GIT:

1. Создание репозитория, первоначальная настройка
2. Создание коммитов, переключение между коммитами
3. Создание веток, переключение между ветками
4. Работа с удаленным репозиторием

Python:

1. Типы данных в Python
2. Циклы и логические операторы
3. Функции, области видимости
4. Контейнеры (массивы)
5. Модули Numpy, Pandas, Matplotlib
6. Объектно-ориентированное программирование

**GIT**

**Создание репозитория**

Если у вас уже есть проект в каталоге, который не находится под версионным контролем Git, то для начала нужно перейти в него.

$ cd C:/Users/user/my\_project

а затем выполните команду:

$ git init

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git, содержащий все необходимые файлы репозитория — структуру Git репозитория. На этом этапе ваш проект ещё не находится под версионным контролем.

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду git add несколько раз, указав индексируемые файлы, а затем выполнив git commit:

$ git add \*.c

$ git add LICENSE

$ git commit -m 'Initial project version'

Клонирование репозитория осуществляется командой git clone <url>. Например, если вы хотите клонировать библиотеку libgit2, вы можете сделать это следующим образом:

$ git clone https://github.com/libgit2/libgit2

Эта команда создаёт каталог libgit2, инициализирует в нём подкаталог .git, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы перейдёте в только что созданный каталог libgit2, то увидите в нём файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в каталог с именем, отличающимся от libgit2, необходимо указать желаемое имя, как параметр командной строки:

$ git clone https://github.com/libgit2/libgit2 mylibgit

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mylibgit.

**Первоначальная настройка**

Теперь, когда Git установлен в вашей системе, самое время настроить среду для работы с Git под себя. Это нужно сделать только один раз — при обновлении версии Git настройки сохранятся. Но, при необходимости, вы можете поменять их в любой момент, выполнив те же команды снова.

В состав Git входит утилита git config, которая позволяет просматривать и настраивать параметры, контролирующие все аспекты работы Git, а также его внешний вид. Эти параметры могут быть сохранены в трёх местах:

1. Файл [path]/etc/gitconfig содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл. Так как этот файл является системным, то вам потребуются права суперпользователя для внесения изменений в него.
2. Файл ~/.gitconfig или ~/.config/git/config хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global и применяется ко **всем** репозиториям, с которыми вы работаете в текущей системе.
3. Файл config в каталоге Git (т. е. .git/config) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория. Вы можете заставить Git читать и писать в этот файл с помощью параметра --local, но на самом деле это значение по умолчанию. Неудивительно, что вам нужно находиться где-то в репозитории Git, чтобы эта опция работала правильно.

Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в .git/config перекрывают соответствующие значения в [path]/etc/gitconfig.

В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Users\$USER для большинства пользователей). Кроме того, Git ищет файл [path]/etc/gitconfig, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git при запуске инсталлятора.

Если вы используете Git для Windows версии 2.х или новее, то так же обрабатывается файл конфигурации уровня системы, который имеет путь C:\Documents and Settings\All Users\Application Data\Git\config в Windows XP или C:\ProgramData\Git\config в Windows Vista и новее. Этот файл может быть изменён только командой git config -f <file>, запущенной с правами администратора.

Чтобы посмотреть все установленные настройки и узнать где именно они заданы, используйте команду:

$ git config --list --show-origin

**Имя пользователя**

Первое, что вам следует сделать после установки Git — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Опять же, если указана опция --global, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра --global в каталоге с нужным проектом.

Многие GUI-инструменты предлагают сделать это при первом запуске.

**Выбор редактора**

Теперь, когда вы указали своё имя, самое время выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git. По умолчанию Git использует стандартный редактор вашей системы, которым обычно является Vim. Если вы хотите использовать другой текстовый редактор, например, Emacs, можно проделать следующее:

$ git config --global core.editor emacs

В системе Windows следует указывать полный путь к исполняемому файлу при установке другого текстового редактора по умолчанию. Пути могут отличаться в зависимости от того, как работает инсталлятор.

В случае с Notepad++, популярным редактором, скорее всего вы захотите установить 32-битную версию, так как 64-битная версия ещё не поддерживает все плагины. Если у вас 32-битная Windows или 64-битный редактор с 64-битной системой, то выполните следующее:

$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

### Настройка ветки по умолчанию

Когда вы инициализируете репозиторий командой git init, Git создаёт ветку с именем **master** по умолчанию. Начиная с версии 2.28, вы можете задать другое имя для создания ветки по умолчанию.

Например, чтобы установить имя **main** для вашей ветки по умолчанию, выполните следующую команду:

$ git config --global init.defaultBranch main

### Проверка настроек

Если вы хотите проверить используемую конфигурацию, можете использовать команду git config --list, чтобы показать все настройки, которые Git найдёт:

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Некоторые ключи (названия) настроек могут отображаться несколько раз, потому что Git читает настройки из разных файлов (например, из /etc/gitconfig и ~/.gitconfig). В таком случае Git использует последнее значение для каждого ключа.

Также вы можете проверить значение конкретного ключа, выполнив git config <key>:

$ git config user.name

John Doe

**Создание коммитов, переключение между коммитами**

Добавить коммит в Git можно следующим образом. Для этого необходимо выполнить 2 команды. Первая команда git add -A добавляет все измененные файлы в индекс. А вторая команда создает коммит.

git add -A

git commit -m "Commit message."

Если вам нужно создать коммит, но добавлять в него не все измененные файлы, а только определенные, то при использовании команды git add необходимо указать через пробел добавляемые в индекс файлы. Например:

git add goodfile.cpp goodfile.h

git commit -m "Commit message."

Можно создать коммит одно командой, но только для отслеживаемых файлов. В данном случае в индекс добавляются все изменения файлов, но только тех файлов, которые уже когда-либо были добавлены в Git репозиторий — отслеживаемые файлы (Tracked files). То есть новые неотслеживаемые файлы (Untracked files) данной командой игнорируются.

git commit -a -m "Commit message."

Когда вы создали один или несколько коммитов в своем локальном репозитории, вам, скорее всего, нужно будет отправить все свои изменения на удаленный репозиторий. Для этого используется команда git push. Например, отправим на удаленный репозиторий origin все наши изменения в ветке master:

git push origin master

Для переключения на нужный коммит используется действие checkout.

После переключения, все файлы в проекте станут такими, какими они были в данном коммите.

git checkout commit

Где commit - это хеш (обозначение, имя) коммита, причем можно указывать не весь хеш, а несколько начальных символов хеша.

Вопрос: А как вернуться в самое свежее состояние?

Ответ. Команда:

git checkout

без дополнительных параметров вернет вас в head ревизию.

**Создание веток, переключение между ветками**

Что же на самом деле происходит при создании ветки? Всего лишь создаётся новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем testing. Вы можете это сделать командой git branch :

$ git branch testing

В результате создаётся новый указатель на текущий коммит.

Для переключения на существующую ветку выполните команду git checkout. Давайте переключимся на ветку testing:

$ git checkout testing

В результате указатель HEAD переместится на ветку testing.

**Работа с удаленным репозиторием**

**Просмотр удалённых репозиториев**

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду git remote. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите как минимум origin — имя по умолчанию, которое Git даёт серверу, с которого производилось клонирование:

$ git clone https://github.com/schacon/ticgit

Cloning into 'ticgit'...

remote: Reusing existing pack: 1857, done.

remote: Total 1857 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.

Resolving deltas: 100% (772/772), done.

Checking connectivity... done.

$ cd ticgit

$ git remote

origin

Вы можете также указать ключ -v, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

Если у вас больше одного удалённого репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удалёнными репозиториями в случае совместной работы нескольких пользователей, вывод команды может выглядеть примерно так:

$ cd grit

$ git remote -v

bakkdoor https://github.com/bakkdoor/grit (fetch)

bakkdoor https://github.com/bakkdoor/grit (push)

cho45 https://github.com/cho45/grit (fetch)

cho45 https://github.com/cho45/grit (push)

defunkt https://github.com/defunkt/grit (fetch)

defunkt https://github.com/defunkt/grit (push)

koke git://github.com/koke/grit.git (fetch)

koke git://github.com/koke/grit.git (push)

origin git@github.com:mojombo/grit.git (fetch)

origin git@github.com:mojombo/grit.git (push)

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториев доступны для записи и в них можно отправлять свои изменения, хотя вывод команды не даёт никакой информации о правах доступа.

Обратите внимание на разнообразие протоколов, используемых при указании адреса удалённого репозитория; подробнее мы рассмотрим протоколы в разделе [Установка Git на сервер](https://git-scm.com/book/ru/v2/ch00/r_git_on_the_server) главы 4.

**Добавление удалённых репозиториев**

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду git remote add <shortname> <url>:

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)

origin https://github.com/schacon/ticgit (push)

pb https://github.com/paulboone/ticgit (fetch)

pb https://github.com/paulboone/ticgit (push)

Теперь вместо указания полного пути вы можете использовать pb. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду git fetch pb:

$ git fetch pb

remote: Counting objects: 43, done.

remote: Compressing objects: 100% (36/36), done.

remote: Total 43 (delta 10), reused 31 (delta 5)

Unpacking objects: 100% (43/43), done.

From https://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

Ветка master из репозитория Пола сейчас доступна вам под именем pb/master. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола. Более подробно работа с ветками рассмотрена в главе Ветвление в Git.

**Получение изменений из удалённого репозитория — Fetch и Pull**

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

$ git fetch [remote-name]

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда clone автоматически добавляет этот удалённый репозиторий под именем «origin». Таким образом, git fetch origin извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью fetch). Важно отметить, что команда git fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если ветка настроена на отслеживание удалённой ветки (см. следующий раздел и главу [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) чтобы получить больше информации), то вы можете использовать команду git pull чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей. Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда git clone автоматически настраивает вашу локальную ветку master на отслеживание удалённой ветки master на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

|  |  |
| --- | --- |
| **Примечание** | Начиная с версии 2.27, команда git pull выдаёт предупреждение, если настройка pull.rebase не установлена. Git будет выводить это предупреждение каждый раз пока настройка не будет установлена.  Если хотите использовать поведение Git по умолчанию (простое смещение вперёд если возможно — иначе создание коммита слияния): git config --global pull.rebase "false"  Если хотите использовать перебазирование при получении изменений: git config --global pull.rebase "true" |

**Отправка изменений в удаленный репозиторий (Push)**

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в удалённый репозиторий. Команда для этого действия простая: git push <remote-name> <branch-name>. Чтобы отправить вашу ветку master на сервер origin (повторимся, что клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

$ git push origin master

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а после него выполнить команду push попытаетесь вы, то ваш push точно будет отклонён. Вам придётся сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить push. Обратитесь к главе [Ветвление в Git](https://git-scm.com/book/ru/v2/ch00/ch03-git-branching) для более подробного описания, как отправлять изменения на удалённый сервер.

**Просмотр удаленного репозитория**

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду git remote show <remote>. Выполнив эту команду с некоторым именем, например, origin, вы получите следующий результат:

$ git remote show origin

\* remote origin

Fetch URL: https://github.com/schacon/ticgit

Push URL: https://github.com/schacon/ticgit

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

Local branch configured for 'git pull':

master merges with remote master

Local ref configured for 'git push':

master pushes to master (up to date)

Она выдаёт URL удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы, находясь на ветке master, выполните git pull, ветка master с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации и вы наверняка встречались с чем-то подобным. Однако, если вы используете Git более интенсивно, вы можете увидеть гораздо большее количество информации от git remote show:

$ git remote show origin

\* remote origin

URL: https://github.com/my-org/complex-project

Fetch URL: https://github.com/my-org/complex-project

Push URL: https://github.com/my-org/complex-project

HEAD branch: master

Remote branches:

master tracked

dev-branch tracked

markdown-strip tracked

issue-43 new (next fetch will store in remotes/origin)

issue-45 new (next fetch will store in remotes/origin)

refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)

Local branches configured for 'git pull':

dev-branch merges with remote dev-branch

master merges with remote master

Local refs configured for 'git push':

dev-branch pushes to dev-branch (up to date)

markdown-strip pushes to markdown-strip (up to date)

master pushes to master (up to date)

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении git push. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере, и для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении git pull.

