Git

Git и GitHub

Git является одной из самых популярных системой контроля версий (SVC). Так называют программу, которая позволяет хранить разные версии одного и того же документа, легко переключаться между ранними и поздними вариантами, вносить и отслеживать изменения.

Установка Git на Linux (Ubuntu 20.04)

Установить Git можно с помощью обычного менеджера пакетов вашего дистрибутива. Откройте терминал и введите подходящие команды:

```
sudo apt-get update
sudo apt-get install git
```

Команды для других версий Linux тут.

После того, как все действия по установке завершены, убедимся, что Git появился в системе компьютера. Откройте терминал и введите:

```
git --version
```

Установка Git на Windows 10

Для установки Git в Windows также имеется несколько способов. Официальная сборка доступна для скачивания на официальном сайте Git. Просто перейдите на страницу https://git-scm.com/download/win, и загрузка запустится автоматически. Дальше запустится Мастер установки и всё продолжится как при установки обычных Windows программ. Установщик спросит добавлять ли в меню проводника возможность запуска файлов с помощью Git Bash (консольная версия) и GUI (графическая версия). Подтвердите действие, чтобы далее вести работу через консоль в Git Bash. Остальные пункты можно оставить по умолчанию.

Для автоматической установки вы можете использовать пакет Git Chocolatey. Обратите внимание, что пакет Chocolatey поддерживается сообществом:

```
choco install git
```

Настройка Git

После того как Git появился на компьютере, нужно ввести свои данные, а именно имя и адрес электронной почты. Ваши действия в Git будут содержать эту информацию.

Откройте терминал и используйте следующую команду, чтобы добавить своё имя:

```
git config --global user.name "ваше имя"
```

Для добавления почтового адреса вводите:

```
git config --global user.email адрес_почты
```

Например:

```
git config --global user.name "pylounge"
git config --global user.email pylounge@mail.ru
```

Обратите внимание, что в командах, указанных выше, есть опция `--global`. Это значит, что такие данные будут сохранены для всех ваших действий в Git и вводить их больше не надо. Если вы хотите менять эту информацию для разных проектов, то в директории проекта вводите эти же команды, только без опции `--global.`

> Желательно, чтобы адрес электронной почти совпадал с адресом, указанным при регистрации на GitHub (аналогично и имя).

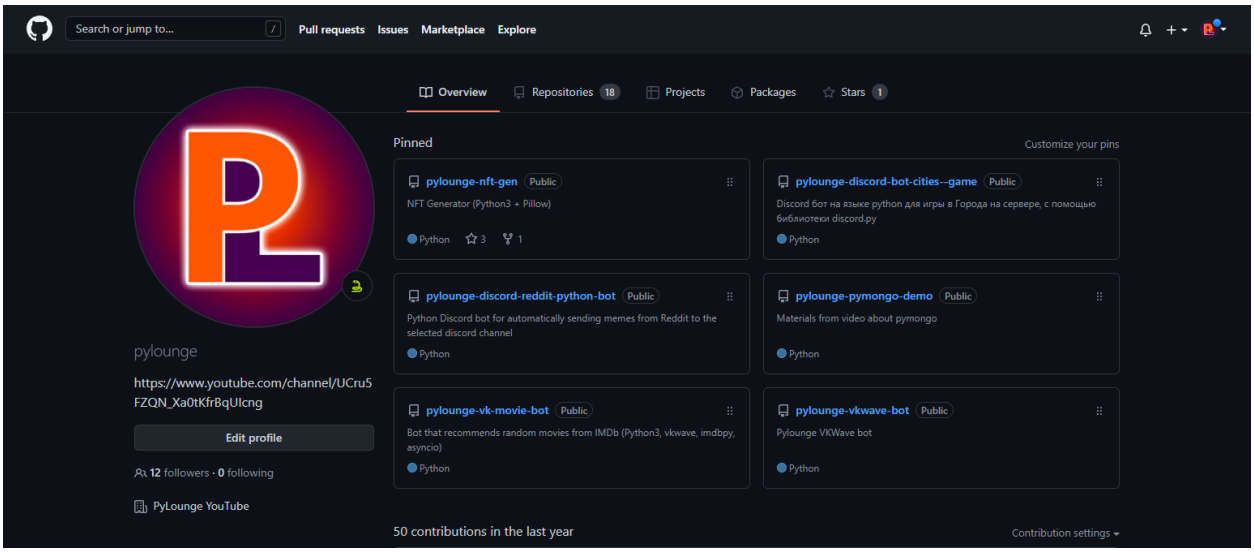
Теперь каждое наше действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто отвечает за какие изменения — это вносит порядок.

Регистрация на GitHub

GitHub — веб-сервис, который основан на системе Git. Это такая социальная сеть для разработчиков, которая помогает удобно вести коллективную разработку IT-проектов. Здесь можно публиковать и редактировать свой код, комментировать чужие наработки, следить за новостями других пользователей. Своего рода Инстаграм, только туда выкладывают не фотки и видео, а программный код. Собственно на GitHub его хранят и коллективно редактируют.

Чтобы начать работу с GitHub, нужно зарегистрироваться на сайте.

1. Переходим на сайт GitHub
2. Для начала регистрации:
   Нажимаем кнопку Sign up (зарегистрироваться), попадаем на страницу регистрации, где вводим обязательные данные: имя пользователя, адрес электронной почты и пароль. После заполнения полей проходим верификацию.
3. После верификации GitHub предложит создать новый репозиторий, организацию или узнать больше о GitHub. Этот пункт пока можно пропустить и перейти в профиль.



