

Скачайте Яндекс Браузер для образования

[Скачать](#)[< Урок Репозитории](#)

## Введение в репозитории. Подключение в PyCharm. Работа с удаленным репозиторием

- 1 Введение
- 2 История Git
- 3 Установка и настройка Git
- 4 Основы локального использования Git
- 5 Создание локального репозитория
- 6 Отслеживание версий файлов
- 7 Начало работы с удаленным репозиторием
- 8 Создание аккаунта на GitHub
- 9 Синхронизация с сетевыми репозиториями
- 10 Клонирование сетевого репозитория



### Аннотация

В этом уроке рассматриваются цели систем контроля версий, принципы работы с ними и производится разбор системы Git. Научимся создавать локальные репозитории, работать с версиями, подключать удаленный репозиторий, клонировать удаленный репозиторий.

## 1. Введение

На предыдущих занятиях вы создали первый проект на языке программирования Python с использованием библиотеки PyQT, теперь нас ждет проект по созданию своей собственной игры. Проект на PyQT был достаточно большим и потребовал от вас значительных усилий, но разработка игр, даже самых простых, имеет свои особенности. Поэтому на текущем уроке мы уйдем в сторону от особенностей языка, алгоритмов и библиотек, и погрузимся в отдельную важную тему. То, что мы будем рассматривать, применяется в любой области разработки, независимо от языка и стека технологий. Сегодня мы поговорим про **Системы контроля версий** (Version Control Systems или VCS).

Представьте: вы работаете над новым хобби-проектом — программой, выводящей на экран изображение снежного человека. Как сделать процесс удобным и максимально продуктивным?

Когда появляются новые идеи, хочется быстрее их проверить, изменить код. Но важно при этом сохранить и промежуточные результаты, чтобы затем, если понадобится, к ним вернуться. Например, чтобы сравнить версии и выбрать более удачную.

В определенный момент становится ясно, какой должна быть финальная версия программы. Дальше работа продолжается вместе с другом из Лицея — и с ним надо поделиться исходным кодом.

Когда каждый из вас захочет поэкспериментировать с результатом со своего устройства, у обоих должен быть доступ к данным.

Потом, когда оба поработали отдельно, получившееся надо объединить в одно целое. И, конечно, совсем не хочется, чтобы все усилия пропали из-за сломавшейся техники.

Тогда на помощь приходят системы контроля версий. Что они делают:

1. Сохраняют код при поломках
2. Хранят много версий кода программы и позволяют легко переключаться между ними
3. Помогают разработчикам обмениваться кодом и редактировать один и тот же код с разных устройств
4. Объединяют результаты труда нескольких разработчиков

Чаще всего используют системы контроля версий Git, Mercurial (ее еще называют Hg) и несколько устаревшие Subversion (она же SVN) и CVS. У каждой из них есть свои плюсы и минусы, мы же подробнее остановимся только на одной из них — на системе Git.

Давайте посмотрим, из чего состоит и как работает эта система.

## 2. История Git

Жили-были разработчики ядра ОС Linux. Проект в 2005 году был хоть и значительно меньше по объему, чем сейчас, но тоже более чем внушительный. Разумеется, разработчики использовали систему контроля версий. Тогда это была проприетарная система **BitKeeper**. В какой-то момент из-за лицензионных разногласий дальнейшее использование BitKeeper стало невозможным, а ни одна из открытых систем контроля версий в то время не позволяла тысячам программистов удобно работать над одним проектом. Не осталось выбора кроме как меньше, чем за неделю, создать прототип Git. Первая версия появилась 3 апреля 2005 года, и с тех пор система часто менялась.

Сайт системы — <https://git-scm.com/>.

## 3. Установка и настройка Git

Для начала — кратко об установке.

Git создавался в первую очередь для операционной системы Linux, но без проблем подходит как для macOS, так и для ОС Windows.

Для установки Git в ОС Linux добавьте в систему пакет Git. Для семейства debian/ubuntu выполните в командной строке команду:

```
> sudo apt install git
```

В macOS ситуация похожая. Выполните команду:

```
> brew install git
```

А вот для установки Git для ОС Windows скачайте сборку с **сайта системы** и следуйте документации.

Многие программисты пользуются Git через интерфейс командной строки, но мы с вами рассмотрим вариант работы с системой контроля версий через графический интерфейс. Работа через командную строку более гибкая и позволяет реализовывать некоторые очень сложные сценарии, но нам для наших проектов будет достаточно функций, которые предоставляются графическими обертками для Git.

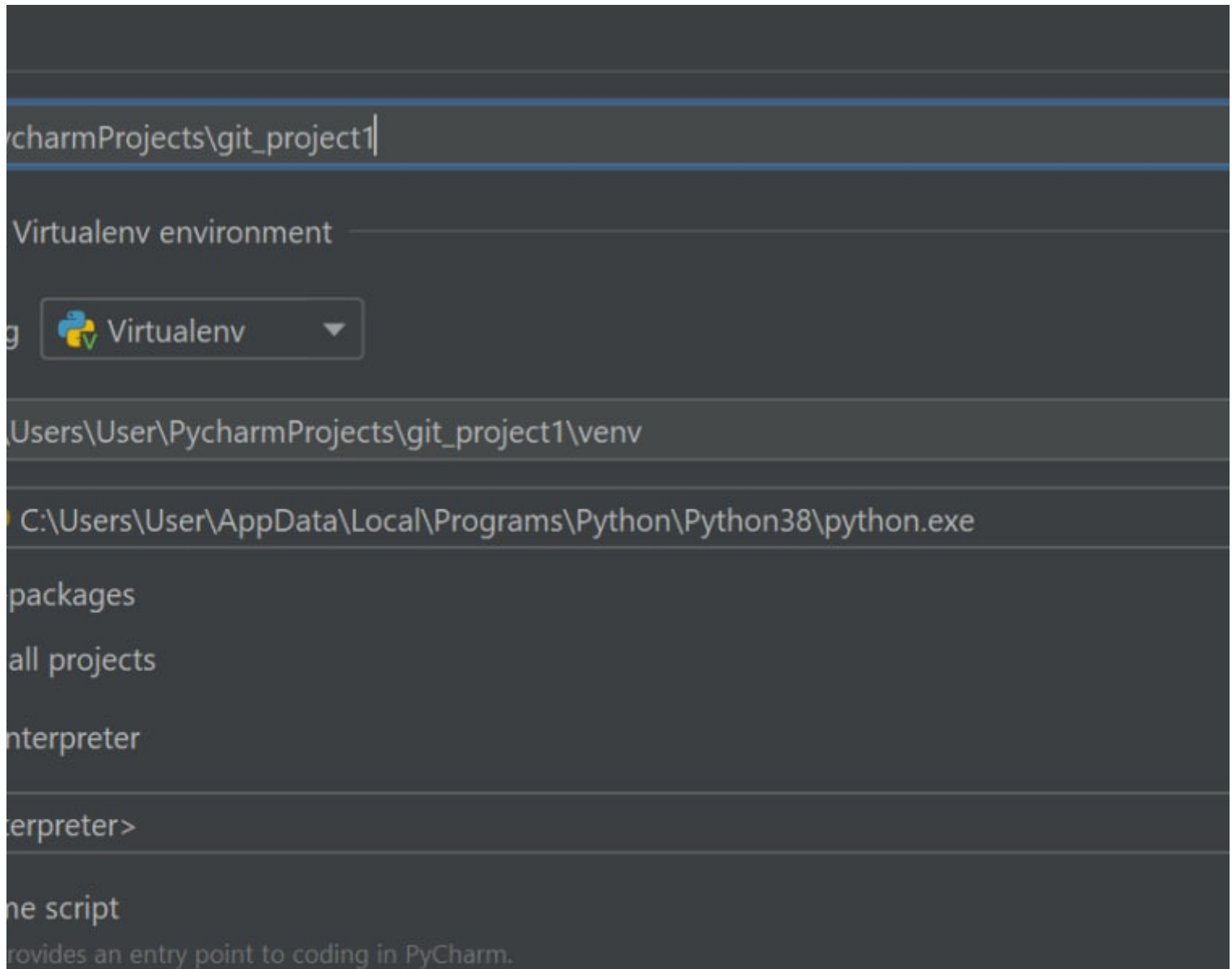
## 4. Основы локального использования Git

Теперь мы готовы изучать Git. На первом занятии мы поработаем с системой из IDE PyCharm. Как работать с Git с использованием командной строки, вы сможете узнать из дополнительных материалов.

Итак, запустите PyCharm.