**Удаление и переименование удалённых репозиториев**

Для переименования удалённого репозитория можно выполнить git remote rename. Например, если вы хотите переименовать pb в paul, вы можете это сделать при помощи git remote rename:

$ git remote rename pb paul

$ git remote

origin

paul

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как pb/master, теперь стало paul/master.

Если по какой-то причине вы хотите удалить удаленный репозиторий — вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения — вы можете использовать git remote rm:

$ git remote remove paul

$ git remote

origin

При удалении ссылки на удалённый репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, так же будут удалены.

**PYTHON**

**Типы данных в Python**

В Python есть несколько стандартных типов данных:

* Numbers (числа)
* Strings (строки)
* Lists (списки)
* Dictionaries (словари)
* Tuples (кортежи)
* Sets (множества)
* Boolean (логический тип данных)

Эти типы данных можно, в свою очередь, классифицировать по нескольким признакам:

* изменяемые (списки, словари и множества)
* неизменяемые (числа, строки и кортежи)
* упорядоченные (списки, кортежи, строки и словари)
* неупорядоченные (множества)

**Циклы и логические операторы**

## **Оператор цикла**while

Оператор цикла while  выполняет указанный набор инструкций до тех пор, пока условие цикла истинно. Истинность условия определяется также как и в операторе if. Синтаксис оператора while  выглядит так.

while выражение:

инструкция\_1

инструкция\_2

...

инструкция\_n

Выполняемый набор инструкций называется телом цикла.

Пример.

a = **0**

while a < **7**:

print("A")

a += **1**

Буква “А” будет выведена семь раз в столбик.

Пример бесконечного цикла.

a = **0**

while a == **0**:

print("A")

### Операторы break и continue

При работе с циклами используются операторы **break**и **continue**.

Оператор **break**предназначен для досрочного прерывания работы цикла while.

Пример.

a = **0**

while a >= **0**:

if a == **7**:

break

a += **1**

print("A")

В приведенном выше коде, выход из цикла произойдет при достижении переменной a  значения 7. Если бы не было этого условия, то цикл выполнялся бы бесконечно.

Оператор ***continue***  запускает цикл заново, при этом код, расположенный после данного оператора, не выполняется.

Пример.

a = -**1**

while a < **10**:

a += **1**

if a >= **7**:

continue

print("A")

При запуске данного кода символ “А” будет напечатан 7 раз, несмотря на то, что всего будет выполнено 11 проходов цикла.

## **Оператор цикла**for

Оператор for  выполняет указанный набор инструкций заданное количество раз, которое определяется количеством элементов в наборе.

Пример.

for i in range(**5**):

print("Hello")

В результате “Hello” будет выведено пять раз.

Внутри тела цикла можно использовать операторы **break**и **continue**, принцип работы их точно такой же как и в операторе while.

Если у вас есть заданный список, и вы хотите выполнить над каждым элементом определенную операцию (возвести в квадрат и напечатать получившееся число), то с помощью for  такая задача решается так.

lst = [**1**, **3**, **5**, **7**, **9**]

for i in lst:

print(i \*\* **2**)

Также можно пройти по всем буквам в строке.

word\_str = "Hello, world!"

for l in word\_str:

print(l)

Строка “Hello, world!” будет напечатана в столбик.

На этом закончим краткий обзор операторов ветвления и цикла.

Логические операторы

Говоря на естественном языке (например, русском) мы обозначаем сравнения словами "равно", "больше", "меньше". В языках программирования используются специальные знаки, подобные тем, которые используются в математике: > (больше), < (меньше), >= (больше или равно), <= (меньше или равно), == (равно), != (не равно).

Не путайте операцию присваивания значения переменной, обозначаемую в языке Python одиночным знаком "равно", и операцию сравнения (два знака "равно"). Присваивание и сравнение – разные операции.

**Функции, области видимости**

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

Определим простейшую функцию:

**def** add(x, y):

**return** x + y

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

**>>>** add(1, 10)

11

**>>>** add('abc', 'def')

'abcdef'

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

**>>> def** newfunc(n):

**...**  **def** myfunc(x):

**...**  **return** x + n

**...**  **return** myfunc

**...**

**>>>** new = newfunc(100) *# new - это функция*

**>>>** new(200)

300

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение [None](https://pythonworld.ru/tipy-dannyx-v-python/none.html):

**>>> def** func():

**...**  **pass**

**...**

**>>>** print(func())

None

## Аргументы функции

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

**>>> def** func(a, b, c=2): *# c - необязательный аргумент*

**...**  **return** a + b + c

**...**

**>>>** func(1, 2) *# a = 1, b = 2, c = 2 (по умолчанию)*

5

**>>>** func(1, 2, 3) *# a = 1, b = 2, c = 3*

6

**>>>** func(a=1, b=3) *# a = 1, b = 3, c = 2*

6

**>>>** func(a=3, c=6) *# a = 3, c = 6, b не определен*

Traceback (most recent call last):

File "", line 1, in

func(a=3, c=6)

TypeError: func() takes at least 2 arguments (2 given)

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится \*:

**>>> def** func(\*args):

**...**  **return** args

**...**

**>>>** func(1, 2, 3, 'abc')

(1, 2, 3, 'abc')

**>>>** func()

()

**>>>** func(1)

(1,)

Как видно из примера, args - это [кортеж](https://pythonworld.ru/tipy-dannyx-v-python/kortezhi-tuple.html) из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится \*\*:

**>>> def** func(\*\*kwargs):

**...**  **return** kwargs

**...**

**>>>** func(a=1, b=2, c=3)

{'a': 1, 'c': 3, 'b': 2}

**>>>** func()

{}

**>>>** func(a='python')

{'a': 'python'}

В переменной kwargs у нас хранится [словарь](https://pythonworld.ru/tipy-dannyx-v-python/slovari-dict-funkcii-i-metody-slovarej.html), с которым мы, опять-таки, можем делать все, что нам заблагорассудится.

## Анонимные функции, инструкция lambda

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func():

**>>>** func = **lambda** x, y: x + y

**>>>** func(1, 2)

3

**>>>** func('a', 'b')

'ab'

**>>>** (**lambda** x, y: x + y)(1, 2)

3

**>>>** (**lambda** x, y: x + y)('a', 'b')

'ab'

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

**>>>** func = **lambda** \*args: args

**>>>** func(1, 2, 3, 4)

(1, 2, 3, 4)

**Контейнеры (массивы)**

**Одномерные массивы в Python** представляют собой список элементов. Значения указываются внутри квадратных скобок, где перечисляются через запятую. Как правило, любой элемент можно вызвать по индексу и присвоить ему новое значение.

Пустой список:

a **=** []

Массив строк в **Python**:

Prime **=** ['string1', 'string2', 'string3']

Prime[1] **=** 'string2'; **//**true

Чтобы возвратить число элементов внутри списка, используют функцию len():

len(Prime) **==** 4; **//** true

Когда нужно перечислить элементы массива, применяют цикл for. В «**Питоне**» этот цикл перебирает элементы, а не индексы, как в Pascal:

**for** elem **in** [1, 4, 67]

Идём дальше. Создать и добавить цикл в **Python** можно с помощью генератора заполнения списков. Записывается он в следующем виде: [значение массива for имя переменной in число элементов];

Если говорить про создание не одномерного, а **двумерного массива**, то он в **Python** создаётся путём использования вложенных генераторов, и выглядит это так:

[[0 **for** j **in** range(m)] **for** i **in** range(n)]

## Как создаются матрицы в Python?

Добавление и модификация массивов или матриц (**matrix**) в **Python** осуществляется с помощью библиотеки **NumPy**. Вы можете создать таким образом и одномерный, и двумерный, и многомерный массив. Библиотека обладает широким набором пакетов, которые необходимы, чтобы успешно решать различные математические задачи. Она не только поддерживает создание двумерных и многомерных массивов, но обеспечивает работу однородных многомерных матриц.

Чтобы получить доступ и начать использовать функции данного пакета, его импортируют:

**import** numpy **as** np

Функция array() — один из самых простых способов, позволяющих динамически задать одно- и двумерный массив в **Python**. Она создаёт объект типа ndarray:

array **=** np**.**array(**/\*** множество элементов **\*/**)

Для проверки используется функция array.type() — принимает в качестве аргумента имя массива, который был создан.

Если хотите сделать переопределение типа массива, используйте на стадии создания dtype=np.complex:

array2 **=** np**.**array([ **/\***элементы**\*/**, dtype**=**np**.**complex)

Когда стоит задача задать одномерный или двумерный массив определённой длины в **Python**, и его значения на данном этапе неизвестны, происходит его заполнение нулями функцией zeros(). Кроме того, можно получить матрицу из единиц через функцию ones(). При этом в качестве аргументов принимают число элементов и число вложенных массивов внутри:

np**.**zeros(2, 2, 2)

К примеру, так в Python происходит задание двух массивов внутри, которые по длине имеют два элемента:

array([

[[0, 0]]

[[0, 0]]]

)

Если хотите вывести одно- либо двумерный массив на экран, вам поможет функция print(). Учтите, что если матрица слишком велика для печати, **NumPy** скроет центральную часть и выведет лишь крайние значения. Дабы увидеть массив полностью, используется функция set\_printoptions(). При этом по умолчанию выводятся не все элементы, а происходит вывод только первой тысячи. И это значение массива указывается в качестве аргумента с ключевым словом **threshold**.

## Базовые операции в NumPy

Все действия, производимые над компонентами массива, оборачиваются созданием нового массива. При этом массивы и матрицы взаимодействуют в том случае, если имеют один и тот же размер:

array1 **=** np**.**array([[1, 2, 3], [1, 2, 3]])

array2 **=** np**.**array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])

Если выполнить array1 + array2, компилятор скажет об ошибке, а всё потому, что размер первого matrix равен двум, а второго трём.

array1 **=** np**.**array([1, 2, 5, 7])

array2 **=** arange([1, 5, 1])

В данном случае array1 + array2 вернёт нам массив со следующими элементами: 2, 4, 8, 11. Здесь не возникнет ошибки, т. к. матрицы имеют одинаковые размеры. Причём вместо ручного сложения часто применяют функцию, входящую в класс ndarray sum():

np**.**array(array1 **+** array1) **==** array1 **+** array2

В ndarray входит большая библиотека методов, необходимых для выполнения математических операций.

## Форма матрицы в Python

Lenght matrix (длина матрицы) в Python определяет форму. Длину матрицы проверяют методом shape().

Массив с 2-мя либо 3-мя элементами будет иметь форму (2, 2, 3). И это состояние изменится, когда в shape() будут указаны аргументы: первый — число подмассивов, второй — размерность каждого подмассива.

Те же задачи и ту же операцию выполнит reshape(). Здесь **lenght** и другие параметры matrix определяются числом столбцов и строк.

Есть методы и для манипуляции формой. Допустим, при манипуляциях с двумерными или многомерными массивами можно сделать одномерный путём выстраивания внутренних значений последовательно по возрастанию. А чтобы поменять в матрице строки и столбцы местами, применяют transpose().

## Операции со срезами matrix в Python

Часто мы работаем не с целым массивом, а с его компонентами. Эти операции выполняются с помощью метода слайс (срез). Он пришел на замену циклу for, при котором каждый элемент подвергался перебору. Метод позволяет получать копии matrix, причём манипуляции выполняются в виде [start:stop:step]. В данном случае **start** — индекс элемента, с которого берётся отсчёт, **stop** — последний элемент, **step** — размер шага или число пропускаемых значений элемента при каждой итерации. Изначально start равен нулю, stop — индексу последнего элемента, step — единице. Если выполнить операции без аргументов, копирование и добавление списка произойдёт полностью.