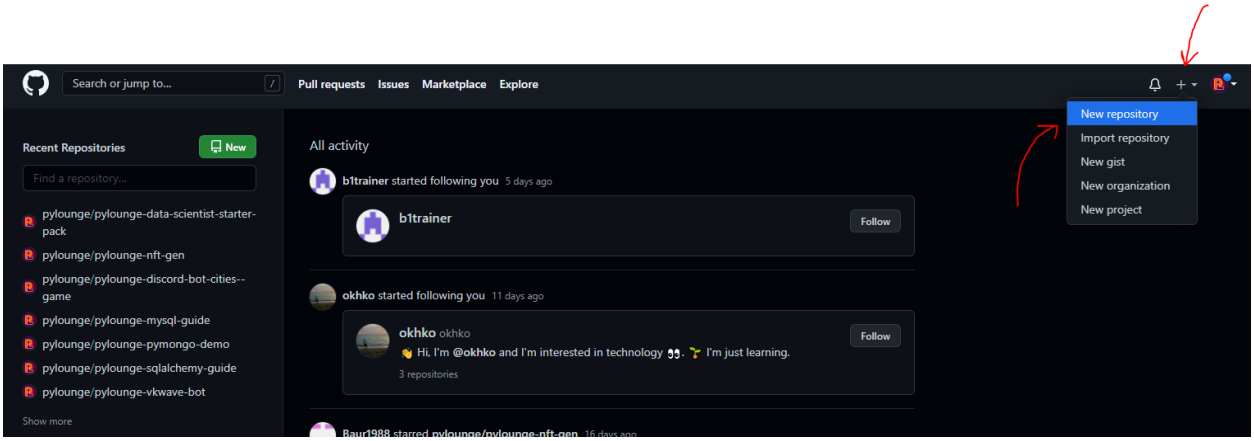
Профиль пользователя на GitHub

Создание репозитория на GitHub

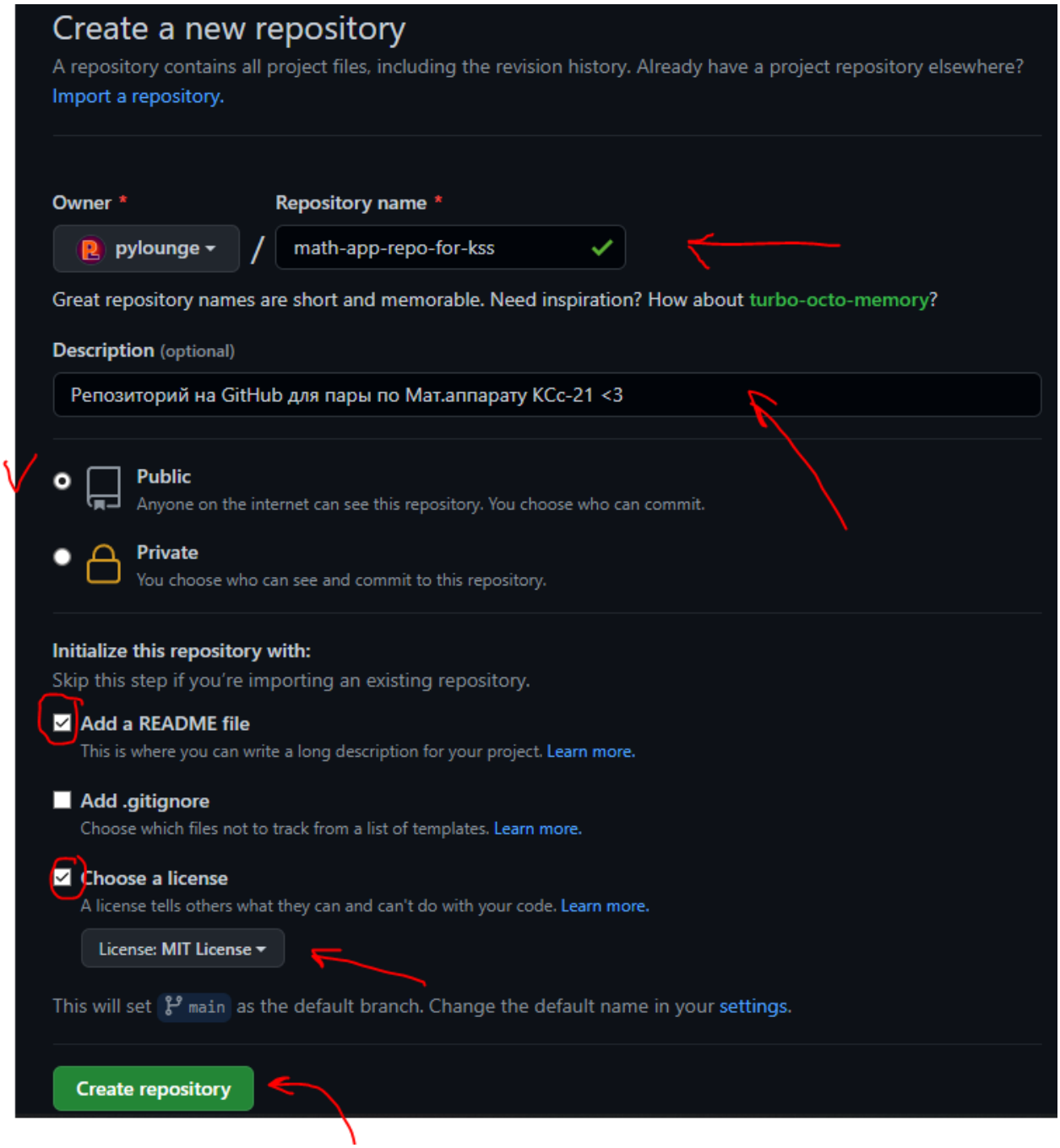
Репозиторий — это место, в котором вы систематизируете свой проект. Здесь вы храните файлы, папки, видео, изображения, блокноты Jupyter Notebook, наборы данных и т.д. Перед началом работы с Git необходимо инициализировать репозиторий для проекта и правильно его подготовить. Это можно сделать на сайте GitHub.

Лучше сразу добавлять в репозиторий **README**-файл с информацией о проекте. Это можно сделать в момент создания репозитория, поставив галочку в соответствующем поле.

- Перейдите на сайт GitHub. Нажмите на значок **+** в верхнем правом углу, а затем выберите **New repository**.
- Придумайте имя репозитория и добавьте короткое описание.
- Решите, будет ли этот репозиторий размещаться в открытом доступе или останется закрытым для просмотра.
- Нажмите **Initialize this repository with a README** для добавления README-файла. Настоятельно рекомендую снабжать все ваши проекты файлом-описанием, ведь README — это первая вещь, на которую люди обращают внимание при просмотре репозитория. К тому же, здесь можно разместить нужную информацию для понимания или запуска проекта.
- Также выбираем необходимую лицензию, которая покрывает код вашего проекта.



Создание нового репозитория



Настройка репозитория

При желании можете уже сейчас начинать работать над проектом. Добавляйте файлы, вносите в них изменения и т.д. напрямую с сайта GitHub.

Вносить изменения в проект можно двумя способами:

1. Вы можете изменять файлы/блокноты на компьютере (затем синхронизировать изменения с репозиторием на сайте).
2. Делать все изменения сразу на сайте GitHub.

Создание и работа с локальным репозиторием Git

Инициализация репозитория

Как мы отметили ранее, git **хранит свои файлы и историю прямо в папке проекта**. Чтобы создать новый репозиторий, нам нужно открыть терминал, зайти в папку нашего проекта и выполнить команду `init`. Это включит приложение в этой конкретной папке и создаст скрытую директорию *.git*, где будет храниться история репозитория и настройки.

Создайте папку своего проекта, например, pylounge_project и перейдите в неё. Для этого в окне терминала введите:

```
mkdir pylounge_project
cd pylounge_project
```

Инициализируем наш проект в качестве локального репозитория:

```
git init
```

После этого в папке проекта появилась папка .git. В неё будет писаться вся информация, все изменения, которые происходят с файлами и файлах этого проекта (своего рода журнал изменений или дневник).

Теперь создайте файл (пока что пустой) программы на языке Python под названием `app.py` и сохраните его в в папке проекта:

```
touch app.py
```

Состояние репозитория

status — это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нём, нет ли чего-то нового, что поменялось, и так далее. Посмотрим статус репозитория:

```
git status
```

В ответ получим нечто такое:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  app.py

nothing added to commit but untracked files present (use "git add" to track)
```

Сообщение говорит о том, что файл app.py ***неотслеживаемый***. Это значит, что файл **новый** и система еще не знает, ***нужно ли следить за изменениями в файле или его можно просто игнорировать***.

Для того, чтобы ***начать отслеживать новый файл***, нужно его специальным образом объявить (указать Git'у, что этот файл теперь полноценный файл проекта и его надо записать в "книжечку" - `.git`).

Каталог .git отслеживает файлы проекта

Подготовка файлов проекта для отслеживания

Сперва “книжечка” пустая, но затем мы добавляем в неё файлы (или части файлов, или даже одиночные строчки) командой `add` и, наконец, ***коммитим*** все нужное в репозиторий (создаем снимок нужного нам состояния) командой ***commit***.

В нашем случае у нас только один файл, так что добавим его в “книжечку”:

```
git add app.py
```

Если нам нужно добавить все, что лежит в папке проекта, мы можем использовать:

```
git add -A

# или

git add .
```

Снова смотрим статус:

```
git status
```

И на этот раз мы должны получить другой ответ:

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   app.py
```

Сообщение о состоянии говорит нам о том, какие изменения относительно файла/папки были проведены в проекте — в данном случае это появление нового файла ***(файлы ещё могут быть отредактированы или удалены)***.

Теперь файл под наблюдением и он готов к фиксации изменений - ***коммиту***.

Фиксация изменений (commit)

Коммит представляет собой состояние репозитория в определенный момент времени. Это похоже на снимок виртуальной машины, к которому мы можем вернуться и увидеть состояние объектов на определенный момент времени.

Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого мы может коммитить:

```
git commit -m "Добавил файл app.py"
```

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла app.py). Ключ ***-m*** и сообщение ***«Добавил файл app.py»*** — это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.

```
git status
```

```
On branch master
nothing to commit, working tree clean
```

Таким образом, мы не просто добавили файл в отслеживаемые, а зафиксировали, что в нашем проекте теперь появился файл app.py (состояние которого будет также отслеживаться в дальнейшем).

Внесение изменений в файл

Внесём следующие изменения в файл app.py:

```
nano app.py
```

```
import antigravity
```

Посмотрим статус:

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  modified:   app.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Видим, что в статусе указано, что файл app.py был модифицирован.

Добавляем и фиксируем изменения:

```
git add app.py
git commit -m "Написал код в app.py. Только не запускай файл!"
```

Проверим статус:

```
On branch master
nothing to commit, working tree clean
```

Отслеживание изменений, сделанных в коммитах (Логи)

У каждого коммита есть свой уникальный идентификатор в виде строки цифр и букв. Чтобы просмотреть список всех коммитов и их идентификаторов, можно использовать команду log:

```
git log
```

Получим такой вывод:

```
commit 50ce4a1ab0649c116ec1608f168b0aad8343a610 (HEAD -> master)
Author: pylounge <pylounge@mail.ru>
Date:   Thu Mar 3 10:07:02 2022 -0800

    Написал код в app.py. Только не запускай файл!

commit 1086245bb2d7c7a3eed91b3ffe0fa0b917d6cfb6
Author: pylounge <pylounge@mail.ru>
Date:   Thu Mar 3 09:57:00 2022 -0800

    Добавил файл app.py
```

Как вы можете заметить, идентификаторы довольно длинные, но для работы с ними не обязательно копировать их целиком — первых нескольких символов будет вполне достаточно. Чтобы посмотреть, что нового появилось в коммите, мы можем воспользоваться командой show:

```
git show 50ce4a1a
```

Здесь будут указаны все изменения, которые были зафиксированы эти коммитом:

```
Author: pylounge <pylounge@mail.ru>
Date:   Thu Mar 3 10:07:02 2022 -0800

    Написал код в app.py. Только не запускай файл!
```

```
diff --git a/app.py b/app.py
index e69de29..674b0b4 100644
--- a/app.py
+++ b/app.py
@@ -0,0 +1 @@
+import antigravity
```

Чтобы увидеть разницу между двумя коммитами, используется команда diff (с указанием промежутка между коммитами):

```
git diff 50ce4a1a..1086245b
```

Мы сравнили первый коммит с последним, чтобы увидеть все изменения, которые были когда-либо сделаны. Обычно проще использовать git difftool, так как эта команда запускает графический клиент, в котором наглядно сопоставляет все изменения.

```
diff --git a/app.py b/app.py
index 674b0b4..e69de29 100644
--- a/app.py
+++ b/app.py
@@ -1 +0,0 @@
-import antigravity
```

Возвращение файла к предыдущему состоянию

Git позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается командой ***checkout.*** Она используется для переключения между коммитами (это довольно распространенная ситуация для Гита - использование одной команды для различных, на первый взгляд, слабо связанных задач).

