Введение в репозитории. Основные команды при одиночной работе

План урока

Аннотация

*В этом уроке рассматриваются цели систем контроля версий, принципы работы с ними и производится разбор системы **Git**. Мы научимся создавать локальные репозитории, работать с версиями, ветками и разбирать простые конфликты.*

1. Введение

На предыдущих занятиях вы уже познакомились с основами языка Python, попробовали писать программы и узнали базовые принципы работы с IDE. На ближайших двух уроках мы уйдём в сторону от особенностей языка и алгоритмов и погрузимся в отдельную большую тему. Её применяют в любой области разработки, независимо от языка и технологий. Тема называется **Системы контроля версий**.

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, то хочется побыстрее их проверить, изменить код. Но важно сохранить и промежуточные результаты. Потом, если понадобится, к ним можно вернуться. Например, для того, чтобы сравнить версии и выбрать более удачную.

В определённый 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 из дома — 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 легко переключаются между ними;

3. Помогают разработчикам обмениваться кодом и редактировать один и тот же код с разных устройств;

4. Объединяют труд нескольких разработчиков.

Чаще всего используют системы контроля версий **Git**, **Mercurial** (её ещё называют Hg) и несколько устаревшие **Subversion** (она же SVN) и **CVS**.

Давайте посмотрим, из чего состоит и как работает система **Git**.

2. История Git

Однажды разработчикам ОС Linux стало тяжело вместе работать над проектом. **BitKeeper** для этого тоже не годился. И им ничего не оставалось делать, как меньше, чем за неделю, создать прототип **Git**. Первая версия появилась 3 апреля 2005 года, и с тех пор система часто менялась. Актуальная на сегодня версия (2.19) вышла 27 сентября 2018 года.

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 и для macOS. Для работы в ОС Windows рекомендуется использовать среду **Cygwin** как эмулятор Linux под Windows.

Для установки Git в ОС Linux добавьте в систему пакет **git**. Для семейства debian/ubuntu (пару уроков назад вы могли скачать образ такой системы) выполните команду:

```
> sudo apt install git
```

В macOS ситуация похожая. Выполните команду:

```
> brew install git
```

А вот для установки **git** для ОС Windows придётся немного попотеть. Скачайте сборку с этого или этого сайтов и следуйте документации.

Пошаговая инструкция по установке лежит здесь.

Есть клиенты, которые позволяют работать с системами контроля версий в графическом режиме (например, SourceTree). Во время установки они самостоятельно инсталлируют **git**. Это удобно, но последствия контролировать сложней. Поэтому на наших уроках мы будем самостоятельно устанавливать нужные системы и пакеты.

Как работать с Git с помощью таких клиентов мы рассмотрим в одном из следующих уроков.

4. Основы локального использования Git

Теперь мы готовы изучать **Git**. На первом занятии мы поработаем с системой из командной строки, а в одном из следующих уроков — попробуем повторить все действия через IDE.

Итак, запустите командную строку (программа **Терминал** в Linux или macOS, или **Cmd** в Windows).

5. Создание локального репозитория

Сначала мы создадим для нашего проекта новую папку — **git_project_1** — и перейдём в неё. Вспоминаем:

— команда **pwd** показывает наше текущее положение в дереве каталогов;

— команда **mkdir <имя каталога>** создаёт новый каталог;

— команда **cd <имя каталога>** переводит в указанную папку.

```
> pwd
    /files

> mkdir git_project_1
> cd git_project_1
```

Убедимся, что текущий каталог пуст — команда **ls** (**dir** — для ОС Windows).

```
> ls -lsa

    total 0
    0 drwxr-xr-x  2 user  593637566   68  9 ноя 23:16 .
    0 drwxr-xr-x@ 3 user  593637566  102  9 ноя 23:16 ..
```

Теперь создадим первый файл нашей программы **program.py** в папке **git_project_1** со следующим содержимым:

```
print("My first Git program")
```

Теперь каталог не пустой. Убедимся в этом:

```
> ls -lsa

    total 8
    0 drwxr-xr-x  3 user  593637566  102  9 ноя 23:17 .
    0 drwxr-xr-x@ 3 user  593637566  102  9 ноя 23:16 ..
    8 -rw-r--r--@ 1 user  593637566   44  9 ноя 23:17 program.py
```

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.

```
> git init

    Initialized empty Git repository in /files/git_project_1/.git/
```

Выполнив эту команду, Git создаст в текущей директории служебную папку **.git** со служебными структурами репозитория. Сейчас мы не будем её трогать.