Допустим, имеем целочисленный массив otus = [1, 2, 3, 4]. Для копирования и вывода используем otus[:]. В итоге произойдёт вывод последовательности [1, 2, 3, 4]. Но если аргументом станет отрицательное значение, допустим, -2, произойдёт вывод уже других данных:

otus[**-**2]; **//**[4]

Возможны и другие операции. Например, если добавить ещё одно двоеточие, будет указан шаг копируемых элементов. Таким образом, otus[::2] позволит вывести матрицу [1, 3].

Если ввести отрицательное значение, к примеру, [::-2] отсчёт начнётся с конца, и в результате произойдёт вывод [3, 1]. Остаётся добавить, что метод среза позволяет гибко работать с матрицами и вложенными списками в **Python**.

**Numpy**

NumPy это open-source модуль для python, который предоставляет общие математические и числовые операции в виде пре-скомпилированных, быстрых функций. Они объединяются в высокоуровневые пакеты. Они обеспечивают функционал, который можно сравнить с функционалом MatLab. NumPy (Numeric Python) предоставляет базовые методы для манипуляции с большими массивами и матрицами. SciPy (Scientific Python) расширяет функционал numpy огромной коллекцией полезных алгоритмов, таких как минимизация, преобразование Фурье, регрессия, и другие прикладные математические техники.

**import** numpy **as** np

Это выражение позволяет нам получать доступ к numpy объектам используя np.X вместо numpy.X.

Главной особенностью numpy является объект array. Массивы схожи со списками в python, исключая тот факт, что элементы массива должны иметь одинаковый тип данных, как float и int. С массивами можно проводить числовые операции с большим объемом информации в разы быстрее и, главное, намного эффективнее чем со списками.  
Создание массива из списка:

a = np.array([1, 4, 5, 8], float)

>>> a

array([ 1., 4., 5., 8.])

>>> type(a)

<**class** '**numpy**.**ndarray**'>

Здесь функция array принимает два аргумента: список для конвертации в массив и тип для каждого элемента. Ко всем элементам можно получить доступ и манипулировать ими так же, как вы бы это делали с обычными списками:

>>> a[:2]

array([ 1., 4.])

>>> a[3]

8.0

>>> a[0] = 5.

>>> a

array([ 5., 4., 5., 8.])

Массивы могут быть и многомерными. В отличии от списков можно использовать запятые в скобках. Вот пример двумерного массива (матрица):

>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)

>>> a

array([[ 1., 2., 3.],

[ 4., 5., 6.]])

>>> a[0,0]

1.0

>>> a[0,1]

2.0

Array slicing работает с многомерными массивами аналогично, как и с одномерными, применяя каждый срез, как фильтр для установленного измерения. Используйте ":" в измерении для указывания использования всех элементов этого измерения

>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)

>>> a[1,:]

array([ 4., 5., 6.])

>>> a[:,2]

array([ 3., 6.])

>>> a[-1:, -2:]

array([[ 5., 6.]])

Метод shape возвращает количество строк и столбцов в матрице:

>>> a.shape

(2, 3)

Метод dtype возвращает тип переменных, хранящихся в массиве:

>>> a.dtype

dtype('float64')

Тут float64, это числовой тип данных в numpy, который используется для хранения вещественных чисел двойной точности. Так же как float в Python.  
Метод len возвращает длину первого измерения (оси):

a = np.array([[1, 2, 3], [4, 5, 6]], float)

>>> len(a)

2

Метод in используется для проверки на наличие элемента в массиве:

>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)

>>> 2 **in** a

True

>>> 0 **in** a

False

Массивы можно переформировать при помощи метода, который задает новый многомерный массив. Следуя следующему примеру, мы переформатируем одномерный массив из десяти элементов во двумерный массив, состоящий из пяти строк и двух столбцов:

>>> a = np.array(range(10), float)

>>> a

array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

>>> a = a.reshape((5, 2))

>>> a

array([[ 0., 1.],

[ 2., 3.],

[ 4., 5.],

[ 6., 7.],

[ 8., 9.]])

>>> a.shape

(5, 2)

Обратите внимание, метод reshape создает новый массив, а не модифицирует оригинальный.  
Имейте ввиду, связывание имен в python работает и с массивами. Метод copy используется для создания копии существующего массива в памяти:

>>> a = np.array([1, 2, 3], float)

>>> b = a

>>> c = a.copy()

>>> a[0] = 0

>>> a

array([0., 2., 3.])

>>> b

array([0., 2., 3.])

>>> c

array([1., 2., 3.])

Списки можно тоже создавать с массивов:

>>> a = np.array([1, 2, 3], float)

>>> a.tolist()

[1.0, 2.0, 3.0]

>>> list(a)

[1.0, 2.0, 3.0]

Можно также переконвертировать массив в бинарную строку (то есть, не human-readable форму). Используйте метод tostring для этого. Метод fromstring работает в для обратного преобразования. Эти операции иногда полезны для сохранения большого количества данных в файлах, которые могут быть считаны в будущем.

>>> a = array([1, 2, 3], float)

>>> s = a.tostring()

>>> s

'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'

>>> np.fromstring(s)

array([ 1., 2., 3.])

Заполнение массива одинаковым значением.

>>> a = array([1, 2, 3], float)

>>> a

array([ 1., 2., 3.])

>>> a.fill(0)

>>> a

array([ 0., 0., 0.])

Транспонирование массивов также возможно, при этом создается новый массив:

>>> a = np.array(range(6), float).reshape((2, 3))

>>> a

array([[ 0., 1., 2.],

[ 3., 4., 5.]])

>>> a.transpose()

array([[ 0., 3.],

[ 1., 4.],

[ 2., 5.]])

Многомерный массив можно переконвертировать в одномерный при помощи метода flatten:

>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)

>>> a

array([[ 1., 2., 3.],

[ 4., 5., 6.]])

>>> a.flatten()

array([ 1., 2., 3., 4., 5., 6.])

Два или больше массивов можно сконкатенировать при помощи метода concatenate:

>>> a = np.array([1,2], float)

>>> b = np.array([3,4,5,6], float)

>>> c = np.array([7,8,9], float)

>>> np.concatenate((a, b, c))

array([1., 2., 3., 4., 5., 6., 7., 8., 9.])

Если массив не одномерный, можно задать ось, по которой будет происходить соединение. По умолчанию (не задавая значения оси), соединение будет происходить по первому измерению:

>>> a = np.array([[1, 2], [3, 4]], float)

>>> b = np.array([[5, 6], [7,8]], float)

>>> np.concatenate((a,b))

array([[ 1., 2.],

[ 3., 4.],

[ 5., 6.],

[ 7., 8.]])

>>> np.concatenate((a,b), axis=0)

array([[ 1., 2.],

[ 3., 4.],

[ 5., 6.],

[ 7., 8.]])

>>>

np.concatenate((a,b), axis=1)

array([[ 1., 2., 5., 6.],

[ 3., 4., 7., 8.]])

В заключении, размерность массива может быть увеличена при использовании константы newaxis в квадратных скобках:

>>> a = np.array([1, 2, 3], float)

>>> a

array([1., 2., 3.])

>>> a[:,np.newaxis]

array([[ 1.],

[ 2.],

[ 3.]])

>>> a[:,np.newaxis].shape

(3,1)

>>> b[np.newaxis,:]

array([[ 1., 2., 3.]])

>>> b[np.newaxis,:].shape

(1,3)

Заметьте, тут каждый массив двумерный; созданный при помощи newaxis имеет размерность один. Метод newaxis подходит для удобного создания надлежаще-мерных массивов в векторной и матричной математике.

# **Другие пути создания массивов**

Функция arange аналогична функции range, но возвращает массив:

>>> np.arange(5, dtype=float)

array([ 0., 1., 2., 3., 4.])

>>> np.arange(1, 6, 2, dtype=int)

array([1, 3, 5])

Функции zeros и ones создают новые массивы с установленной размерностью, заполненные этими значениями. Это, наверное, самые простые в использовании функции для создания массивов:

>>> np.ones((2,3), dtype=float)

array([[ 1., 1., 1.],

[ 1., 1., 1.]])

>>> np.zeros(7, dtype=int)

array([0, 0, 0, 0, 0, 0, 0])

Функции zeros\_like и ones\_like могут преобразовать уже созданный массив, заполнив его нулями и единицами соответственно:

>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)

>>> np.zeros\_like(a)

array([[ 0., 0., 0.],

[ 0., 0., 0.]])

>>> np.ones\_like(a)

array([[ 1., 1., 1.],

[ 1., 1., 1.]])

Также есть некоторое количество функций для создания специальных матриц. Для создания квадратной матрицы с главной диагональю, которая заполненная единицами, воспользуемся методом identity:

>>> np.identity(4, dtype=float)

array([[ 1., 0., 0., 0.],

[ 0., 1., 0., 0.],

[ 0., 0., 1., 0.],

[ 0., 0., 0., 1.]])

Функция eye возвращает матрицу с единичками на к-атой диагонали:

>>> np.eye(4, k=1, dtype=float)

array([[ 0., 1., 0., 0.],

[ 0., 0., 1., 0.],

[ 0., 0., 0., 1.],

[ 0., 0., 0., 0.]])

**Математические операции над массивами**

Когда для массивов мы используем стандартные математические операции, должен соблюдаться принцип: элемент--элемент. Это означает, что массивы должны быть одинакового размера во время сложения, вычитания и тому подобных операций:

>>> a = np.array([1,2,3], float)

>>> b = np.array([5,2,6], float)

>>> a + b

array([6., 4., 9.])

>>> a – b

array([-4., 0., -3.])

>>> a \* b

array([5., 4., 18.])

>>> b / a

array([5., 1., 2.])

>>> a % b

array([1., 0., 3.])

>>> b\*\*a

array([5., 4., 216.])

Для двухмерных массивов, умножение остается поэлементным и не соответствует умножению матриц. Для этого существуют специальные функции, которые мы изучим позже.

>>> a = np.array([[1,2], [3,4]], float)

>>> b = np.array([[2,0], [1,3]], float)

>>> a \* b

array([[2., 0.], [3., 12.]])

При несоответствии в размере выбрасываются ошибки:

>>> a = np.array([1,2,3], float)

>>> b = np.array([4,5], float)

>>> a + b

Traceback (most recent call last):

File "<stdin>", line 1, **in** <module>

ValueError: operands could **not** be broadcast together **with** shapes (3,) (2,)

Однако, если размерность массивов не совпадает, они будут преобразованы для выполнения математических операций. Это зачастую означает, что меньший массив будет использован несколько раз для завершения операций. Рассмотрим такой пример:

>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)

>>> b = np.array([-1, 3], float)

>>> a

array([[ 1., 2.],

[ 3., 4.],

[ 5., 6.]])

>>> b

array([-1., 3.])

>>> a + b

array([[ 0., 5.],

[ 2., 7.],

[ 4., 9.]])

Тут, одномерный массив b был преобразован в двухмерный, который соответствует размеру массива a. По существу, b был повторен несколько раз, для каждой «строки» a. Иначе его можно представить так:

array([[-1., 3.],

[-1., 3.],

[-1., 3.]])

Python автоматически преобразовывает массивы в этом случае. Иногда, однако, когда преобразование играет роль, мы можем использовать константу newaxis, чтобы изменить преобразование:

>>> a = np.zeros((2,2), float)

>>> b = np.array([-1., 3.], float)

>>> a

array([[ 0., 0.],

[0., 0.]])

>>> b

array([-1., 3.])

>>> a + b

array([[-1., 3.],

[-1., 3.]])

