## Как сдавать ревью

Дедлайн на первую посылку 20 апреля 23:59. Первая посылка обязательно должна работать, иначе ревью не принимается! Заметьте, не “не падать”, а именно корректно работать. pep8 - обязательный код стайл, приведенная команда не должна ничего вывести в вашей директории:

pycodestyle . --max-line-length=120 -r

В первой посылке обязательна только базовая функциональность, бонусы можно досылать в последующих итерациях. Только учтите, что они тоже подлежат ревью. Если они не успеют его успешно пройти, они не засчитаны. А в конце семестра, сами понимаете, число сдающих может вырасти вместе со временем ожидания комментариев от проверяющего) Так что, чем раньше вы сделаете бонусы, тем лучше.

Сам процесс сдачи устроен так:

* Выбираем одну из трех задач
* Создаем пустой **приватный** репозиторий на github
* Добавляем туда семинариста (Onaga1958): Settings->Manage Access->Invite a collaborator
* Коммитим в ветку **master** README.md с небольшим описанием
* Создаем ветку **dev** от **master**
* Пишем код в ветке **dev** и пушим в репозиторий
* Создаем pull request из **dev** в **master** и **не вливаем** его. В нем будет проходить обсуждение задачи.
* И дальше пока семинарист не принял ревью повторяем следующие шаги:
  + Ждем комментариев
  + Исправляем код, коммитим в **dev**, пушим. На исправление замечаний дается неделя.
* Успеть пройти ревью нужно до начала зачетной сессии, так что чем быстрее будет исправлять замечания тем лучше

## Генерация текста

Нужно написать автоматический генератор текста с помощью простой статистической языковой модели. Примерный алгоритм такой. На первой стадии, условно стадии обучения, мы по какому-то массиву текста, например “Война и Мир”, определяем насколько часто после слова w\_1 встречается слово w\_2. На стадии генерации мы генерируем очередное слово с соответствующим распределением частот. Аналогично, можно генерировать слово в зависимости от двух последних слов, от трех и так далее.

### Алгоритм более подробно

Будем использовать предположение [Маркова](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BF%D1%8C_%D0%9C%D0%B0%D1%80%D0%BA%D0%BE%D0%B2%D0%B0). Идея проста: давайте считать, что вероятность i-го слова, зависит не от всех предыдущих слов, а лишь от нескольких последних. Это упрощение реальной картины, понятно, что на самом деле все сложнее. Но как оценить вероятность? Вот у нас есть слово “каша”, как подсчитать вероятность встретить после него слово “гречневая”? Мы можем взять большой текст, найти в нем все упоминания слова “каша” и посмотреть какие слова идут за ним. Доля случаев, когда после “каша” мы обнаружим “гречневая”, и будет искомой оценкой вероятности.

Теперь более формально. Алгоритм состоит из двух стадий: подсчет вероятностей и генерация текста. В первой стадии для начала нужно разбить текст на **токены**. Для простоты можем считать токенами последовательности из букв (то есть слова) и пунктуацию, а остальные символы отфильтровывать. Дальше, считаем частоту встречаемости в тексте для всех последовательностей токенов длины не больше глубины depth. Последовательность из n токенов, кстати, называют n-граммой. n-граммы короче depth нужны для генерации начала текста. Затем, частоты приводим к вероятностям: n-граммы с одинаковыми n-1 начальными токенами нужно объединить в одну группу и их частоты отнормировать. Таким образом, для истории из n-1 токена мы получим распределение вероятней для следующего токена. В том числе нужно подсчитать распределение и для пустой истории, то есть просто частоты токенов. Вычисленный словарь с вероятностями сохраняем в файл. Замечу, что код не обязан работать ровно таким образом. Главное, чтобы на выходе получился такой же результат.

Давайте посмотрим на пример. Пусть входной текст: “a b c a b a”. Тогда с depth равной трем у нас будут следующие вероятности:

* для пустой истории - {a: 0.5, b: 1/3, c: 1/6}, так как всего токенов 6, “а” встретилась 3 раза, “b” два раза, “c” один;
* для истории из одного токена “a” - {“b”: 1}, после токена “a” мы видели только токен “b”;
* для (b,) - {“c”: 0.5, “a”: 0.5}, после токена “b” по разу встретились “c” и “a”;
* для (c,) - {“a”: 1};
* для (a, b) - {“c”: 0.5, “a”: 0.5}, комбинация из идущих подряд “a” и “b” встречается дважды, один раз после них идет “c”, а в другой раз - “a”;
* для (b, c) - {“a”: 1};
* для (с, a) - {“b”: 1}.