В следующем примере мы возьмем файл app.py и откатим все изменения, совершенные над ним к первому коммиту. Чтобы сделать это, мы подставим в команду идентификатор нужного коммита, а также путь до файла:

```
git checkout 1086245b app.py
```

Теперь заглянем в файл app.py и убедимся, что там пусто.

```
git status

On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  modified:   app.py
```

Если мы хотим сохранить изменения, которые мы откатили, то используем уже знакомую комбинацию add и commit:

```
git add .
git commit -m "Убрал код из app.py"
```

Отмена Git Add (необязательно)

Вы можете отменить действия над конкретным файлом или все внесенные изменения.

Решение, которое потребуется для этого, весьма простое. Чтобы откатить один файл, просто вызовите команду `git reset`:

```
git reset <file>
```

Для отмены всех изменений запустите следующее:

```
git reset
```

Необходимость в откате становится неизбежной, если вы сделали коммит слишком рано и забыли добавить еще несколько файлов. В подобных случаях используется команда `amend`, чтобы внести изменения, зафиксировать их и сделать коммит снова.

Исправление коммита (необязательно)

Если вы опечатались в комментарии или забыли добавить файл и заметили это сразу после того, как закоммитили изменения, вы легко можете это поправить при помощи commit —amend. Эта команда добавит все из последнего коммита в область подготовленных файлов и попытается сделать новый коммит. Это дает вам возможность поправить комментарий или добавить недостающие файлы в область подготовленных файлов.Для более сложных исправлений, например, не в последнем коммите или если вы успели отправить изменения на сервер, нужно использовать revert. Эта команда создаст коммит, отменяющий изменения, совершенные в коммите с заданным идентификатором. Самый последний коммит может быть доступен по псевдониму (алиасу) HEAD:

```
$ git revert HEAD
```

Для остальных будем использовать идентификаторы:

```
$ git revert 1086245b
```

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строчки, состояние которых нужно откатить, так как они больше не существуют.

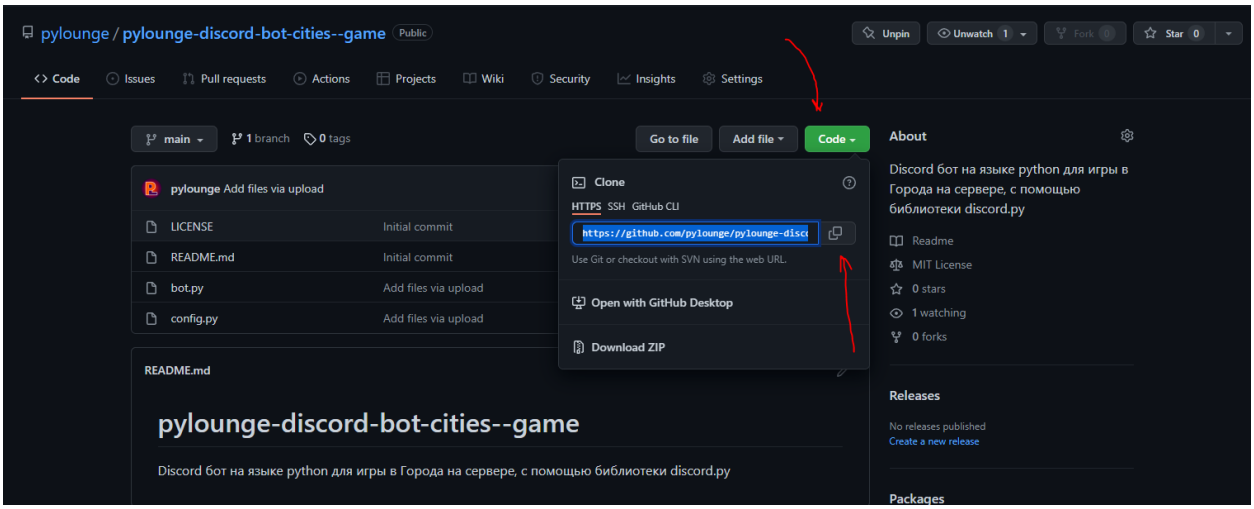
Подключение к репозиторию на GitHub

Сейчас наш коммит является локальным — существует только в папке .git на нашем компьютере. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случаев мы хотим поделиться нашей работой или доставить код на сервер, где он будет выполняться.

Давайте загрузим наш проект со всеми изменениями на созданный GitHub-репозиторий.

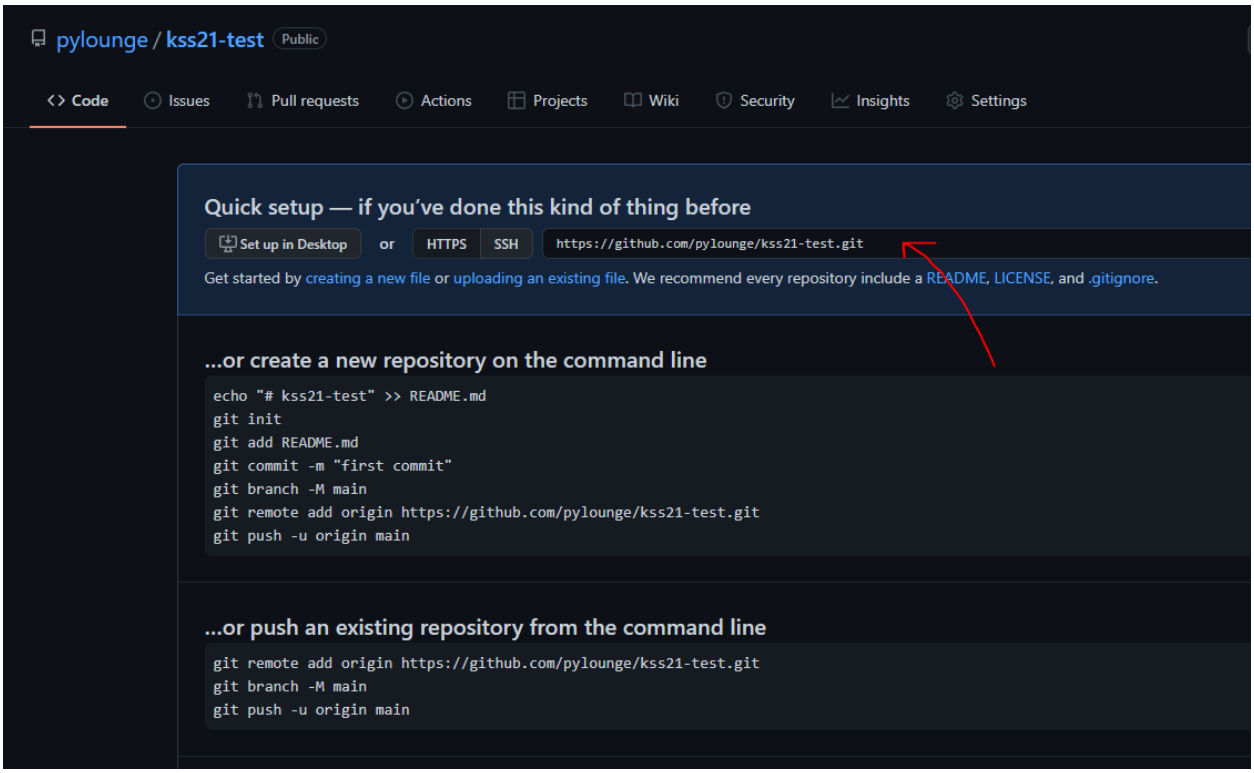
Для этого нужно к нему подключиться. Нам понадобиться URL-адрес репозитория. Переходим на страницу созданного репозитория.

Если в репозитории создан файл README, то нажимаем кнопку ***Code*** и копируем ссылку из раздела HTTPS:



Получение URL-адреса репозитория для доступа через протокол HTTPS

Если же README не создан, то на странице репозитория сразу можно взять ссылку:

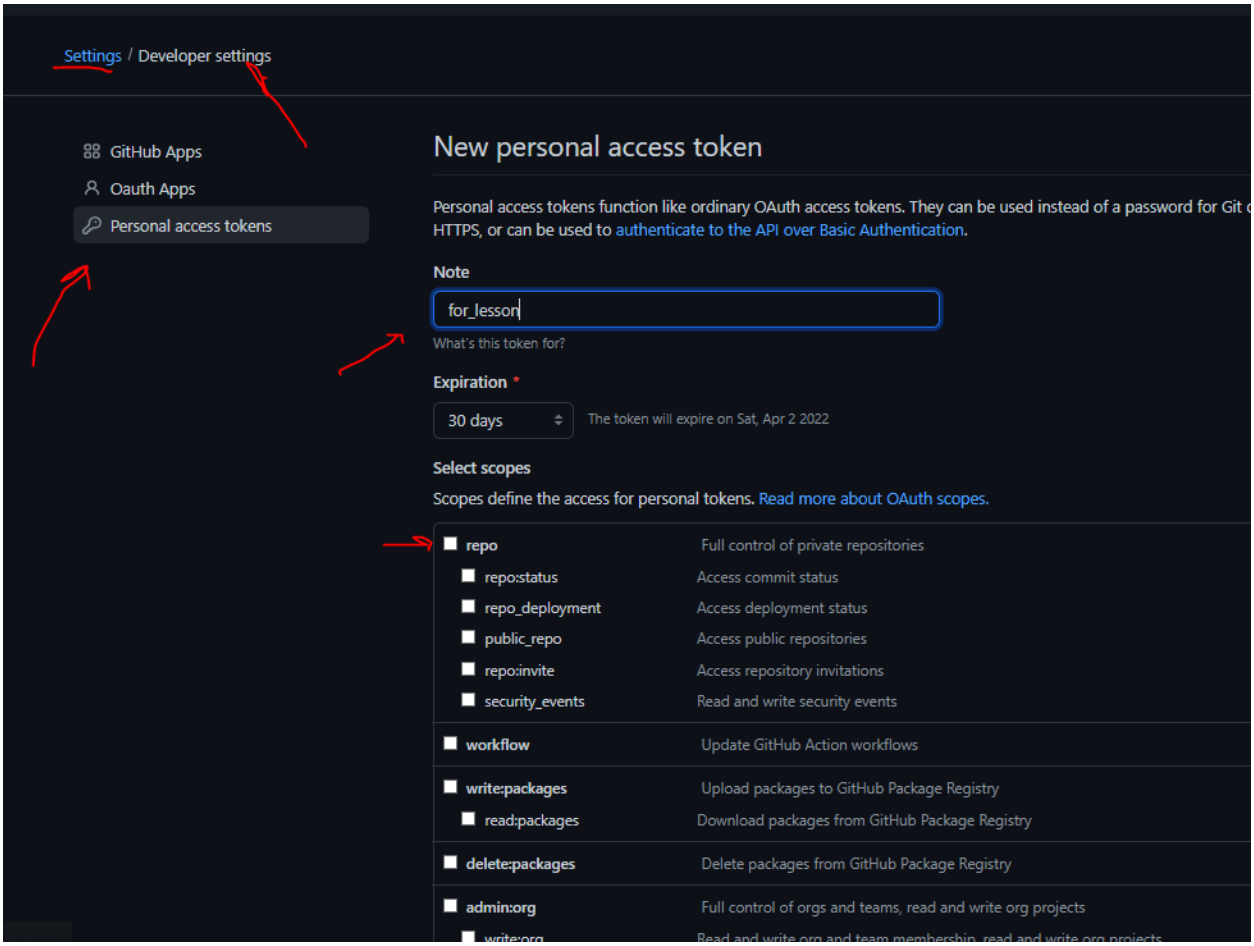


Получение URL-адреса репозитория для доступа через протокол HTTPS (2)

Для авторизации на GitHub нам необходимо ***сгенерировать токен доступа***.
Для этого заходим в ***Settings → Developer settings → Personal access tokens →***

Generate new token.

Вводим имя токена, отмечаем галочками необходимые права доступа и генерируем токен:



В общем случае для связи с удалённым репозиторием используется команда следующего вида:

```
git remote add origin <https://<token>@github.com/><username>/<repo>
```

Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория на свой:

```
git remote add origin https://0NKkKPyWrDZaqcLn@github.com/pylounge/kss21-test.git
```

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется ***origin***.

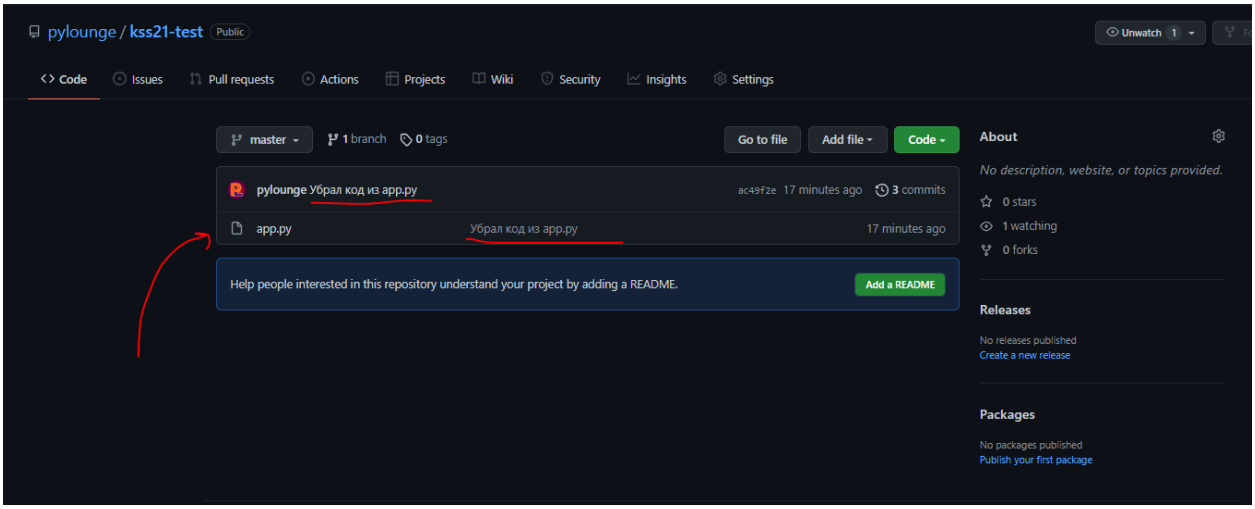
Отправка изменений на сервер GitHub

Сейчас самое время переслать наш локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим обновить данные в удаленном репозитории.

Команда, предназначенная для этого - `push`. Она принимает два параметра: имя удаленного репозитория (мы назвали наш ***origin***) и ветку, в которую необходимо внести изменения (***master/main*** — это ветка по умолчанию для всех репозиториев).

```
git push origin master
# или
git push origin main
```

В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправились. Если все сделано правильно, то когда вы посмотрите в удаленный репозиторий при помощи браузера, вы увидите файл app.py



Запрос изменений с GitHub

Предположим кто-то внёс изменения в файл app.py, который лежит в нашем GitHub репозитории. В таком случае мы и другие пользователи могут скачать изменения при помощи команды pull и тем самым обновить наш локальный проект на компьютере до актуального состояния:

```
git pull origin master
```

И если мы теперь заглянем внутрь файла app.py:

```
nano app.py

#или

cat app.py
```

То увидим следующее содержимое:

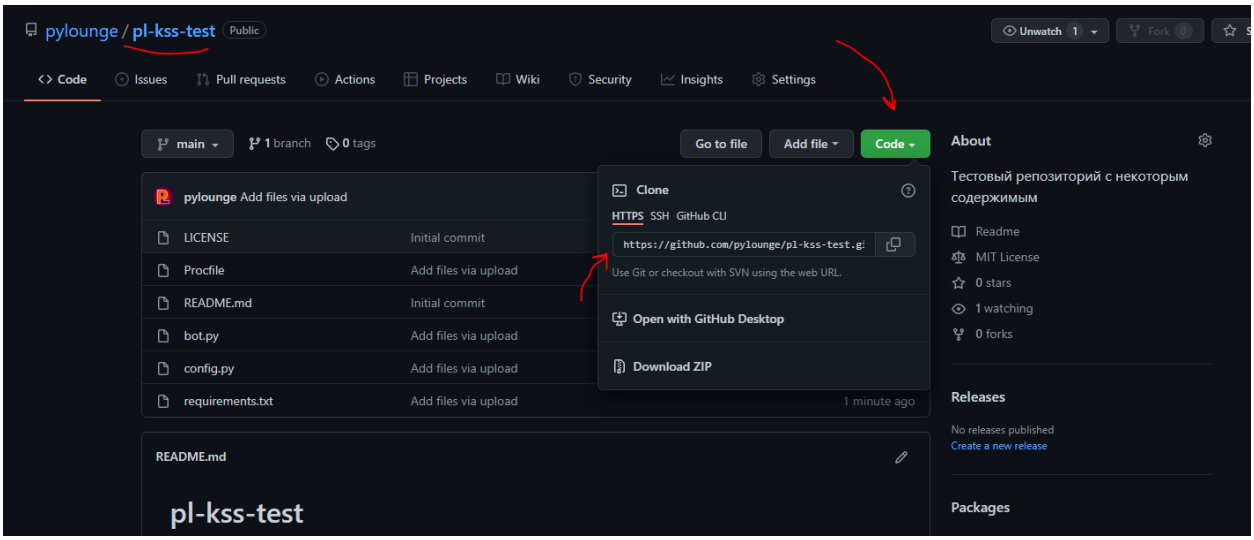
```
print('Hello from PyLounge!')
```

Создание и работа с удалённым репозиторием Git

Клонирование чужого/своего репозитория

Теперь другие пользователи GitHub могут просматривать ваш репозиторий. Они (и вы) могут скачать из него данные и получить полностью работоспособную копию вашего проекта при помощи команды ***clone***.