>>> a + b[np.newaxis,:]

array([[-1., 3.],

[-1., 3.]])

>>> a + b[:,np.newaxis]

array([[-1., -1.],

[ 3., 3.]])

Вдобавок к стандартным операторам, в numpy включена библиотека стандартных математических функций, которые могут быть применены поэлементно к массивам. Собственно функции: abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, и arctanh.

>>> a = np.array([1, 4, 9], float)

>>> np.sqrt(a)

array([ 1., 2., 3.])

Функции floor, ceil и rint возвращают нижние, верхние или ближайшие (округлённое) значение:

>>> a = np.array([1.1, 1.5, 1.9], float)

>>> np.floor(a)

array([ 1., 1., 1.])

>>> np.ceil(a)

array([ 2., 2., 2.])

>>> np.rint(a)

array([ 1., 2., 2.])

Также в numpy включены две важные математические константы:

>>> np.pi

3.1415926535897931

>>> np.e

2.7182818284590451

**Перебор элементов массива**

Проводить итерацию массивов можно аналогично спискам:

>>> a = np.array([1, 4, 5], int)

>>> **for** x **in** a:

... **print** x

1

4

5

Для многомерных массивов итерация будет проводиться по первой оси, так, что каждый проход цикла будет возвращать «строку» массива:

>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)

>>> **for** x **in** a:

... **print** x

[ 1. 2.]

[ 3. 4.]

[ 5. 6.]

Множественное присваивание также доступно при итерации:

>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)

>>> **for** (x, y) **in** a:

... **print** x \* y

2.0

12.0

30.0

**Базовые операции над массивами**

Для получения каких-либо свойств массивов существует много функций. Элементы могут быть суммированы или перемножены:

>>> a = np.array([2, 4, 3], float)

>>> a.sum()

9.0

>>> a.prod()

24.0

В этом примере были использованы функции массива. Также можно использовать собственные функции numpy:

>>> np.sum(a)

9.0

>>> np.prod(a)

24.0

Для большинства случаев могут использоваться оба варианта.  
Некие функции дают возможность оперировать статистическими данными. Это функции mean (среднее арифметическое), вариация и девиация:

>>> a = np.array([2, 1, 9], float)

>>> a.mean()

4.0

>>>

a.var()

12.666666666666666

>>> a.std()

3.5590260840104371

Можно найти минимум и максимум в массиве:

>>> a = np.array([2, 1, 9], float)

>>> a.min()

1.0

>>> a.max()

9.0

Функции argmin и argmax возвращают индекс минимального или максимального элемента:

>>> a = np.array([2, 1, 9], float)

>>> a.argmin()

1

>>> a.argmax()

2

Для многомерных массивов каждая из функций может принять дополнительный аргумент axis и в зависимости от его значения выполнять функции по определенной оси, помещая результаты исполнения в массив:

>>> a = np.array([[0, 2], [3, -1], [3, 5]], float)

>>> a.mean(axis=0)

array([ 2., 2.])

>>> a.mean(axis=1)

array([ 1., 1., 4.])

>>> a.min(axis=1)

array([ 0., -1., 3.])

>>> a.max(axis=0)

array([ 3., 5.])

Как и списки, массивы можно отсортировать:

>>> a = np.array([6, 2, 5, -1, 0], float)

>>> sorted(a)

[-1.0, 0.0, 2.0, 5.0, 6.0]

>>> a.sort()

>>> a

array([-1., 0., 2., 5., 6.])

Значения в массиве могут быть «сокращены», чтобы принадлежать заданному диапазону. Это тоже самое что применять min(max(x, minval), maxval) к каждому элементу x:

>>> a = np.array([6, 2, 5, -1, 0], float)

>>> a.clip(0, 5)

array([ 5., 2., 5., 0., 0.])

Уникальные элементы могут быть извлечены вот так:

>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)

>>> np.unique(a)

array([ 1., 4., 5., 7.])

Для двухмерных массивов диагональ можно получить так:

>>> a = np.array([[1, 2], [3, 4]], float)

>>> a.diagonal()

array([ 1., 4.])

**Pandas**

Pandas — это библиотека для работы с данными на Python. Она упрощает жизнь аналитикам: где раньше использовалось 10 строк кода теперь хватит одной.

Например, чтобы прочитать данные из csv, в стандартном Python надо сначала решить, как хранить данные, затем открыть файл, прочитать его построчно, отделить значения друг от друга и очистить данные от специальных символов.

> with open('file.csv') as f:

... content = f.readlines()

... content = [x.split(',').replace('\n','') for x in content]

В Pandas всё проще. Во-первых, не нужно думать, как будут храниться данные — они лежат в датафрейме. Во-вторых, достаточно написать одну команду:

> data = pd.read\_csv('file.csv')

Pandas добавляет в Python новые структуры данных — серии и датафреймы. Расскажу, что это такое.

## **Структуры данных: серии и датафреймы**

Серии — одномерные массивы данных. Они очень похожи на списки, но отличаются по поведению — например, операции применяются к списку целиком, а в сериях — поэлементно.

То есть, если список умножить на 2, получите тот же список, повторенный 2 раза.

> vector = [1, 2, 3]

> vector \* 2

[1, 2, 3, 1, 2, 3]

А если умножить серию, ее длина не изменится, а вот элементы удвоятся.

> import pandas as pd

> series = pd.Series([1, 2, 3])

> series \* 2

0 2

1 4

2 6

dtype: int64

Обратите внимание на первый столбик вывода. Это индекс, в котором хранятся адреса каждого элемента серии. Каждый элемент потом можно получать, обратившись по нужному адресу.

> series = pd.Series(['foo', 'bar'])

> series[0]

'foo'

Еще одно отличие серий от списков — в качестве индексов можно использовать произвольные значения, это делает данные нагляднее. Представим, что мы анализируем помесячные продажи. Используем в качестве индексов названия месяцев, значениями будет выручка:

> months = ['jan', 'feb', 'mar', 'apr']

> sales = [100, 200, 300, 400]

> data = pd.Series(data=sales, index=months)

> data

jan 100

feb 200

mar 300

apr 400

dtype: int64

Теперь можем получать значения каждого месяца:

> data['feb']

200

Так как серии — одномерный массив данных, в них удобно хранить измерения по одному. На практике удобнее группировать данные вместе. Например, если мы анализируем помесячные продажи, полезно видеть не только выручку, но и количество проданных товаров, количество новых клиентов и средний чек. Для этого отлично подходят датафреймы.

Датафреймы — это таблицы. У их есть строки, колонки и ячейки.

Технически, колонки датафреймов — это серии. Поскольку в колонках обычно описывают одни и те же объекты, то все колонки делят один и тот же индекс:

> months = ['jan', 'feb', 'mar', 'apr']

> sales = {

... 'revenue': [100, 200, 300, 400],

... 'items\_sold': [23, 43, 55, 65],

... 'new\_clients': [10, 20, 30, 40]

...}

> sales\_df = pd.DataFrame(data=sales, index=months)

> sales\_df

revenue items\_sold new\_clients

jan 100 23 10

feb 200 43 20

mar 300 55 30

apr 400 65 40

Объясню, как создавать датафреймы и загружать в них данные.

## **Создаем датафреймы и загружаем данные**

Бывает, что мы не знаем, что собой представляют данные, и не можем задать структуру заранее. Тогда удобно создать пустой датафрейм и позже наполнить его данными.

> df = pd.DataFrame()

А иногда данные уже есть, но хранятся в переменной из стандартного Python, например, в словаре. Чтобы получить датафрейм, эту переменную передаем в ту же команду:

> df = pd.DataFrame(data=sales, index=months))

Случается, что в некоторых записях не хватает данных. Например, посмотрите на список goods\_sold — в нём продажи, разбитые по товарным категориям. За первый месяц мы продали машины, компьютеры и программное обеспечение. Во втором машин нет, зато появились велосипеды, а в третьем снова появились машины, но велосипеды исчезли:

> goods\_sold = [

... {'computers': 10, 'cars': 1, 'soft': 3},

... {'computers': 4, 'soft': 5, 'bicycles': 1},

... {'computers': 6, 'cars': 2, 'soft': 3}

... ]

Если загрузить данные в датафрейм, Pandas создаст колонки для всех товарных категорий и, где это возможно, заполнит их данными:

> pd.DataFrame(goods\_sold)

bicycles cars computers soft

0 NaN 1.0 10 3

1 1.0 NaN 4 5

2 NaN 2.0 6 3

Обратите внимание, продажи велосипедов в первом и третьем месяце равны NaN — расшифровывается как Not a Number. Так Pandas помечает отсутствующие значения.

Теперь разберем, как загружать данные из файлов. Чаще всего данные хранятся в экселевских таблицах или csv-, tsv- файлах.

Экселевские таблицы читаются с помощью команды pd.read\_excel(). Параметрами нужно передать адрес файла на компьютере и название листа, который нужно прочитать. Команда работает как с xls, так и с xlsx:

> pd.read\_excel('file.xlsx', sheet\_name='Sheet1')

Файлы формата csv и tsv — это текстовые файлы, в которых данные отделены друг от друга запятыми или табуляцией:

# CSV

month,customers,sales

feb,10,200

# TSV

month\tcustomers\tsales

feb\t10\t200

Оба читаются с помощью команды .read\_csv(), символ табуляции передается параметром sep (от англ. separator — разделитель):

> pd.read\_csv('file.csv')

> pd.read\_csv('file.tsv', sep='\t')

При загрузке можно назначить столбец, который будет индексом. Представьте, что мы загружаем таблицу с заказами. У каждого заказа есть свой уникальный номер, Если назначим этот номер индексом, сможем выгружать данные командой df[order\_id]. Иначе придется писать фильтр df[df[‘id’] == order\_id ].

О том, как получать данные из датафреймов, я расскажу в одном из следующих разделов. Чтобы назначить колонку индексом, добавим в команду read\_csv() параметр index\_col, равный названию нужной колонки:

> pd.read\_csv('file.csv', index\_col='id')

После загрузки данных в датафрейм, хорошо бы их исследовать — особенно, если они вам незнакомы.

## **Исследуем загруженные данные**

Представим, что мы анализируем продажи американского интернет-магазина. У нас есть данные о заказах и клиентах. Загрузим файл с продажами интернет-магазина в переменную orders. Раз загружаем заказы, укажем, что колонка id пойдет в индекс:

> orders = pd.read\_csv('orders.csv', index\_col='id')

Расскажу о четырех атрибутах, которые есть у любого датафрейма: .shape, .columns, .index и .dtypes.

.shape показывает, сколько в датафрейме строк и колонок. Он возвращает пару значений (n\_rows, n\_columns). Сначала идут строки, потом колонки.

> orders.shape

(5009, 5)

В датафрейме 5009 строк и 5 колонок.

Окей, масштаб оценили. Теперь посмотрим, какая информация содержится в каждой колонке. С помощью .columns узнаем названия колонок:

> orders.columns

Index(['order\_date', 'ship\_mode', 'customer\_id', 'sales'], dtype='object')

Теперь видим, что в таблице есть дата заказа, метод доставки, номер клиента и выручка.

С помощью .dtypes узнаем типы данных, находящихся в каждой колонке и поймем, надо ли их обрабатывать. Бывает, что числа загружаются в виде текста. Если мы попробуем сложить две текстовых значения '1' + '1', то получим не число 2, а строку '11':

> orders.dtypes

order\_date object

ship\_mode object

customer\_id object

sales float64

dtype: object

Тип object — это текст, float64 — это дробное число типа 3,14.

C помощью атрибута .index посмотрим, как называются строки:

> orders.index

Int64Index([100006, 100090, 100293, 100328, 100363, 100391, 100678, 100706,

100762, 100860,

...