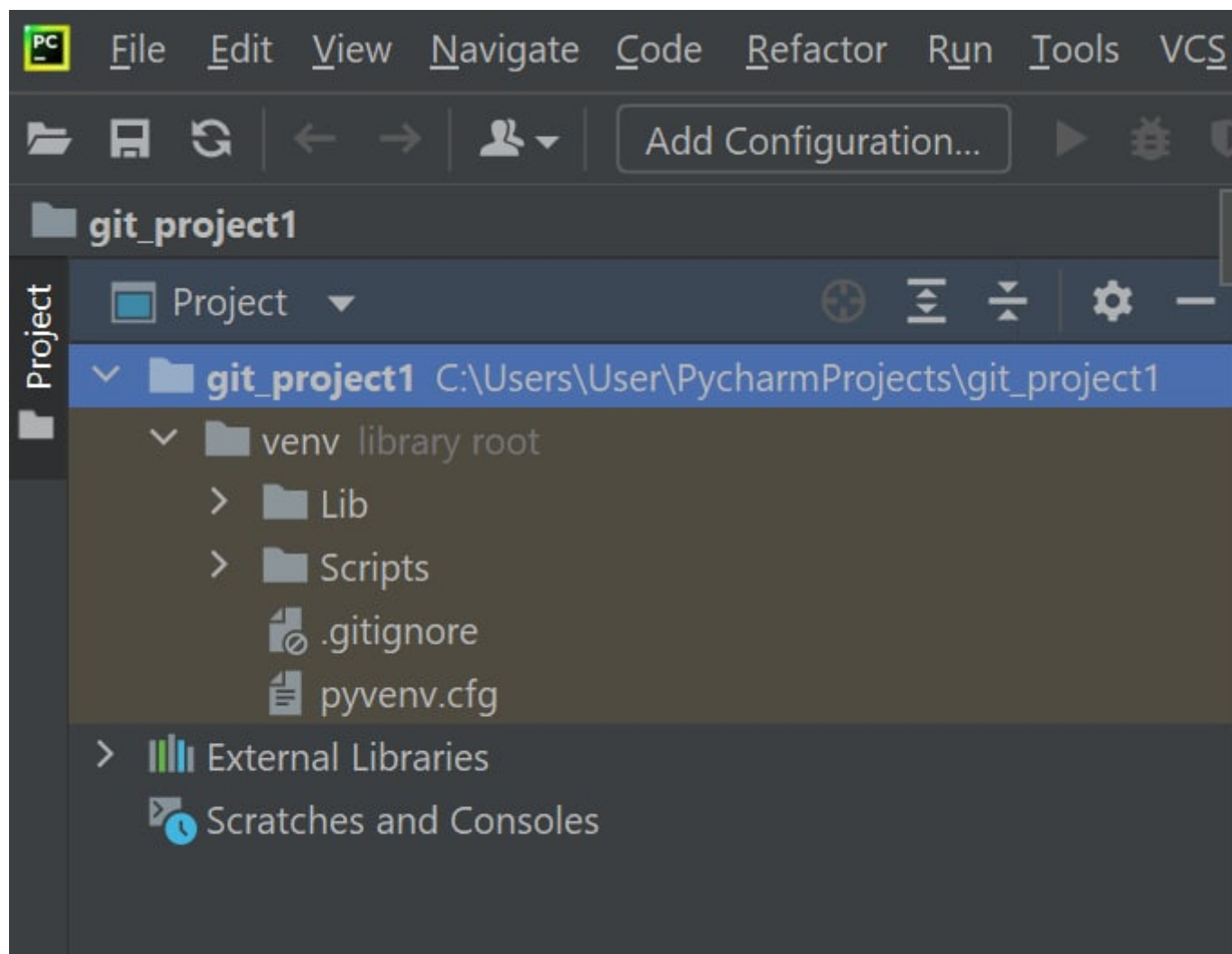
## 5. Создание локального репозитория

Сначала мы создадим новый Python-проект (File → New project) в IDE и назовем его git\_project1.



После этого нажмем кнопку This window, чтобы новый проект открылся в этом же окне PyCharm.

PyCharm создаст на диске папку с названием проекта, которая будет либо пустой (если при создании проекта вы укажете существующий в системе интерпретатор), либо в ней будет папка venv, если для проекта вы решили создать новое виртуальное окружение (свой отдельный от других проектов Python со своими копиями установленных библиотек). Во втором случае придется еще немного подождать, пока происходит создание копии Python в директории с вашим проектом.

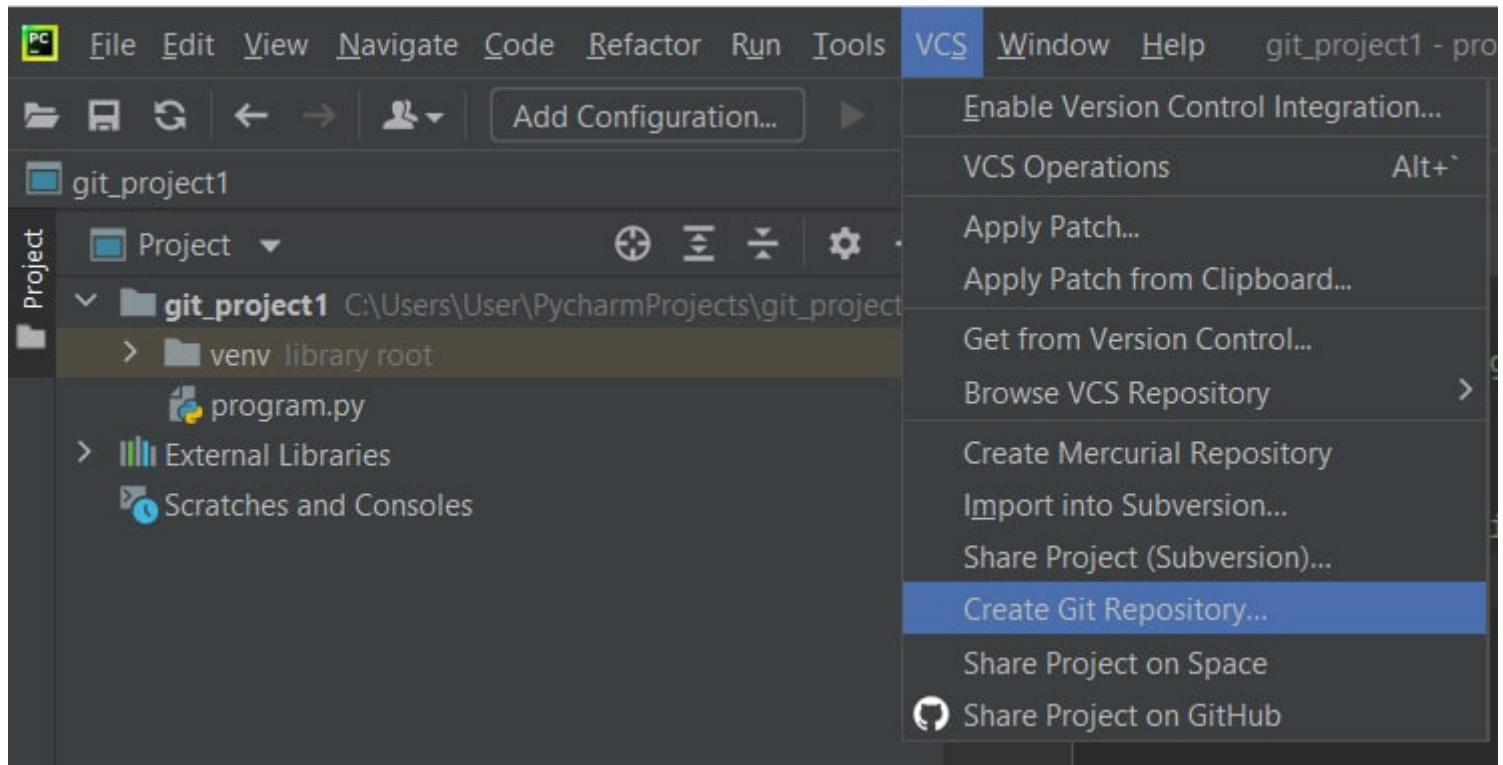


Теперь создадим первый файл нашей программы program.py в папке git\_project1 со следующим содержимым:

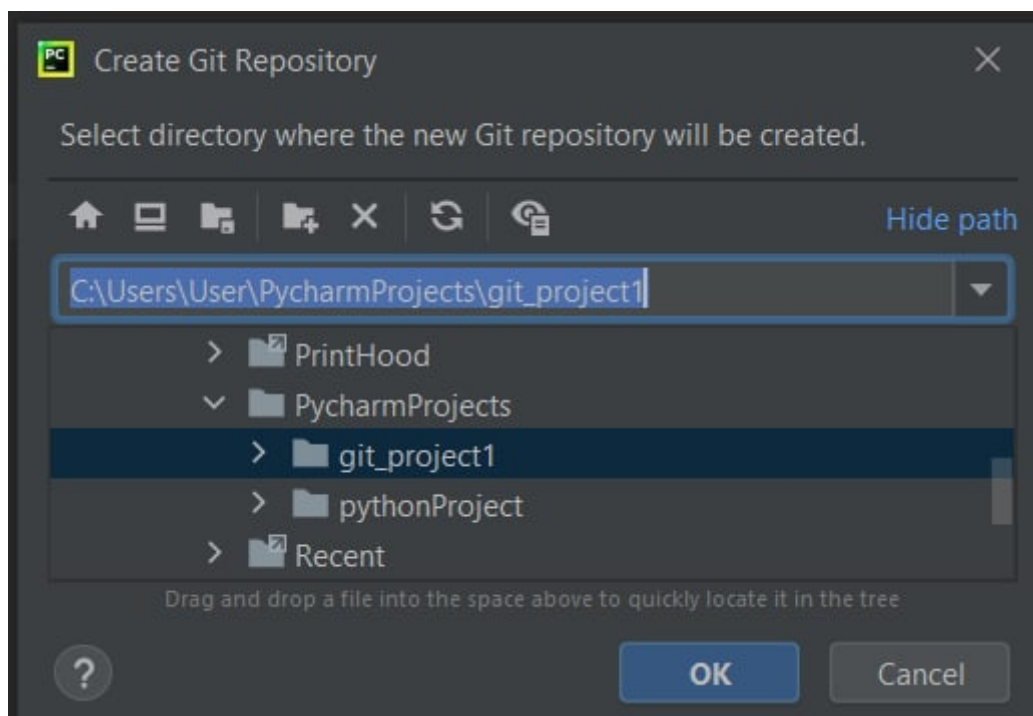
```
def main():  
    print('My first git program')  
  
if __name__ == '__main__':  
    main()
```

И, наконец, инициализируем (создадим новый) в этом каталоге пустой репозиторий Git. **Репозиторием** Git называют каталог (папку, директорию), содержащий отслеживаемые файлы, папки и служебные структуры Git.

Для того чтобы подключить систему контроля версий к проекту, перейдите в пункт меню VCS → Create Git Repository.

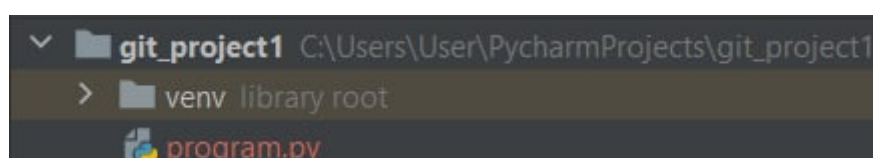


Появится диалоговое окно с вопросом, в каком каталоге мы хотим инициализировать репозиторий:



В нем уже будет выбран текущий каталог, который нам и нужен. Так что просто нажимаем кнопку ОК.

После этого Git создаст в текущей директории скрытую папку .git со служебными структурами репозитория. Сейчас мы не будем ее трогать. Обратите внимание: созданный нами ранее файл program.py стал красным. Так отмечают файлы, не отслеживаемые системой контроля версий.



После инициализации репозитория нам станет доступно меню системы контроля версий. Вызвать его можно с помощью пункта меню View → Tool Windows → Git или сочетанием клавиш Alt + 9, или выбрав в нижней части окна IDE вкладку Git. В нем нас интересует вкладка Log: там хранится история нашего репозитория. Так как мы пока ничего не делали с репозиторием, история пустая.

В Git есть такое понятие, как ветки.

## Ветка

Ветка (branch) — это именованная версия (направление) разработки программы, с которой сейчас работает программист.

Например, представьте, что вы работаете над программой, у которой уже есть стабильная версия. Вам надо ее изменить, и это можно сделать двумя способами. Чтобы выбрать лучший, не повредив текущему состоянию, можно создать два варианта развития программы — две ветки — с именами **Вариант 1** и **Вариант 2**.

Как только мы создаем репозиторий, появляется автоматически сформированная ветка с названием master — главная ветка нашего проекта. В реальной жизни в ней хранится только протестированный код, из которого можно в любой момент времени собрать рабочее, готовое к использованию приложение.

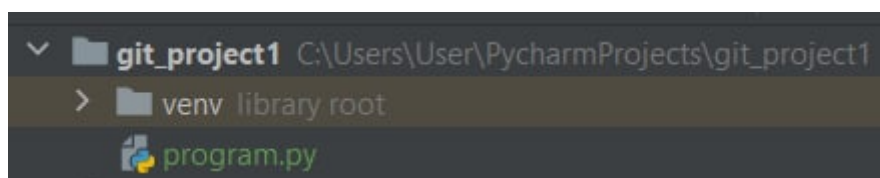
Вы всегда можете **переключаться** между ветками, а каждое подтверждение изменений в терминологии Git называется **коммит** (от англ. Commit).

Подробнее работа с ветками будет рассмотрена немного позже. Поэтому сегодня мы будем делать то, что в реальных проектах делать очень не рекомендуется (существует множество шуток, связанных с этим): мы будем коммитить в master.

Запомните: по умолчанию Git **не отслеживает** новые файлы в репозитории до того момента, пока мы чётко не укажем ему обратное.

## 6. Отслеживание версий файлов

Сообщим Git, что теперь ему необходимо **отслеживать** файл program.py. Кликнем на файле правой кнопкой мыши и в выпадающем меню выберем пункт Git → Add (Для этого действия есть «горячие клавиши» — Ctrl + Alt + A). Цвет, которым написан program.py, сменится на зеленый. Зеленый цвет означает, Git увидел новый файл, ни одной версии которого не зафиксировано в репозитории.



Если просто внести изменения в файл, Git никак не отреагирует. Чтобы система сохранила текущую версию программы, ей нужно дать сигнал через **коммит**.

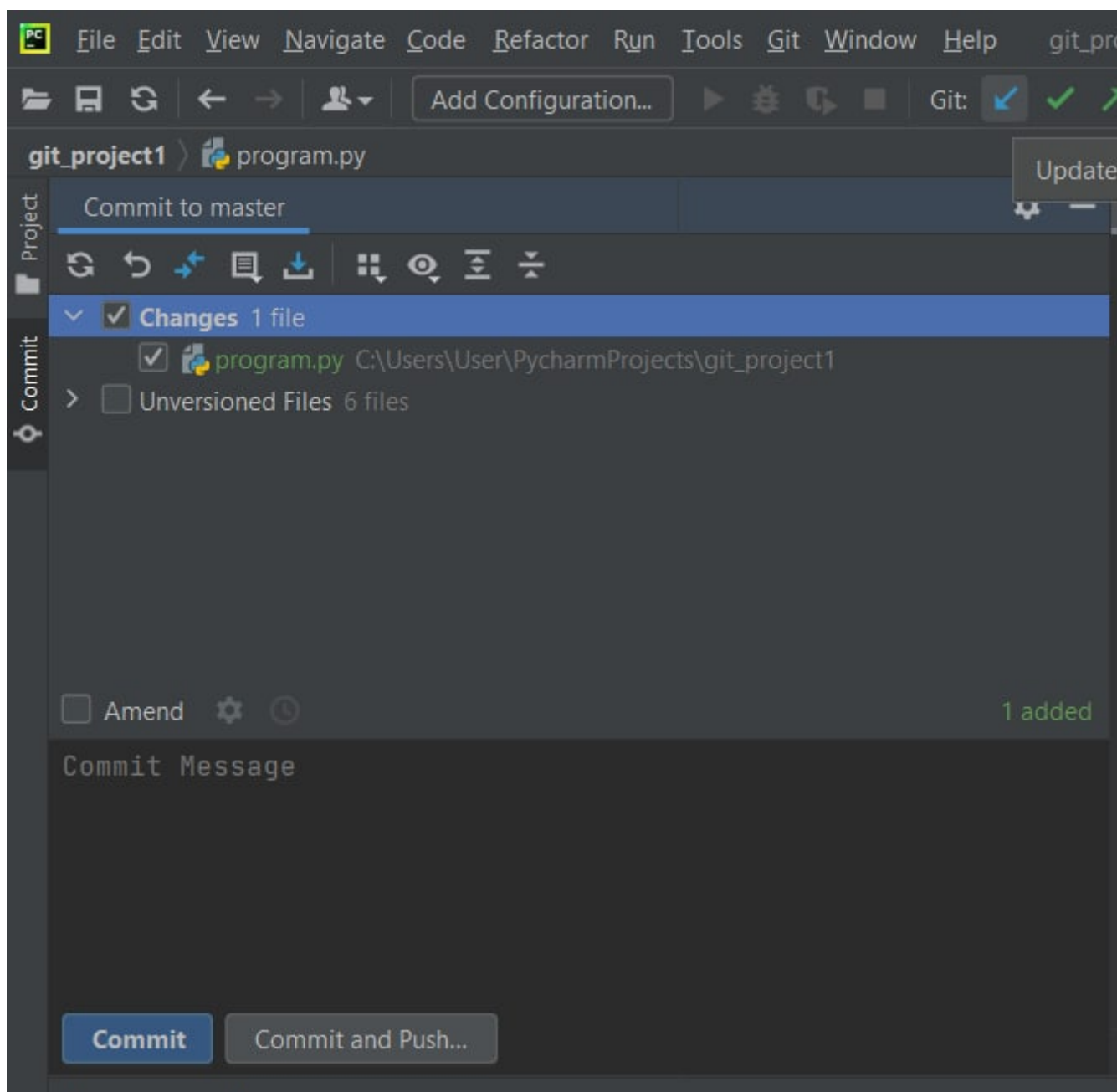
Для этого предназначена команда Git → commit (Ctrl + K). Также можно воспользоваться меню быстрого доступа к командам системы контроля версий (зеленая галочка):



Каждый коммит в Git обязательно сопровождается коротким текстовым сообщением, в котором разработчик описывает внесенные изменения.