Аналогичным образом получаем ссылку на интересующий на репозиторий:



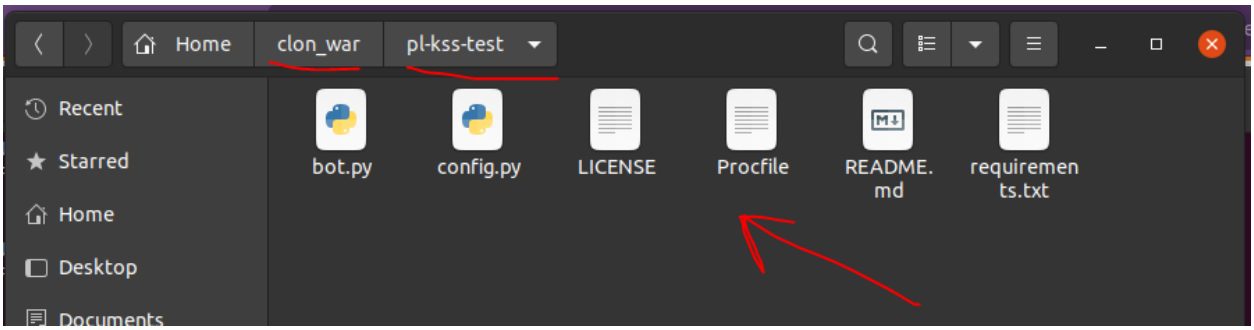
Создадим папку, куда будут загружаться файлы проекта из выбранного репозитория и перейдём в неё:

```
cd ..
mkdir clon_war
cd clon_war
```

Затем выполним команду clone:

```
git clone https://github.com/pylounge/pl-kss-test.git
```

Новый локальный репозиторий создается автоматически с GitHub в качестве удаленного репозитория и внутри папки clon_war появится папка с проектом pl-kss-test со всеми файлами:



Теперь у нас есть копия чужого проекта. Мы можем использовать файлы проекта, вносить изменения и (или) пересохранить себе.

Ветки (branches)

Допустим мы решили доработать клонированный нами проект. Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

- Уже рабочая, стабильная версия кода сохраняется.
- Различные новые функции могут разрабатываться параллельно разными программистами.
- Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
- В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

Создание новой ветки

Основная ветка в каждом репозитории называется master. Чтобы создать еще одну ветку, используем команду ***branch*** <name>

```
git branch my_super_branch
```

Это создаст новую ветку, пока что точную копию ветки master.

Переключение между ветками

Сейчас, если мы запустим branch, мы увидим две доступные опции:

```
git branch
```

```
* main
  my_super_branch
```

master — это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей ***новой суперветкой***, так что нам понадобится переключиться на другую ветку. Для этого воспользуемся командой ***checkout***, она принимает один параметр — имя ветки, на которую необходимо переключиться.

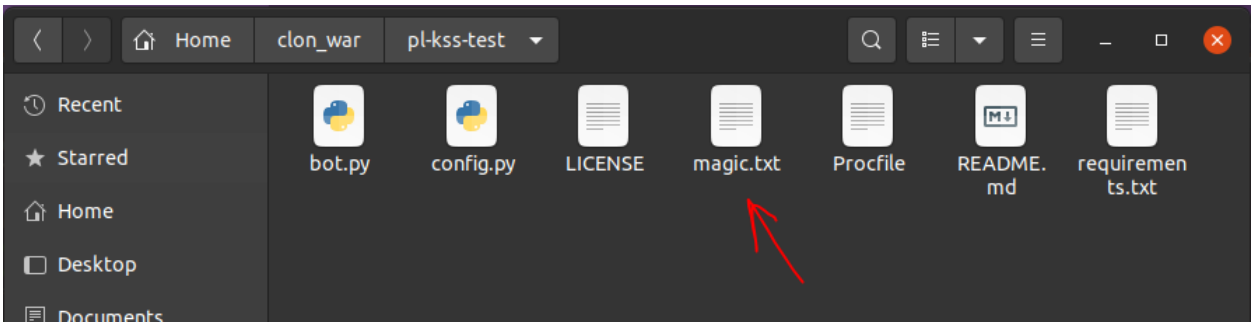
```
git checkout my_super_branch
```

Теперь мы работает с веткой my_super_branch.

Слияние веток

Добавим в нашу ветку текстовым файлом под названием *magic.txt*. Мы создадим его, добавим и закоммитим:

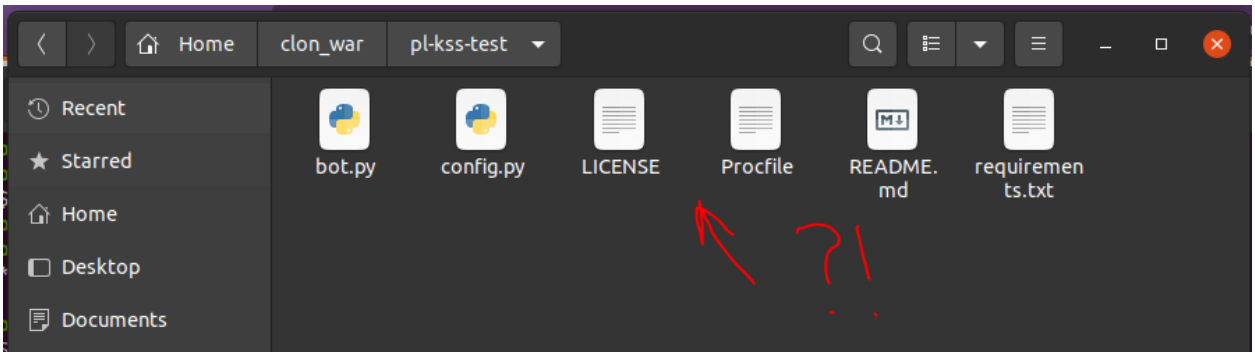
```
touch magic.txt

git add magic.txt
git commit -m "Oy my"
```



Изменения завершены, теперь мы можем переключиться обратно на ветку master.

```
git checkout main
```

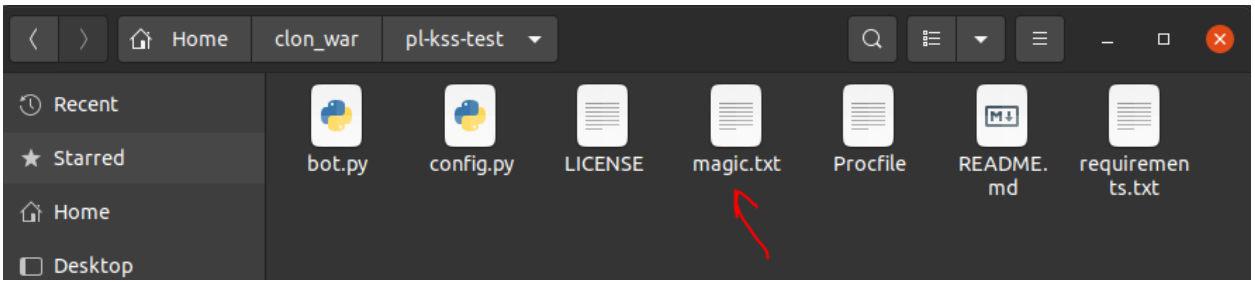
Теперь, если мы откроем наш проект в файловом менеджере, мы не увидим файла magic.txt, потому что мы переключились обратно на ветку master, в которой такого файла не существует.



Чтобы он появился, нужно воспользоваться ***merge*** для объединения веток (применения изменений из ветки my_super_branch к основной версии проекта).

```
git merge my_super_branch
```

Теперь ветка master актуальна.

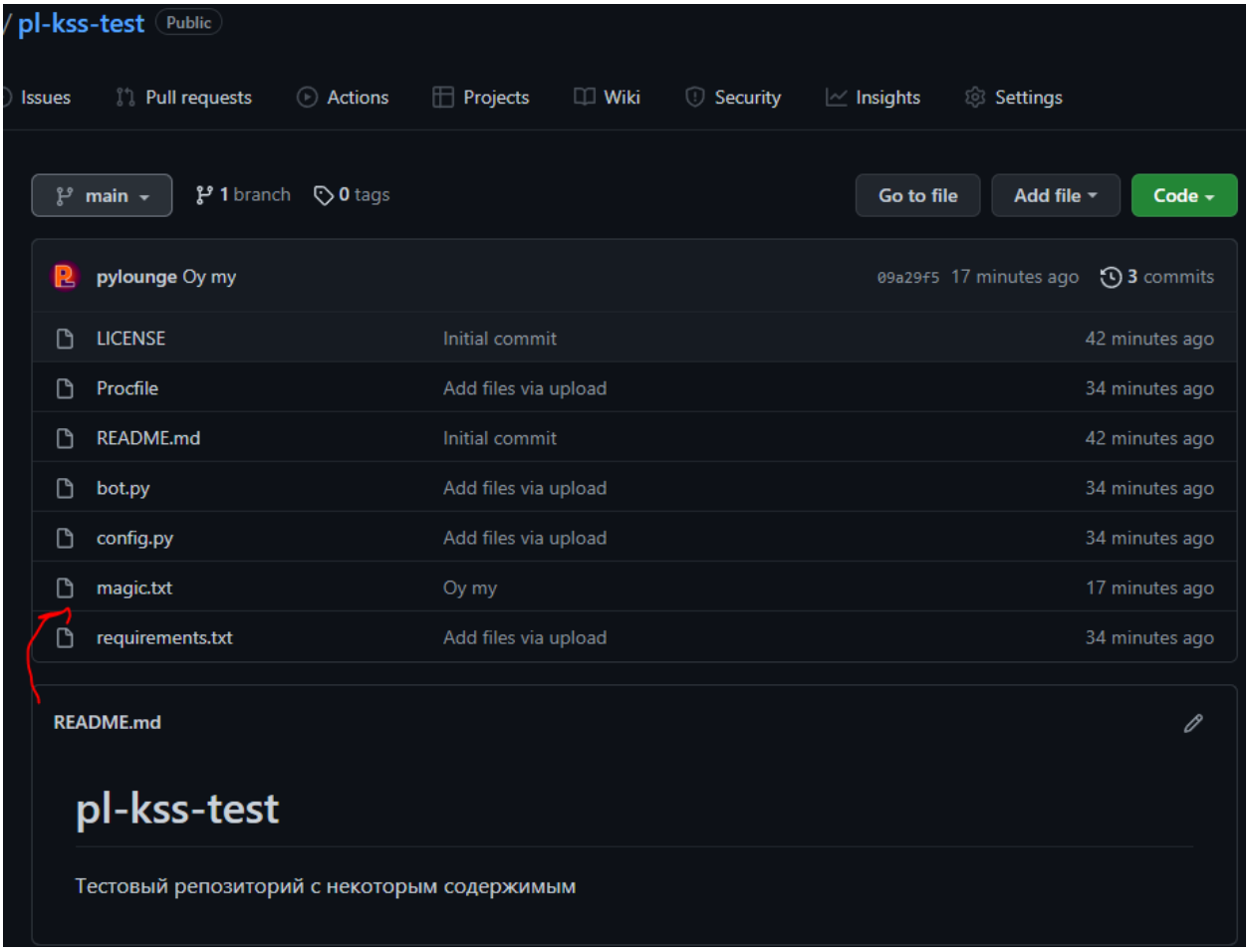


Ветка my_super_branch больше не нужна, и ее можно удалить.

```
git branch -d my_super_branch
```