Посмотрим на текущее состояние репозитория:

```
> git status

    On branch master

    No commits yet

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

        program.py

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

В выводе команды **git status** мы видим информацию по текущей **ветке** и состоянию отслеживаемой файловой системы.

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 её изменить — и это можно сделать двумя способами. Чтобы выбрать лучший,

не повредив текущему состоянию, создайте два варианта развития программы — две ветки — с именами **Вариант 1** и **Вариант 2**.

Как только мы создаем репозиторий, то у нас появляется автоматически сформированная ветка с названием **master**.

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

Подробнее работу с ветками мы разберём немного позже.

Здесь видно, что Git нашёл в папке репозитория только один файл, но пока его не отслеживает. Запомните: по умолчанию Git **не отслеживает** новые файлы в репозитории до того момента, пока мы чётко не укажем ему на обратное.

6. Отслеживание версий файлов

Сообщим нашему Git, что теперь ему необходимо **отслеживать** файл program.py с помощью команды **git add <имя/маска файла>**, и снова проверим статус репозитория:

```
> git add program.py
> git status

    On branch master

    No commits yet

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

        new file:   program.py
```

Git увидел новый файл и сообщает, что готов отслеживать изменения в нём. Удалить файл из списка отслеживаемых можно с помощью команды **git rm --cached <имя/маска файла>,** о чём Git нам любезно поведал.

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**. Каждый коммит в Git обязательно сопровождается коротким текстовым сообщением, в котором разработчик описывает внесённые изменения. Давайте попробуем:

```
> git commit -m "Мой первый коммит"

    [master (root-commit) f96f82d] Мой первый коммит
     1 file changed, 2 insertions(+)
     create mode 100644 program.py
```

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

Дело в том, что Git любит вежливых разработчиков, и ему надо сначала **представиться**. Тогда Git расскажет, что делать: выполнить команду **git config**.

Чтобы представиться системе, выполните команды:

```
> git config --global user.email "developer@yandex.ru"
> git config --global user.name "Smart developer"
```

А затем снова повторите команду **commit**.

Удалось? Теперь проверим статус:

```
> git status

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

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

У каждой зафиксированной версии в Git есть свой идентификатор, называемый **хэшом**. Посмотреть историю версий можно с помощью команды **git log**:

```
> git log

    commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167 (HEAD -> master)
    Author: Smart developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:18:21 2017 +0300

        Мой первый коммит
```

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;

— комментарий, который написал автор коммита.

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

Изменим содержимое файла program.py на:

```
print("My first Git program!!!")
```

то есть добавим ещё три восклицательных знака в конец.

Убедимся, что Git отследил изменение файла:

```
> git status

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

        modified:   program.py

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

Git заметил изменения и теперь предлагает на выбор два варианта:

— Отменить изменения — **git checkout -- program.py**. Эта команда откатит файл к последней зафиксированной версии. Вы можете проверить это самостоятельно;

— Зафиксировать изменения **git commit -a -m "Добавили восклицательные знаки в конце предложения"**. Параметр -a указывает, что нужно зафиксировать изменения всех изменившихся файлов. Вместо этого параметра можно просто перечислить имена нужных файлов: **git commit program.py -m "Добавили восклицательные знаки в program.py"**. Параметр -m задаёт комментарий к коммиту.

Зафиксируем новую версию, а затем посмотрим статус и обновлённую историю коммитов:

```
> git commit -a -m "Добавили восклицательные знаки в конце предложения."
```

```
    [master b6e8fa1] Добавили восклицательные знаки в конце предложения.
     1 file changed, 1 insertion(+), 1 deletion(-)

> git status

    On branch master
    nothing to commit, working tree clean

> git log

    commit b6e8fa1fa53a9d4970994d6139fbe51caae99751(HEAD -> master)
    Author: Smart Developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:19:11 2017 +0300