167570, 167920, 168116, 168613, 168690, 168802, 169320, 169488,

169502, 169551],

dtype='int64', name='id', length=5009)

Ожидаемо, в индексе датафрейма номера заказов: 100762, 100860 и так далее.

В колонке sales хранится стоимость каждого проданного товара. Чтобы узнать разброс значений, среднюю стоимость и медиану, используем метод .describe():

> orders.describe()

sales

count 5009.0

mean 458.6

std 954.7

min 0.6

25% 37.6

50% 152.0

75% 512.1

max 23661.2

Наконец, чтобы посмотреть на несколько примеров записей датафрейма, используем команды .head() и .sample(). Первая возвращает 6 записей из начала датафрейма. Вторая — 6 случайных записей:

> orders.head()

order\_date ship\_mode customer\_id sales

id

100006 2014-09-07 Standard DK-13375 377.970

100090 2014-07-08 Standard EB-13705 699.192

100293 2014-03-14 Standard NF-18475 91.056

100328 2014-01-28 Standard JC-15340 3.928

100363 2014-04-08 Standard JM-15655 21.376

Получив первое представление о датафреймах, теперь обсудим, как доставать из него данные.

## **Получаем данные из датафреймов**

Данные из датафреймов можно получать по-разному: указав номера колонок и строк, использовав условные операторы или язык запросов. Расскажу подробнее о каждом способе.

### Указываем нужные строки и колонки

Продолжаем анализировать продажи интернет-магазина, которые загрузили в предыдущем разделе. Допустим, я хочу вывести столбец sales. Для этого название столбца нужно заключить в квадратные скобки и поставить после них названия датафрейма: orders['sales']:

> orders['sales']

id

100006 377.970

100090 699.192

100293 91.056

100328 3.928

100363 21.376

100391 14.620

100678 697.074

100706 129.440

...

Обратите внимание, результат команды — новый датафрейм с таким же индексом.

Если нужно вывести несколько столбцов, в квадратные скобки нужно вставить список с их названиями: orders[['customer\_id', 'sales']]. Будьте внимательны: квадратные скобки стали двойными. Первые — от датафрейма, вторые — от списка:

> orders[['customer\_id', 'sales']]

customer\_id sales

id

100006 DK-13375 377.970

100090 EB-13705 699.192

100293 NF-18475 91.056

100328 JC-15340 3.928

100363 JM-15655 21.376

100391 BW-11065 14.620

100363 KM-16720 697.074

100706 LE-16810 129.440

...

Перейдем к строкам. Их можно фильтровать по индексу и по порядку. Например, мы хотим вывести только заказы 100363, 100391 и 100706, для этого есть команда .loc[]:

> show\_these\_orders = ['100363', '100363', '100706']

> orders.loc[show\_these\_orders]

order\_date ship\_mode customer\_id sales

id

100363 2014-04-08 Standard JM-15655 21.376

100363 2014-04-08 Standard JM-15655 21.376

100706 2014-12-16 Second LE-16810 129.440

А в другой раз бывает нужно достать просто заказы с 1 по 3 по порядку, вне зависимости от их номеров в таблицемы. Тогда используют команду .iloc[]:

> show\_these\_orders = [1, 2, 3]

> orders.iloc[show\_these\_orders]

order\_date ship\_mode customer\_id sales

id

100090 2014-04-08 Standard JM-15655 21.376

100293 2014-04-08 Standard JM-15655 21.376

100328 2014-12-16 Second LE-16810 129.440

Можно фильтровать датафреймы по колонкам и столбцам одновременно:

> columns = ['customer\_id', 'sales']

> rows = ['100363', '100363', '100706']

> orders.loc[rows][columns]

customer\_id sales

id

100363 JM-15655 21.376

100363 JM-15655 21.376

100706 LE-16810 129.440

...

Часто вы не знаете заранее номеров заказов, которые вам нужны. Например, если задача — получить заказы, стоимостью более 1000 рублей. Эту задачу удобно решать с помощью условных операторов.

### Если — то. Условные операторы

Задача: нужно узнать, откуда приходят самые большие заказы. Начнем с того, что достанем все покупки стоимостью более 1000 долларов:

> filter\_large = orders['sales'] > 1000

> orders.loc[filter\_slarge]

order\_date ship\_mode customer\_id sales

id

101931 2014-10-28 First TS-21370 1252.602

102673 2014-11-01 Standard KH-16630 1044.440

102988 2014-04-05 Second GM-14695 4251.920

103100 2014-12-20 First AB-10105 1107.660

103310 2014-05-10 Standard GM-14680 1769.784

...

Помните, в начале статьи я упоминал, что в сериях все операции применяются по-элементно? Так вот, операция orders['sales'] > 1000 идет по каждому элементу серии и, если условие выполняется, возвращает True. Если не выполняется — False. Получившуюся серию мы сохраняем в переменную filter\_large.

Вторая команда фильтрует строки датафрейма с помощью серии. Если элемент filter\_large равен True, заказ отобразится, если False — нет. Результат — датафрейм с заказами, стоимостью более 1000 долларов.

Интересно, сколько дорогих заказов было доставлено первым классом? Добавим в фильтр ещё одно условие:

> filter\_large = df['sales'] > 1000

> filter\_first\_class = orders['ship\_mode'] == 'First'

> orders.loc[filter\_large & filter\_first\_class]

order\_date ship\_mode customer\_id sales

id

101931 2014-10-28 First TS-21370 1252.602

103100 2014-12-20 First AB-10105 1107.660

106726 2014-12-06 First RS-19765 1261.330

112158 2014-12-02 First DP-13165 1050.600

116666 2014-05-08 First KT-16480 1799.970

...

Логика не изменилась. В переменную filter\_large сохранили серию, удовлетворяющую условию orders['sales'] > 1000. В filter\_first\_class — серию, удовлетворяющую orders['ship\_mode'] == 'First'.

Затем объединили обе серии с помощью логического ‘И’: filter\_first\_class & filter\_first\_class. Получили новую серию той же длины, в элементах которой True только у заказов, стоимостью больше 1000, доставленных первым классом. Таких условий может быть сколько угодно.

### Язык запросов

Еще один способ решить предыдущую задачу — использовать язык запросов. Все условия пишем одной строкой 'sales > 1000 & ship\_mode == 'First' и передаем ее в метод .query(). Запрос получается компактнее.

> orders.query('sales > 1000 & ship\_mode == First')

order\_date ship\_mode customer\_id sales

id

101931 2014-10-28 First TS-21370 1252.602

103100 2014-12-20 First AB-10105 1107.660

106726 2014-12-06 First RS-19765 1261.330

112158 2014-12-02 First DP-13165 1050.600

116666 2014-05-08 First KT-16480 1799.970

...

Отдельный кайф: значения для фильтров можно сохранить в переменной, а в запросе сослаться на нее с помощью символа @: sales > @sales\_filter.

> sales\_filter = 1000

> ship\_mode\_filter = 'First'

> orders.query('sales > @sales\_filter & ship\_mode > @ship\_mode\_filter')

order\_date ship\_mode customer\_id sales

id

101931 2014-10-28 First TS-21370 1252.602

103100 2014-12-20 First AB-10105 1107.660

106726 2014-12-06 First RS-19765 1261.330

112158 2014-12-02 First DP-13165 1050.600

116666 2014-05-08 First KT-16480 1799.970

...

Разобравшись, как получать куски данных из датафрейма, перейдем к тому, как считать агрегированные метрики: количество заказов, суммарную выручку, средний чек, конверсию.

## **Считаем производные метрики**

Задача: посчитаем, сколько денег магазин заработал с помощью каждого класса доставки. Начнем с простого — просуммируем выручку со всех заказов. Для этого используем метод .sum():

> orders['sales'].sum()

2297200.8603000003

Добавим класс доставки. Перед суммированием сгруппируем данные с помощью метода .groupby():

> orders.groupby('ship\_mode')['sales'].sum()

ship\_mode

First 3.514284e+05

Same Day 1.283631e+05

Second 4.591936e+05

Standard 1.358216e+06

3.514284e+05 — научный формат вывода чисел. Означает 3.51 \* 105. Нам такая точность не нужна, поэтому можем сказать Pandas, чтобы округлял значения до сотых:

> pd.options.display.float\_format = '{:,.1f}'.format

> orders.groupby('ship\_mode')['sales'].sum()

ship\_mode

First 351,428.4

Same Day 128,363.1

Second 459,193.6

Standard 1,358,215.7

Другое дело. Теперь видим сумму выручки по каждому классу доставки. По суммарной выручке неясно, становится лучше или хуже. Добавим разбивку по датам заказа:

> orders.groupby(['ship\_mode', 'order\_date'])['sales'].sum()

ship\_mode order\_date

First 2014-01-06 12.8

2014-01-11 9.9

2014-01-14 62.0

2014-01-15 149.9

2014-01-19 378.6

2014-01-26 152.6

...

Видно, что выручка прыгает ото дня ко дню: иногда 10 долларов, а иногда 378. Интересно, это меняется количество заказов или средний чек? Добавим к выборке количество заказов. Для этого вместо .sum() используем метод .agg(), в который передадим список с названиями нужных функций.

> orders.groupby(['ship\_mode', 'order\_date'])['sales'].agg(['sum', 'count'])

sum count

ship\_mode order\_date

First 2014-01-06 12.8 1

2014-01-11 9.9 1

2014-01-14 62.0 1

2014-01-15 149.9 1

2014-01-19 378.6 1

2014-01-26 152.6 1

...

Ого, получается, что это так прыгает средний чек. Интересно, а какой был самый удачный день? Чтобы узнать, отсортируем получившийся датафрейм: выведем 10 самых денежных дней по выручке:

> orders.groupby(['ship\_mode', 'order\_date'])['sales'].agg(['sum']).sort\_values(by='sum', ascending=False).head(10)

sum

ship\_mode order\_date

Standard 2014-03-18 26,908.4

2016-10-02 18,398.2

First 2017-03-23 14,299.1

Standard 2014-09-08 14,060.4

First 2017-10-22 13,716.5

Standard 2016-12-17 12,185.1

2017-11-17 12,112.5

2015-09-17 11,467.6

2016-05-23 10,561.0

2014-09-23 10,478.6

Команда разрослась, и её теперь неудобно читать. Чтобы упростить, можно разбить её на несколько строк. В конце каждой строки ставим обратный слеш \:

> orders \

... .groupby(['ship\_mode', 'order\_date'])['sales'] \

... .agg(['sum']) \

... .sort\_values(by='sum', ascending=False) \

... .head(10)

sum

ship\_mode order\_date

Standard 2014-03-18 26,908.4

2016-10-02 18,398.2

First 2017-03-23 14,299.1

Standard 2014-09-08 14,060.4

First 2017-10-22 13,716.5

Standard 2016-12-17 12,185.1

2017-11-17 12,112.5

2015-09-17 11,467.6

2016-05-23 10,561.0

2014-09-23 10,478.6

В самый удачный день — 18 марта 2014 года — магазин заработал 27 тысяч долларов с помощью стандартного класса доставки. Интересно, откуда были клиенты, сделавшие эти заказы? Чтобы узнать, надо объединить данные о заказах с данными о клиентах.