Если мы имеем доступ к репозиторию, то можем сразу же добавить все изменения на GitHub:

```
git remote add my_target https://cLnpmRgv3F2peq@github.com/pylounge/pl-kss-test.git
git push my_target
```



Pull request

Допустим, мы хотим помочь кому-то с разработкой проекта. Для этого нам надо клонировать репозиторий его проекта, внести свои изменения и отправить автору запрос, чтобы он добавил наши изменения к себе в проект на GitHub. Для этого используется понятие Fork и Pull Request.

Когда нам нравится чей-то репозиторий и мы хотели бы иметь его в собственном аккаунте на GitHub, мы делаем ***форк («вилку»)*** этого репозитория, чтобы иметь возможность работать с ним отдельно.

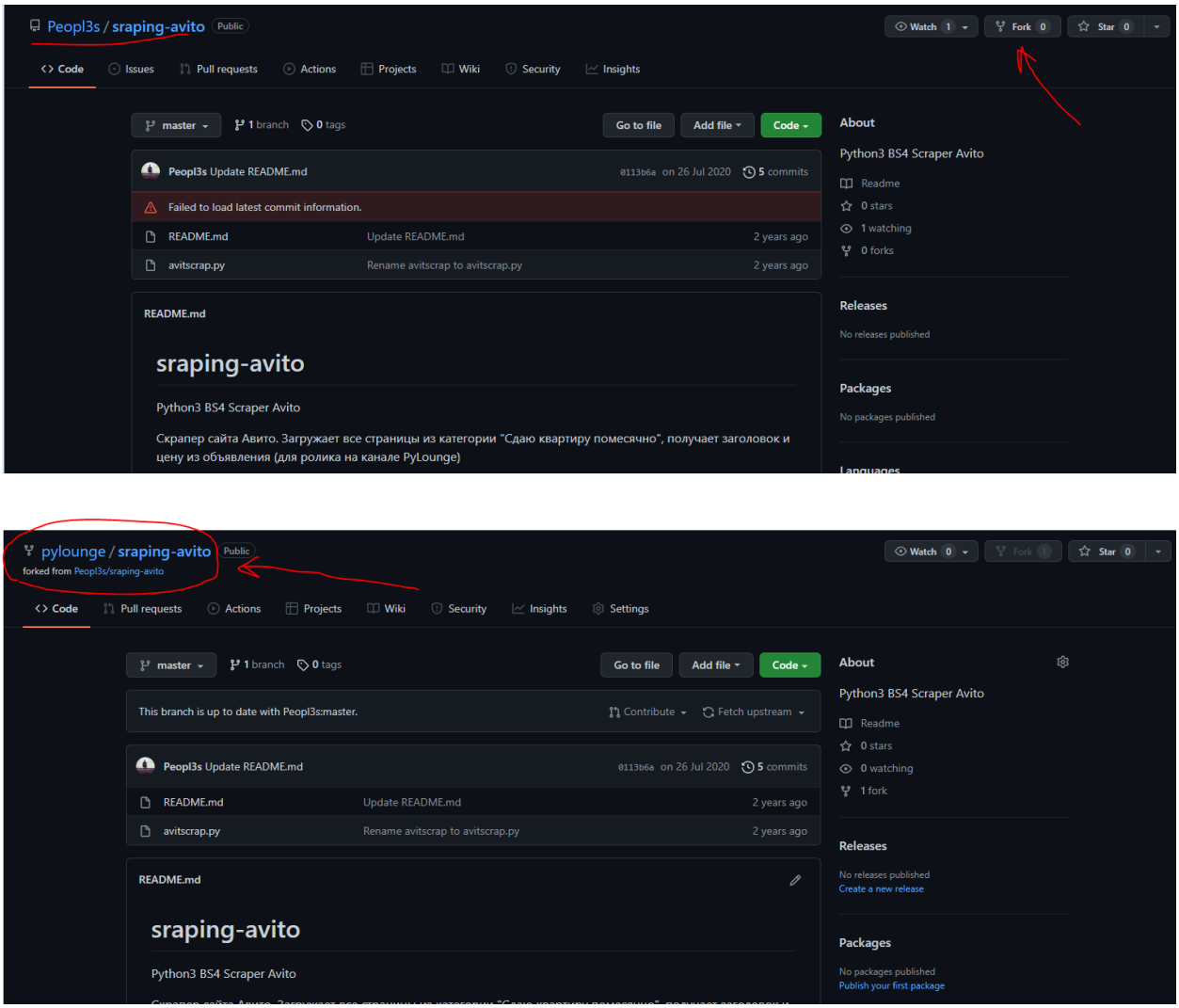
Когда мы делаем форк репозитория, мы получаем экземпляр всего репозитория, со всей его историей. После форка мы можем делать с ним все, что угодно: оригинальная версия при этом не будет задета.

Пул-реквесты нужны. Когда мы хотим принять участие в групповой разработке проектов с открытым исходным кодом.

Например, пользователь Павел делает форк репозитория ThanoshanMV и вносит изменения в свой экземпляр. После этого Павел отсылает пул-реквест ThanoshanMV, который может либо принять его, либо отклонить. По сути это что-то вроде письма «Не будете ли вы так любезны, уважаемый ThanoshanMV, внести мои изменения в свой оригинальный репозиторий?»

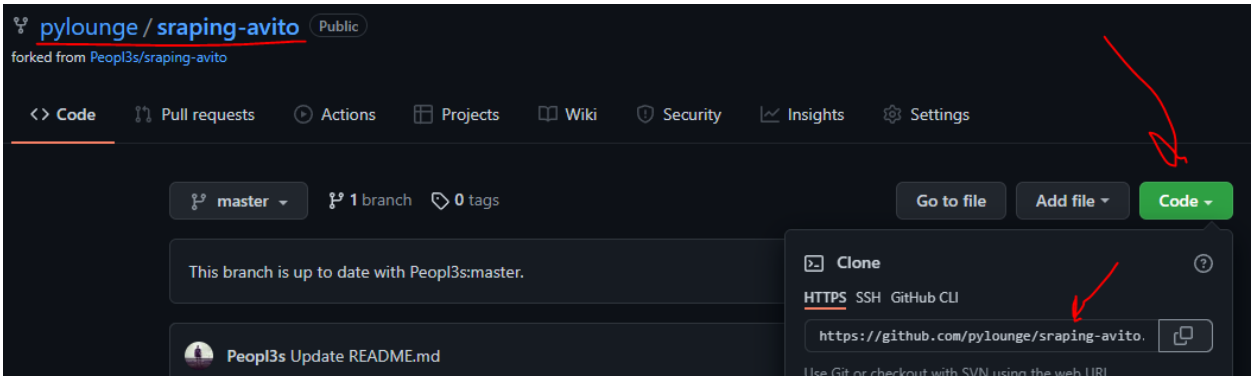
Таким образом, переходим в репозиторий интересующего нас проекта на GitHub, нажмите кнопку «Fork», чтобы заполучить собственную копию проекта.

Мы зарегистрированы на GitHub под именем «pylounge», так что наша копия окажется по адресу `https://github.com/pylounge/project` и отобразится у нас на GitHub аккаунте, где мы сможем редактировать её.