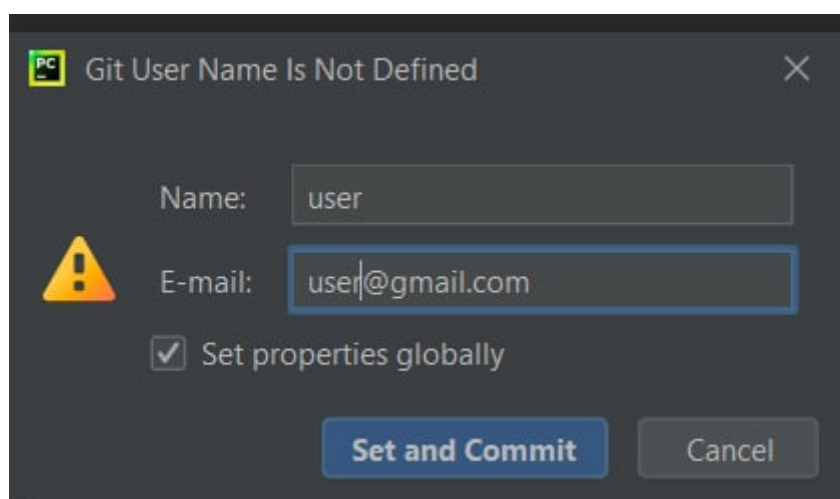


Давайте попробуем:



В верхней части окна также показываются файлы, которые вы собираетесь закоммитить. Если дважды щелкнуть на измененном или добавленном файле, можно посмотреть изменения, внесенные в каждый из файлов. Укажите в качестве сообщения текст "First commit" и нажмите кнопку "Commit".

Если вы запустили Git на своем компьютере в первый раз, то у вас не получится сделать коммит, о чем Git снова любезно проинформирует.

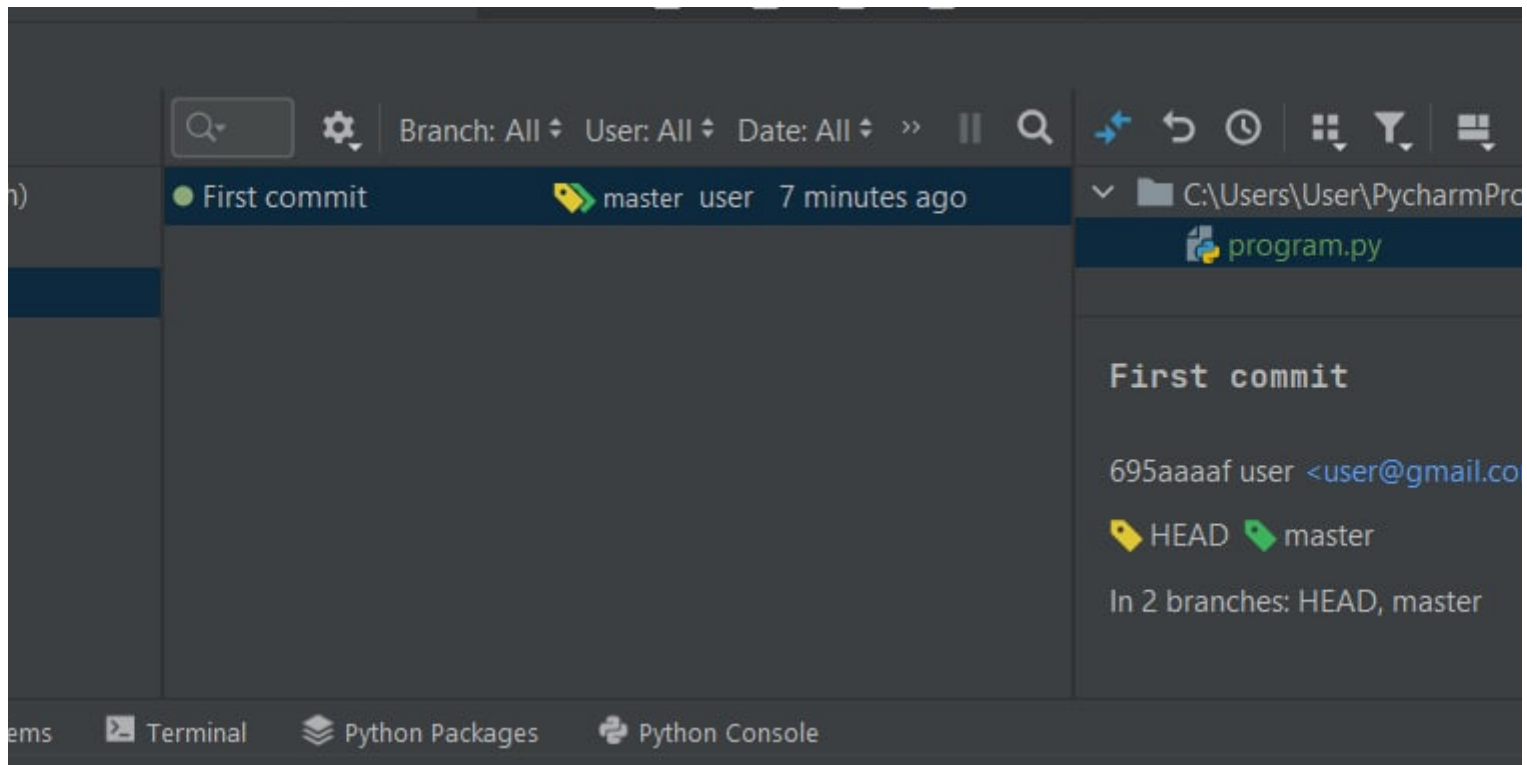


Дело в том, что Git любит вежливых разработчиков, и ему надо сначала **представиться**. Вас попросят ввести

ваше имя и адрес электронной почты.

А затем нажмите кнопку "Set and commit".

Удалось? Снова перейдем на вкладку Log меню системы контроля версий, где увидим наш коммит, а также суть внесенных изменений и некоторую дополнительную информацию:



Ура! Наша первая версия зафиксирована. **Коммит** получился.

У каждой зафиксированной версии в Git есть свой идентификатор, называемый **хэшем**. Посмотреть часть хэша коммита можно в правой нижней части вкладки Log после выбора интересующего коммита, а скопировать целиком — после клика правой кнопкой мышки на коммите в левой части вкладки Log и выбора пункта меню Copy Revision Number. У вас он будет уникальный — последовательность букв и цифр, похожая на 17b655ee91ed6e3483b8bd7e642f2793f7815b33.

Рядом с нашим коммитом мы можем увидеть две бирочки: желтую и зеленую. Зеленая — ветка master, желтая — HEAD.

HEAD — это своего рода указатель. Он сообщает нам, на какой версии мы сейчас находимся и связана ли она с веткой.

В истории версий мы видим:

- Уникальный идентификатор (**хэш**) коммита (версии)
- Направление коммита — из какой ветки в какую мы сохраняем изменения. Сейчас мы сохранили из HEAD в master
- Автора изменения
- Дату изменения
- Детали изменений
- Комментарий, который написал автор коммита

Обратите внимание, что файл program.py в обзоре проекта перестал быть зеленым. Это означает, что в репозитории хранится последняя версия файла.

Теперь внесем изменения в файл program.py и попробуем зафиксировать следующую версию.