На стадии генерации читаем из файла вероятности и начинаем генерировать. Смотрим на предыдущие depth-1 сгенерированных токенов, достаем из словаря распределение следующего токена и семплируем из него. В начале генерации у нас более короткая история, поэтому смотрим не на depth-1 токен, а на сколько есть. В процессе генерации могут возникать depth-1-граммы, которых в истории не было. Тогда смотрим на распределение следующего токена для depth-2-граммы. Если и такой в нашем словаре нет, будем продолжать уменьшать историю, пока не дойдем до пустой.

Теперь давайте посмотрим на пример генерации с вероятностями из предыдущего примера. Пусть нам нужно сгенерировать 5 токенов. В начале, когда мы еще ничего не сгенерировали, история пустая. Значит берем распределение {a: 0.5, b: 1/3, c: 1/6}. Пусть нам повезло и мы с вероятностью ⅙ выбрали “c”. Теперь история состоит из одного токена “c”. “c” - редкий токен, он встретился всего 1 раз, поэтому для него мало статистики и распределение {“a”: 1} получилось тривиальным. Деваться некуда, следующий токен “a”. Теперь история состоит из двух токенов: “c” и “a”, для такого префикса распределение тоже тривиальное, генерируем “b”. Теперь, так как глубина равна трем, мы смотрим не на все токены, а лишь на два последних. То есть история теперь состоит из токенов “a” и “b”, для них распределение такое {“c”: 0.5, “a”: 0.5}. Пусть получилось “a”, теперь история (b, a), для неё распределения нет, ведь на этих токенах заканчивался исходный текст. Значит придется уменьшить историю до одного токена “a”, а значит следующий токен без вариантов “b”. Итого у нас вышла последовательность “c a b a b”.

При генерации также нужно сделать предложения и знаки препинания похожими на нормальные тексты: предложение должно начинаться с заглавный буквы, пробел стоять только после знака препинания, а не до него, не должно быть несколько знаков препинания подряд, если в предложении открываются кавычки, они должны закрываться в том же предложении, для открывающей кавычки/скобки пробел должен стоять перед ней, а не после, и в каждом предложении скобки должны образовывать правильную скобочную последовательность и т.п. Пример **плохой** генерации:  
" Ok, i agree", said Mark( he always goes to the club)." Bla bla", said Mark( he always goes to the club", said Tom( he always agrees).

Пример **хорошей** генерации:

"Ok, i agree", said Mark (he always goes to the club). "Bla bla", said Mark (he always goes to the club, said Tom (he always agrees)).

### Формат решения

Программа должна вызываться из консоли в двух режимах. В режиме подсчета вероятностей она должна принимать файл с входным текстом, файл, куда нужно сохранить вероятности, и глубину depth. Например, так:

python3 main.py calculate\_probabilities --input\_file <path\_1> --probabilities\_file <path\_2> --depth <depth>

В режиме генерации она должна принимать файл с вероятностями, глубину, сколько токенов нужно сгенерировать и опционально файл для вывода сгенерированного текста. Если он не указан - выводить в консоль.

В файл README.md нужно добавить ссылку на текст, который вы нашли, с его небольшим описанием, несколько примеров генерации, которые были получены на основе этого текста, и пример команд для запуска вашей программы в двух режимах.

### Технические детали

Для сохранения и загрузки подсчитанных вероятностей можно использовать библиотеку [pickle](https://docs.python.org/3/library/pickle.html#examples) или [json](https://docs.python.org/3/library/json.html). Последняя, правда, работает только со словарями, причем только с теми, где в качестве ключей используются строки.

Чтобы парсить аргументы командной строки, рекомендуется использовать библиотеку [argparse](https://docs.python.org/3/library/argparse.html). Обратите внимание на subparser’ы, их довольно естественно использовать, когда программа запускается в нескольких разных режимах.

