«Введение в информационные технологии»

ЛЕКЦИЯ 1. СИСТЕМА КОНТРОЛЯ ВЕРСИЙ GIT.

Цель и задачи курса:

Цель курса – изучение особенностей теоретических и методологических основ современных информационных систем.

Задачи курса:

приобретение теоретических знаний и практических навыков в области информационных технологий.

ЛИТЕРАТУРА:

1. Информатика. Базовый курс: учебник / под ред. С. В. Симоновича. - 3-е изд. - Москва: 2016. - 637 с. (11 экз.)
2. Информатика, автоматизированные информационные технологии и системы: учебник / В.А. Гвоздева. - Москва: Издательский Дом "ФОРУМ", 2021. – 542 с. http://znanium.com/catalog/document/?pid=1220288&id=368655
3. Информатика: учебник / В. А. Острейковский. - 5-е изд., стер. - Москва: Высшая школа, 2009. - 510 с. (14 экз.)
4. Информатика (курс лекций): учебное пособие / В.Т. Безручко. - 1. - Москва: Издательский Дом "ФОРУМ", 2020. – 432 с. http://znanium.com/catalog/document/?pid=1036598&id=344072
5. Информатика: учеб. для бакалавров / С.-Пб. гос. ун-т экономики и финансов; под ред. В. В. Трофимова. - Москва: Юрайт, 2012. - 910 с. (8 экз.)

ПЛАН ЛЕКЦИОННЫХ ЗАНЯТИЙ:

- Система контроля версий Git
- Типы данных, арифметические операции, имена, значения, переменные, управляющие конструкции языка Python
- Работа со строками и файлами, списками, срезы списков, функции и декораторы, коллекции, модули
- Множества, словари, операции со словарями
- Базы данных, основы реляционной алгебры, основные типы запросов
- Веб фреймворки
- Библиотеки для создания GUI

СОДЕРЖАНИЕ ЛЕКЦИИ:

- Система контроля версий.
- Понятия git систем.
- Модель ветвления.

1. Version Control System

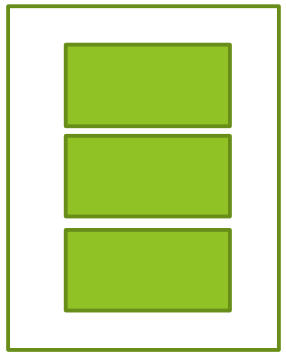
Система контроля версий (Version Control System)

программное обеспечение хранящее все версии возможного файла, и дающее возможность получить к ним доступ

системы

- ***git***
- *mercurial*
- *subversion(svn)*
- *Team Foundation server (TFS)*

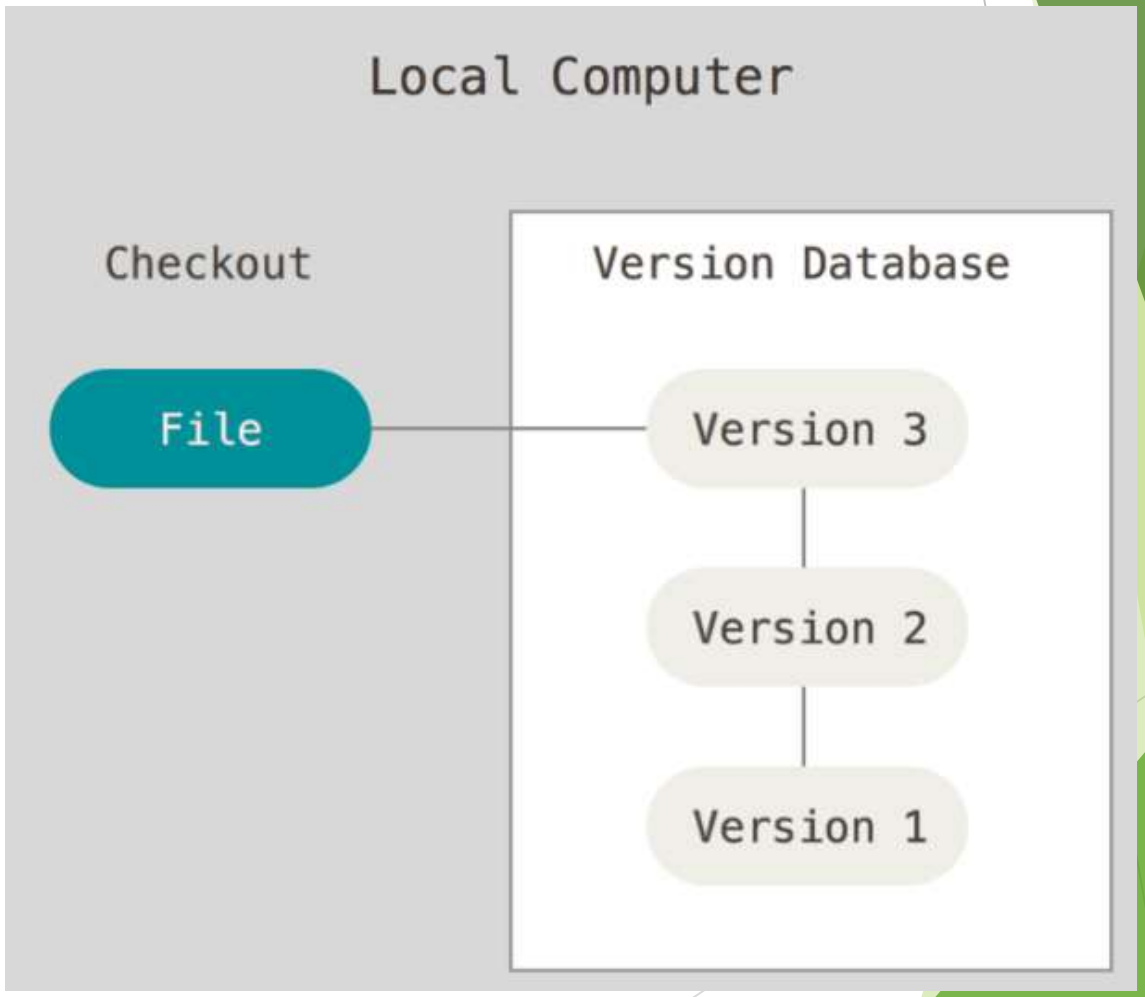
VCS



простые архивы кодов, которые разработчики передавали между собой, а если требовалось ***версионирование*** *- оно создавалось при помощи различных наименований файлов. При этом итоговая сборка проекта была сильно затруднена, поскольку никто из разработчиков по сути не знал над чем и как работает его коллега, и не был уверен в совместимости кода. Поэтому на итоговую компиляцию и исправление ошибок уходило значительное время*

Локальные системы контроля версий

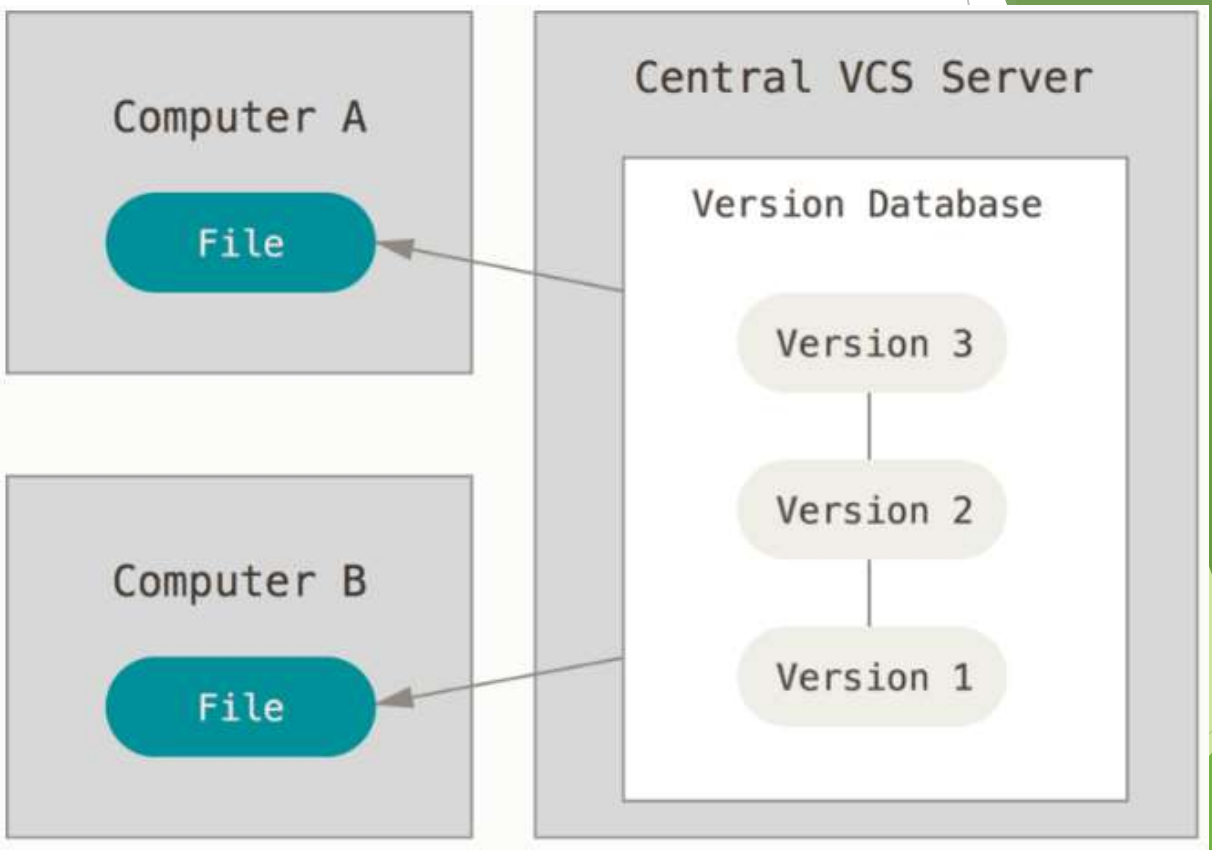
Локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий



Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени

Централизованные системы контроля версий

Взаимодействие с другими разработчиками



разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища

Недостатки:

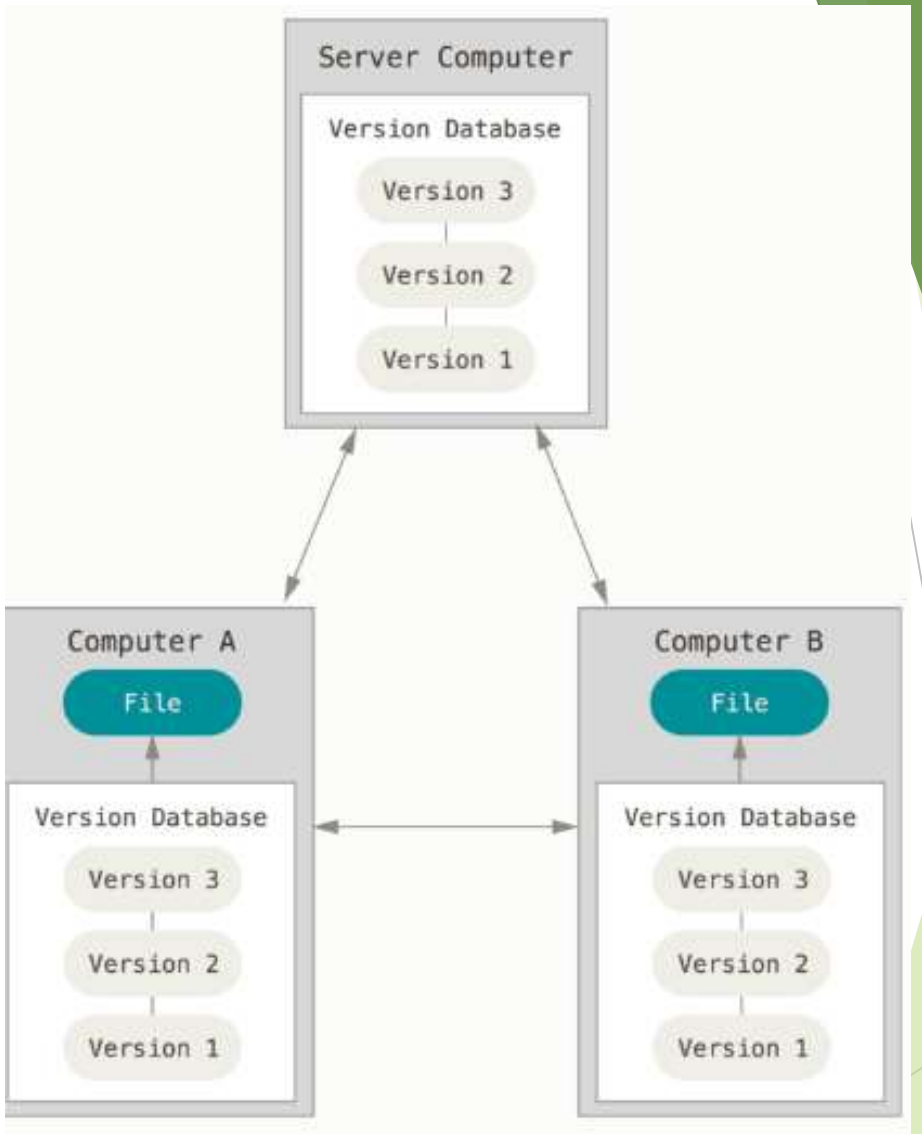
единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков

Распределённые системы контроля версий

В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий.

В этом случае, если один из серверов, через который разработчики обменивались данными, выйдет из строя, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы.

Каждая копия репозитория является полным бэкапом всех данных.



директории на жестком диске для хранения данных

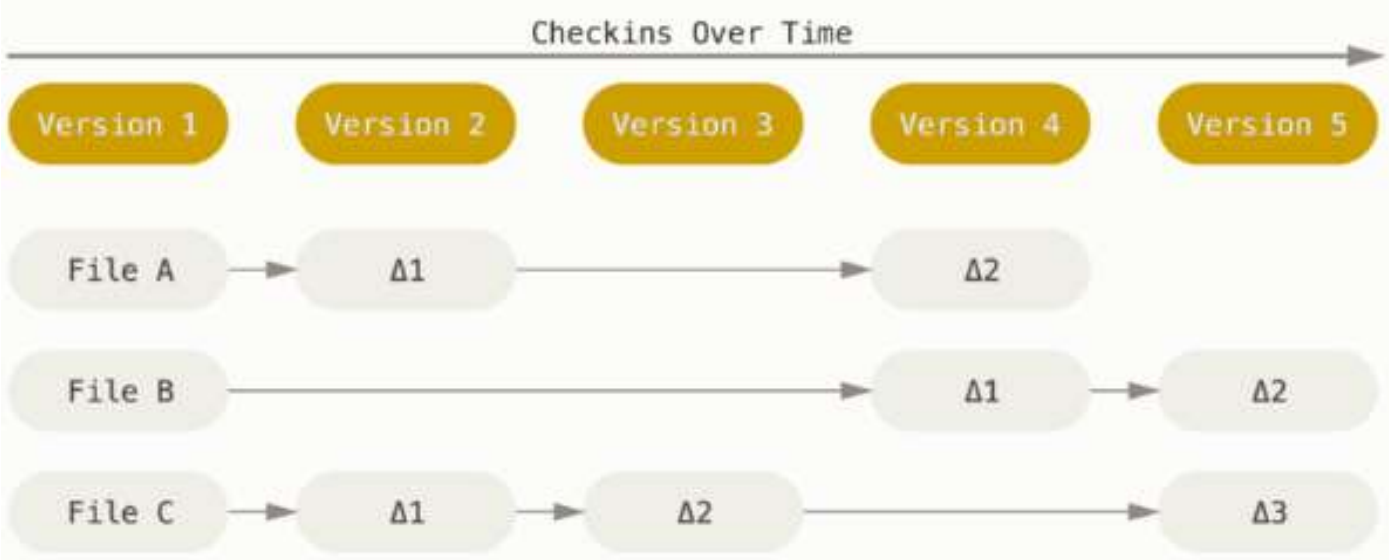
github и bitbucket

копия репозитория хранится онлайн

Цели:

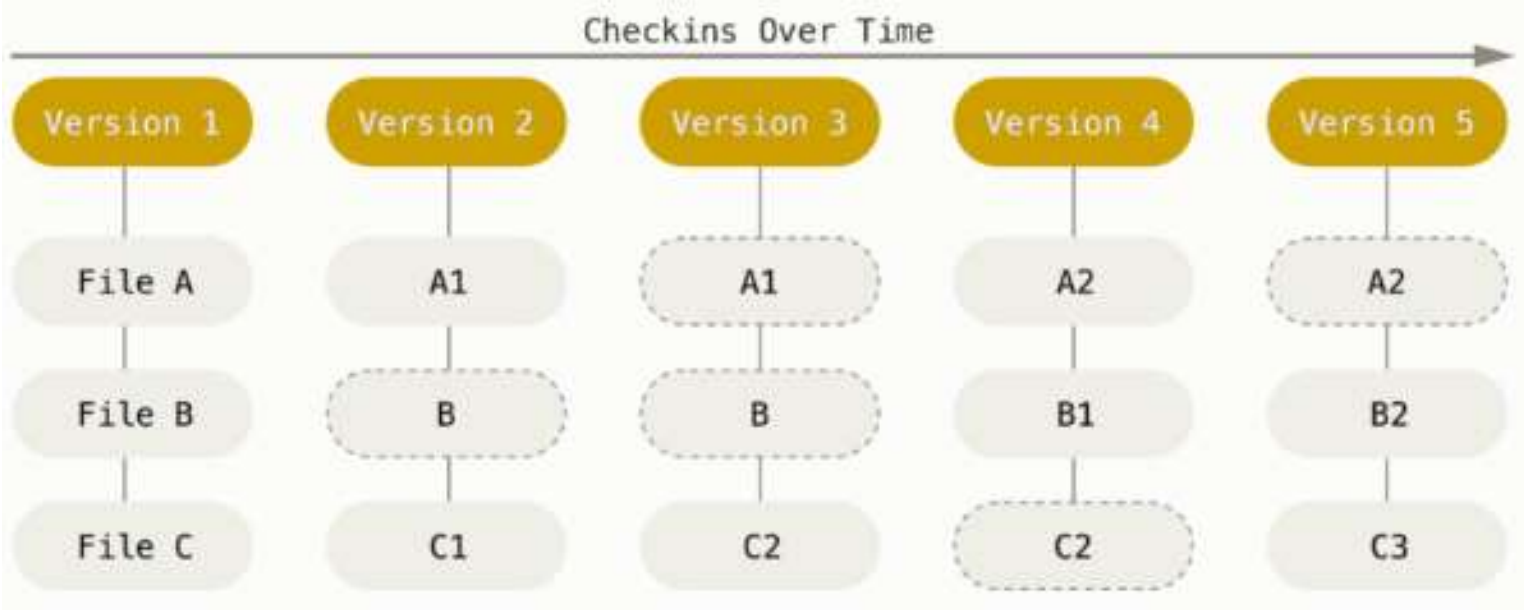
- *Скорость*
- *Простая архитектура*
- *Хорошая поддержка нелинейной разработки (тысячи параллельных веток)*
- *Полная децентрализация*
- *Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства)*

Большинство систем хранят информацию в виде списка изменений в файлах. Эти системы (CVS, Subversion, Perforce, Bazaar и т. д.) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, ***основанным на различиях)***



2. Понятия git систем

Подход Git к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, ***поток снимков***



Целостность Git

В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом

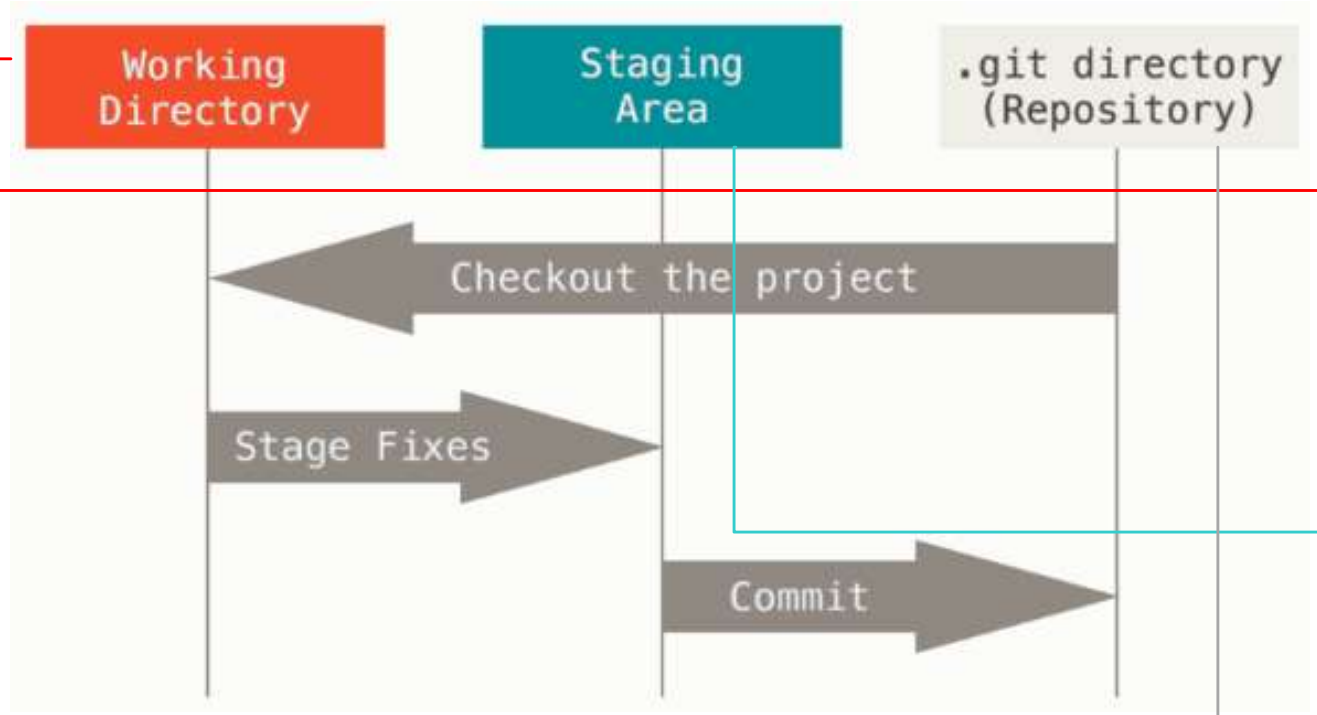
SHA-1 хеш

24b9da6552252987aa493b52f8696cd6d3b00373

40 шестнадцатеричных символов

Состояния Git:

- *изменён (modified)*
  - *•К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.*
- *индексирован (staged)*
  - *•Индексированный — это изменённый файл в его текущей версии, отмеченный для включения в следующий коммит.*
- *зафиксирован (committed)*
  - *•Зафиксированный значит, что файл уже сохранён в локальной базе*



рабочая копия, *область индексирования*, *каталог*

является снимком одной версии проекта. Эти файлы извлекаются из сжатой базы данных в каталоге Git и помещаются на диск, для того чтобы их можно было использовать или редактировать

это файл, обычно находящийся в каталоге Git, в нём содержится информация о том, что попадёт в следующий коммит. Её техническое название на языке Git — «индекс», но фраза «область индексирования» также работает

это то место, где Git хранит метаданные и базу объектов проекта. Это самая важная часть Git и это та часть, которая копируется при клонировании репозитория с другого компьютера

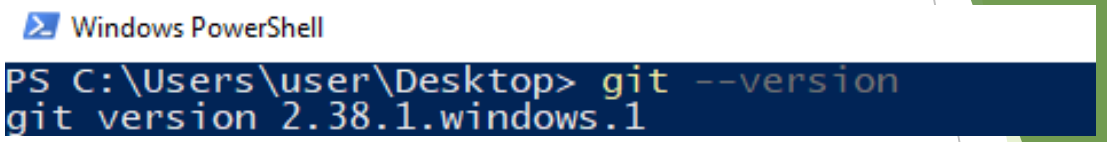
Действия:

Изменяете файлы рабочей копии.

Выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки только этих изменений в индекс

Когда делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в ваш каталог Git

Командная строка:



Linux *sudo dnf install git-all*

Mac *git --version*

Документация в различных форматах (doc, html, info), понадобится установить дополнительные зависимости:

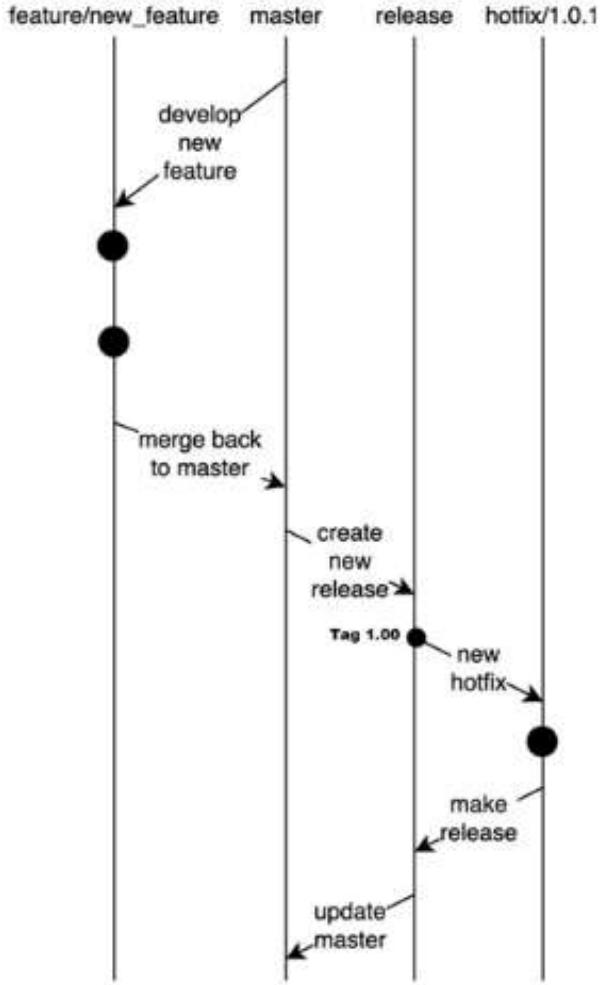
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x