Форк проекта

Мы клонируем “нашу копию” проекта себе на компьютер в отдельную папку:

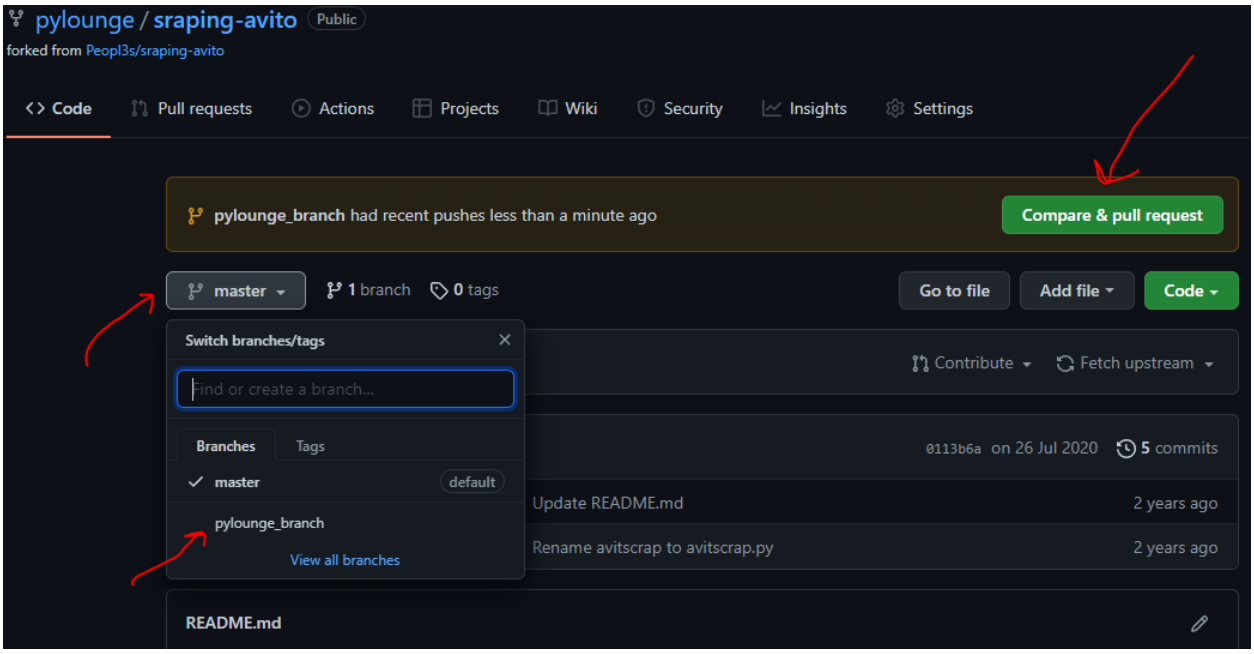



```
mkdir max_project
cd max_project

git clone https://github.com/pylounge/sraping-avito.git
cd sraping-avito/
```

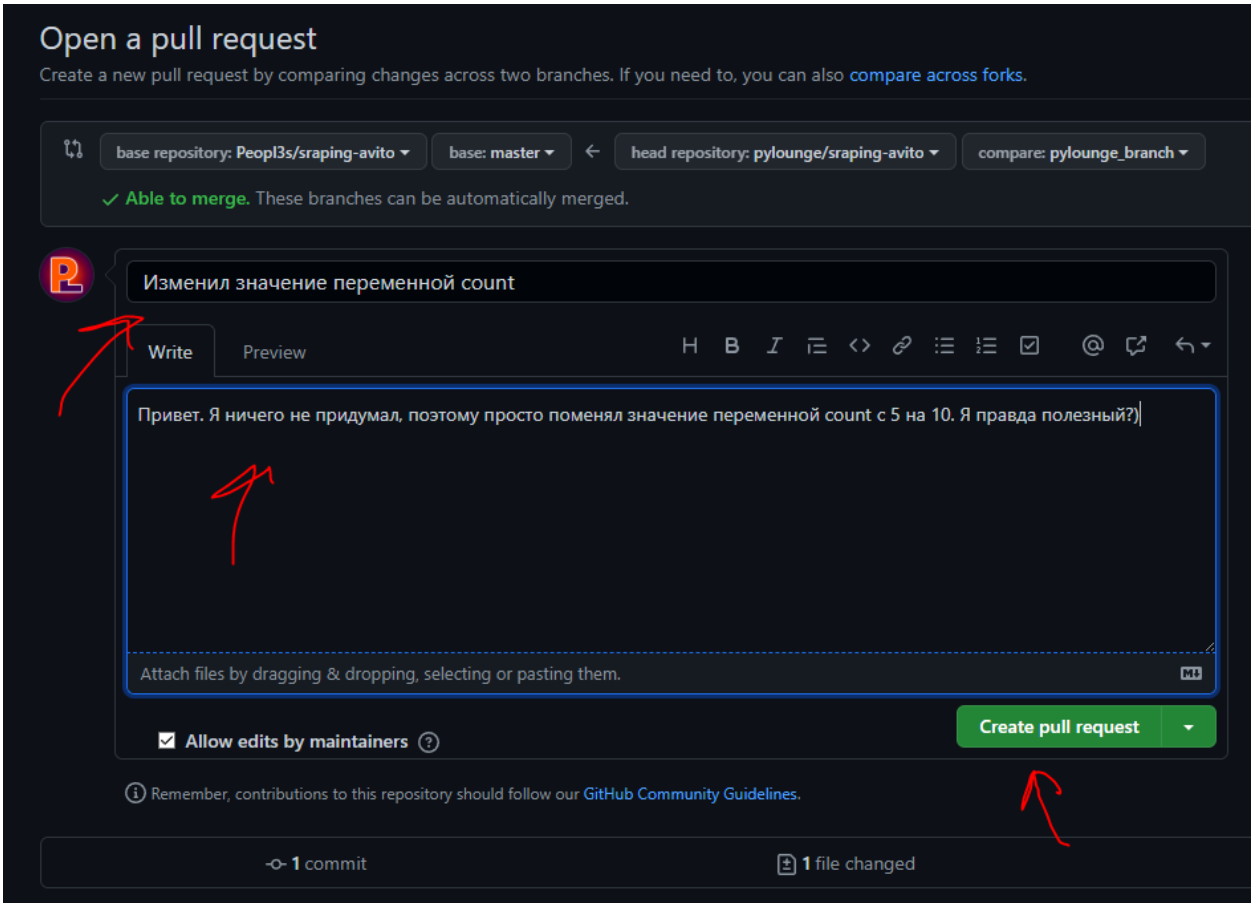
Создадим тематическую ветку, внесём необходимые изменения и, наконец, отправим их на GitHub:

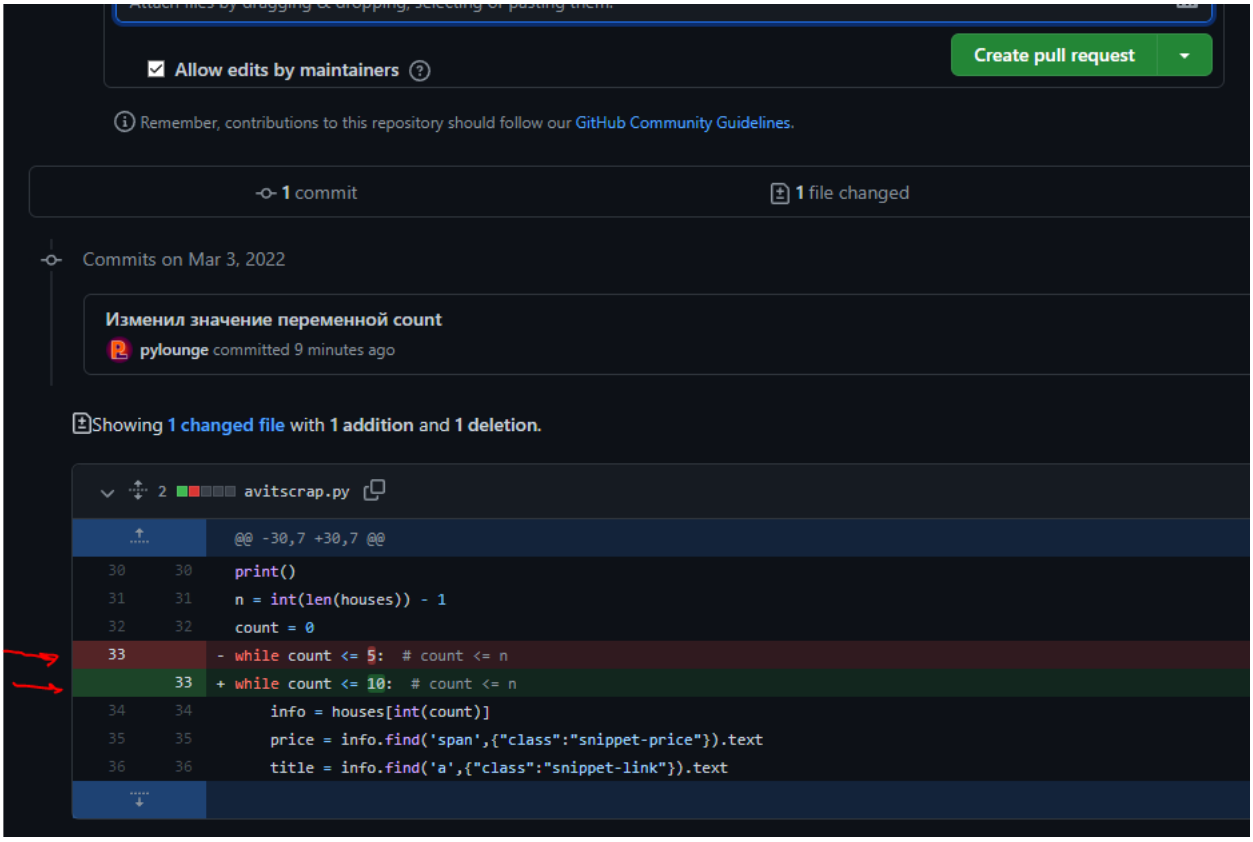
```
git checkout -b pylounge_branch

nano avito.py
# вносим каакие-то изменения ...

git add .
git commit -a -m 'Изменил значение переменной count'

git remote add my_clone_project https://v3F2peq@github.com/pylounge/sraping-avito.git

git push my_clone_project pylounge_branch
```

