README.md

Задание

Poker

Задание: дополнить код в скрипте `poker.py`, инструкции в шапке файла

Цель задания: применить знания о итераторах\генераторах, получить (освежить) навык обращения с `itertools`. Умение умеренно использовать итераторы в правильных местах позволяет повысить читаемость и идиоматичность кода, а также облегчит чтение чужого кода.

Критерии успеха: задание **опционально**, минимальным критерием успеха является прохождение тестов функции `test_best_hand`. Дальнейшая успешность определяется code review, если здание отправлено на проверку.

Deco

Задание: дополнить код в скрипте `deco.py`. Там готовый шаблон кода, нужно реализовать несколько декораторов. Примеры запуска уже есть

Цель задания: получить (освежить) навык обращения с декораторами. В реальности декораторы используется достаточно часто для подключения функционала не связанного непосредственно с логикой самой функции (например кеширующий декоратор, декоратор замеряющий время работы или проверяющий соединие с сервером), так что их понимание и навык использования важны для любого Python разработчика.

Критерии успеха: задание **опционально**, минимальным критерием успеха является то, что `main` выводит сообразный логике кода результат. Дальнейшая успешность определяется code review, если здание отправлено на проверку.

Log Analyzer

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


# log_format ui_short '$remote_addr $remote_user  $http_x_real_ip [$time_local] "$request" '
#                     '$status $body_bytes_sent "$http_referer" '
#                     '"$http_user_agent" "$http_x_forwarded_for" "$http_X_REQUEST_ID" "$http_X_RB_USER"
'
#                     '$request_time';

config = {
    "REPORT_SIZE": 1000,
    "REPORT_DIR": "./reports",
    "LOG_DIR": "./log"
}


