







Полная комплектация структуры проекта:

 MANUAL.pdf	Microsoft Edge PDF Document	309 КБ
 record_logs.py	Python File	8 КБ
 sort_items.py	Python File	3 КБ
 tests.py	Python File	3 КБ
 config.ini	Параметры конфигурации	1 КБ
 logging.ini	Параметры конфигурации	1 КБ

Зависимости:

- Python 3.8.3;
- requests 2.25.1;
- sqlalchemy 1.3.23;
- psycopg2 2.8.6;

1. Получение логов выводом в файл **logs.json** после предварительного успешного тестирования данных в логах:

- 1) Переход в директорию **to_path** проекта со скриптом **tests.py** (при необходимости):

```
>>cd to_path
```

- 2) Запуск скрипта **tests.py** с реализацией API по выводу отсортированных логов в файл **logs.json** с протестированными в них данными:

```
>>python -m unittest
```

или

```
>>python tests.py
```

Набор логов получается запросом к REST API:
<http://www.dsdev.tech/logs/<date logs>>,

Где **date_logs** – дата создания логов. Параметр задаётся в **[logs]** в файле **config.ini**.

Набор логов сериализуется в список из структур данных python – **dict** и сортируется алгоритмом сортировки, реализованном в скрипте **sort_items.py**, после чего данные логов будут протестированы unittests. Данные в логах тестируются, на то, что:

- 1) Каждый лог содержит все необходимые поля ***first_name***, ***second_name***, ***created_at***, ***user_id***, ***message***.
- 2) Поле ***user_id*** успешно может быть преобразовано в тип ***int***.
- 3) Дата создания логa ***created_at*** действительно в формате ISO 8601.

Запуская каждый раз команду получения логов по п. 1 существующий файл полученных и протестированных логов ***logs.json*** удаляется и заменяется на новый с заново полученными логами.

2. Запись логов в базу данных:

- 1) Переход в директорию проекта ***to_path*** со скриптом ***record_logs.py*** (при необходимости):

```
>>cd to_path
```

- 2) Запуск скрипта с реализацией API по подключению и записи логов в базу данных postgres:

```
>>python record_logs.py
```

Параметры подключения к базе данных postgres – имя пользователя ***USERNAME***, пароль ***PASSWORD***, номер порта ***PORT***, имя базы данных ***DB_NAME*** заданы в ***[pgs_db]*** в файле ***config.ini***.

Данные записываются в таблицы:

logs:

<i>id</i>	<i>created_at</i>	<i>message</i>	<i>user_id</i>
Integer, Primary Key	TimeStamp Without TZ	Text	Integer, Foreign Key users

users:

<i>id</i>	<i>first_name</i>	<i>second_name</i>
Integer, Primary Key	CharCharacter Varying	CharCharacter Varying

Имена таблиц также можно задать в ***[tab_names]*** в файле ***config.ini***.

3. Логирование вызова метода **RecordLogs.record** добавления логов в базу данных модуля **record_logs.py**.

Все сообщения с приоритетом **DEBUG** и выше заносятся в журнал **config.log**.

Все сообщения с приоритетом **WARNING** и выше отображаются в stdout в консоли.

При каждом запуске модуля журнал **config.log** перезаписывается.

* * *

Алгоритм сортировки:

Вход: Последовательность элементов, над любыми парами которых можно совершать операции сравнения в арифметическом смысле. Пусть это будет последовательность логов **Logs** = {**Logs₁**, ..., **Logs_n**}, значения которых – дата их создания.

Выход: Упорядоченная последовательность логов **Logs**, такая что: **Logs₁** ≤ , ..., ≤ **Logs_n**

Над множеством элементов сортируемой последовательности логов **Logs** задано соотношение:

$$h(a, b) = \begin{cases} 1, & \text{если } a \geq b; \\ -1, & \text{если иначе} \end{cases}$$

Инициализация переменных алгоритма:

```
Logs ← {Logs2, ..., Logsn};  
rest_part ← ∅;  
i ← 1;  
stack ← {Logs1};
```

Где **n** = |**Logs**|

Идея алгоритма:

Каждый следующий **j**-й элемент **Logs_j** из **Logs** последовательно сравнивается с элементами стека **stack**, начиная с конца стека. Стек **stack** сортируется, перемещением из него

элементов в начало списка *rest_part*, которые меньше чем *Logs_j*. Если же *Logs_j* в какой-то момент становится больше или равным элементу из *stack*, то добавляем *Logs_j* в конец стека и так далее, пока не пройдем все элементы *Logs*. После того, как прошли все элементы *Logs*, у нас все его элементы находятся в *stack* или/и в *rest_part*, то есть какие-то уже отсортированы, а какие-то нет. Далее, просто переопределяем *Logs* новым значением, которым будет являться *rest_part* и повторяем предыдущие вышеописанные действия до тех пор, пока в *rest_part* остаются элементы.