Принципы	Команды
Репозиторий - обособленное хранилище кода внутри которого будут отслеживаться изменения файлов **Ветвь** (branch) - отдельная цепочка (ветка) отслеживаемых изменений **Мастер** branch - главная ветка в репозитории **Коммит** - зафиксированная точка изменений **Удаленный сервер** (remote server) - сервер на котором хранится репозиторий **Merge (слияние)** - процесс слияния двух ветвей **tag** - ссылка на определенный коммит	**git pull** - забрать изменения с удалённого сервера **git push** - отправить локальные изменения на удалённый сервер **git branch** - показать текущую ветку/список веток **git merge** <branch_name> - влить в текущую ветку указанную **git add** - добавить в текущее состояние изменённые файлы **git commit** - зафиксировать изменения и сделать коммит **git stash** - сохранить локальные изменения не создавая коммит и откатиться до состояния удалённого сервера

3. Модель ветвления

GitFlow

методология работы с Git



Существует master ветка в которой всегда находится рабочий код

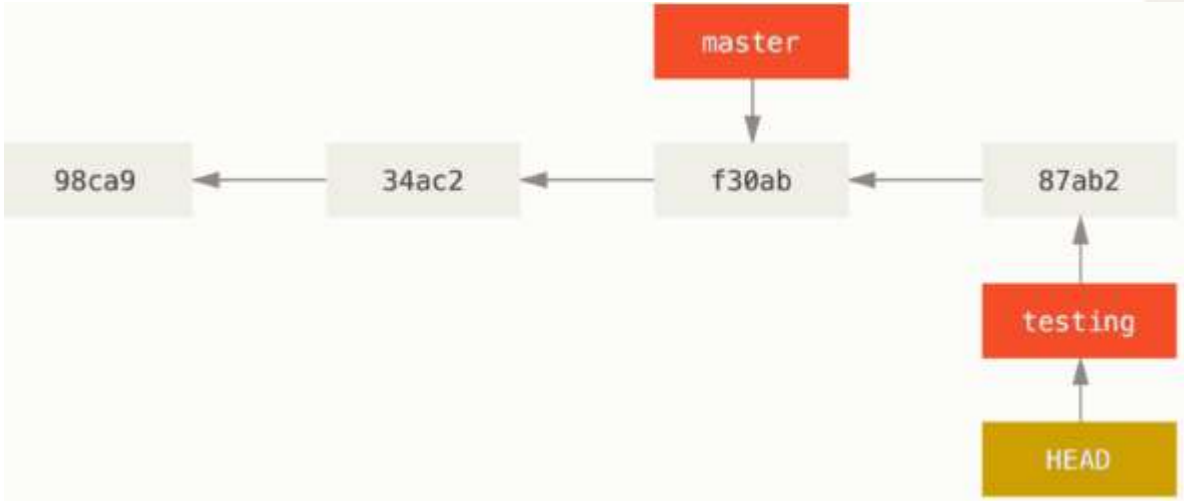
Из нее в нужные моменты создается релиз ветки, из которого собирается результирующий артефакт

Для разработки нового компонента используется feature-ветки которые создаются из мастера, проходят стадии разработки и затем вливаются обратно в мастер. (Опционально master ветка может быть заменена на dev ветку в которой собирается весь разрабатываемый код, и которая в свою очередь перед релизом вливается в master ветку)

Для создания срочных патчей создаются Ehotfix-ветки из релиза, разрабатываются и вливаются обратно в релиз, после чего релиз ветка вливается в мастер ветку

Переключение веток

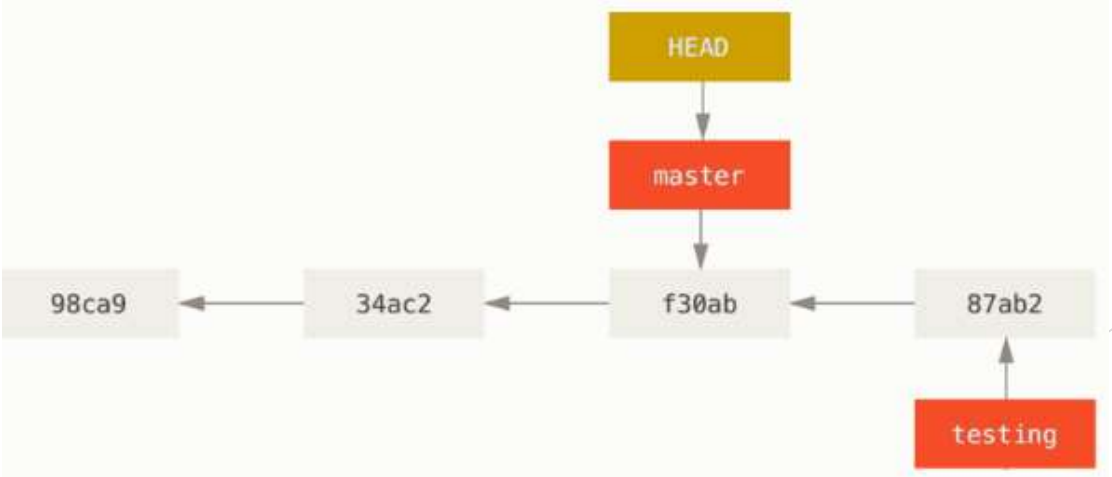
vim test.rb
git commit -a -m 'made a change'



git checkout testing

указатель на ветку testing переместился вперёд, а master указывает на тот же коммит, где были до переключения веток командой git checkout