## **Объединяем несколько датафреймов**

До сих пор мы смотрели только на таблицу с заказами. Но ведь у нас есть еще данные о клиентах интернет-магазина. Загрузим их в переменную customers и посмотрим, что они собой представляют:

> customers = pd.read\_csv('customers.csv', index='id')

> customers.head()

name segment state city

id

CG-12520 Claire Gute Consumer Kentucky Henderson

DV-13045 Darrin Van Huff Corporate California Los Angeles

SO-20335 Sean O'Donnell Consumer Florida Fort Lauderdale

BH-11710 Brosina Hoffman Consumer California Los Angeles

AA-10480 Andrew Allen Consumer North Carolina Concord

Мы знаем тип клиента, место его проживания, его имя и имя контактного лица. У каждого клиента есть уникальный номер id. Этот же номер лежит в колонке customer\_id таблицы orders. Значит мы можем найти, какие заказы сделал каждый клиент. Например, посмотрим, заказы пользователя CG-12520:

> cust\_filter = 'CG-12520'

> orders.query('customer\_id == @cust\_filter')

order\_date ship\_mode customer\_id sales

id

CA-2016-152156 2016-11-08 Second CG-12520 993.90

CA-2017-164098 2017-01-26 First CG-12520 18.16

US-2015-123918 2015-10-15 Same Day CG-12520 136.72

Вернемся к задаче из предыдущего раздела: узнать, что за клиенты, которые сделали 18 марта заказы со стандартной доставкой. Для этого объединим таблицы с клиентами и заказами. Датафреймы объединяют с помощью методов .concat(), .merge() и .join(). Все они делают одно и то же, но отличаются синтаксисом — на практике достаточно уметь пользоваться одним из них.

Покажу на примере .merge():

> new\_df = pd.merge(orders, customers, how='inner', left\_on='customer\_id', right\_index=True)

> new\_df.columns

Index(['order\_date', 'ship\_mode', 'customer\_id', 'sales', 'name', 'segment',

'state', 'city'],

dtype='object')

В .merge() я сначала указал названия датафреймов, которые хочу объединить. Затем уточнил, как именно их объединить и какие колонки использовать в качестве ключа.

Ключ — это колонка, связывающая оба датафрейма. В нашем случае — номер клиента. В таблице с заказами он в колонке customer\_id, а таблице с клиентами — в индексе. Поэтому в команде мы пишем: left\_on='customer\_id', right\_index=True.

## **Решаем задачу**

Закрепим полученный материал, решив задачу. Найдем 5 городов, принесших самую большую выручку в 2016 году.

Для начала отфильтруем заказы из 2016 года:

> orders\_2016 = orders.query("order\_date >= '2016-01-01' & order\_date <= '2016-12-31'")

> orders\_2016.head()

order\_date ship\_mode customer\_id sales

id

100041 2016-11-20 Standard BF-10975 328.5

100083 2016-11-24 Standard CD-11980 24.8

100153 2016-12-13 Standard KH-16630 63.9

100244 2016-09-20 Standard GM-14695 475.7

100300 2016-06-24 Second MJ-17740 4,823.1

Город — это атрибут пользователей, а не заказов. Добавим информацию о пользователях:

> with\_customers\_2016 = pd.merge(customers, orders\_2016, how='inner', left\_index=True, right\_on='customer\_id')

Cруппируем получившийся датафрейм по городам и посчитаем выручку:

> grouped\_2016 = with\_customers\_2016.groupby('city')['sales'].sum()

> grouped\_2016.head()

city

Akron 1,763.0

Albuquerque 692.9

Amarillo 197.2

Arlington 5,672.1

Arlington Heights 14.1

Name: sales, dtype: float64

Отсортируем по убыванию продаж и оставим топ-5:

> top5 = grouped\_2016.sort\_values(ascending=False).head(5)

> print(top5)

city

New York City 53,094.1

Philadelphia 39,895.5

Seattle 33,955.5

Los Angeles 33,611.1

San Francisco 27,990.0

Name: sales, dtype: float64

**Matplotlib**

Matplotlib — библиотека на языке программирования Python для визуализации данных двумерной (2D) графикой (3D графика также поддерживается).

## **2.1. График линии**

Метод построения линии очень прост:

* есть массив абсцис (x);
* есть массив ординат (y);
* элементы с одинаковым индексом в этих массивах - это координаты точек на плоскости;
* последовательные точки соединяются линией.

Под массивами, подразумеваются списки, кортежи или массивы NumPy. Кстати, последние предоставляют гораздо больше удобств чем списки и кортежи, поэтому знание пакета NumPy может значительно упростить вам жизнь.

Давайте выполним следующий код:

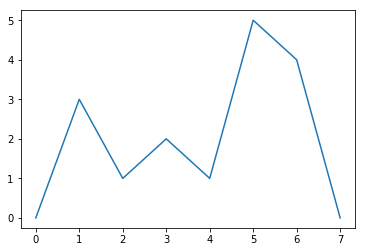
%matplotlib inline

import matplotlib.pyplot as plt

plt.plot((0, 1, 2, 3, 4, 5, 6, 7), (0, 3, 1, 2, 1, 5, 4, 0))

plt.show()

В результате мы получим вот такой простой график:



Метод plt.plot(), в простейшем случае, принимает один аргумент - последовательность чисел, которая соответствует оси ординат (y), ось абсцис (x) строится автоматически от 0 до n, где n - это длинна массива ординат. Следующий код построит точно такой же график:

%matplotlib inline

import matplotlib.pyplot as plt

plt.plot((0, 3, 1, 2, 1, 5, 4, 0))

plt.show()

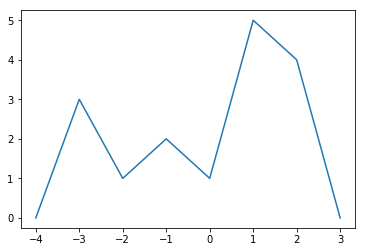
Такой способ может оказаться полезным, если диапазон чисел на оси абцис для вас не важен. Однако, если диапазон или шаг все-таки важны, то их все же необходимо указать:

%matplotlib inline

import matplotlib.pyplot as plt

plt.plot((-4, -3, -2, -1, 0, 1, 2, 3), (0, 3, 1, 2, 1, 5, 4, 0))

plt.show()

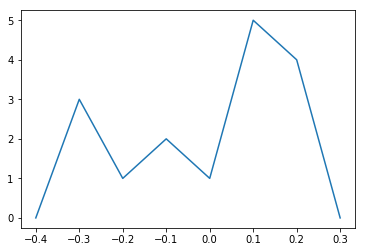


%matplotlib inline

import matplotlib.pyplot as plt

plt.plot((-0.4, -0.3, -0.2, -0.1, 0., 0.1, 0.2, 0.3), (0, 3, 1, 2, 1, 5, 4, 0))

plt.show()



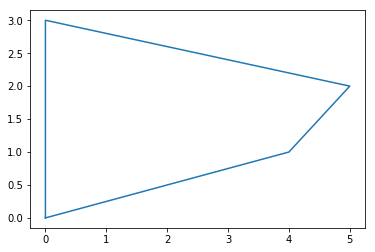
Еще один интересный момент - числа в массиве абсцис не обязательно должны быть последовательными, т.е. могут быть абсолютно произвольными, а соединяться линией будут так же только последовательные точки. Например:

%matplotlib inline

import matplotlib.pyplot as plt

plt.plot((0, 0, 5, 4, 0), (0, 3, 2, 1, 0))

plt.show()



Такое поведение очень удобно, когда вам необходимо строить плоские, замкнутые кривые или геометрические фигуры:

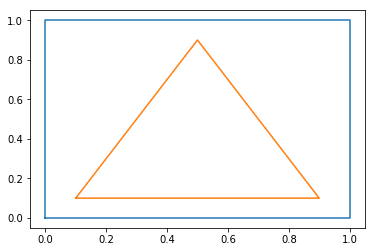
%matplotlib inline

import matplotlib.pyplot as plt

plt.plot((0, 0, 1, 1, 0), (0, 1, 1, 0, 0))

plt.plot((0.1, 0.5, 0.9, 0.1), (0.1, 0.9, 0.1, 0.1))

plt.show()



Как вы заметили мы два раза использовали метод plt.plot(), передавая разные данные. С определенной натяжкой, можна сказать, что метод plt.plot() и занимается прорисовкой наших линий, а plt.show() отображением самого графика. Но мы оставим все подробности на потом и двинемся дальше.

## **2.2. График множества точек**

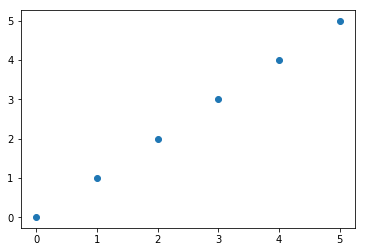
Единственное отличие графика множества точек от графика линии - точки не соединяются линией. Вот и все.

%matplotlib inline

import matplotlib.pyplot as plt

plt.scatter([0, 1, 2, 3, 4 , 5], [0, 1, 2, 3, 4 , 5])

plt.show()



Все как и прежде двум соответствующим значениям из массивов соответствуют координаты точки.

Если у вас несколько множеств, то все их так же можно построить на одном графике:

%matplotlib inline

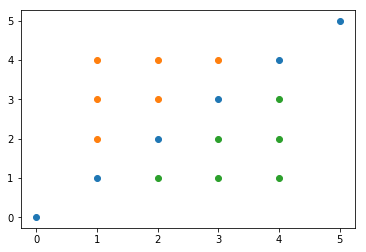
import matplotlib.pyplot as plt

plt.scatter([0, 1, 2, 3, 4 , 5], [0, 1, 2, 3, 4 , 5])

plt.scatter([1, 2, 3, 1, 2 , 1], [2, 3, 4, 3, 4 , 4])

plt.scatter([2, 3, 4, 3, 4 , 4], [1, 2, 3, 1, 2 , 1])

plt.show()



## **2.3. Гистограммы**

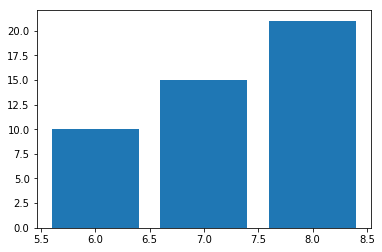
Очень часто, данные удобно представлять в виде гистограмм. В самом простом случае, гистограмма - это множество прямоугольников, площадь которых (или высота) пропорциональна какой-нибудь величине. Например, осадки за 3 месяца: в июне выпало 10 мм, в июле - 15мм, в августе - 21 мм.

%matplotlib inline

import matplotlib.pyplot as plt

plt.bar([6, 7, 8], [10, 15, 21])

plt.show()



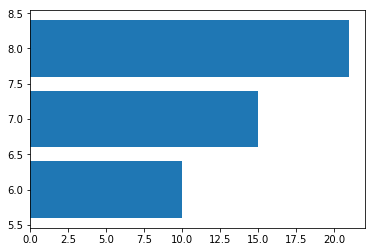
Первый массив содержит номера месяцев, а второй массив - значения показателей. Эти прямоугольники построены вертикально, но их можно отображать и в горизонтальном виде:

%matplotlib inline

import matplotlib.pyplot as plt

plt.barh([6, 7, 8], [10, 15, 21])

plt.show()



Гистограммы могут отображать несколько наборов данных, что очень удобно для их сравнения:

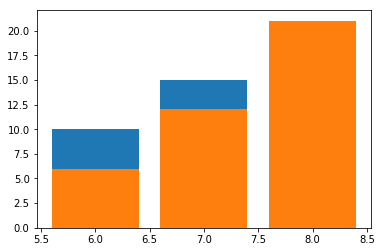
%matplotlib inline

import matplotlib.pyplot as plt

plt.bar([6, 7, 8], [10, 15, 21])

plt.bar([6, 7, 8], [6, 12, 21])

plt.show()



Такой график мог бы отображать летние осадки за два года. Но вот в чем дело, прямоугольники строятся поверх друг друга и если они равны, как в случае наших осадков за август, то прямоугольники друг друга перекроют. Если указать небольшое смещение по оси x, то ситуация не улучшится:

%matplotlib inline

import matplotlib.pyplot as plt

plt.bar([6, 7, 8], [10, 15, 21])

plt.bar([6.5, 7.5, 8.5], [6, 12, 28])

plt.show()