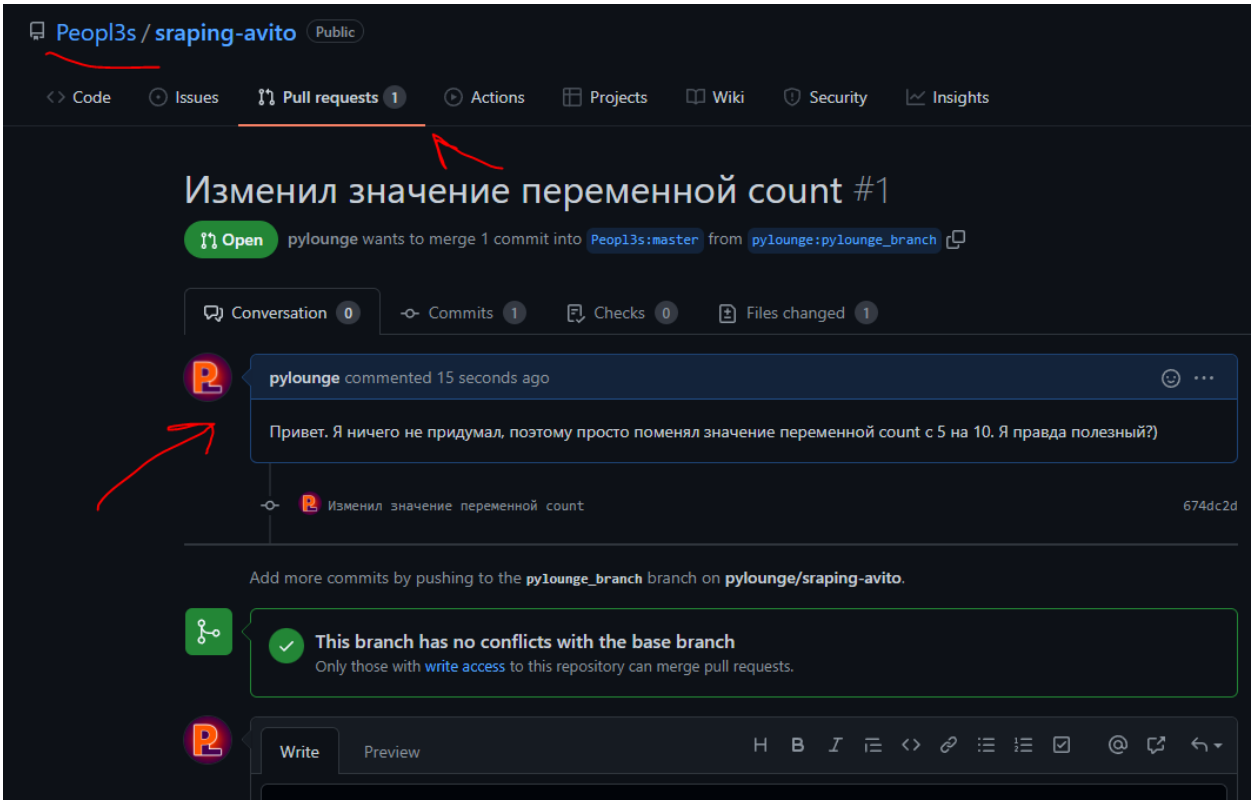
Теперь в нашем репозитории появилась вторая ветка:



Перейдите в свой репозиторий на GitHub. Там есть кнопка «Compare & pull request» — кликните ее. Откроется форма запроса на изменение. Введём необходимые детали относительно того, что именно вы сделали (чтобы поставить ссылку на issues, воспользуйтесь знаком «решетки»). После этого можно нажать кнопку подтверждения внизу.

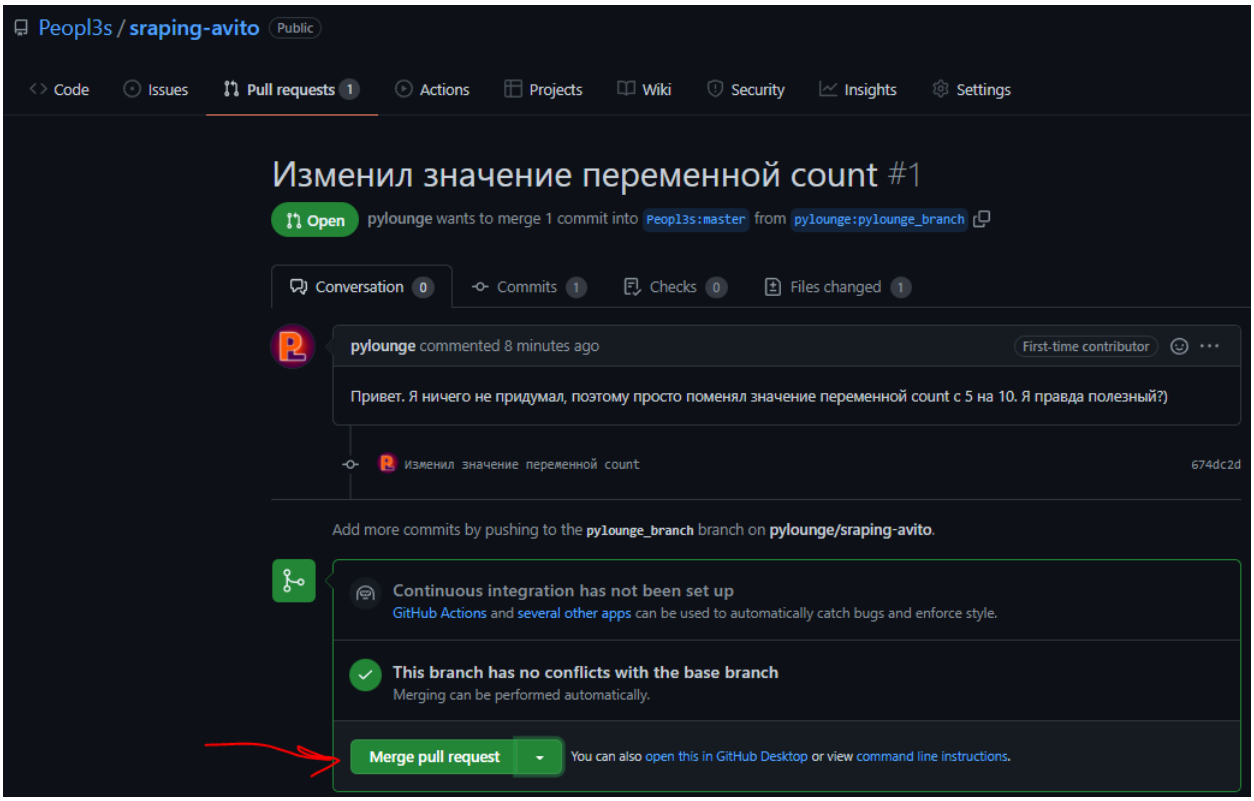

Open a pull request
Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.
base repository: Peopl3s/sraping-avito
base: master
head repository: pylounge/sraping-avito
compare: pylounge_branch
Able to merge. These branches can be automatically merged.
Изменил значение переменной count
Write
Preview
Привет. Я ничего не придумал, поэтому просто поменял значение переменной count с 5 на 10. Я правда полезный?)
Attach files by dragging & dropping, selecting or pasting them.
Allow edits by maintainers
Create pull request
Remember, contributions to this repository should follow our GitHub Community Guidelines.
1 commit
1 file changed


Allow edits by maintainers
Create pull request
Remember, contributions to this repository should follow our GitHub Community Guidelines.
1 commit
1 file changed
Commits on Mar 3, 2022
Изменил значение переменной count
pylounge committed 9 minutes ago
Showing 1 changed file with 1 addition and 1 deletion.
2 avitscrap.py
@@ -30,7 +30,7 @@
print()
n = int(len(houses)) - 1
count = 0
- while count <= 5: # count <= n
+ while count <= 10: # count <= n
info = houses[int(count)]
price = info.find('span',{"class":"snippet-price"}).text
title = info.find('a',{"class":"snippet-link"}).text


Peopl3s / sraping-avito
Public
Code
Issues
Pull requests 1
Actions
Projects
Wiki
Security
Insights
Изменил значение переменной count #1
Open
pylounge wants to merge 1 commit into Peopl3s:master from pylounge:pylounge_branch
Conversation 0
Commits 1
Checks 0
Files changed 1
pylounge commented 15 seconds ago
Привет. Я ничего не придумал, поэтому просто поменял значение переменной count с 5 на 10. Я правда полезный?)
Изменил значение переменной count
674dc2d
Add more commits by pushing to the pylounge_branch branch on pylounge/sraping-avito.
This branch has no conflicts with the base branch
Only those with write access to this repository can merge pull requests.
Write
Preview

Мы создали свой первый пул-реквест. Если автор его примет, вы получите уведомление по электронной почте.

Чтобы принять и “слить” изменения автор исходного репозитория должен нажать нажать ***Merge pull request***.



Git для VS Code

https://www.digitalocean.com/community/tutorials/how-to-use-git-integration-in-visual-studio-code-ru

Заключение

Полезные материалы

Список использованных источников

!! https://proglib.io/p/git-for-half-an-hour

!!! https://techrocks.ru/2021/04/04/how-to-use-git-part-1/

!!! https://medium.com/nuances-of-programming/знакомство-с-git-и-github-руководство-для-начинающих-54ea2567d76c

!!! https://htmlacademy.ru/blog/boost/frontend/git-console