git checkout master

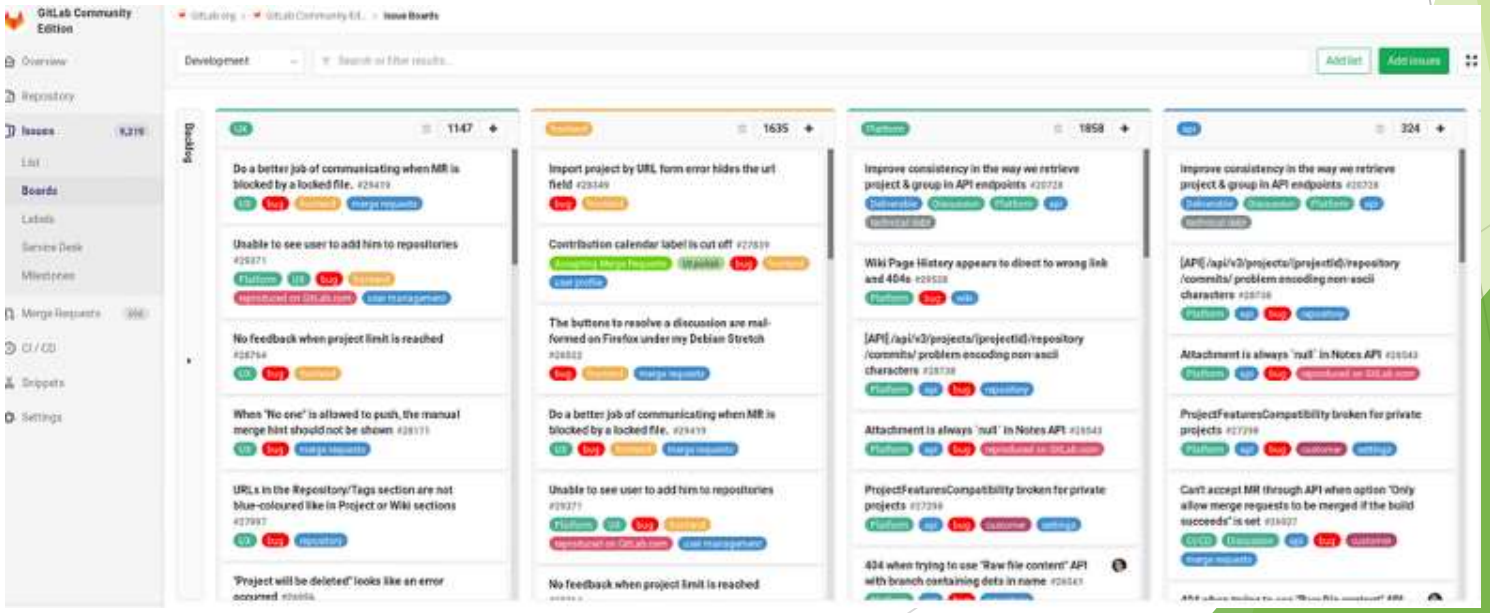


Github-подобные системы

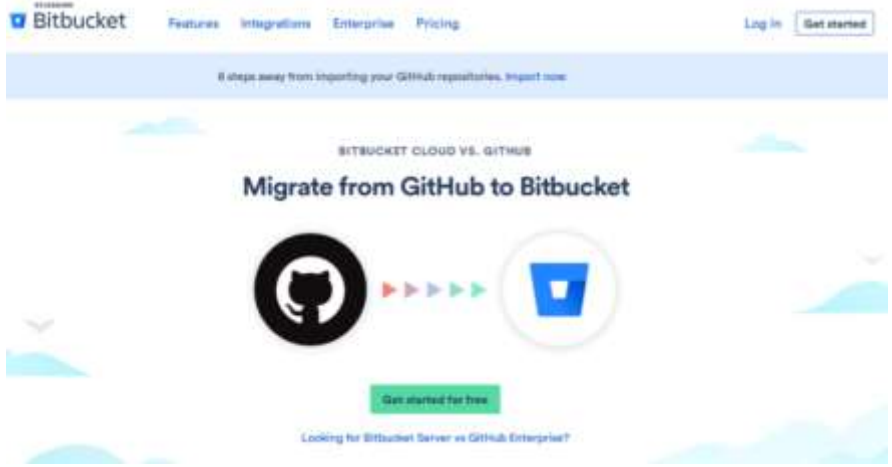
Github, gitlab, bitbucket, gitea, Azure Repos *и т.д. - это программные продукты которые предоставляют функционал удаленного сервера с веб-интерфейсом для управления git-репозиториями*

- *Размещение кода*
- *Создание **Issue** (описанных проблем)*
- *Создание Pull/Merge Request'ов - Issue с подготовленными ветками для слияния*
- *Wiki - база знаний основанная на разметке markdown*
- *В некоторых системах - CI-системы для сборки, тестирования и развертывания кода.*

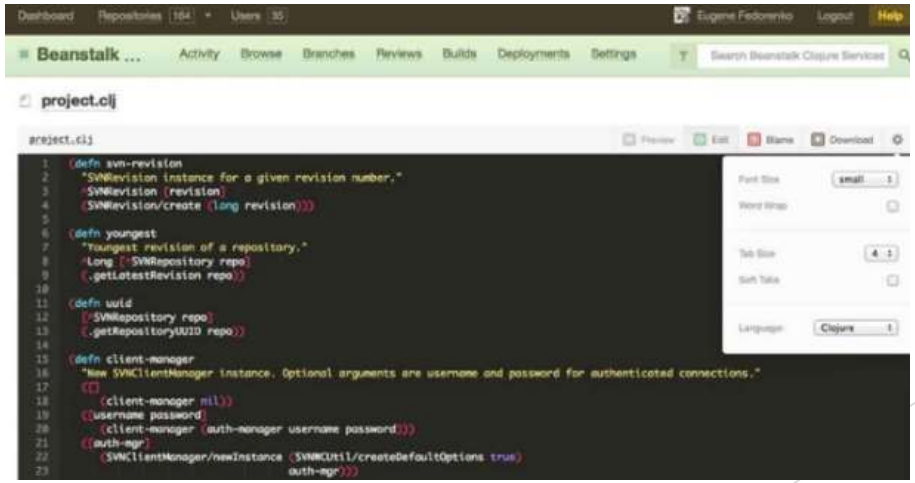
Платформа поддерживает учет рабочего времени, предоставляет мощные инструменты для разветвления, возможность защитить ветви и тэги, функции блокировки файлов, объединения запросов, персонализации уведомлений, создания дорожных карт проектов, выставления приоритетов по тикетам, создания конфиденциальных и связанных тикетов, графика выполнения работ по проектам и этапов milestones.



Bitbucket *мощная, полностью масштабируемая и высокопроизводительная платформа для разработчиков, предназначенная специально для профессиональных команд.*

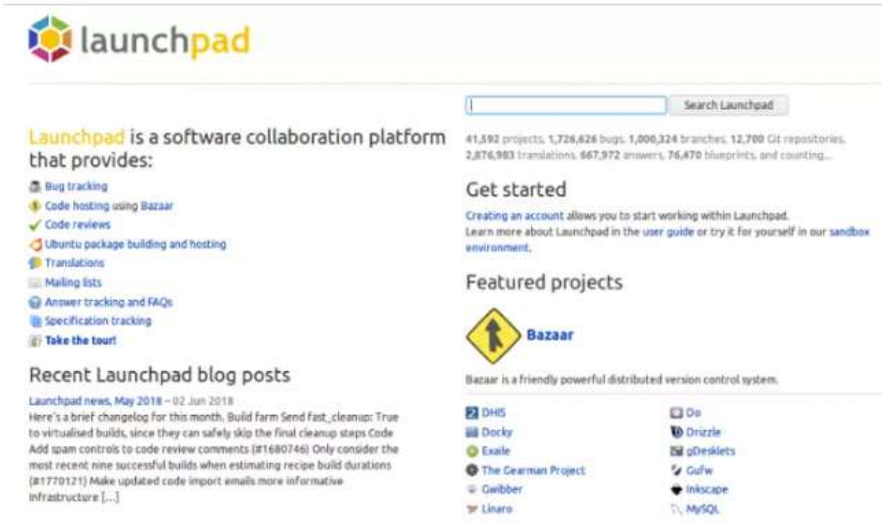


Beanstalk *предназначен для совершенствования процесса разработки с помощью такого функционала, как анализ кода, отслеживание ошибок, статистика хранилища, заметки, отправка уведомлений по электронной почте, дайджесты, сравнения, а также полная история по всем изменениям и файлам*



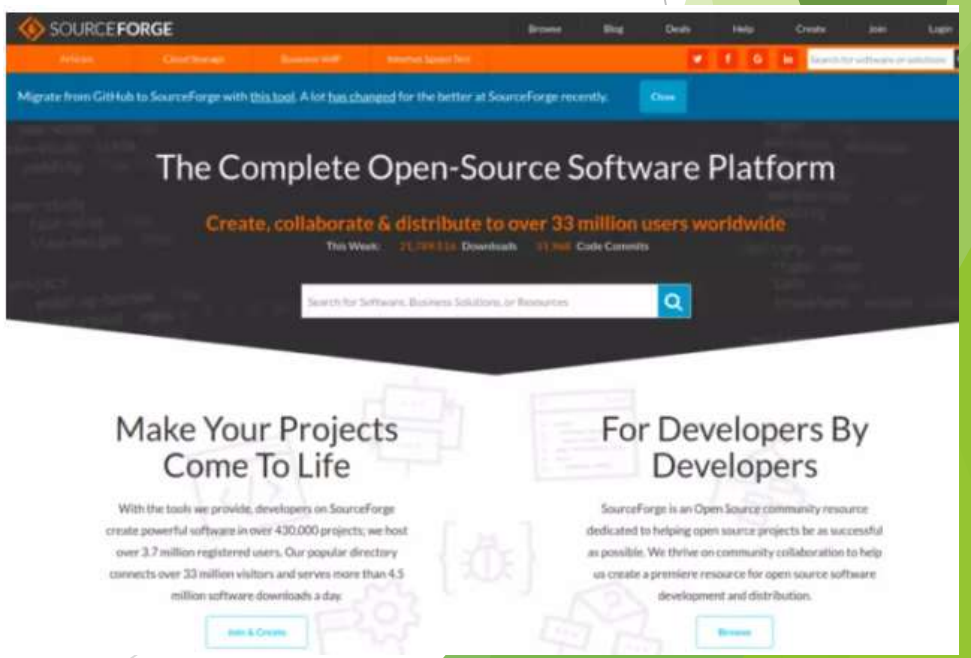
Launchpad

бесплатная, популярная платформа для создания, управления и совместной работы над проектами, созданная компанией Canonical — создателем Ubuntu Linux. Платформа предлагает следующий функционал: хостинг кода, создание пакетов Ubuntu, проверка кода, рассылка писем, отслеживание тикетов. Кроме того, Launchpad поддерживает переводы, позволяет отслеживать ответы и раздел «Вопросы и ответы»