Псевдокод алгоритма:

1	while True:
2	
3	for j = 1 to Logs :
4	if h(Logs _j , stack _i) == -1:
5	stack ← stack ∪ {Logs _j };
6	i ← i + 1;
7	else:
8	while h(Log _i , stack _i) == 1 & i > 0:
9	for k = 1 to rest_part + 1:
10	next ← rest_part _k ;
11	if k == 1:
12	rest_part _k ← stack _i ;
13	else:
14	rest_part _{k+1} ← next;
15	i ← i - 1;
16	else:
17	stack ← {stack ₁ , ..., stack _{i+1} };
18	stack ← stack ∪ {Logs _j };
19	i ← i + 1;
20	If rest_part == 0:
21	return stack;
22	break;
23	Logs ← rest_part;
24	rest_part ← ∅;

Подробное описание:

Стек *stack* инициализирован первым элементом списка *Logs*, указатель *i* инициализирован 1, *rest_part* – список логов, которые не удалось отсортировать на строках 4-19, инициализирован как пустое множество (пустой контейнер).

Алгоритм проходит последовательно по каждому элементу списка *Logs*, сравнивая текущий элемент списка *Logs_j*, на который указывает указатель *j* и элемент стека *stack_i*, на который указывает указатель *i*, помещая элемент *Logs_j* в стек *stack*, который будет являться отсортированным списком логов. Помещение элемента *Logs_j* в стек *stack* происходит, принимая во внимание следующие условия:

- 1) В случае, когда текущий элемент списка *Logs_j* меньше чем элемент стека *stack_i*, в стек *stack* в конец добавляется *Logs_j*, инкрементируем указатель *i* на 1 и двигаемся далее, сравнивая следующий элемент неотсортированного списка логов *Logs* с элементом *stack_i*.
- 2) В случае, когда текущий элемент списка *Logs_j* больше или равен элементу стека *stack_i*, то в начало списка неотсортированных логов *rest_part* добавляем *stack_i* (строки 9-15). Это производится до тех пор, пока удовлетворяется условие $h(Logs_j, stack_i) = 1 \ \& \ i > 0$ (строка 8 псевдокода выше). Если же условие в строке 8 не выполняется, то мы удаляем из стека *stack* элементы, которые добавили ранее в *rest_part* на строках псевдокода 9-15 и в стек *stack* добавляется текущий элемент списка логов *Logs_j*, с которым и производили сравнение по условию $h(Logs_j, stack_i) = 1$. После чего, инкрементируем указатель *i* на 1 и двигаемся далее, сравнивая следующий элемент неотсортированного списка логов *Logs* с элементом *stack_i*.

Пройдя весь список неотсортированных логов *Logs*, переходим к выполнению строк 20-24. Таким образом, у нас есть уже отсортированная по возрастанию часть логов *stack*, размер которой с ростом указателя *i* может быть оценён как величина порядка $\log(i)$, а список *rest_part* растёт как величина $i - \log(i)$. Что нам нужно далее, так это переопределить список неотсортированных логов *Logs* неотсортированной частью логов *rest_part*, которые ещё не отсортированы (строка 23), а из самого списка *rest_part* все логи удаляются, и он снова становится пуст (строка 24). Если список *rest_part* части логов, которые отсортировать не удалось на строках 4-19 стал пуст (строка 20), то считается, что все логи отсортированы и находятся в стеке *stack*, тогда и завершается алгоритм.

Таким образом на каждой итерации внешнего цикла (строка 1) мы совершаем не менее чем *p* сравнений *Logs_j* и *stack_i*. Величина *p* оценивается как: $p = \sum_{i=1}^n \log i$, а $n = |Logs|$ уменьшается на каждой следующей итерации внешнего цикла (строка 1) на величину $\log(n)$, где n – размер списка *rest_part* – части логов, которые не удалось отсортировать. Обращаю внимание, что $n = |Logs|$ справедливо для первой итерации внешнего цикла (строка 1).