Чтобы выбирать следующий токен, можно использовать функцию [random](https://docs.python.org/3/library/random.html#random.random) из стандартной библиотеки random, которая возвращает случайное число от 0 до 1.

Нестандартные библиотеки использовать нельзя.

### Где взять текст?

Это может быть любой осмысленный текст на русском или английском, длинной минимум в 10 000 слов. Литературное произведение, сборник анекдотов, сборник статей по биологии, журнал для автолюбителей. Проще всего, конечно, взять какую-нибудь относительно большую книгу в текстовом формате. [Вот](http://www.gutenberg.org/files/11/11-0.txt), например, “Алиса в стране чудес” на английском.

### Бонусы

* 0.2 балла. В первом режиме добавить в качестве еще одного аргумента командной строки регулярное выражение для определения токена. Например, если указать “\d+”, только последовательности цифр будут токенами, а остальные символы фильтруются
* 0.2 балла. Сделать в режиме генерации дополнительный аргумент, вероятность uniform\_proba. Перед генерацией следующего токена, мы “подкидываем” еще одну “монетку”. С вероятностью 1 - uniform\_proba следующий токен генерируется как и ранее, а с вероятностью uniform\_proba он выбирается из равномерного распределения по всем токенам. Таким образом, вы реализуете [сглаживание Лапласа](https://en.wikipedia.org/wiki/Additive_smoothing). Заметьте, текст не должен перестать быть “нормальным”, то есть по-прежнему не должно быть двух запятых подряд и т.п.
* 0.6 балла. Сделать второй режим интерактивным. Теперь вместо генерации указанного числа токенов, мы начинаем принимать команды из консоли, примерно как это делает ipython. Пользователь должен с помощью команд уметь:
  + генерировать указанное число токенов. Если пользователь делает две генерации подряд, вторая продолжает первую, то есть история не сбрасывается;
  + сбрасывать текущую историю, то есть начинать генерацию заново;
  + давать “затравку”, то есть несколько токенов, с которых нужно начать следующую генерацию;
  + посмотреть распределение для следующего токена, причем, если возможностей для продолжения слишком много, выводим только несколько наиболее вероятных;
  + изменить глубину;
  + посмотреть помощь по имеющимся командам;
  + завершить генерацию.
* 1 балл. Реализовать [Kneser–Ney smoothing](https://en.wikipedia.org/wiki/Kneser%E2%80%93Ney_smoothing)

## Шифрование

Нужно реализовать несколько простых шифров для латинского алфавита: [Шифр Цезаря](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80_%D0%A6%D0%B5%D0%B7%D0%B0%D1%80%D1%8F) и [Шифр Виженера](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80_%D0%92%D0%B8%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D0%B0). Программа должна уметь шифровать, дешифровать, а также взламывать шифр, последнее только для Цезаря. Для простоты, пусть программа работает только с буквами латинского алфавита, а остальные символы оставляет как есть.

### Как работает взлом

Предлагается расшифровать шифр Цезаря без знания сдвига. Известно, что разные буквы имеют разную частоту. Для английского языка, например, самой частотной является буква “e”. Зная, какая должна быть частота встречаемости у каждого символа в “обычном” для языка тексте, легко подобрать нужный сдвиг. Мы можем перебрать все возможные 26 сдвигов и выбрать тот из них, для которого частоты символов будут ближе всего к нормальным для языка. Не имея же под рукой частоты символов, можно вычислить их по какому-нибудь тексту на английском, например по ["Алисе в стране чудес"](http://www.gutenberg.org/files/11/11-0.txt).

Остался вопрос, что значит “ближе всего” для частот букв в языке. На самом деле нам подойдет любое вменяемое расстояние между векторами чисел, например сумма квадратов разностей .