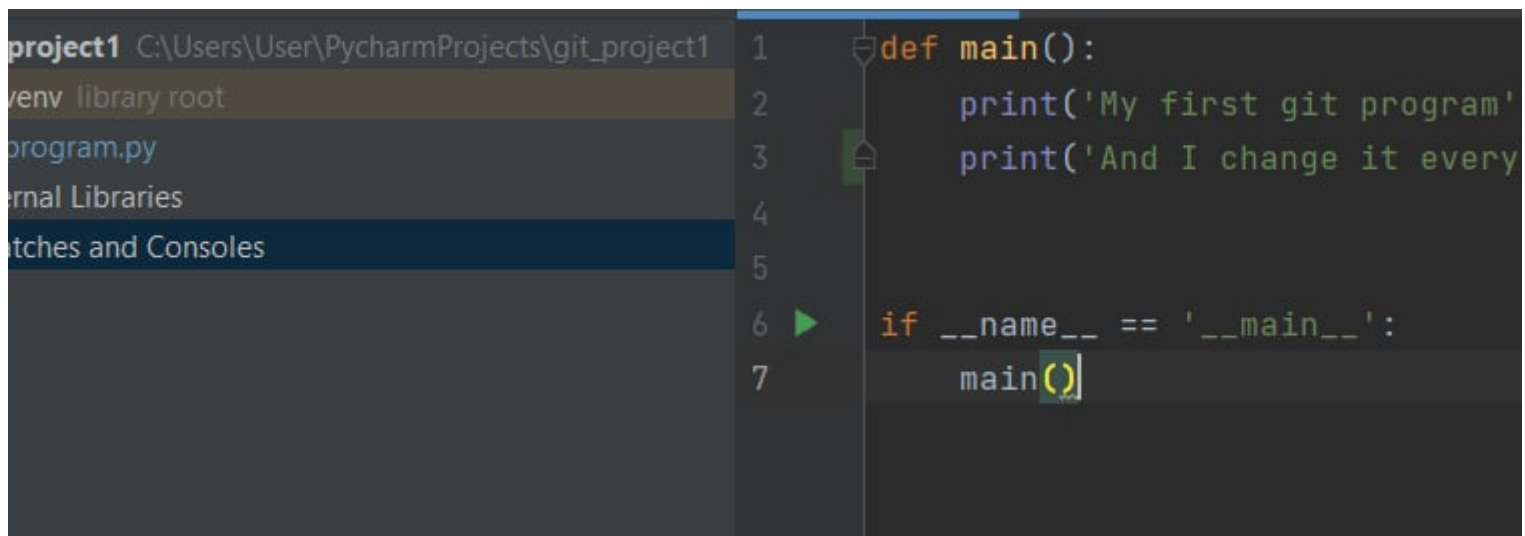
Изменим содержимое файла `program.py` на следующее:

```
def main():
    print('My first git program')
    print('And I change it every day')

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

То есть добавим ещё один `print` в нашу функцию `main`.

Убедимся, что Git отследил изменение файла: цвет файла изменился на синий. Это означает, что в отслеживаемом файле есть незафиксированные изменения. А в редакторе кода поле рядом со строкой, которую мы добавили, стало зеленым, сигнализируя, что это новый код, который пока не сохранен в репозитории:



Теперь у нас есть несколько вариантов действий:

- Отменить изменения — Git → Uncommitted Changes → Rollback (стрелочка разворота на меню быстрого доступа). Эта команда откатит файл к последней зафиксированной версии. *Вы можете проверить это самостоятельно*
- Зафиксировать изменения — Git → Commit

Зафиксируем новую версию с комментарием "New day — new print", а затем посмотрим статус и обновленную историю коммитов.

Как видно из истории, теперь в нашем репозитории есть две зафиксированные (разработчики говорят **закоммиченные**) версии: одна — текущая, на которой есть указатель HEAD (с этой версией мы сейчас работаем), и вторая — наш первый коммит.

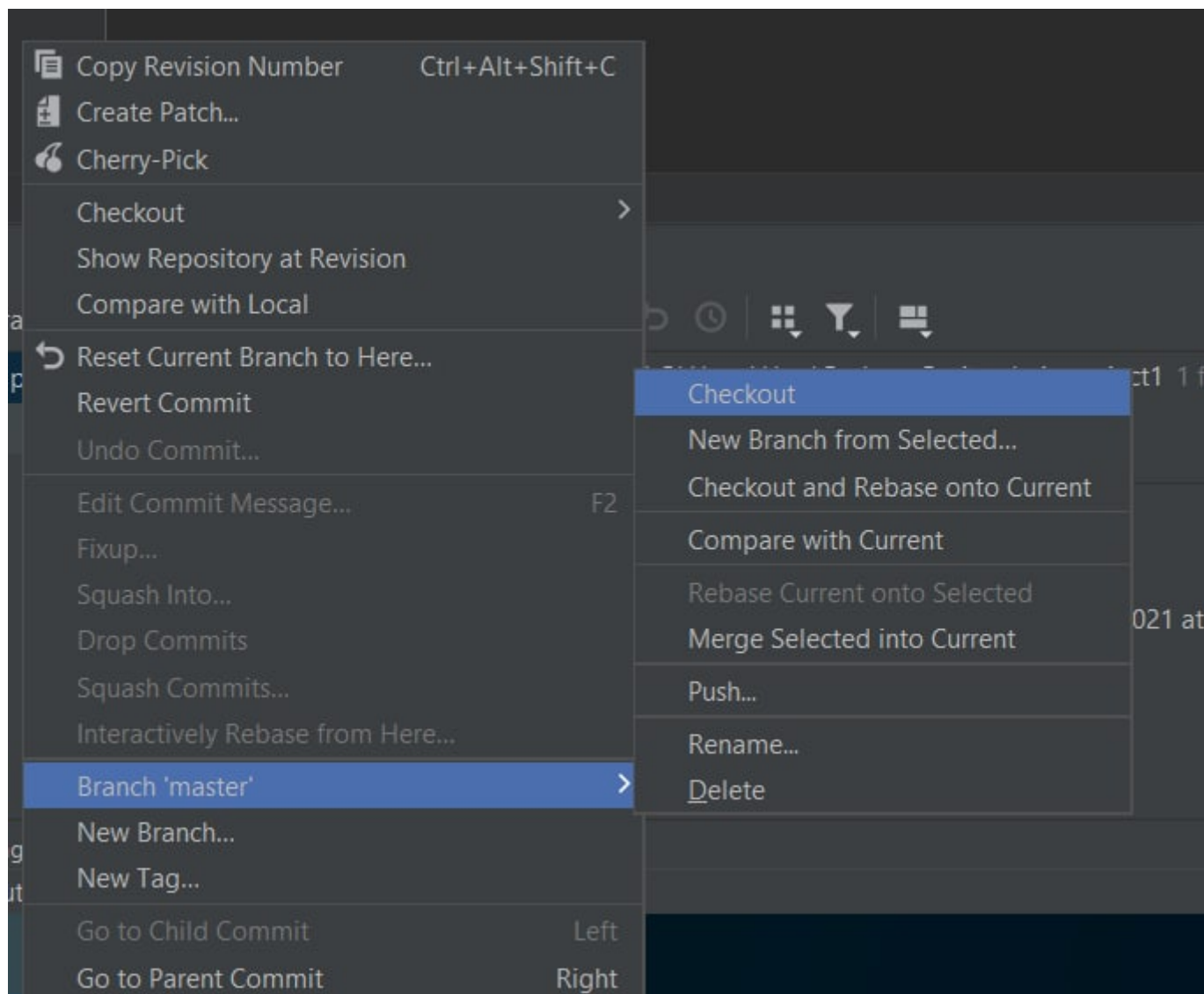
Переключимся на предыдущую версию. Для этого выберем в списке предыдущий коммит, кликнем на нем правой кнопкой мыши и выберем пункт меню Checkout Revision.

Как видим, Git откатил файл к предыдущей версии: в программе нет добавленного нами ранее второго `print`.

Проверим статус в истории Git: метка HEAD находится не на последнем коммите, то есть мы работаем не с последней версией кода в этой ветке. Этого лучше избегать, чтобы не получалось лишних ветвей и высокой степени неопределенности.

Для продолжения работы вернемся к последнему коммиту, выбрав его в истории изменений и переключившись с помощью команды Branch 'master' → Checkout, и убедимся, что мы снова работаем

с наиболее актуальной версией файла.

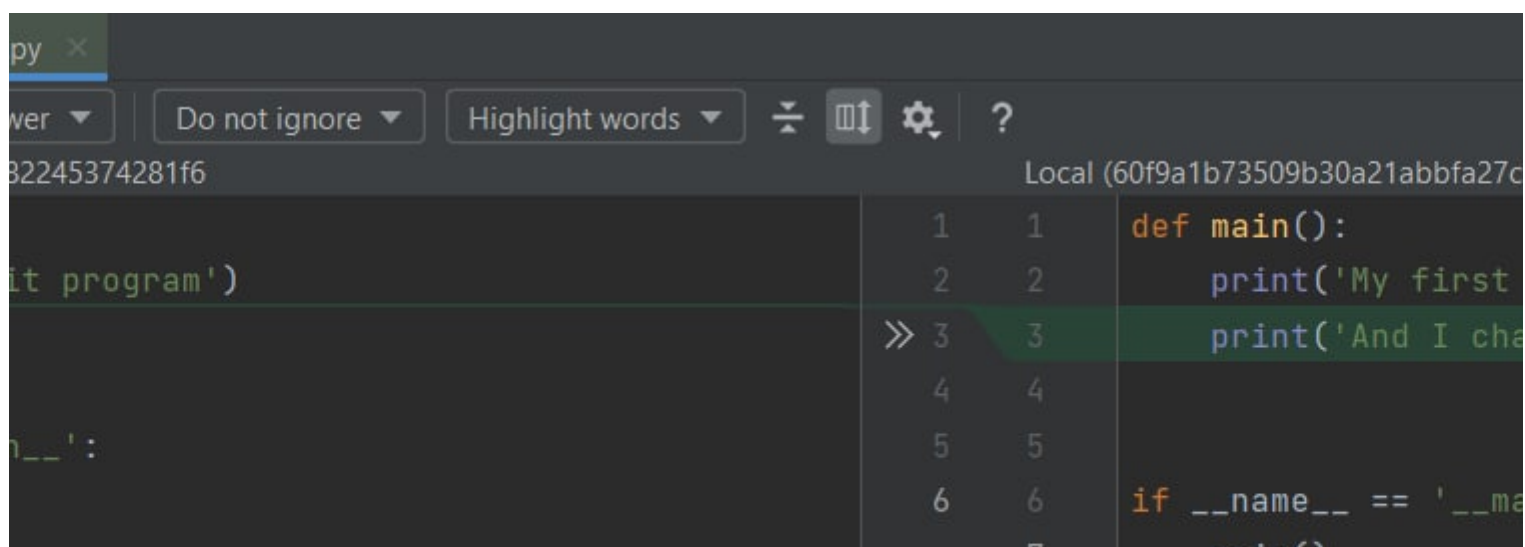


Итак, мы вернулись к последним изменениям.

Возможность откатиться к предыдущей версии файла бывает крайне полезна: например, когда нужно увидеть, как программа работала до последних изменений. Однако постоянное переключение между коммитами не слишком удобно, особенно если нужно отследить изменения многих файлов. Приходится где-то хранить старую и новую версии, как-то искать между ними расхождения, что само по себе — задача трудоемкая.

В PyCharm есть более мощный инструмент для поиска различий в версиях Git. Он вызывается командой Git → Current file → Compare with Revision..., который позволяет сравнить текущую версию файла с версией из любого другого коммита (сначала надо выделить нужный файл в обзоре проекта).

