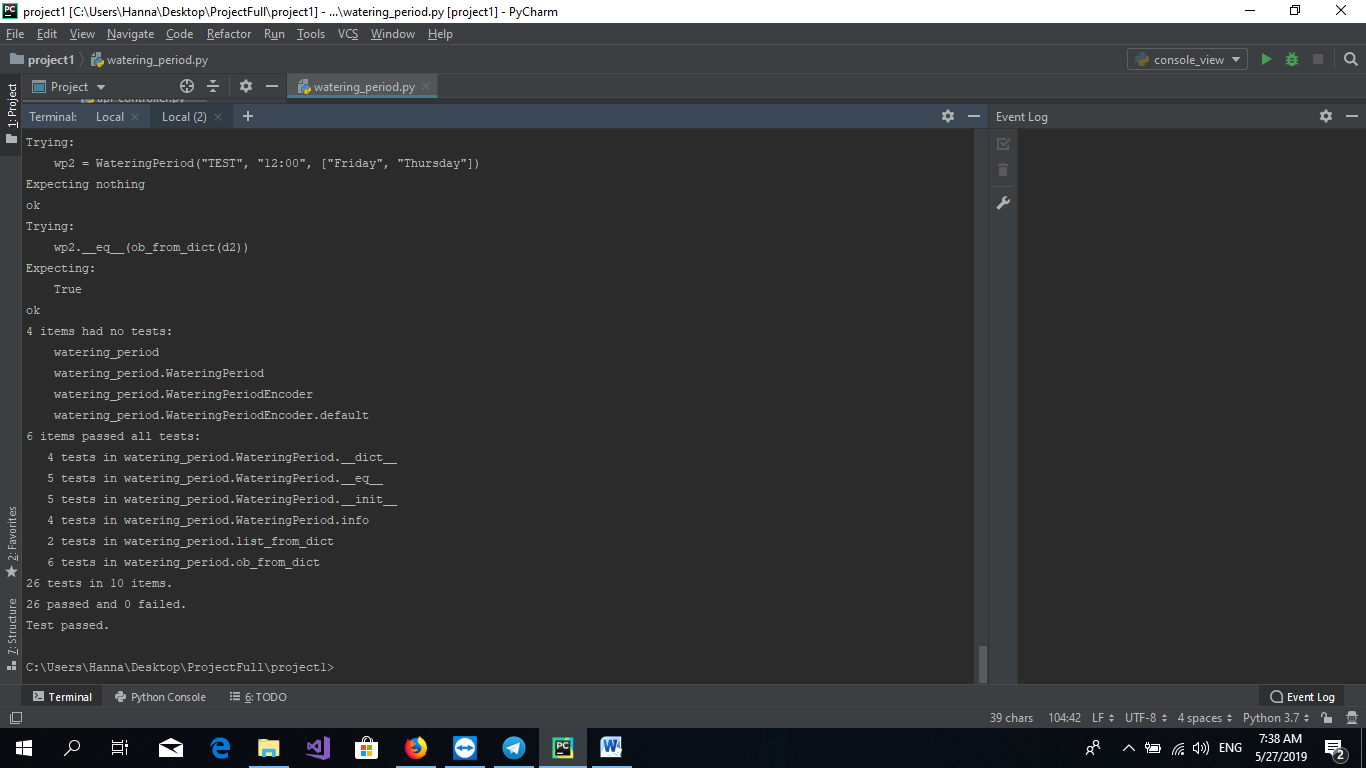


Рис. 9 Покриття тестами модуля watering\_period

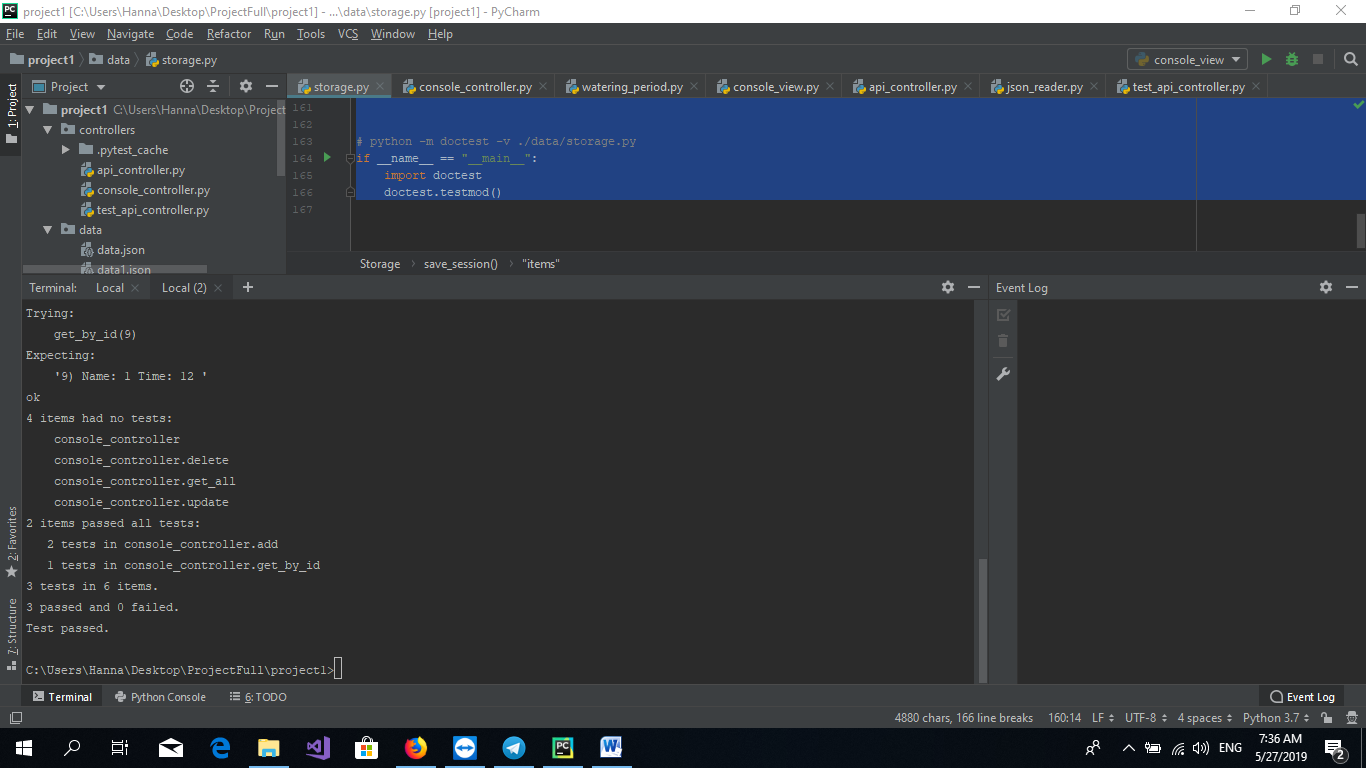


Рис. 10 Покриття тестами модуля console\_controller

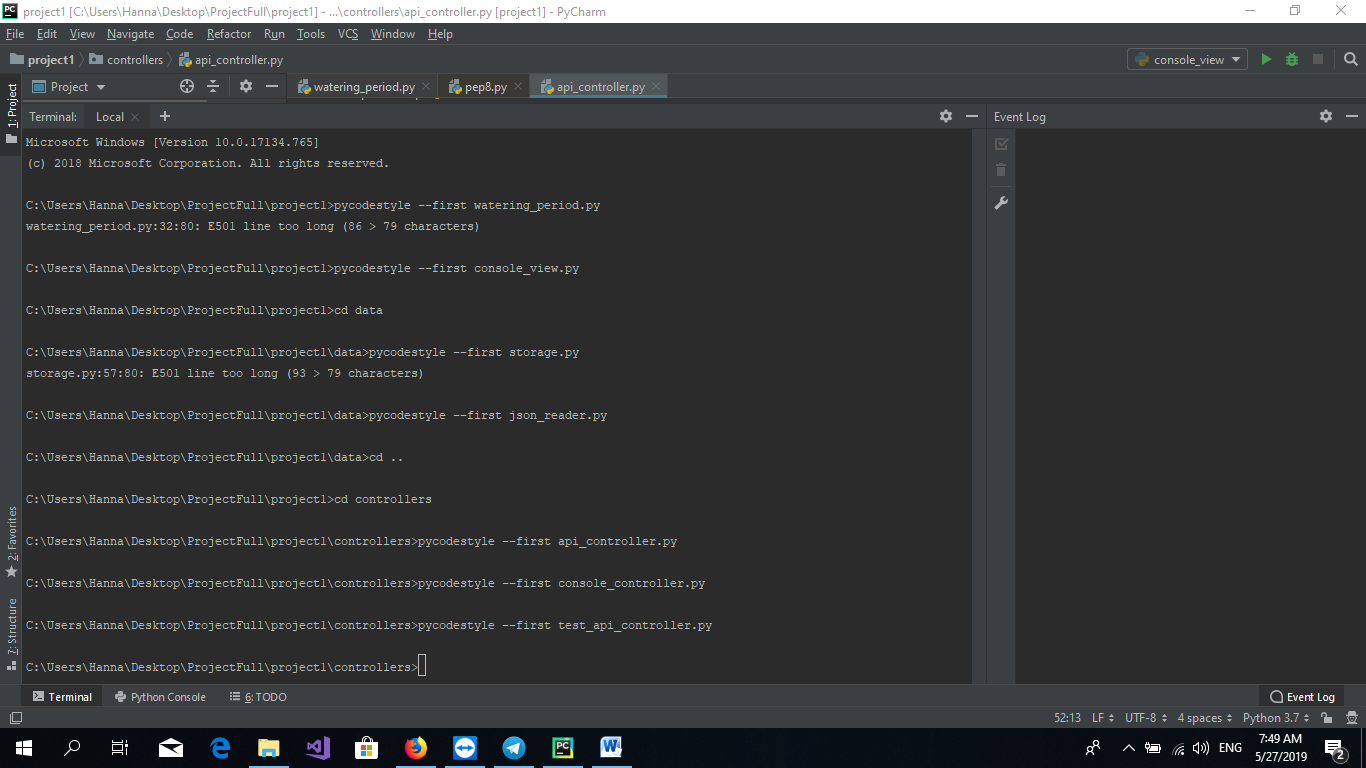


Рис.11 Звіт pep8

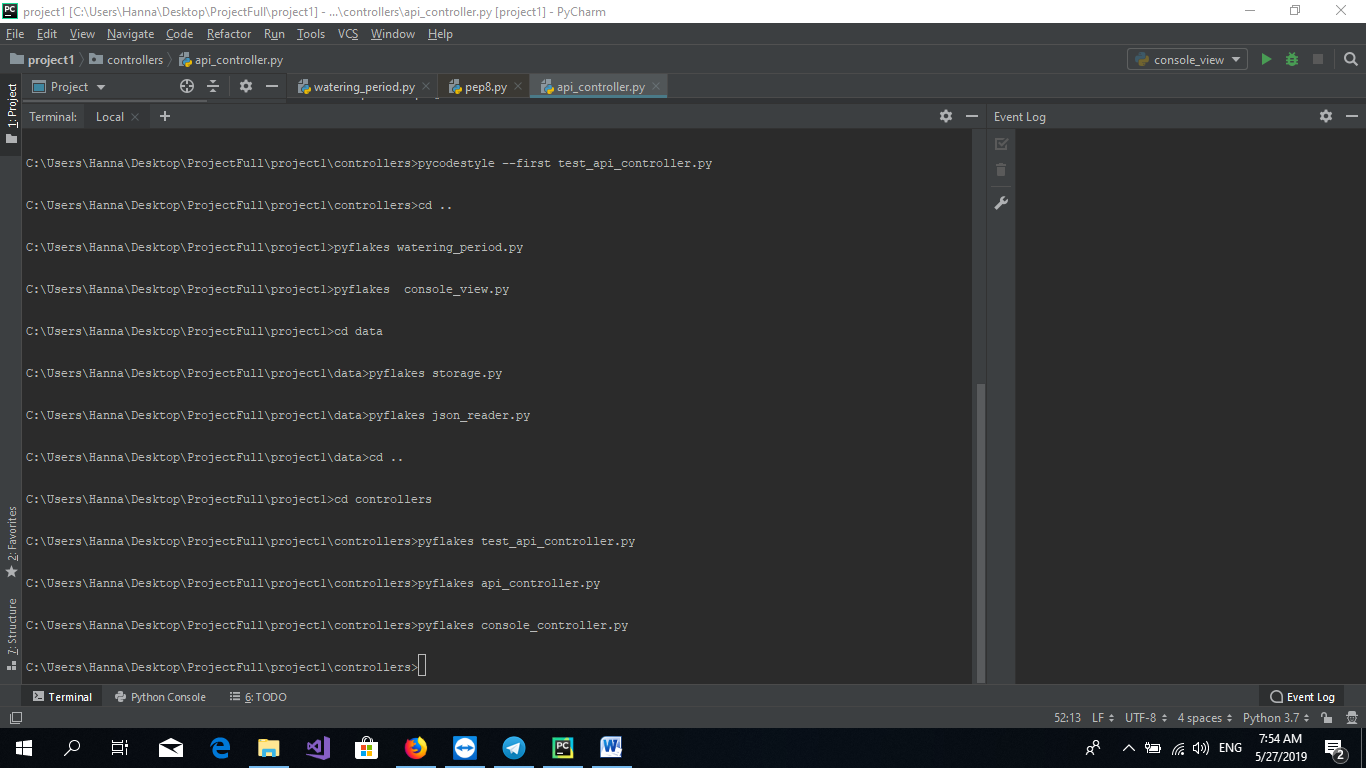


Рис.12 Звіт pyflakes

**Висновки**

Отже, в результаті виконання цієї лабораторної роботи, було опрацьовано процедурні конструкції мови Python та застосовано на практиці основний інструментарій розробника.

Було створено ПЗ, що має консольний інтерфейс, співпрацює з JSON документами та маэ REST-api.

Використано фреймворк flask-restful для реалізації REST-інтерфейсу для розроблених програмних засобів, до якого написано тести, за допомогою бібліотек pytest та requests.

За допомогою засобів статичного аналізу pep8 та pyflakes забезпечено сумісність програмного коду із стандартами кодування.

Розроблені програмні засоби розміщено у публічному сховищі програмного коду github.