### Формат решения

Программа должна вызываться из консоли в четырех режимах. В режиме шифрования она принимает входной файл с текстом, который нужно зашифровать, выходной файл, куда будет записан результат, шифр (Цезарь или Виженер) и ключ шифра. Входной и выходной файл могут быть не указаны, тогда ввод или вывод осуществляется через консоль. Пример того, как может выглядеть вызов программы:

python3 main.py encode --input\_file <path\_1> --output\_file <path\_2> --cipher <caesar или vigenere> --key <ключ шифра: число в случае Цезаря, слово в случае Виженера>

Второй режим - дешифрование, у него такие же аргументы, только, естественно, входной файл содержит зашифрованный текст, а выходной файл - дешифрованный. Имейте в виду, дешифрование должно быть полностью обратно к шифрованию. То есть последовательность команд encode + decode должна давать на выходе исходный файл.

Третий режим - подсчет частот букв в тексте. В этом режиме два аргумента: входной файл с текстом и выходной файл, куда нужно сохранить подсчитанные частоты.

И наконец взлом. В этом режиме программа принимает три аргумента: входной файл с текстом, который нужно расшифровать, выходной файл, куда нужно записать ответ, и файл с частотами букв.

В README.md нужно привести пример команд для запуска вашего кода в каждом из режимов.

### Технические детали

Для сохранения и загрузки подсчитанных вероятностей можно использовать библиотеку [pickle](https://docs.python.org/3/library/pickle.html#examples) или [json](https://docs.python.org/3/library/json.html). Последняя, правда, работает только со словарями, причем только с теми, где в качестве ключей используются строки.

Чтобы парсить аргументы командной строки, рекомендуется использовать библиотеку [argparse](https://docs.python.org/3/library/argparse.html). Обратите внимание на subparser’ы, их довольно естественно использовать, когда программа запускается в нескольких разных режимах.

Нестандартные библиотеки использовать нельзя.

### Бонусы

* 0.4 балла. Поддержать [шифр Вернама](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D1%84%D1%80_%D0%92%D0%B5%D1%80%D0%BD%D0%B0%D0%BC%D0%B0) в шифровании и дешифровании
* 0.2 балла. Поддержать буквы русского алфавита, пунктуацию и пробелы во всех режимах. Сдвиг можете сделать как вам больше нравится. Можно отдельные циклы для латинских, русских букв и остальных символов, можно все вместе в один цикл. Но внутри цикла буквы должны идти по алфавиту. И разумеется в дешифровщике должна быть такая же логика, что и в шифраторе
* 0.4 балла. Если запомнить из большого текста помимо частот букв еще и слова, можно выбирать сдвиг с помощью поиска по словарю. Это может помочь расшифровать короткие тексты, для которых частоты символов дают слишком шумный сигнал. Так что такой режим взлома можно включать автоматически, если длина входного текста меньше некого порога. Тут есть несколько подводных камней. Во-первых, нельзя смотреть на все слова из большого текста, ведь слово может совпасть случайно, так что нужно оставить в словаре только самые частотные слова. Количество или процент слов, которые мы оставляем, нужно сделать параметром. Во-вторых, слова из словаря могут найтись для разных сдвигов, а некоторые последовательности букв вообще могут попадать в словарь при разных сдвигах, например cf может быть or или be. В этом случае можно отдавать предпочтение сдвигу, где, например, нашлось больше словарных слов.
* 1 балл. Взлом шифра Виженера с помощью [индексов совпадений](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81_%D1%81%D0%BE%D0%B2%D0%BF%D0%B0%D0%B4%D0%B5%D0%BD%D0%B8%D0%B9)

## Игра “Лабиринт” (нужно ООП)

Тут вам предстоит реализовать консольную версию игры на бумаге “Лабиринт”. Суть игры такова: есть некое поле со стенами, ловушками и выходом, которое известно только ведущему. Остальные игроки в тайне друг от друга указывают ему координаты стартовой локации. Дальше они начинают по очереди ходить по лабиринту, а ведущий сообщает им, уперлись они в стену или смогли пройти. Таким образом, игроки исследуют карту, внимательно озираясь по сторонам, думая, где же расположены ловушки, и пытаясь добраться до выхода раньше соперников. Это продолжается до тех пор, пока какой-нибудь игрок не наткнется на выход и в этом случае он победил!