        Добавили восклицательные знаки в конце предложения.

    commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167
    Author: Smart Developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:18:21 2017 +0300

        Мой первый коммит
```

Как видно из истории, теперь в нашем репозитории есть две зафиксированные (разработчики говорят **закоммиченные**) версии:

1. commit **b6e8fa1fa53a9d4970994d6139fbe51caae99751** (HEAD → master) — запись в скобках обозначает, что с этой версией мы сейчас работаем;

2. commit **f96f82d3c9c2ceec1c8a789ead881ce0d146f167** — предыдущая версия.

Переключимся на предыдущую версию. Для этого выполним команду **git checkout <хеш (имя) версии, на которую мы хотим переключиться>**.

Имя можно сокращать до нескольких первых символов — лишь бы их было достаточно, чтобы отличить нужную версию от других. Меньше четырёх символов вводить нельзя, даже если они образуют уникальную последовательность.

```
> git checkout f96f82d

    Note: checking out 'f96f82d'.

    You are in 'detached HEAD' state. You can look around, make
    experimental changes and commit them, and you can discard any
    commits you make in this state without impacting any branches
    by performing another checkout.

    If you want to create a new branch to retain commits you create,
```

```
    you may do so (now or later) by using -b with the checkout command
    again. Example:

      git checkout -b <new-branch-name>

    HEAD is now at f96f82d... Мой первый коммит
```

Заботливый Git продолжает подсказывать вам, что же можно сейчас сделать. Например, осмотреться, поэкспериментировать и т.д. Постарайтесь понять подсказки самостоятельно, а мы посмотрим, что сейчас есть в тексте нашей программы program.py:

```
> cat "program.py"

    print("My first Git program")
```

Как мы видим, Git откатил файл к предыдущей версии: в выводимой строке нет восклицательных знаков.

Проверим статус:

```
> git status
> git log
    HEAD detached at f96f82d
    nothing to commit, working tree clean
    commit f96f82d3c9c2ceec1c8a789ead881ce0d146f167 (HEAD)
    Author: Smart Developer <developer@yandex.ru>
    Date:   Thu Nov 9 23:18:21 2017 +0300

        Мой первый коммит
```

Сообщения показывают, что сейчас указатель текущей версии **HEAD** стоит на промежуточном коммите. Этого лучше избегать, чтобы не получалось лишних ветвей и высокой степени неопределённости.

Для продолжения работы вернёмся к последнему коммиту, указав при переключении хэш последнего коммита, и убедимся, что мы снова работаем с наиболее актуальной версией файла:

```
> git checkout b6e8fa1

    Previous HEAD position was f96f82d... Мой первый коммит
    HEAD is now at b6e8fa1... Добавили восклицательные знаки
                              в конце предложения.
```

```
> cat program.py

    print("My first Git program!!!")
```

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 трудоёмкая.

В Git есть более мощный инструмент для поиска различий в версиях. Он вызывается командой **diff**.

Команду diff можно записать как **git diff <коммит 1> <коммит 2>**, либо как **git diff <коммит 2>**.

Во втором случае коммитом для сравнения будет выбран текущий активный коммит, на который указывает HEAD.

Если вы меняли файлы после последнего коммита, то можете дать команду **git diff**. В этом случае Git сравнит активную ветку (HEAD) с текущим состоянием репозитория.

Сравним наши два коммита:

```
> git diff f96f82 b6e8fa1f

    diff --git a/program.py b/program.py
    index 90ab278..7da6c40 100644
    --- a/program.py
    +++ b/program.py
     @@ -1,2 +1,2 @@
    -print("My first Git program")
    \ No newline at end of file
    +print("My first Git program!!!")
    \ No newline at end of file
```

Вывод команды diff интуитивно понятен. Мы видим список файлов, различающихся в двух версиях, и список изменений в каждом из них. Плюсами обозначены новые строки, минусами — удалённые строки.

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

Теперь подробнее остановимся на вопросах ветвления, работы с сетевыми репозиториями и объединения изменений.

7. Ветки в Git

Мы уже немного затронули тему о **ветках** в Git. Настало время внимательней разобрать это понятие.

Если бы система контроля версий ограничивалась только ведением истории коммитов, то она бы никак не помогла нескольким разработчикам работать с одним репозиторием. Или одному разработчику — над несколькими задачами в одном.

Для всего этого (и не только) в Git и существуют ветки. Ветки позволяют:

— давать имена версиям;

— иметь одновременно несколько **рабочих** версий (рабочей версией называется та, над которой в определенный момент времени трудится разработчик);

— объединять результаты деятельности нескольких разработчиков.

Допустим, мы хотим создать версию нашей программы, выводящую надпись «Hello, python!!!». Старую версию мы тоже хотим сохранить.

Сначала заведём новую ветку программы и сразу на неё переключимся.

Это делается командой **git checkout -b <имя новой ветки>**:

```
> git checkout -b "python3branch"

    Switched to a new branch 'python3branch'
```

Создать новую ветку можно и **git branch <имя ветки>**, но эта команда не переключает нас на созданную ветку.

Заведём ветку demobranch:

```
> git branch demobranch
```

Посмотрим на список веток, которые сейчас есть в нашем репозитории. Для этого нужна команда **git branch** (без параметров).

```
> git branch

      demobranch
      master
    * python3branch
```

Мы видим, что обе ветки demobranch и python3branch созданы. python3branch — текущая активная ветка, она помечена звёздочкой. Это именно та ветка, которую мы создали командой checkout -b 'python3branch'.

Ещё мы видим, что, кроме созданных нами веток demobranch и python3branch, в списке есть master — её Git всегда автоматически создаёт для нового проекта. Именно в ней мы и работали, пока не завели новые ветки.

Переключаться между ветками можно с помощью команды **git checkout <имя ветки>** — той же самой, что позволяет переключаться между коммитами. Для Git ветки — просто коммиты особого типа.

Перейдём на ветку master, а затем вернёмся к python3branch:

```
> git checkout master

    Switched to branch 'master'

> git branch

     demobranch
    * master
      python3branch

> git checkout python3branch

    Switched to branch 'python3branch'

> git branch

      demobranch
      master
    * python3branch
```

Теперь удалим неиспользуемую ветку demobranch командой **git branch -d <имя ветки>** и убедимся, что у нас их осталось только две:

```
> git branch -d demobranch

    Deleted branch demobranch (was b6e8fa1).

> git branch

      master
    * python3branch
```

Теперь мы умеем создавать и удалять ветки, а также переключаться между ветками.

А сейчас в ветке python3branch модифицируем нашу программу так, чтобы она выводила нужную надпись, и зафиксируем версию.

В любом текстовом редакторе изменим содержимое program.py на:

```
print("Hello, python")
```

а после зафиксируем версию и проверим текущее состояние репозитория:

```
> git commit program.py -m "Python3 version"

    [python3branch 96aaee0] Python3 version
     1 file changed, 1 insertion(+), 1 deletion(-)

> git status

    On branch python3branch
    nothing to commit, working tree clean
```

Версии программы, зафиксированные в разных ветках, можно сравнивать между собой с помощью уже знакомой нам команды **diff**:

```
> git diff master python3branch

    diff --git a/program.py b/program.py
    index 7da6c40..1ce4f9a 100644
    --- a/program.py
    +++ b/program.py
    @@ -1,2 +1,2 @@
    -print("My first Git program!!!")
    \ No newline at end of file
    +print("Hello, python")
    \ No newline at end of file
```

Обратите внимание, что, создавая ветку любой из описанных выше команд, мы получаем **полную копию** ветки, в которой находились в этот момент.

Для закрепления успехов перейдём на ветку master, создадим новую addAuthorBranch и переключимся на неё:

```
> git checkout master

    Switched to branch 'master'

> git checkout -b addAuthorBranch

    Switched to a new branch 'addAuthorBranch'

> git status

    On branch addAuthorBranch
    nothing to commit, working tree clean

> git branch

    * addAuthorBranch
      master
      python3branch
```

Изменим содержимое файла program.py, добавив в первую строчку комментарий с именем автора программы:

```
# I am author!
print("My first Git program!!!")
```

И зафиксируем версию в ветке addAuthorBranch:

```
> git commit -a -m "Add author"

    [addAuthorBranch bc31053] Add author
     1 file changed, 1 insertion(+)
```

Переключимся на ветку master и проанализируем, что же у нас в итоге получилось.

Для наглядного отображения введём команду **git log** с параметрами:

--all — показывать историю всех веток;

--graph — показывать ветки в виде дерева;

--oneline — не показывать комментарии к коммитам;

--abbrev-commit — показывать сокращённые имена коммитов.

Полный список опций доступен по команде **git help log**:

```
> git checkout master

    Switched to branch 'master'

> git log --graph --oneline --all --abbrev-commit

    * bc31053 (addAuthorBranch) Add author
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 (HEAD -> master) Добавили восклицательные
                               знаки в конце предложения.
    * f96f82d Мой первый коммит
```

Проанализируем вывод последней команды.

Сначала был **Мой первый коммит**. Потом мы сделали следующий, добавив восклицательные знаки в конце. Текущая главная ветка master сейчас указывает на эту версию (обратите внимание на комментарий). При этом наша последовательность изменений пока ещё оставалась линейной.

Из master'а у нас получилось два ответвления: python3branch и addAuthorBranch. Каждое из них является продолжением версии master, но вносит в код программы свои уникальные изменения. При этом две версии в настоящий момент никак не связаны друг с другом, и работать над каждой можно совершенно независимо.

Для следующего задания создадим ещё одно ответвление от ветки master:

```
> git checkout -b "addFooter"

    Switched to a new branch 'addFooter'
```

Изменим программу program.py следующим образом:

```
print("My first Git program!!!")
# 2017 (c) Me
```

Зафиксируем изменения и вернёмся на ветку master:

```
> git commit -a -m 'Add footer'

    [addFooter 1c43860] Add footer
     1 file changed, 2 insertions(+), 1 deletion(-)

> git checkout master
```

```
Switched to branch 'master'
```

И снова посмотрим на дерево коммитов:

```
> git log --graph --oneline --all --abbrev-commit

* 1c43860 (addFooter) Add footer
| * bc31053 (addAuthorBranch) Add author
|/
| * 96aaee0 (python3branch) Python3 version
|/
* b6e8fa1 (HEAD -> master) Добавили восклицательные
                           знаки в конце предложения.
* f96f82d Мой первый коммит
```

8. Объединение (слияние) изменений

Итак, Git позволяет независимо разрабатывать несколько версий программы в разных **ветках** одного большого дерева.

Теперь разберёмся, как из двух веток собрать единую версию. Попробуем получить программу как с приветствием «Hello, python!!!» из ветки python3branch, так и с именем автора из ветки addAuthorBranch.

Для объединения нескольких веток в одну используется команда **git merge <имя ветки>** — она добавляет изменения из ветки <имя ветки> в текущую (в которой мы сейчас находимся).

Сначала добавим в master коммиты из addAuthorBranch и посмотрим на результат:

```
> cat program.py

print("My first Git program!!!")

> git merge addAuthorBranch

Updating b6e8fa1..bc31053
Fast-forward
 program.py | 1 +
 1 file changed, 1 insertion(+)

> cat program.py
```

```
    # I am author!
    print("My first Git program!!!")

> git log --graph --oneline --all --abbrev-commit

    * 1c43860 (addFooter) Add footer
    | * bc31053 (HEAD -> master, addAuthorBranch) Add author
    |/
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 Добавили восклицательные знаки в конце предложения.
    * f96f82d Мой первый коммит
```

Изменения из addAuthorbranch попали в master. Надпись **Fast-forward** говорит о том, что добавляемая ветка — **дочерняя** ветка текущей. Для такого объединения Git копирует изменения из указанной ветки без сложных сопоставлений — это самый простой и удобный случай.

Теперь добавим изменения из ветки addFooter. Обратите внимание, что в данном случае мы объединяем master с веткой, которая уже не её **дочка** — master уже там, где addAuthorBranch.

Ещё добавим комментарий с помощью параметра -m, чтобы потом понимать, какие действия мы совершали.