SourceForge

создана на базе Apache Allura, и поддерживает любое количество частных проектов



Работа с Cit

Создание нового репозитория

```
    Каталог: C:\Users\user\Desktop\Desktop

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        23.10.2022     19:27                git_exercise
```

$ mkdir Desktop/git_exercise/
$ cd Desktop/git_exercise/
$ git init

```
PS C:\Users\user\Desktop>  cd Desktop/git_exercise/
PS C:\Users\user\Desktop\Desktop\git_exercise> git init
Initialized empty Git repository in C:/Users/user/Desktop/Desktop/git_exercise/.git/
PS C:\Users\user\Desktop\Desktop\git_exercise>
```

Реопзиторий создан, но пока пуст

Определение состояния

$ git status On branch master Initial commit Untracked files: (use "git add ..." to include in what will be committed) hello.txt

```
PS C:\Users\user\Desktop\Desktop\git_exercise> git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
PS C:\Users\user\Desktop\Desktop\git_exercise>
```

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

Подготовка файлов

git add hello.txt
Если нужно добавить все, что находится в директории git add –A

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached ..." to unstage)
new file: hello.txt
```

Работа с Cit

добавить пару новых блоков в html-разметку (index.html) и стилизовать их в файле style.css. Для сохранения изменений, их необходимо закоммитить. Но сначала, должны обозначить эти файлы для Гита, при помощи команды git add, добавляющей (или подготавливающей) их к коммиту. Добавлять их можно по отдельности:

Фиксация изменений

git add index.html
git add css/style.css

Все файлы

```
git add .
```

Убрать лишний

```
git reset:


git reset css/style.css
```

Создание коммита

git commit -m 'Add some code'

Флажок -m задаст commit message - комментарий разработчика. Он необходим для описания закоммиченных изменений.

Просмотр истории коммитов

git log
Отследить интересующие операции в списке изменений, можно по хэшу коммита, при помощи команды git show

```
git show hash_commit
```

Удаленные репозитории

Подключение к удаленному репозиторию

$ git remote add origin https://github.com/tutorialzine/awesome-project.git

Чтобы связать локальный репозиторий с репозиторием на GitHub, выполнить следующую команду в терминале. Необходимо указать свой URI репозитория

Отправка изменений на сервер

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

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/tutorialzine/awesome-project.git
* [new branch] master -> master
```

git push - команда выгрузки, а git pull и git fetch - команда загрузки или скачивания. После этого необходимо внедрить или интегрировать, при помощи команды слияния git merge.

Запрос изменений с сервера

```
$ git pull origin master
From https://github.com/tutorialzine/awesome-project
* branch master -> FETCH_HEAD
Already up-to-date.
```

пользователи могут скачать изменения при помощи команды pull

Ветвление

Создание новой ветки

```
$ git branch amazing_new_feature
```

Основная ветка в каждом репозитории называется master.
Чтобы создать еще одну ветку, используется команда branch <name>

Переключение между ветками

если запустить branch, увидим две доступные опции, master — это активная ветка. Для переключения используется команда checkout, она принимает один параметр — имя ветки, на которую необходимо переключиться

```
$ git branch
amazing_new_feature
* master
```

```
$ git checkout amazing_new_feature
```

Слияние веток

```
$ git add feature.txt
$ git commit -m "New feature complete."
```

если нужна только определенная его ветка, а не все хранилище - после git clone необходимо выполнить следующую команду в соответствующем репозитории: git checkout -b <имя ветки> origin/<имя ветки>

```
$ git checkout master
 $ git merge amazing_new_feature
$ git branch -d awesome_new_feature
```

Удаление веток

при удалении ветвей, необходимо переключиться на другой branch

```
git branch -d local_branch_name
```

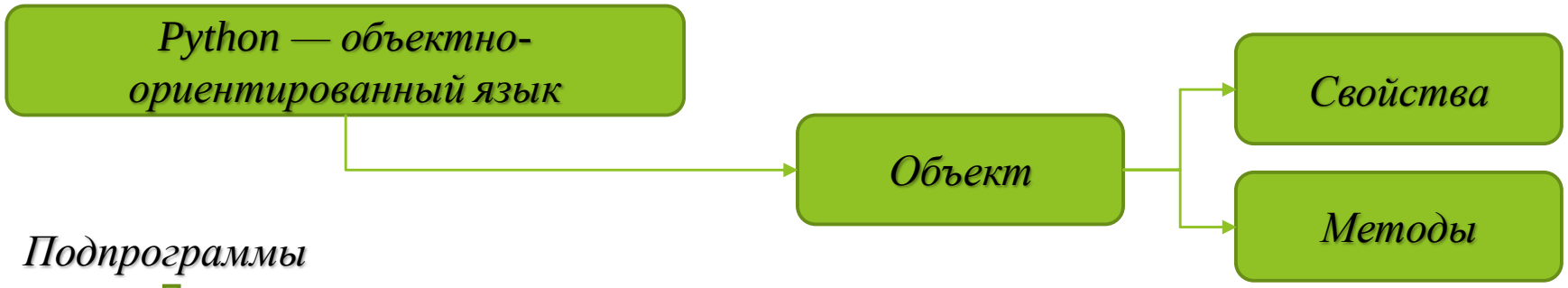
«Введение в информационные технологии»

ЛЕКЦИЯ 2. ТИПЫ ДАННЫХ, АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ, ИМЕНА, ЗНАЧЕНИЯ, ПЕРЕМЕННЫЕ, УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА PYTHON.

СОДЕРЖАНИЕ ЛЕКЦИИ:

- Понятие типа данных и переменной.
- Поток команд (управляющие структуры).

1. Понятие типа данных и переменной



Подпрограммы

Функции

Функции, оформленные по определённым правилам, объединяются в модули (библиотеки функций). Модули (или некоторые функции из модулей) по мере необходимости подключаются к пользовательским программам. Такой подход позволяет экономить память вычислительной системы и не занимать её ненужным кодом.

Модули подключаются в начале программы с помощью команды:
import *имя_модуля*
А отдельные функции — с помощью команды:
from *имя_модуля* **import** *функция1, . . . функцияN*

В Python отсутствуют "операторные скобки" типа begin ... end или DO ... LOOP. Вместо них в составных операторах (ветвления, циклы, определения функций) используются отступы от начала строки (пробелы)

Типы данных

Числа в Python могут быть обычными целыми (тип int), длинными целыми (тип long), вещественными (тип float) и комплексными

Для преобразования чисел из вещественных в целые и наоборот в Python определены функции ***int() и float().*** *Например, int(17.6) даст в результате 17, а float(15) даёт в результате 15.0*

Операция	*Описание*
x + y	*Сложение (сумма x и y)*
x - y	*Вычитание (разность x и y)*
*x * y*	*Умножение (произведение x и y)*
x/y	*Деление x на y (частное).* ***Внимание!*** *Если x и y целые, то результат всегда будет целым числом! Для получения вещественного результата хотя бы одно из чисел должно быть вещественным. Пример:100/8 → 12, а вот100/8.0 → 12.5*
x//y	*Целочисленное деление (результат — целое число). Если оба числа в операции вещественные, получается вещественное число с дробной частью, равной нулю. Пример:100//8 → 12 101.8//12.5 → 8.0 (для сравнения101.8/12.5 → 8.1440000000000001)*
x%y	*Остаток от целочисленного деления x на y Пример:10%4 → 2*
*x**y*	*Возведение в степень (x в степени y). Работает и для вещественных чисел. Примеры:2**3 → 8 2.3**(-3.5) → 0.05419417057580235*
-x	*Смена знака числа*

Для операций с числами используются функции abs() (вычисление абсолютного значения — модуля, abs(-3)→ 3), pow() (возведение в степень, pow(2,3) → 8), divmod() (вычисление результата целочисленного деления и остатка, divmod(17,5)→ (3,2)) и round() (округление, round(100.0/6) → 17.0). Эти функции являются "встроенными", что означает, что для их использования нет необходимости подключать дополнительные модули.

Все прочие функции для работы с числами (математические), такие как вычисление квадратного корня, синуса и пр. требуют подключения модуля math

Логические значения

Операция или выражение	Описание
>	Условие "больше" (например, проверяем, что a > b)
<	Условие "меньше" (например,проверяем, что a < b)
==	Условие равенства (проверяем, что a равно b)
!=	Условие неравенства (проверяем, что a не равно b)
not x	Отрицание (условие x не выполняется)
x and y	Логическое "И" (умножение). Чтобы выполнилось условие x and y, необходимо, чтобы одновременно выполнялись условия x и y.
x or y	Логическое "ИЛИ" (сложение). Чтобы выполнилось условие x or y, необходимо, чтобы выполнилось одно из условий.
x in A	Проверка принадлежности элемента x множеству (структуре) A (см. "Структуры данных").
a < x < b	Эквивалентно (x > a) and (x < b)

Структуры данных

позволяют устанавливать связи (ассоциации) "ключ—значение" (например, "Фамилия—Адрес"), поэтому с их помощью создаются так называемые ассоциативные массивы

подразделяются на изменяемые и неизменяемые. Под изменяемостью (изменчивостью) последовательности понимается возможность добавлять или удалять элементы этой последовательности (т.е. изменять количество элементов последовательности).