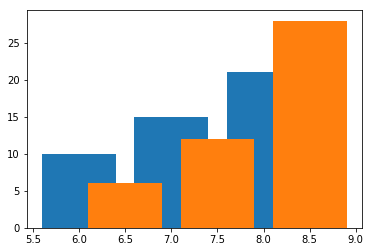


График станет привлекательней, если сузить прямоугольники и расположить их без наложения друг на друга:

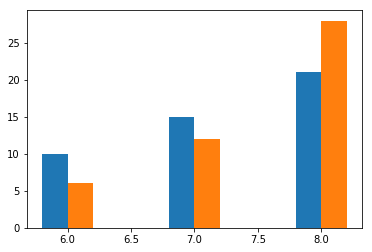
%matplotlib inline

import matplotlib.pyplot as plt

plt.bar([5.9, 6.9, 7.9], [10, 15, 21], width = 0.2)

plt.bar([6.1, 7.1, 8.1], [6, 12, 28], width = 0.2)

plt.show()



[[](https://an.yandex.ru/count/WvuejI_zO9y3fHm0L31Gvwur8Are9WK0dmGnvpVlOG00000utBaoXfFitvZMxPkV0O01ZfojBOW1jiJetbMG0SYZhvdMW8200fW1oAElcLQW0S2e0S2u0QAcg-qVm042s06CjUGWu06sYOqEw07U0VW1WWYW0ixKgHQv0iDzNY9b3uuZy0AXvDQb3y3nOO03WwIYlm-80vt2suW2jWBu19gBWWU81RNApW6G1PgBWWUW1Oh_aGAe1VEfv06m1VEfv06u1VEfv0701OR9-W781PNQFz05X_LFWk0Eg0Q2u0x91dczGVoC76Gta0SSgGTSvcovPl-G0RW7j0QdwfJgSkuuw1mCGCaGgWiGzzLfzLpc001-BSzkNDRe2xNApW7m2mQ83DBvuRu1w0oR1fWDhgW_-Vg6ui-HXx8te0x0X3tSpFFbw-o_o0604DYwxHcG4D3WqkZLWgQTkC6W4D0GrOQO4O7n7Q4HKhWUXUjHoE3Hth8Fu16xliW1w17PWvZiWyxoweWWX2L1xuaP-sQuvWQe4w30miVFp8VT5k0Jywda0O0KW22858IfmyYTulcjym6W5FEfv06e59gBWWUkpEeHs1IQyBQF1k0K0UWKZ0B85TtMk8RY1T0Lcl2sZmRO5S6AzkoZZxpyO_2O5gxtX8y6eB0MiWF95j0MqldXlW7O5e4Nc1U_lBeFg1S9k1S1m1UrCj0Nq8O3s1UVe3xe5m6P6A0O2h0OgTwyZmQu607G627u6EI8jVctsjUlL80PgE-wefAIYvOTYHa080S000000A0Peh81i1cu6S3I6H9vOM9pNtDbSdPbSYzoDZ4rBJ7W6GBe6T06y1c0mWFu6QZSgXo16l__jrye3JE5Y1h0X3sO6jJ3Kx0QvOZOXhQLbuxZ0VKQ0G0009WRhgWti1jRk1i3wHi0000W8QRqGF0RWQSKWXmDEMn2EbLZC5fXDrfND-aSW1t_VvaTu1tLhfu5g1u1q1xdwVU2a__Zyf81s1xwsXw87____m6W7zYwxHcm7m787yQQiX3I7mKrCp0mDoeWGGGH1Xa4aNICw3P03eUDa8nbpBTBZcgB2MnywLpWwX47IB0GzWbYKiG6hsCNJNytoD30qHqKlJq12P7dSkFt3FLhIyGKNFTb-insw4kkpXVAAm4qAdbc9QpLUm_WboqXoyaN7bELz1ichxD1Oy2PhFkSmR4TVmVJ8-CBTxGnKbHMapSurX0TjSt30m00~1?stat-id=8&test-tag=430510568431121&banner-sizes=eyI3MjA1NzYwNTYwNzI1NDI0NCI6Ijg0OHg5MCJ9&format-type=119&actual-format=12&pcodever=53007&banner-test-tags=eyI3MjA1NzYwNTYwNzI1NDI0NCI6IjU3MzYxIn0%3D&pcode-active-testids=487926%2C0%2C36%3B458752%2C0%2C40&width=848&height=90)](https://an.yandex.ru/count/WvuejI_zO9y3fHm0L31Gvwur8Are9WK0dmGnvpVlOG00000utBaoXfFitvZMxPkV0O01ZfojBOW1jiJetbMG0SYZhvdMW8200fW1oAElcLQW0S2e0S2u0QAcg-qVm042s06CjUGWu06sYOqEw07U0VW1WWYW0ixKgHQv0iDzNY9b3uuZy0AXvDQb3y3nOO03WwIYlm-80vt2suW2jWBu19gBWWU81RNApW6G1PgBWWUW1Oh_aGAe1VEfv06m1VEfv06u1VEfv0701OR9-W781PNQFz05X_LFWk0Eg0Q2u0x91dczGVoC76Gta0SSgGTSvcovPl-G0RW7j0QdwfJgSkuuw1mCGCaGgWiGzzLfzLpc001-BSzkNDRe2xNApW7m2mQ83DBvuRu1w0oR1fWDhgW_-Vg6ui-HXx8te0x0X3tSpFFbw-o_o0604DYwxHcG4D3WqkZLWgQTkC6W4D0GrOQO4O7n7Q4HKhWUXUjHoE3Hth8Fu16xliW1w17PWvZiWyxoweWWX2L1xuaP-sQuvWQe4w30miVFp8VT5k0Jywda0O0KW22858IfmyYTulcjym6W5FEfv06e59gBWWUkpEeHs1IQyBQF1k0K0UWKZ0B85TtMk8RY1T0Lcl2sZmRO5S6AzkoZZxpyO_2O5gxtX8y6eB0MiWF95j0MqldXlW7O5e4Nc1U_lBeFg1S9k1S1m1UrCj0Nq8O3s1UVe3xe5m6P6A0O2h0OgTwyZmQu607G627u6EI8jVctsjUlL80PgE-wefAIYvOTYHa080S000000A0Peh81i1cu6S3I6H9vOM9pNtDbSdPbSYzoDZ4rBJ7W6GBe6T06y1c0mWFu6QZSgXo16l__jrye3JE5Y1h0X3sO6jJ3Kx0QvOZOXhQLbuxZ0VKQ0G0009WRhgWti1jRk1i3wHi0000W8QRqGF0RWQSKWXmDEMn2EbLZC5fXDrfND-aSW1t_VvaTu1tLhfu5g1u1q1xdwVU2a__Zyf81s1xwsXw87____m6W7zYwxHcm7m787yQQiX3I7mKrCp0mDoeWGGGH1Xa4aNICw3P03eUDa8nbpBTBZcgB2MnywLpWwX47IB0GzWbYKiG6hsCNJNytoD30qHqKlJq12P7dSkFt3FLhIyGKNFTb-insw4kkpXVAAm4qAdbc9QpLUm_WboqXoyaN7bELz1ichxD1Oy2PhFkSmR4TVmVJ8-CBTxGnKbHMapSurX0TjSt30m00~1?stat-id=8&test-tag=430510568431121&banner-sizes=eyI3MjA1NzYwNTYwNzI1NDI0NCI6Ijg0OHg5MCJ9&format-type=119&actual-format=12&pcodever=53007&banner-test-tags=eyI3MjA1NzYwNTYwNzI1NDI0NCI6IjU3MzYxIn0%3D&pcode-active-testids=487926%2C0%2C36%3B458752%2C0%2C40&width=848&height=90" \t "_blank)

[**Вебинар для iOS-разработчиков с опытом**](https://an.yandex.ru/count/WvuejI_zO9y3fHm0L31Gvwur8Are9WK0dmGnvpVlOG00000utBaoXfFitvZMxPkV0O01ZfojBOW1jiJetbMG0SYZhvdMW8200fW1oAElcLQW0S2e0S2u0QAcg-qVm042s06CjUGWu06sYOqEw07U0VW1WWYW0ixKgHQv0iDzNY9b3uuZy0AXvDQb3y3nOO03WwIYlm-80vt2suW2jWBu19gBWWU81RNApW6G1PgBWWUW1Oh_aGAe1VEfv06m1VEfv06u1VEfv0701OR9-W781PNQFz05X_LFWk0Eg0Q2u0x91dczGVoC76Gta0SSgGTSvcovPl-G0RW7j0QdwfJgSkuuw1mCGCaGgWiGzzLfzLpc001-BSzkNDRe2xNApW7m2mQ83DBvuRu1w0oR1fWDhgW_-Vg6ui-HXx8te0x0X3tSpFFbw-o_o0604DYwxHcG4D3WqkZLWgQTkC6W4D0GrOQO4O7n7Q4HKhWUXUjHoE3Hth8Fu16xliW1w17PWvZiWyxoweWWX2L1xuaP-sQuvWQe4w30miVFp8VT5k0Jywda0O0KW22858IfmyYTulcjym6W5FEfv06e59gBWWUkpEeHs1IQyBQF1k0K0UWKZ0B85TtMk8RY1T0Lcl2sZmRO5S6AzkoZZxpyO_2O5gxtX8y6eB0MiWF95j0MqldXlW7O5e4Nc1U_lBeFg1S9k1S1m1UrCj0Nq8O3s1UVe3xe5m6P6A0O2h0OgTwyZmQu607G627u6EI8jVctsjUlL80PgE-wefAIYvOTYHa080S000000A0Peh81i1cu6S3I6H9vOM9pNtDbSdPbSYzoDZ4rBJ7W6GBe6T06y1c0mWFu6QZSgXo16l__jrye3JE5Y1h0X3sO6jJ3Kx0QvOZOXhQLbuxZ0VKQ0G0009WRhgWti1jRk1i3wHi0000W8QRqGF0RWQSKWXmDEMn2EbLZC5fXDrfND-aSW1t_VvaTu1tLhfu5g1u1q1xdwVU2a__Zyf81s1xwsXw87____m6W7zYwxHcm7m787yQQiX3I7mKrCp0mDoeWGGGH1Xa4aNICw3P03eUDa8nbpBTBZcgB2MnywLpWwX47IB0GzWbYKiG6hsCNJNytoD30qHqKlJq12P7dSkFt3FLhIyGKNFTb-insw4kkpXVAAm4qAdbc9QpLUm_WboqXoyaN7bELz1ichxD1Oy2PhFkSmR4TVmVJ8-CBTxGnKbHMapSurX0TjSt30m00~1?stat-id=8&test-tag=430510568431121&banner-sizes=eyI3MjA1NzYwNTYwNzI1NDI0NCI6Ijg0OHg5MCJ9&format-type=119&actual-format=12&pcodever=53007&banner-test-tags=eyI3MjA1NzYwNTYwNzI1NDI0NCI6IjU3MzYxIn0%3D&pcode-active-testids=487926%2C0%2C36%3B458752%2C0%2C40&width=848&height=90)