### Какие есть клетки

* Оглушение. Игрок, пришедший на эту клетку, пропускает следующие несколько шагов. Его извещают о том, куда он попал, и сколько ходов пропустит.
* Резиновая комната. Игрок может выйти из этой клетки только в одном заданном, известном только ведущему, направлении. Если он попытается пойти в другую сторону, то просто останется на месте, а ведущий нагло ему соврет, что он успешно продвинулся. Когда игрок все же сходит в нужном направлении, ему сообщается, что он вышел из резиновой комнаты. Сколько он там пробыл не уточняется.
* Телепорт. Игрок, пришедший на эту клетку, переносится на клетку с заданными, известными только ведущему, координатами. Игроку сообщается, что его телепортировало, но не сообщается куда. Эффект клетки, куда он попал, не срабатывает.
* Оружейная. Если у игрока, пришедшего на эту клетку, меньше трех патронов (на старте игры их ни у кого нет), то число патронов в его инвентаре становится равно трем. Теперь он может стрелять в заданном направлении. Если он падает в другого игрока, тот погибает, его инвентарь становится пустым, он отправляется на свою стартовую точку и пропускает свой следующий ход. Если игроки стоят на одной клетке, то выстрел в любую сторону - попадание.
* Выход. Если игрок двигается из этой клетки в заданном направлении, он выигрывает. И ему об этом даже сообщается.

### Формат решения

Программа должна вызываться из консоли в двух режимах: проверка карты и игра. В первом режиме она принимает на вход файл с полем и проверяет, что из любой клетки можно добраться до выхода. Сделать это можно с помощью обхода в ширину или глубину. Карта при этом полностью известна, то есть, после попадания в телепорт, вы знаете на каких координатах оказались. Если проверка прошла успешно, нужно вывести на экран “OK”, если нет, выводим “FAILED” и координаты точки, из которой нельзя добраться до выхода. Пример запуска:

python3 main.py check --field <path\_1>

В режиме игры запускается игра в интерактивном режиме, то есть игроки будут вводить команды в консоли, а им туда же будут выводиться ответы “ведущего”. В качестве аргумента программа принимает файл с полем, число игроков и стартовые позиции. Последние можно передать явно, а можно через файл, чтобы игроки не могли подсмотреть их друг друга. Число игроков и стартовые позиции - опциональные аргументы. Если они не указаны, программа запрашивает эти данные при старте игры, разумеется, сначала сообщив размер поля. Когда стартовые позиции заданы, игроки начинают по очереди ходить. В свой ход игрок может ввести такие команды:

* в/н/п/л - сходить вверх, вниз, вправо, влево. Для удобства можно сделать возможность вводить направления по-разному, например вверх, up, и т.п.
* патрон <направление> - выстрел в указанном направлении. На это тратится ход. Если у игрока нет патронов, сообщить ему об этом вместо выстрела.
* инвентарь - узнать, что лежит в инвентаре, в базовом варианте - сколько у игрока патронов. На это не тратится ход
* помощь - посмотреть список доступных команд с описанием. На это, разумеется, тоже не тратится ход

Когда один из игроков находит выход, пишем, что он победитель, и завершаемся.

В README.md нужно привести пример команд для запуска вашего кода в каждом из режимов.

### Формат описания поля

Давайте сразу на примере.

3 3

.|S E

. . \_

S . L

. . .

A R .

E Exit(UP)

S Stun(2)

A Armory()

R RubberRoom(RIGHT)

L RubberRoom(LEFT)

В первой строчке указываем размер поля, в данном случае 3 на 3. Дальше идет описание поля. На нечетных линиях описаны клетки и вертикальные стены, на четных линиях - горизонтальные стены. На нечетной линии нечетные символы задают клетки: “.” означает, что это обычная пустая клетка, а буква - означает особую клетку, значение которой описано ниже. Четные символы нечетной линии задают вертикальные стены: пробел означает отсутствие стены, а “|” означает, что стена есть. На четной линии четный символы ничего не задают, а нечетные задают горизонтальные стены: пробел означает отсутствие стены, а “\_” или “-” означает, что стена есть. В данном примере, стена есть между левым верхним углом (координаты 0,0) и соседней справа клеткой (0, 1) и между правым верхним углом (0, 2) и соседней снизу клеткой (1, 2).