Сравним наши два коммита:



Мы видим исходный код каждой из версий, а также строки, которыми они различаются.

На этом мы закончим рассматривать основные команды для работы с Git из PyCharm как с системой локального хранения и управления версиями файлов.

Подробнее остановимся на вопросах работы с сетевыми репозиториями.

## 7. Начало работы с удаленным репозиторием

Для начала давайте определимся с терминологией. **Локальный репозиторий** — это тот репозиторий, который размещен на конкретной машине разработчика. **Удаленный репозиторий** (он же сетевой репозиторий) — это репозиторий, расположенный на удаленном сервере, в который вносят изменение все разработчики проекта.

В произвольный момент времени состав веток и коммитов во всех репозиториях может различаться, но в некоторые оговоренные моменты времени (как правило, перед началом работы над задачей, после окончания работы над задачей и в конце дня) разработчики синхронизируют свои локальные репозитории с удаленным.

Для этого они выполняют два действия:

1. Скачивают изменения с удаленного репозитория (pull)
2. Отправляют туда свои (push)

Перед отправкой своих изменений разработчик должен объединить (часто говорят «смержить» от английского слова merge) их с изменениями, подтянутыми с сервера.

Чаще всего для ведения сетевых репозиториях используют сервисы GitHub, Gitlab или Bitbucket. На этом уроке мы будем использовать GitHub, потому что он распространен среди разработчиков ПО с открытым кодом, и вы наверняка с ним неоднократно столкнетесь. Кроме того, хорошо выглядящий профиль на GitHub с большим числом полезных коммитов в свои и чужие открытые проекты — это, считайте, половина резюме успешного разработчика.

Поэтому сейчас мы создадим себе аккаунт на GitHub.

## 8. Создание аккаунта на GitHub

Первым делом перейдите по **ссылке**. Заполните Username — ваше имя пользователя в GitHub. Это **ник**, по нему вас будут узнавать. Имя пользователя видят все посетители ваших репозиториях.

Укажите адрес электронной почты, придумайте пароль и нажмите кнопку.

Затем выберите Unlimited public repositories for free. Остальные галочки пока не нужны.

На третьем шаге пока можно ничего не трогать, а просто нажать Submit.

Теперь проверьте электронный почтовый ящик: вам придет ссылка для подтверждения аккаунта.

Войдя на сайт, нажмите на ссылку **Start a project**, чтобы завести свой первый сетевой репозиторий. Введите имя репозитория git\_project1 и нажмите Create repository.

Поздравляем — ваш первый репозиторий создан!

Сохраните ссылку на него — она имеет вид [https://github.com/<ваш\\_username>/git\\_project1.git](https://github.com/<ваш_username>/git_project1.git).

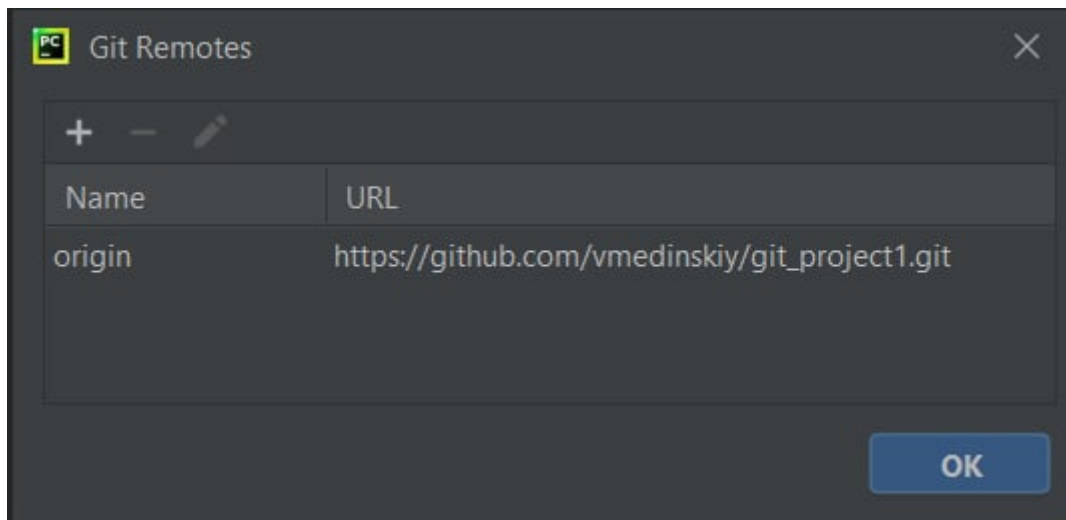
Работая с примерами, не забывайте подставлять в ссылку свое имя пользователя.