Строки (последовательности символов — букв и других значков, которые можно найти на клавиатуре компьютера) могут состоять из символов английского и любого другого алфавита

Структуры данных

последовательности

подразделяются на изменяемые и неизменяемые. Под изменяемостью (изменчивостью) последовательности понимается возможность добавлять или удалять элементы этой последовательности (т.е. изменять количество элементов последовательности).

Числа могут быть преобразованы в строки с помощью функции str().
Например, str(123) даст строку '123'. Если строка является последовательностью знаков-цифр, то она может быть преобразована в целое число в помощью функции int() (int('123') даст в результате число 123), а в вещественное — с помощью функции float() (float('12.34') даст в результате число 12.34).
Для любого символа можно узнать его номер (код символа) с помощью функции ord() (например, ord('s') даст результат 115).
И наоборот, получить символ по числовому коду можно с помощью функции chr() (например chr(100) даст результат 'd').

Операции со строками

Функция или операция	Описание и результат
`len(s)`	Вычисляется длина строки `s` как число символов
`s1 + s2`	Конкатенация. К концу строки `s1` присоединяется строка `s2`, в результате получается новая строка, например, `'вы' + 'года' → 'выгода'`
`s * n` (или `n * s`)	`n` -кратное повторение строки `s`, в результате получается новая строка, например `'кан'*2 → 'канкан'`

`s[i]`	Выбор из `s` элемента с номером `i`, нумерация начинается с 0 (первый элемент имеет номер 0). Результатом является символ. Если $i < 0$, отсчёт идёт с конца (первый символ строки имеет номер 0, последний имеет номер -1). Пример:
	`s='дерево'`
	`s[2] → 'р'`
	`s[-2] → 'в'`

`s[i : j :k]`	Срез — подстрока, содержащая символы строки `s` с номерами от i до j с шагом k (элемент с номером i входит в итоговую подстроку, а элемент с номером j уже не входит). Если k не указан (использован вариант `s[i : j]`), то символы идут подряд (равносильно `s[i : j :1]`). Примеры:
	`s='derevo'`
	`s[3:5] → 'ev'`
	`s[1:5:2] → 'ee'`

`min(s)`	Определяет и выводит (возвращает) символ с наименьшим значением (кодом – номером в кодовой таблице) Пример:
	`s='derevo'`
	`min(s) → 'd'`

`max(s)`	Возвращает символ с наибольшим значением (кодом) Пример:
	`s='derevo'`
	`max(s) → 'v'`

Методы строк

Метод	Описание и результат

`s1.center(n)`	Возвращается строка `s1`, дополненная пробелами справа и слева до ширины в n символов. Исходная строка не изменяется. Если $n \leq len(s1)$, пробелы не добавляются. Пример:
	`s1='Zoom-Zoom'`
	`s1.center(15) → '  Zoom_Zoom  '`

`s1.ljust(n)`	Строка `s1` выравнивается по левому краю (дополняется пробелами справа) в пространстве шириной n символов. Если $n < len(s1)$, пробелы не добавляются. Пример:
	`s1='Zoom_Zoom'`
	`s1.ljust(15) → 'Zoom-Zoom      '`

`s1.rjust(n)`	Строка `s1` выравнивается по правому краю (дополняется пробелами слева) в пространстве шириной n символов. Если n < len(s1), пробелы не добавляются. Пример:
	`s1='Zoom_Zoom'`
	`s1.rjust(15) → '      Zoom-Zoom'`

`s1.count(s[, i, j])`	Определяется количество вхождений подстроки `s` в строку `s1`. Результатом является число. Можно указать позицию начала поиска i и окончания поиска j (по тем же правилам, что и начало и конец среза). Примеры:
	`s1='abrakadabra'`
	`s1.count('ab') → 2`
	`s1.count('ab',1) → 1`
	`s1.count('ab',1,-3) → 0, потому что`
	`s1[1:-3] → 'brakada'`

`s1.find(s[, i, j])`	Определяется позиция первого (считая слева) вхождения подстроки `s` в строку `s1`. Результатом является число. Необязательные аргументы i и j определяют начало и конец области поиска (как в предыдущем случае). Пример:
	`s1='abrakadabra'`
	`s1.find('br') → 1`

`s1.find(s[, i, j])`	Определяется позиция первого (считая слева) вхождения подстроки `s` в строку `s1`. Результатом является число. Необязательные аргументы i и j определяют начало и конец области поиска (как в предыдущем случае). Пример:
	`s1='abrakadabra'`
	`s1.find('br') → 1`

`s1.rfind(s[, i, j])`	Определяется позиция последнего (считая слева) вхождения подстроки `s` в строку `s1`. Результатом является число. Необязательные аргументы i и j определяют начало и конец области поиска (как в предыдущем случае). Пример:
	`s1='abrakadabra'`
	`s1.rfind('br') → 8`

`s1.strip()`	Создаётся копия строки, в которой удалены пробелы в начале и в конце (если они есть или образовались в результате каких-то операций). Пример:
	`s1='  breKeKeKeKs  '`
	`s2=s1.strip()`
	`s2 → 'breKeKeKeKs'`

`s1.lstrip()`	Создаётся копия строки, в которой удалены пробелы в начале (если они есть или образовались в результате каких-то операций). Пример:
	`s1='  breKeKeKeKs  '`
	`s2=s1.lstrip()`
	`s2 → 'breKeKeKeKs  '`

`s1.rstrip()`	Создаётся копия строки, в которой удалены пробелы в конце (если они есть или образовались в результате каких-то операций). Пример:
	`s1='  breKeKeKeKs  '`
	`s2=s1.rstrip()`
	`s2 → '  breKeKeKeKs'`

Кортежи

упорядоченный набор объектов, в который могут одновременно входить объекты разных типов (числа, строки и другие структуры, в том числе и кортежи)

a =(32, 'b', 37.6, 'derevo ')

a=(x, s1, y, s2) =(32, 'b', 37.6, 'derevo ')

a [0]=x…

Функция или операция	Описание и результат
`len(t)`	Определяется количество элементов кортежа (результатом является число) `t`

`t1 + t2`	Объединение кортежей. Получается новый кортеж, в котором после элементов кортежа `t1` находятся элементы кортежа `t2` . Пример:
	`t1=(1,2,3)`
	`t2=('raz', 'dva')`
	`t3=t1+t2`
	`t3 → (1, 2, 3, 'raz', 'dva')`

`t * n` или `n * t`	n-кратное повторение кортежа `t` Пример:
	`t2=('raz', 'dva')`
	`t2*3 → ('raz','dva', 'raz', 'dva', 'raz', 'dva')`

`t[i]`	Выбор из `t` элемента с номером i , нумерация начинается с 0 (первый элемент имеет номер 0) Если $i < 0$, отсчёт идёт с конца (первый элемент кортежа имеет номер 0, последний имеет номер -1). Пример:
	`t3=(1, 2, 3, 'raz', 'dva')`
	`t3[2] → 3`
	`t3[-2] → 'raz'`

`t[i : j :k]`	Срез — кортеж, содержащий элементы кортежа `t` с номерами от i до j с шагом k (элемент с номером i входит в итоговый кортеж, а элемент с номером j уже не входит). Если k не указан (использован вариант `t[i : j]`), то элементы идут подряд (равносильно `t[i : j :1]`). Пример:
	`t3=(1, 2, 3, 'raz', 'dva')`
	`t3[1:4] → (2, 3, 'raz')`

`min(t)`	Определяется элемент с наименьшим значением в соответствии с алфавитным ("словарным") порядком. Пример:
	`t3=(1, 2, 3, 'raz', 'dva')`
	`min(t3) → 1`

`max(t)`	Определяется элемент с наибольшим значением в соответствии с алфавитным ("словарным") порядком. Пример:
	`t3=(1, 2, 3, 'raz', 'dva')`
	`max(t3) → 'raz'`

Изменяемые последовательности — списки

упорядоченный набор объектов, в список могут одновременно входить объекты разных типов (числа, строки и другие структуры, в частности, списки и кортежи)

`lst=[12, 'b', 34.6, 'derevo ']`

`lst=[x, s1, y, s2]=[12, 'b', 34.6, 'derevo ']`

элемент списка и соответствующая переменная будут указывать на одни и те же значения, т.е. значение lst[0] будет равно значению x,, а lst[3]. соответственно, s2.
Однако эти переменные могут изменяться независимо от элементов списка. Присвоение нового значения переменной s1 никак не влияет на элемент lst[1]. В отличие от кортежа, значения элементов списка можно изменять, добавлять элементы в список и удалять их.