Затем, идет описание особых клеток, каждая буква описывается на отдельной строчке. В этих строчках идет буква, а дальше вызов класса особой клетки с необходимыми ему аргументами. В нашем примере мы одну из клеток (0, 2) сделали выходом, причем чтобы выйти нужно пойти наверх, другие две сделали оглушением на два хода ((0, 1) и (1, 0)) и так далее. Заметьте, что одна буква в поле может использоваться несколько раз.

Классы в вашей программе могут называться по-другому и иметь другие аргументы, если вам так удобней, это просто пример.

### Технические детали

Чтобы парсить аргументы командной строки, рекомендуется использовать библиотеку [argparse](https://docs.python.org/3/library/argparse.html). Обратите внимание на subparser’ы, их довольно естественно использовать, когда программа запускается в нескольких разных режимах.

В поле часть карты задается с помощью кусочков кода, а именно задание особых клеток. В питоне код, помещенный в строку, можно запустить с помощью встроенной функции [eval](https://docs.python.org/3/library/functions.html#eval). Имейте в виду, что в “дикой природе” использовать эту функцию нужно с осторожностью. Если строка, которую вы хотите выполнить с помощью eval, приходит от внешнего пользователя, он туда может положить код, который делает что-то плохое. К этой задаче это не относится, jfyi.

Нестандартные библиотеки использовать нельзя.

### Бонусы

* 0.4 балла. На карту добавляется ключ. Теперь чтобы выйти, нужно сначала найти и подобрать ключ, а лишь потом направиться к выходу. А что делать другим игрокам, когда один из них уже подобрал ключ? Конечно же пристрелить счастливчика! Если игрок погибает, его инвентарь остается лежать на той клетке, где его настигла пуля.
* 0.6 балла. Новая клетка - сон. Когда игрок переходит на неё, он “засыпает” и переносится на другое поле. Разумеется, ему об этом не сообщают. Следующие несколько ходов он, ничего не подозревая, будет бродить по другому полю. Там тоже могут быть стены, ловушки, там можно найти ключ, можно умереть, ведь другие игроки заснув попадают на ту же карту, или даже выиграть! Но чтобы с вами не случилось, когда вы проснетесь, по истечению времени или из-за смерти или победы во сне, ваш инвентарь вернется к состоянию, в котором он был до сна, и вы вновь окажетесь на исходной карте, в той точке, где уснули. Ведущий сообщает, что игрок проснулся и что находится у него в инвентаре на самом деле, но не говорит, сколько ходов игрок проспал и где он находится. Пока игрок спит, его настоящее тело продолжает стоять на месте. Это значит, что во время сна игрока могут застрелить и наяву. Иными словами, по сну на самом деле ходит не сам игрок, а его “дух”. Теперь, как задать карту для сна? Это делается следующим образом:

2

2 2

. E

Z .

1 1

.

E Exit(1)

Z Sleep(2, (1, 0, 0))

В первой строчке задаётся число полей, дальше размер первого (нулевого вернее), затем его описание, а потом размер следующего поля и так далее. Координаты становятся трехмерными, первый индекс - индекс поля. Стартовые позиции игроков по-прежнему двумерные, они начинают на поле с нулевым индексом. В данном примере клетка сна на два хода переносит “дух” игрока на поле с индексом 1 в точку с координатами (0, 0). Обратите внимание, что внутри сна может быть другой сон! Единственное ограничение - между разными полями не должно быть циклов, в частности из сна не должно быть переходов обратно на “настоящее” поле.

* 1 балл. Еще один режим работы: случайная генерация карты. На вход принимаем размер карты, файл, куда нужно записать созданное поле, и опционально [random seed](https://docs.python.org/3/library/random.html#random.seed) для возможности воспроизвести результат. В лабиринте должны с какой-то вероятностью появляться особые клетки разных типов. Алгоритм можете выбрать какой хотите, главное чтобы в итоге из любой точки можно было дойти до выхода. Вот несколько источников, где можно почитать про методы генерации обычных лабиринтов:
  + <https://en.wikipedia.org/wiki/Maze_generation_algorithm>
  + <http://www.astrolog.org/labyrnth/algrithm.htm>
  + <https://habr.com/ru/post/262345/>
  + <https://habr.com/ru/post/176671/>
  + <https://habr.com/ru/post/319532/>