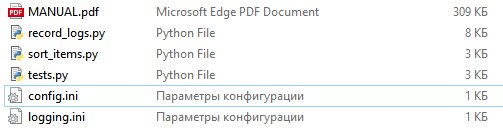
Полная комплектация структуры проекта:



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:
2. Переход в директорию ***to\_path*** проекта со скриптом ***tests.py*** (при необходимости):

>>cd ***to\_path***

1. 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](http://www.dsdev.tech/logs/%3cdate_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 лога ***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.

1. Запись логов в базу данных:
2. Переход в директорию проекта ***to\_path*** со скриптом ***record\_logs.py*** (при необходимости):

>>cd ***to\_path***

1. 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:**

|  |  |  |  |
| --- | --- | --- | --- |
| ***id*** | ***created\_at*** | ***message*** | ***user\_id*** |
| **Integer, Primary Key** | **TimeStamp Without TZ** | **Text** | **Integer, Foreign Key users** |

**users:**

|  |  |  |
| --- | --- | --- |
| ***id*** | ***first\_name*** | ***second\_name*** |
| **Integer, Primary Key** | **CharCharacter Varying** | **CharCharacter Varying** |

Имена таблиц также можно задать в ***[tab\_names]*** в файле ***config.ini*.**

1. Логирование вызова метода ***RecordLogs.record*** добавления логов в базу данных модуля ***record\_logs.py***.

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

Все сообщения с приоритетом **WARNIG** 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***, …, ***Logsn***}, значения которых – дата их создания.

Выход: Упорядоченная последовательность логов ***Logs***, такая что: ***Logs1*** ≤ , …, ≤ ***Logsn***

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

**h**(a, b) =

*Инициализация переменных алгоритма:*

***Logs*** ← {***Logs2***, …, ***Logsn***};

***rest\_part*** ← ∅;

***i*** ← 1;

***stack*** ← {***Logs1***};

Где ***n*** = |***Logs***|

*Идея* *алгоритма*:

Каждый следующий элемент ***Logsj*** 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 ***Logsj***. Если же ***Logsj*** в какой-то момент становится больше или равным элементу из ***stack***, то добавляем ***Logsj*** в конец стека и так далее, пока не пройдём 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(Logsj, stacki) == -1: | | | |  |  |
| 5 |  |  |  | stack ← stack ∪ {Logsj}; | | | |  |
| 6 |  |  |  | i ← i + 1; | |  |  |  |
| 7 |  |  | else: | |  |  |  |  |
| 8 |  |  |  | while h(Logj, stacki) == 1 & i > 0: | | | |  |
| 9 |  |  |  |  | for k = 1 to|rest\_part| + 1: | | |  |
| 10 |  |  |  |  |  | next ← rest\_partk; | | |
| 11 |  |  |  |  |  | if k == 1: | |  |
| 12 |  |  |  |  |  |  | rest\_partk ← stacki; | |
| 13 |  |  |  |  |  | else: |  |  |
| 14 |  |  |  |  |  |  | rest\_partk+1  ← next; | |
| 15 |  |  |  |  | i ← i - 1; |  |  |  |
| 16 |  |  |  | else: |  |  |  |  |
| 17 |  |  |  |  | stack ← {stack1, …, stacki+1}; | | | |
| 18 |  |  |  |  | stack ← stack ∪ {Logsj}; | | |  |
| 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 ***Logsj***, на который указывает указатель ***j*** и элемент стека ***stacki***, на который указывает указатель ***i***, помещая элемент ***Logsj*** в стек ***stack***, который будет являться отсортированным списком логов. Помещение элемента ***Logsj*** в стек ***stack*** происходит, принимая во внимание следующие условия:

1. В случае, когда текущий элемент списка ***Logsj*** меньше чем элемент стека ***stacki***, в стек ***stack*** в конец добавляется ***Logsj***, инкрементируем указатель ***i*** на **1** и двигаемся далее, сравнивая следующий элемент неотсортированного списка логов ***Logs*** с элементом ***stacki***.
2. В случае, когда текущий элемент списка ***Logsj*** больше или равен элементу стека stacki, то в начало списка неотсортированных логов ***rest\_part*** добавляем ***stacki*** (строки 9-15). Это производится до тех пор, пока удовлетворяется условие **h**(***Logsj***, ***stacki***) = **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 ***Logsj***, с которым и производили сравнение по условию **h**(***Logsj***, ***stacki***) = **1**. После чего, инкрементируем указатель ***i*** на **1** и двигаемся далее, сравнивая следующий элемент неотсортированного списка логов ***Logs*** с элементом ***stacki***.

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 ***Logsj*** и ***stacki***. Величина ***p*** оценивается как: ***p*** =, 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).