def main():
    pass

if __name__ == "__main__":
    main()
```

Жил-был чудный веб-интерфейса и все у него было хорошо: в него ходили пользователи, что-то там кликали, переходили по ссылкам, получали результат. Но со временем некоторые его странички стали тупить и долго грузиться. Менеджеры часто жалуются, мол "вот тут список долго грузился" или "интерфейсик тупит, поиск не работает". Но так трудно отделить те случаи, где проблемы на их стороне, а где действительно виноват веб-сервис. В логи интерфейса добавили время запроса (`$request_time` в `nginx` http://nginx.org/en/docs/http/ngx_http_log_module.html#log_format). Теперь можно распарсить логи и провести первичный анализ, выявив подозрительные URL'ы.

Про логи:

- семпл лога: `nginx-access-ui.log-20170630.gz`
- шаблон названия логов интерфейса соответствует названию сэмпла (ну, только время меняется)
- так вышло, что логи могут быть и plain и gzip
- лог ротируется раз в день
- опять же, так вышло, что логи интерфейса лежат в папке с логами других сервисов

Про отчет:

- count - сколько раз встречается URL, абсолютное значение
- count_perc - сколько раз встречается URL, в процентнах относительно общего числа запросов
- time_sum - суммарный $request_time для данного URL'a, абсолютное значение
- time_perc - суммарный $request_time для данного URL'a, в процентах относительно общего $request_time всех запросов
- time_avg - средний $request_time для данного URL'a
- time_max - максимальный $request_time для данного URL'a
- time_med - медиана $request_time для данного URL'a

Задание: реализовать анализатор логов `log_analyzer.py` .

Основная функциональность:

1. Скрипт обрабатывает при запуске последний (со самой свежей датой в имени, не по mtime файла!) лог в `LOG_DIR` , в результате работы должен получится отчет как в `report-2017.06.30.html` (для корректной работы нужно будет найти и принести себе на диск `jquery.tablesorter.min.js`). То есть скрипт читает лог, парсит нужные поля, считает необходимую статистику по url'ам и рендерит шаблон `report.html` (в шаблоне нужно только подставить `$table_json`). Ситуация, что логов на обработку нет возможна, это не должно являться ошибкой.
2. Если удачно обработал, то работу не переделывает при повторном запуске. Готовые отчеты лежат в `REPORT_DIR` . В отчет попадает `REPORT_SIZE` URL'ов с наибольшим суммарным временем обработки (`time_sum`).
3. Скрипту должно быть возможно указать считать конфиг из другого файла, передав его путь через `--config` . У пути конфига должно быть дефолтное значение. Если файл не существует или не парсится, нужно выходить с ошибкой.
4. В переменной `config` находится конфиг по умолчанию (и его не надо выносить в файл). В конфиге, считанном из файла, могут быть переопределены перменные дефолтного конфига (некоторые, все или никакие, т.е. файл может быть пустой) и они имеют более высокий приоритет по сравнению с дефолтным конфигом. Таким образом, результирующий конфиг получается слиянием конфига из файла и дефолтного, с приоритетом конфига из файла.
5. Использовать конфиг как глобальную переменную запрещено, т.е. обращаться в своем функционале к нему так, как будто он глобальный - нельзя. Нужно передавать как аргумент.
6. Использовать сторонние библиотеки запрещено.

Мониторинг:

1. скрипт должен писать логи через библиотеку logging в формате `'[%(asctime)s] %(levelname).1s %(message)s'` с датой в виде `'%Y.%m.%d %H:%M:%S'` (logging.basicConfig позволит настроить это в одну строчку).

Допускается только использование уровней `info`, `error` и `exception`. Путь до логфайла указывается в конфиге, если не указан, лог должен писаться в stdout (параметр filename в logging.basicConfig может принимать значение None как раз для этого).

2. все возможные "неожиданные" ошибки должны попадать в лог вместе с трейсбеком (посмотрите на logging.exception). Имеются в виду ошибки непредусмотренные логикой работы, приводящие к остановке обработки и выходу: баги, нажатие ctrl+C, кончилось место на диске и т.п.
3. должно быть предусмотрено оповещение о том, что большую часть анализируемого лога не удалось распарсить (например, потому что сменился формат логирования). Для этого нужно задаться относительным (в долях/процентах) порогом ошибок парсинга и при его превышении писать в лог, затем выходить.

Тестирование:

1. на скрипт должны быть написаны тесты с использованием библиотеки `unittest` (https://pymotw.com/2/unittest/). Имя скрипта с тестами должно начинаться со слова `test`. Тестируемые кейсы и структура тестов определяется самостоятельно (без фанатизма, в принципе достаточно функциональных тестов).

Цель задания: получить (прокачать) навык написания production-ready кода. То есть адекватного кода, который удобно расширять и поддерживать, протестированного и пригодного для мониторинга. Совпадение всех чисел с приведенным примером отчета целью не является (лишь бы похожи были =)

Критерии успеха: задание **обязательно**, критерием успеха является работающий согласно заданию код, для которого написаны тесты, проверено соответствие pep8, написана минимальная документация с примерами запуска (боевого и тестов), в README, например. Далее успешность определяется code review.

Распространенные проблемы:

- не стоит делать свои кастомные классы ошибок, это иногда (!) имеет смысл для библиотек, но не для задач подобного рода.
- ограничьтесь уровнями логирования DEBUG, INFO и ERROR: https://dave.cheney.net/2015/11/05/lets-talk-about-logging
- не выходите через sys.exit не из main. Это затрудняет тестирование и переиспользование кода.
- чтобы отрендерить шаблон не надо итерироваться по всем его строкам и искать место замены, можно воспользоваться, например, https://docs.python.org/2/library/string.html#string.Template.safe_substitute .
- функцию, которая будет парсить лог желательно сделать генератором.
- не забывайте про кодировки, когда читаете лог и пишите отчет.
- из функции, которая будет искать последний лог удобно возвращать namedtuple с указанием пути до него, распаршенной через datetime даты из имени файла и расширением, например.
- распаршенная дата из имени логфайла пригодится, чтобы составить путь до отчета, это можно сделать "за один присест", не нужно проходится по всем файлам и что-то искать.
- протестируйте функцию поиска лога, она не должна возвращать .bz2 файлы и т.п. Этого можно добиться правильной регуляркой.
- найти самый свежий лог можно за один проход по файлам, без использования glob, сортировки и т.п.
- нужный открыватель лога (open/gzip.open) перед парсингом можно выбрать через тернарный оператор.
- проверка на превышение процента ошибок при парсинге выполняетя один раз, в конце чтения файла, а не на каждую строчку/ошибку.

Deadline

Задание нужно сдать через неделю. То есть ДЗ, выданное в понедельник, нужно сдать до следующего занятия в понедельник (NOTE: если группа на "медленном старте", то две недели, соответственно). Код, отправленный на ревью в это время, рассматривается в первом приоритете. Нарушение делайна (пока) не карается, но может повлиять на ранжирование при выборе топа студентов при окончании курса. Пытаться сдать ДЗ можно до конца курсы. Но код, отправленный с опозданием, когда по плану предполагается работа над более актуальным ДЗ, будет рассматриваться в более низком приоритете без гарантий по высокой скорости проверки

Обратная связь

Студент коммитит все необходимое в свой github/gitlab репозитарий (пример структуры репозитария https://github.com/s-stupnikov/otus-python-0717 , http://docs.python-guide.org/en/latest/writing/structure/#structure-of-the-repository, https://realpython.com/python-application-layouts/). Далее необходимо зайти в ЛК, найти занятие, ДЗ по которому выполнялось, нажать “Чат с преподавателем” и отправить ссылку. После этого ревью и общение на тему ДЗ будет происходить в рамках этого чата.