После урока стоит почитать **инструкцию** GitHub для новых участников.

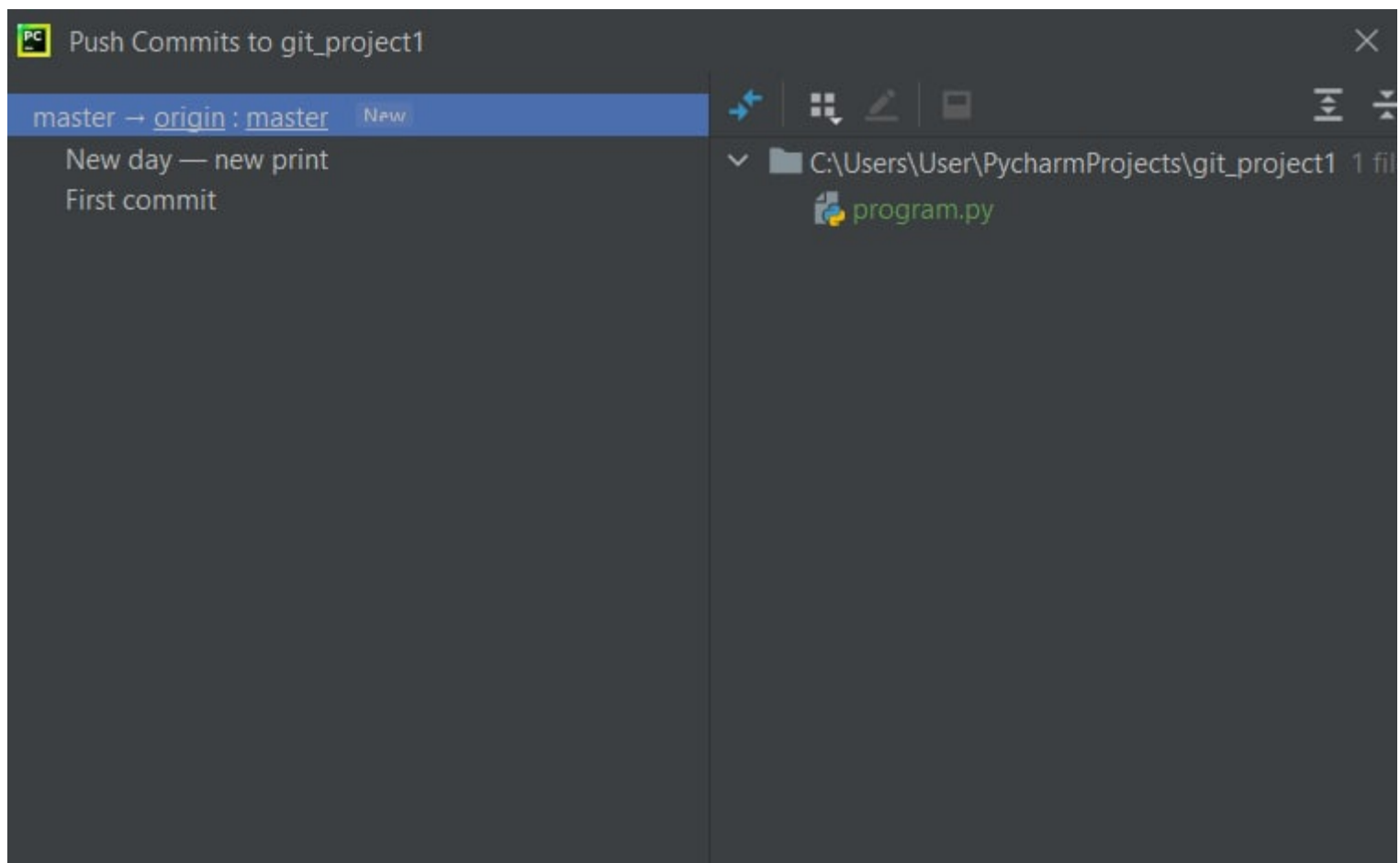
## 9. Синхронизация с сетевыми репозиториями

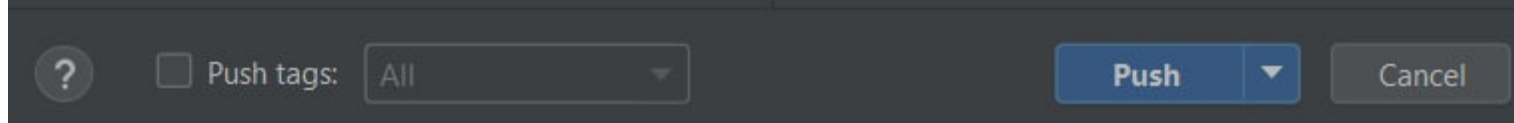
Давайте попробуем выгрузить наш локальный в новый сетевой репозиторий на GitHub. Имейте в виду, что к локальному репозиторию можно подключить несколько удаленных. Сетевые репозитории часто называют **remote-репозиториями**.

Для управления remote-репозиториями используется команда Git → Manage remotes. Подключим к нашему локальному репозиторию удаленный, добавив его в появившемся меню с именем origin. В нашем примере мы подключаем репозиторий с именем github:



Теперь загрузим изменения из нашего локального репозитория в сетевой (удаленный) репозиторий. Для этого используется команда Git → Push (Ctrl + Shift + k). В появившемся окне мы увидим, какие коммиты отправляются из локального в удаленный.

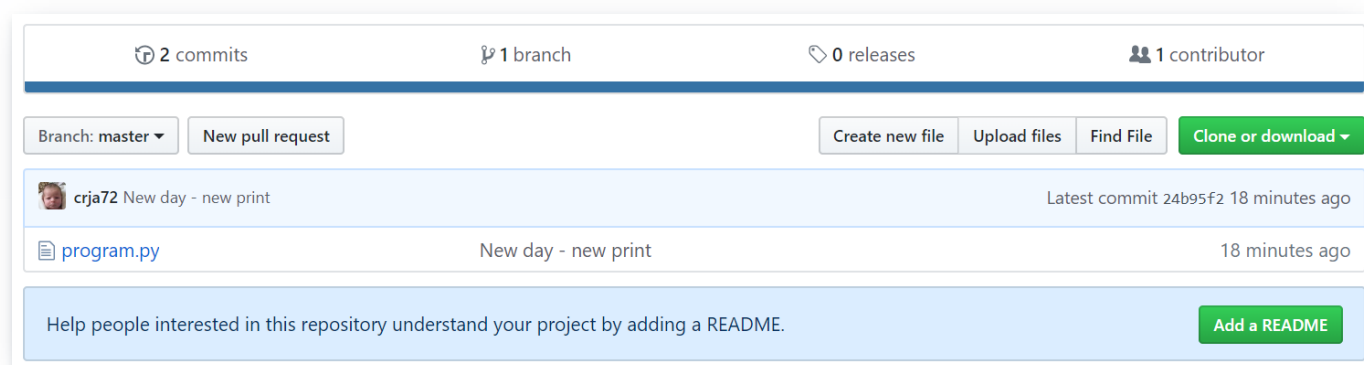




При выполнении этих действий будет запрошен ваши логин и пароль, указанные при регистрации на GitHub.

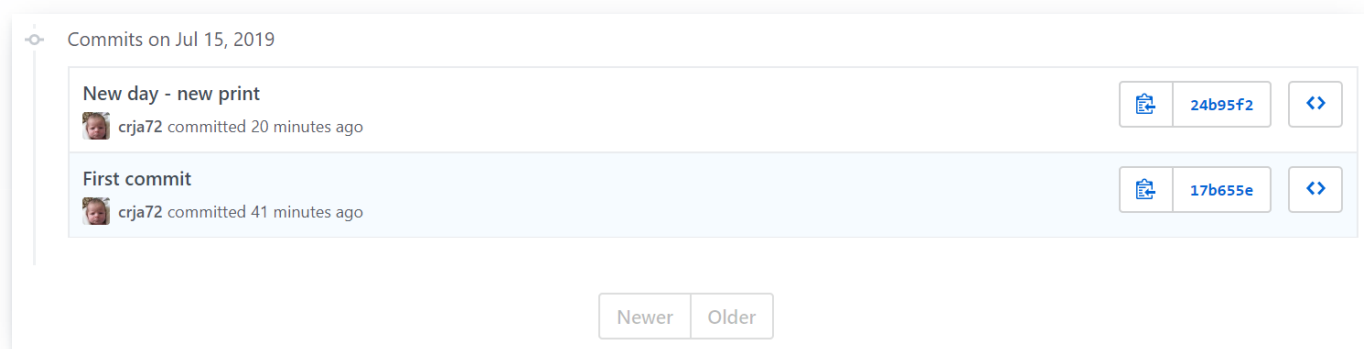
После успешной отправки в удаленном репозитории будет создана ветка `master`, в которую запишутся все наши коммиты для локальной ветки `master`.

Вернитесь в веб-интерфейс Github, открыв в вашем браузере ссылку на репозиторий ([https://github.com/имя\\_пользователя/git\\_lesson\\_repository.git](https://github.com/имя_пользователя/git_lesson_repository.git)), и убедитесь, что в веб-интерфейсе появился наш файл `program.py` из ветки `master`.



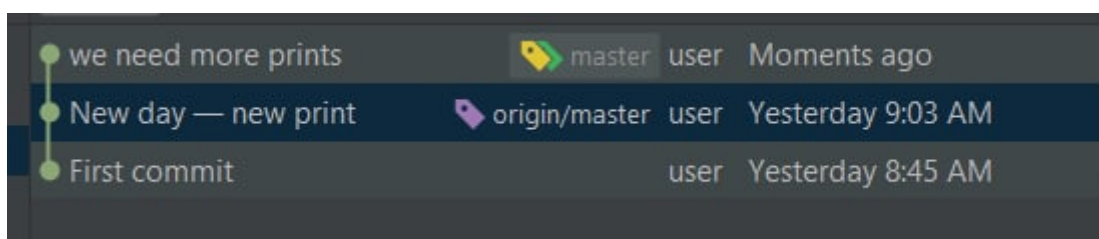
Обратите внимание на ссылку Commits (2) в верхней части репозитория. Перейдя по ней, вы увидите все коммиты, которые мы сделали в ветке `master`.

Важно: при загрузке ветки в удаленный репозиторий копируется не только актуальное состояние ветки, но и вся история коммитов в эту ветку, что позволяет всем пользователям удаленного репозитория легко восстановить хронологию «развития» вашей программы.



Если мы перейдем во вкладку Log меню системы контроля версий, то увидим, что рядом с нашим последним коммитом появилась еще одна фиолетовая бирочка — знак того, что это последний коммит, зафиксированный в удаленном репозитории.

Давайте сделаем еще один коммит и убедимся, что после этого бирочки `HEAD` и `master` передвинулись, а `origin` осталась на месте.

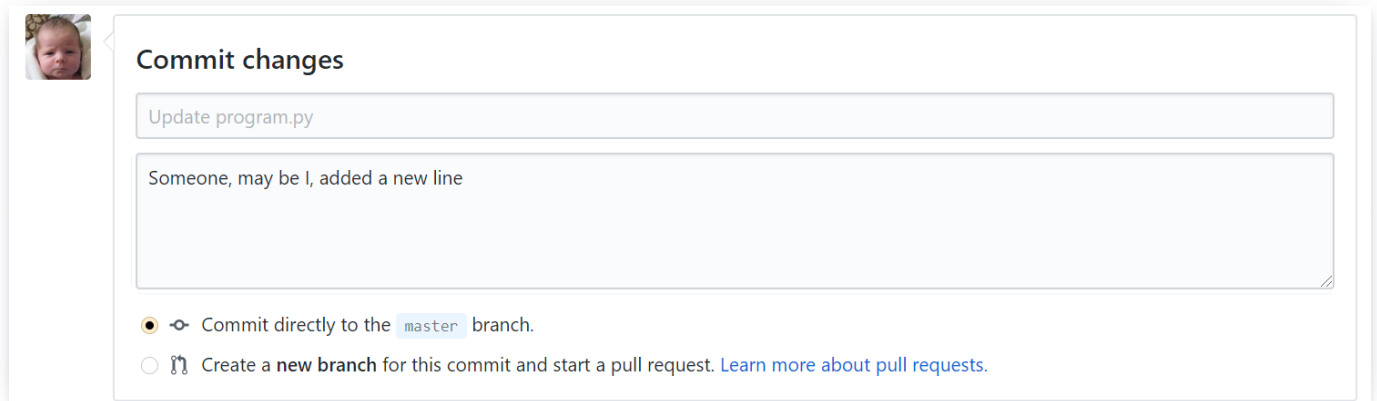


Запустим изменения в удаленный репозиторий.

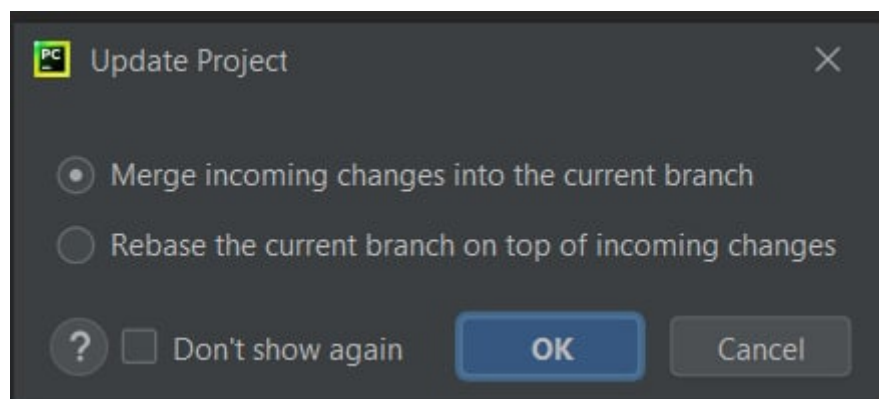
А теперь с использованием веб-интерфейса GitHub симулируем изменения удаленного репозитория другим участником разработки (или нами же, но с другого рабочего места). Кликнем в просмотре репозитория на файл `program.py`, затем на иконку редактирования файла и добавим в файл, какую-либо новую информацию, например еще один `print`:

```
def main():  
    print('My first git program')  
    print('And I change it every day')  
    print('Again')  
    print('UFO came and added this line')  
  
if __name__ == '__main__':  
    main()
```

Добавим комментарий к нашему изменению, так как при изменении файлов через веб-интерфейс GitHub создает в нашем репозитории полноценные коммиты.



Для получения изменений из удаленного в локальный репозиторий используется команда Pull, в PyCharm вокруг нее есть «обертка» в виде меню Git → Update Project (Ctrl + T или стрелочка на «юго-запад» в меню быстрого доступа). Для большинства случаев подойдут настройки по умолчанию.



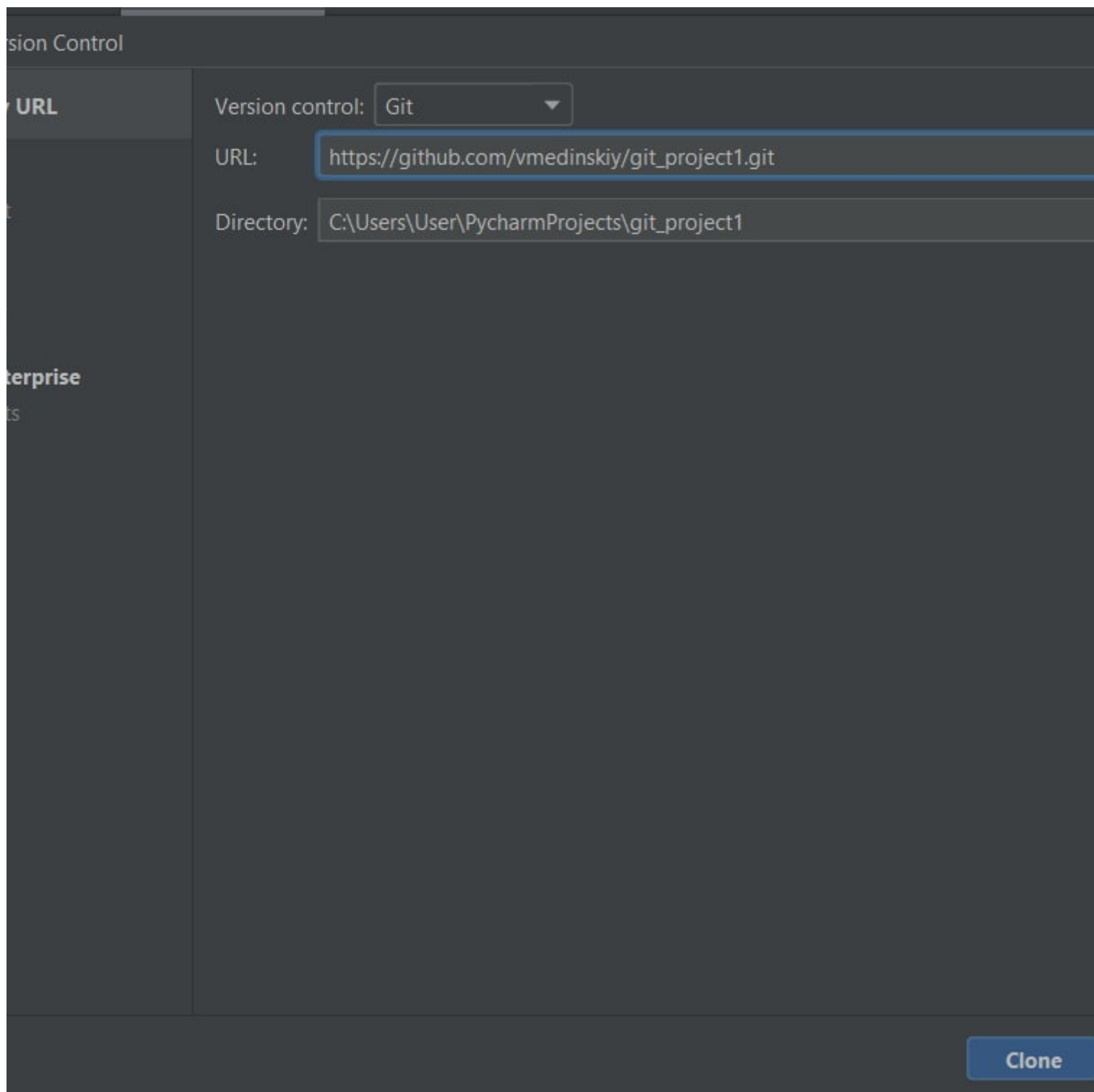
После того, как мы «затянем» изменения из удаленного репозитория в локальный, у нас появятся все коммиты сделанные с других рабочих мест, и все файлы проекта приведутся к актуальным версиям.

## 10. Клонирование сетевого репозитория

Теперь представим, что локального репозитория у вас нет (например, вы начали работать с другого компьютера или к вашему проекту присоединился еще один разработчик). Есть только ссылка на удаленный репозиторий. В этом случае репозиторий необходимо **клонировать**.



Для этого есть команда Git → Clone.



С помощью этой команды вы сможете:

1. Указать адрес удаленного репозитория
2. Указать папку на локальном диске, в которую его необходимо клонировать
3. Проверить доступность репозитория
4. Залогиниться в удаленный Git-репозиторий

Выполняя эту команду, Git проверяет существование удаленного репозитория. Если репозиторий есть, то создается локальный репозиторий, и в него подтягиваются изменения из ветки, на которую указывает HEAD. Как правило, это master удаленного репозитория. Удаленный репозиторий добавляется и как upstream (с возможностью загрузки), и как downstream (с возможностью выгрузки), и получает имя origin.

Попробуйте клонировать свой удаленный репозиторий в другую локальную папку.

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках сервиса, принадлежат АНО ДПО «Образовательные технологии Яндекса». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «Образовательные технологии Яндекса».

[Пользовательское соглашение.](#)

© 2018 – 2024 ООО «Яндекс»