Операции со списками

Функция или операция	Описание и результат
`len(lst)`	Определяется количество элементов списка `lst` . Результат — число.

`lst1+ lst2`	Объединение списков. Получается новый список, в котором после элементов списка `lst1` находятся элементы списка `lst2` . Пример:
	`lst1=[1,2,3]`
	`lst2=['raz', 'dva']`
	`lst3=lst1+lst2`
	`lst3→[1, 2, 3, 'raz', 'dva']`

`lst*n` (или `n* lst`)	`n` -кратное повторение списка `lst` . Результат — новый список. Пример:
	`lst2=['raz', 'dva']`
	`lst2*3 →['raz','dva', 'raz', 'dva', 'raz', 'dva']`

`lst[i]`	Выбор из `lst` элемента с номером i , нумерация начинается с 0 (первый элемент имеет номер 0) Если $i < 0$, отсчёт идёт с конца (последний элемент списка имеет номер -1). Пример:
	`lst3=[1, 2, 3, 'raz', 'dva']`
	`lst3[2] → 3`
	`lst3[-2] → 'raz'`

`max(lst)`	Определяется элемент с наибольшим значением в соответствии с алфавитным ("словарным") порядком. Пример:
	`lst3=[1, 2, 3, 'raz', 'dva']`
	`max(lst3) → 'raz'`

`lst[i]=x`	Замена элемента списка с номером i на значение `x` . Если `x` является списком, то на место элемента списка будет вставлен список. При этом новый список не создаётся. Примеры:
	`lst3=[1, 2, 3, 'raz', 'dva']`
	`lst3[2]='tri'`
	`lst3→[1, 2, 'tri', 'raz', 'dva']`
	`lst3[2]=[7,8]`
	`lst3→[1, 2,[7, 8], 'raz', 'dva']`

`del lst[i]`	Удаление из списка элемента с номером i . Новый список не создаётся. Пример:
	`lst3=[1, 2,[7, 8], 'raz', 'dva']`
	`del lst3[2]`
	`lst3→[1, 2, 'raz', 'dva']`

Методы списков

Метод	*Описание и результат*
lst.append(x)	*Добавление элемента x в конец списка lst. x не может быть списком. Создания нового списка не происходит. Пример:lst=['raz','dva','tri',1,2] lst.append(3) lst→['raz','dva','tri',1,2,3]*
lst.extend(t)	*Добавление кортежа или списка t в конец списка lst (похоже на объединение списков, но создания нового списка не происходит). Пример:lst1=[1,2,3] lst2=['raz','dva'] lst1.extend(lst2) lst1→[1,2,3,'raz','dva']*
lst.count(x)	*Определение количества элементов, равных x, в списке lst. Результат является числом. Пример:lst=[1,2,3,'raz','dva','raz','dva'] lst.count('raz') → 2*
lst.index(x)	*Определение первой слева позиции элемента x в списке lst. Если такого элемента нет, возникает сообщение об ошибке. Пример:lst=[1,2,3,'raz','dva','raz','dva'] lst.index('dva') → 4*
lst.remove(x)	*Удаление элемента x в списке lst в первой слева позиции. Если такого элемента нет, возникает сообщение об ошибке. Пример:lst=[1,2,3,'raz','dva','raz','dva'] lst.remove('dva') lst→[1,2,3,'raz','raz','dva']*
lst.pop(i)	*Удаление элемента с номером i из списка lst. При этом выдаётся значение этого элемента ("извлечение" элемента из списка). Если номер не указан, удаляется последний элемент. Новый список не создаётся. Примеры:lst=[1,2,3,'raz','raz','dva'] \ lstinline lst.pop(3)| → 'raz'| lst→[1,2,3,'raz','dva'] \ lstinline lst.pop()| → 'dva'| lst→[1,2,3,'raz']*
lst.insert(i,x)	*Вставка элемента или списка x в позицию i списка lst. Если i<=0, вставка идёт в начало списка. Если i>len(lst), вставка идёт в конец списка. Новый список не создаётся. Пример:lst=[1,2,3,'raz'] lst.insert(3,'tri') lst→[1,2,3,'tri','raz']*
lst.sort()	*Сортировка списка по возрастанию (в алфавитном порядке). Новый список не создаётся. Пример:lst=[1,2,3,'tri','raz'] lst.sort() lst→[1,2,3,'raz','tri']*
lst.reverse()	*Замена порядка следования элементов на обратный. Новый список не создаётся. Пример:lst=[1,2,3,'raz','tri'] lst.reverse() lst→['tri','raz',3,2,1]*

2. Поток команд (управляющие структуры)

Операторы:

- `if` : условное ветвление;
- `while` : цикл с условием;
- `for` : совместные циклы (циклы по коллекциям).

```
if logical_expression_1:     # (if строго 1) условие для проверки
    suite_1                  # блок команд для выполнения, если условие истинно
elif logical_expression_2:   # (elif - 0 или несколько) дополнительные условия
    suite_2                  # проверка идет, если условия выше ложны
elif logical_expression_N:
    suite_N
else:                        # (else - 0 или 1) блок команд для выполнения,
    else_suite               # если все условия выше оказались ложными
```

```
while logical_expression:  # Если условие истинно, выполняется 'while_suite'
    while_suite            # После выполнения, происходит возврат к проверке
else:                      # (else - 0 или 1)
    else_suite             # Выполняется, если не было прерывания цикла
```

```
for expression in iterable:  # Для каждого элемента 'expression' из 'iterable'
    for_suite                # выполняется 'for_suite'
else:                        # (else - 0 или 1)
    else_suite               # Выполняется, если не было прерывания цикла
```

Функции итераторов

all(*iterable*)

Возвращает `True`, если все элементы `iterable` в логическом контексте оцениваются как `True`.

any(*iterable*)

Возвращает `True`, если хотя бы 1 элемент `iterable` в логическом контексте оцениваются как `True`.

enumerate(*iterable, start=0*)

Возвращает итератор, где каждый элемент является парой «номер» - «значение». Номер отсчитывается от `start`. Обычно используется в циклах `for`, чтобы получить последовательность кортежей `(номер, элемент)`.

sorted(*iterable, key=None, reverse=False*)

Возвращает отсортированный объект в виде списка для итерируемого объекта `iterable`.

Параметры:

- **key** – функция сортировки (по умолчанию не учитывается, сортировка осуществляется поэлементно);
- **reverse** – если равен `True`, сортировка осуществляется в обратном порядке.

reversed(*iterable*)

Возвращает итератор, возвращающий элементы в обратном для исходного `iterable` порядке.

```
>>> fruits = ["апельсины", "ябоки", "мандарины", ""]

>>> any(fruits)
True

>>> all(fruits)  # False (пустая строка == False)
False

>>> list(reversed(fruits))
['', 'мандарины', 'ябоки', 'апельсины']

>>> list(enumerate(fruits, start=1))
[(1, 'апельсины'), (2, 'ябоки'), (3, 'мандарины'), (4, '')]
```

Прерывание и продолжение циклов

break, continue

Не зависимо от способа прерывания цикла, механизм действует одинаково - выполнение цикла прерывается, дополнительная часть цикла else не выполняется и осуществляется выход за пределы цикла.

Для продолжения цикла используется команда continue, передающая управление в начало цикла для выполнения следующей проверки или итерации.

```
nums = [1, 6, 8, 2, 9, 3, 5]

for x in nums:
    if x % 2 == 0:
        print("Число найдено: {}".format(x))
        break
else:
    print("Четное число не найдено!")
```

```
# Поиск простых чисел от 1 до 10
#
# Внешний цикл отвечает за перебор чисел от 1 до 10
# Внутренний цикл подсчитывает количество делителей, перебирая
# все числа от 2 до текущего числа - 1
#
# Данная задача выглядит проще, если решить ее с использованием 2 циклов for,
# данный вариант приведен для демонстрации вложенности разных видов циклов

for i in range(1, 11):
    divisors = 0
    j = 2
    # Внутренний цикл выполняется пока не переберутся все числа [2; i] или
    # не найдется хотя бы 1 делитель
    while j < i and divisors == 0:
        if i % j == 0:
            divisors += 1

        j += 1

    # Если делители для [i] не найдены - это простое число
    if divisors == 0:
        print(i, end=" ")

# -------------
# Пример вывода:

# 1 2 3 5 7
```