```
> git merge addFooter -m "Merge footer"

    Auto-merging program.py
    CONFLICT (content): Merge conflict in program.py
    Automatic merge failed; fix conflicts and then commit the result.
```

Судя по сообщению — что-то пошло не так. Давайте разберёмся.

Git не смог автоматически объединить ветки и предупредил нас о **конфликте**. Он может возникнуть и тогда, когда два разработчика поправили одну и ту же строчку кода.

Некоторые IDE умеют автоматически разрешать такие конфликты. Но в итоге может получиться неправильный код, поэтому лучше всегда проверять конфликты вручную.

Давайте разрешим этот конфликт. Для этого посмотрим на содержание файла program.py:

```
> cat program.py
    <<<<<<< HEAD
    # I am author!
    print("My first Git program!!!")
```

```
=======
print("My first Git program!!!")
# 2017 (c) Me
>>>>>>> addFooter
```

Git просто объединил содержание двух файлов. При этом он разметил части файла признаками, указывающими на ветки, из которых взяты изменения.

Содержимое файла между <<<<<<< HEAD и ======= — это наша текущая ветка, в которую мы пытаемся добавить изменения.

Между ======= и >>>>>>>> <имя ветки> — содержимое новой ветки, которое конфликтует со старым значением.

Теперь вручную отредактируем файл и объединим изменения так, как считаем правильным:

```
# I am author!
print("My first Git program!!!")
# 2017 (c) Me
```

Сохраним файл и сообщим системе, что мы готовы зафиксировать исправленную версию:

```
> git commit -a -m 'Решаем конфликт слияния master и addFooter'

    [master cd7a95b] Решаем конфликт слияния master и addFooter
```

Посмотрим, что получилось:

```
> git log --graph --oneline --all --abbrev-commit

    *   cd7a95b (HEAD->master) Решаем конфликт слияния master и addFooter
    |\
    | * 1c43860 (addFooter) Add footer
    * | bc31053 (addAuthorBranch) Add author
    |/
    | * 96aaee0 (python3branch) Python3 version
    |/
    * b6e8fa1 Добавили восклицательные знаки в конце предложения.
    * f96f82d Мой первый коммит

> cat "program.py"

    # I am author!
```

```
    print("My first Git program!!!")
    # 2017 (c) Me
```

Получилось! Но у нас осталась «висящая» ветка python3branch. Давайте уже и её отправим в master.

```
> git merge python3branch -m "Объединение с python3branch"

    Auto-merging program.py
    CONFLICT (content): Merge conflict in program.py
    Automatic merge failed; fix conflicts and then commit the result.
```

Опять конфликт. Решаем его.

```
> cat "program.py"

    <<<<<<< HEAD
    # I am author!
    print("My first Git program!!!")
    # 2017 (c) Me
    =======
    print("Hello, python")
    >>>>>>> python3branch

> git commit -a -m 'Решаем конфликт слияния master и python3branch'

    [master 9a704f6] Решаем конфликт слияния master и python3branch
```

И смотрим на итоговое дерево, на этот раз опустив параметр «--oneline»:

```
> git log --graph --all --abbrev-commit

        *   commit 9a704f6 (HEAD -> master)
        |\  Merge: cd7a95b 96aaee0
        | | Author: user <user@gmail.com>
        | | Date:   Thu Nov 9 23:27:31 2017 +0300
        | |
        | |     Решаем конфликт слияния master и python3branch
        | |
        | * commit 96aaee0 (python3branch)
        | | Author: user <user@gmail.com>
        | | Date:   Thu Nov 9 23:21:33 2017 +0300
        | |
```

```
| |     Python3 version
| |
* |   commit cd7a95b
|\ \  Merge: bc31053 1c43860
| | | Author: user <user@gmail.com>
| | | Date:   Thu Nov 9 23:25:36 2017 +0300
| | |
| | |     Решаем конфликт слияния master и addFooter
| | |
| * | commit 1c43860 (addFooter)
| |/  Author: user <user@gmail.com>
| |   Date:   Thu Nov 9 23:23:07 2017 +0300
| |
| |       Add footer
| |
* | commit bc31053 (addAuthorBranch)
|/  Author: user <user@gmail.com>
|   Date:   Thu Nov 9 23:22:20 2017 +0300
|
|       Add author
|
* commit b6e8fa1
| Author: user <user@gmail.com>
| Date:   Thu Nov 9 23:19:11 2017 +0300
|
|     Добавили восклицательные знаки в конце предложения.
|
* commit f96f82d
  Author: user <Developer@yandex.ru>
  Date:   Thu Nov 9 23:18:21 2017 +0300

      Мой первый коммит
```

На этом мы заканчиваем изучение основных функций Git, предназначенных для локальной работы. Теперь вы умеете создавать локальный репозиторий, фиксировать изменения, заводить и удалять ветки, объединять изменения из разных веток и даже разрешать простые конфликты.

Здесь перечислены основные изученные команды и их вариации:

Команда	**Применение**
`git init`	Создание пустого репозитория.
`git add <имя/маска файлов>`	Добавление файлов в репозиторий для дальнейшего отслеживания изменений.
`git rm  --cached <имя/маска файлов>`	Удаление файлов из репозитория. Сами файлы при

	этом не удаляются, только прекращается отслеживание изменений в них.
`git commit -a -m "Комментарий"`	Зафиксировать изменения во всех отслеживаемых файлах и добавить к версии комментарий.
`git commit <имя файла> -m "Комментарий"`	Зафиксировать изменения в конкретном файле с комментарием.
`git checkout -b <имя ветки>`	Создание новой ветки и переключение на неё.
`git checkout <имя ветки/хэш версии>`	Переключение на определённую ветку или версию.
`git branch <имя ветки>`	Создание новой ветки.
`git branch`	Вывод списка всех локальных веток, существующих в репозитории.
`git diff <имя ветки/хэш коммита 1> <имя ветки/хэш коммита 2>`	Вывод различий между ветками или версиями. Параметры <имя коммита 1> и <имя коммита 2> можно опускать, в этом случае берутся HEAD и «master».
`git log`	Вывод истории коммитов.
`git log --all --graph`	Вывод всего дерева Git со всеми ветками и зависимостями между ними.
`git status`	Вывод текущего состояния репозитория.
`git merge <имя ветки>`	Добавление изменений из ветки <имя ветки> в текущую активную ветку.
`git reset HEAD~1`	Вернуться к предыдущему коммиту.

Эти команды — далеко не все существующие. Полный список есть в руководстве **man git** или во встроенной справке Git (команда **git help**).

```
> git help

    usage: git [--version] [--help] [-C <path>] [-c name=value]
               [--exec-path[=<path>]] [--html-path] [--man-path]
               [--info-path] [-p | --paginate | --no-pager]
               [--no-replace-objects] [--bare] [--git-dir=<path>]
               [--work-tree=<path>] [--namespace=<name>]
               <command> [<args>]

    These are common Git commands used in various situations:

    start a working area (see also: git help tutorial)
       clone      Clone a repository into a new directory
       init       Create an empty Git repository or reinitialize
```

```
                an existing one

work on the current change (see also: git help everyday)
   add         Add file contents to the index
   mv          Move or rename a file, a directory, or a symlink
   reset       Reset current HEAD to the specified state
   rm          Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect      Use binary search to find the commit that introduced
               a bug
   grep        Print lines matching a pattern
   log         Show commit logs
   show        Show various types of objects
   status      Show the working tree status

grow, mark and tweak your common history
   branch      List, create, or delete branches
   checkout    Switch branches or restore working tree files
   commit      Record changes to the repository
   diff        Show changes between commits, commit and working
               tree, etc
   merge       Join two or more development histories together
   rebase      Reapply commits on top of another base tip
   tag         Create, list, delete or verify a tag object
               signed with GPG

collaborate (see also: git help workflows)
   fetch       Download objects and refs from another repository
   pull        Fetch from and integrate with another repository
               or a local branch
   push        Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

У каждой команды есть множество параметров (например, рассмотренные нами --graph и --all команды git log).

Чтобы узнать подробнее о параметрах конкретной команды и как её применять, обратитесь к расширенной справке **git help <имя команды>** (например, git help merge).

Рекомендуем тем, кто хочет «визуально» поработать с Git и закрепить свои знания, изучить интересный ресурс в сети Интернет.

Помощь