[Ве­би­нар «До­пол­нен­ная ре­аль­ность(AR) в iOS при­ло­же­ниях» Ре­ги­стри­руй­тесь сей­час](https://an.yandex.ru/count/WvuejI_zO9y3fHm0L31Gvwur8Are9WK0dmGnvpVlOG00000utBaoXfFitvZMxPkV0O01ZfojBOW1jiJetbMG0SYZhvdMW8200fW1oAElcLQW0S2e0S2u0QAcg-qVm042s06CjUGWu06sYOqEw07U0VW1WWYW0ixKgHQv0iDzNY9b3uuZy0AXvDQb3y3nOO03WwIYlm-80vt2suW2jWBu19gBWWU81RNApW6G1PgBWWUW1Oh_aGAe1VEfv06m1VEfv06u1VEfv0701OR9-W781PNQFz05X_LFWk0Eg0Q2u0x91dczGVoC76Gta0SSgGTSvcovPl-G0RW7j0QdwfJgSkuuw1mCGCaGgWiGzzLfzLpc001-BSzkNDRe2xNApW7m2mQ83DBvuRu1w0oR1fWDhgW_-Vg6ui-HXx8te0x0X3tSpFFbw-o_o0604DYwxHcG4D3WqkZLWgQTkC6W4D0GrOQO4O7n7Q4HKhWUXUjHoE3Hth8Fu16xliW1w17PWvZiWyxoweWWX2L1xuaP-sQuvWQe4w30miVFp8VT5k0Jywda0O0KW22858IfmyYTulcjym6W5FEfv06e59gBWWUkpEeHs1IQyBQF1k0K0UWKZ0B85TtMk8RY1T0Lcl2sZmRO5S6AzkoZZxpyO_2O5gxtX8y6eB0MiWF95j0MqldXlW7O5e4Nc1U_lBeFg1S9k1S1m1UrCj0Nq8O3s1UVe3xe5m6P6A0O2h0OgTwyZmQu607G627u6EI8jVctsjUlL80PgE-wefAIYvOTYHa080S000000A0Peh81i1cu6S3I6H9vOM9pNtDbSdPbSYzoDZ4rBJ7W6GBe6T06y1c0mWFu6QZSgXo16l__jrye3JE5Y1h0X3sO6jJ3Kx0QvOZOXhQLbuxZ0VKQ0G0009WRhgWti1jRk1i3wHi0000W8QRqGF0RWQSKWXmDEMn2EbLZC5fXDrfND-aSW1t_VvaTu1tLhfu5g1u1q1xdwVU2a__Zyf81s1xwsXw87____m6W7zYwxHcm7m787yQQiX3I7mKrCp0mDoeWGGGH1Xa4aNICw3P03eUDa8nbpBTBZcgB2MnywLpWwX47IB0GzWbYKiG6hsCNJNytoD30qHqKlJq12P7dSkFt3FLhIyGKNFTb-insw4kkpXVAAm4qAdbc9QpLUm_WboqXoyaN7bELz1ichxD1Oy2PhFkSmR4TVmVJ8-CBTxGnKbHMapSurX0TjSt30m00~1?stat-id=8&test-tag=430510568431121&banner-sizes=eyI3MjA1NzYwNTYwNzI1NDI0NCI6Ijg0OHg5MCJ9&format-type=119&actual-format=12&pcodever=53007&banner-test-tags=eyI3MjA1NzYwNTYwNzI1NDI0NCI6IjU3MzYxIn0%3D&pcode-active-testids=487926%2C0%2C36%3B458752%2C0%2C40&width=848&height=90)

[Узнать больше](https://an.yandex.ru/count/WvuejI_zO9y3fHm0L31Gvwur8Are9WK0dmGnvpVlOG00000utBaoXfFitvZMxPkV0O01ZfojBOW1jiJetbMG0SYZhvdMW8200fW1oAElcLQW0S2e0S2u0QAcg-qVm042s06CjUGWu06sYOqEw07U0VW1WWYW0ixKgHQv0iDzNY9b3uuZy0AXvDQb3y3nOO03WwIYlm-80vt2suW2jWBu19gBWWU81RNApW6G1PgBWWUW1Oh_aGAe1VEfv06m1VEfv06u1VEfv0701OR9-W781PNQFz05X_LFWk0Eg0Q2u0x91dczGVoC76Gta0SSgGTSvcovPl-G0RW7j0QdwfJgSkuuw1mCGCaGgWiGzzLfzLpc001-BSzkNDRe2xNApW7m2mQ83DBvuRu1w0oR1fWDhgW_-Vg6ui-HXx8te0x0X3tSpFFbw-o_o0604DYwxHcG4D3WqkZLWgQTkC6W4D0GrOQO4O7n7Q4HKhWUXUjHoE3Hth8Fu16xliW1w17PWvZiWyxoweWWX2L1xuaP-sQuvWQe4w30miVFp8VT5k0Jywda0O0KW22858IfmyYTulcjym6W5FEfv06e59gBWWUkpEeHs1IQyBQF1k0K0UWKZ0B85TtMk8RY1T0Lcl2sZmRO5S6AzkoZZxpyO_2O5gxtX8y6eB0MiWF95j0MqldXlW7O5e4Nc1U_lBeFg1S9k1S1m1UrCj0Nq8O3s1UVe3xe5m6P6A0O2h0OgTwyZmQu607G627u6EI8jVctsjUlL80PgE-wefAIYvOTYHa080S000000A0Peh81i1cu6S3I6H9vOM9pNtDbSdPbSYzoDZ4rBJ7W6GBe6T06y1c0mWFu6QZSgXo16l__jrye3JE5Y1h0X3sO6jJ3Kx0QvOZOXhQLbuxZ0VKQ0G0009WRhgWti1jRk1i3wHi0000W8QRqGF0RWQSKWXmDEMn2EbLZC5fXDrfND-aSW1t_VvaTu1tLhfu5g1u1q1xdwVU2a__Zyf81s1xwsXw87____m6W7zYwxHcm7m787yQQiX3I7mKrCp0mDoeWGGGH1Xa4aNICw3P03eUDa8nbpBTBZcgB2MnywLpWwX47IB0GzWbYKiG6hsCNJNytoD30qHqKlJq12P7dSkFt3FLhIyGKNFTb-insw4kkpXVAAm4qAdbc9QpLUm_WboqXoyaN7bELz1ichxD1Oy2PhFkSmR4TVmVJ8-CBTxGnKbHMapSurX0TjSt30m00~1?stat-id=8&test-tag=430510568431121&banner-sizes=eyI3MjA1NzYwNTYwNzI1NDI0NCI6Ijg0OHg5MCJ9&format-type=119&actual-format=12&pcodever=53007&banner-test-tags=eyI3MjA1NzYwNTYwNzI1NDI0NCI6IjU3MzYxIn0%3D&pcode-active-testids=487926%2C0%2C36%3B458752%2C0%2C40&width=848&height=90" \t "_blank)

[otus.ru](https://an.yandex.ru/count/WvuejI_zO9y3fHm0L31Gvwur8Are9WK0dmGnvpVlOG00000utBaoXfFitvZMxPkV0O01ZfojBOW1jiJetbMG0SYZhvdMW8200fW1oAElcLQW0S2e0S2u0QAcg-qVm042s06CjUGWu06sYOqEw07U0VW1WWYW0ixKgHQv0iDzNY9b3uuZy0AXvDQb3y3nOO03WwIYlm-80vt2suW2jWBu19gBWWU81RNApW6G1PgBWWUW1Oh_aGAe1VEfv06m1VEfv06u1VEfv0701OR9-W781PNQFz05X_LFWk0Eg0Q2u0x91dczGVoC76Gta0SSgGTSvcovPl-G0RW7j0QdwfJgSkuuw1mCGCaGgWiGzzLfzLpc001-BSzkNDRe2xNApW7m2mQ83DBvuRu1w0oR1fWDhgW_-Vg6ui-HXx8te0x0X3tSpFFbw-o_o0604DYwxHcG4D3WqkZLWgQTkC6W4D0GrOQO4O7n7Q4HKhWUXUjHoE3Hth8Fu16xliW1w17PWvZiWyxoweWWX2L1xuaP-sQuvWQe4w30miVFp8VT5k0Jywda0O0KW22858IfmyYTulcjym6W5FEfv06e59gBWWUkpEeHs1IQyBQF1k0K0UWKZ0B85TtMk8RY1T0Lcl2sZmRO5S6AzkoZZxpyO_2O5gxtX8y6eB0MiWF95j0MqldXlW7O5e4Nc1U_lBeFg1S9k1S1m1UrCj0Nq8O3s1UVe3xe5m6P6A0O2h0OgTwyZmQu607G627u6EI8jVctsjUlL80PgE-wefAIYvOTYHa080S000000A0Peh81i1cu6S3I6H9vOM9pNtDbSdPbSYzoDZ4rBJ7W6GBe6T06y1c0mWFu6QZSgXo16l__jrye3JE5Y1h0X3sO6jJ3Kx0QvOZOXhQLbuxZ0VKQ0G0009WRhgWti1jRk1i3wHi0000W8QRqGF0RWQSKWXmDEMn2EbLZC5fXDrfND-aSW1t_VvaTu1tLhfu5g1u1q1xdwVU2a__Zyf81s1xwsXw87____m6W7zYwxHcm7m787yQQiX3I7mKrCp0mDoeWGGGH1Xa4aNICw3P03eUDa8nbpBTBZcgB2MnywLpWwX47IB0GzWbYKiG6hsCNJNytoD30qHqKlJq12P7dSkFt3FLhIyGKNFTb-insw4kkpXVAAm4qAdbc9QpLUm_WboqXoyaN7bELz1ichxD1Oy2PhFkSmR4TVmVJ8-CBTxGnKbHMapSurX0TjSt30m00~1?stat-id=8&test-tag=430510568431121&banner-sizes=eyI3MjA1NzYwNTYwNzI1NDI0NCI6Ijg0OHg5MCJ9&format-type=119&actual-format=12&pcodever=53007&banner-test-tags=eyI3MjA1NzYwNTYwNzI1NDI0NCI6IjU3MzYxIn0%3D&pcode-active-testids=487926%2C0%2C36%3B458752%2C0%2C40&width=848&height=90" \t "_blank)

РЕКЛАМА

## **2.4. Круговые диаграммы**

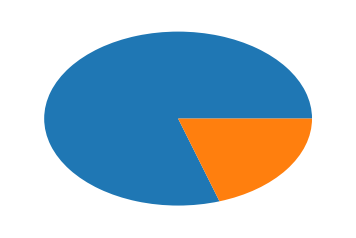
Если вам необходимо наглядно отобразить соотношение частей целого, то лучше воспользоваться круговой диаграммой. Например, в компании работают 50 человек из них 40 женщи и 10 мужчин:

%matplotlib inline

import matplotlib.pyplot as plt

plt.pie([40, 10])

plt.show()



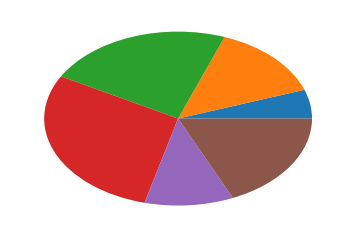
Все предельно просто - количество элементов в массиве определяет количество клиньев, а величина значений определяет их площадь:

%matplotlib inline

import matplotlib.pyplot as plt

plt.pie([5, 13, 21, 27, 10, 17])

plt.show()



## **2.5. Ящик с усами**

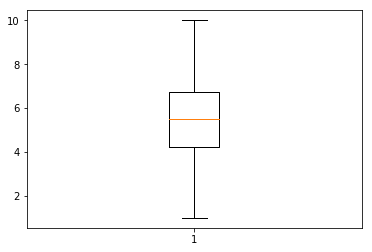
Данный тип графиков действительно похож на ящик с усами (если повернуть монитор на 90o):

%matplotlib inline

import matplotlib.pyplot as plt

plt.boxplot([1, 5, 7, 4, 6, 10])

plt.show()



Но на самом деле этот ящик с усами является диаграммой размаха, служит для отображения случайной величины и несет в себе достаточно много информации. Во первых, внутри ящика оранжевой линией отмечена медиана элементов массива - это такое значение которое меньше и больше ровно половины элементов массива. В нашем случае, это значение равно 5.5 и как нетрудно заметить половина элементов меньше его, а другая больше. Его границами служат 25-й и 75-й процентили (4.25 и 6.75 для нашего массива). Ну а усами, собственно (как правило) максимальное и минимальное значение в наборе данных. Вот такой непростой, но очень полезный ящик.

Иногда на графике, рядом с усами появляются одна или две точки. Такие точки обозначают выбросы - значения которые находятся очень далеко от статистически значимой части данных:

%matplotlib inline

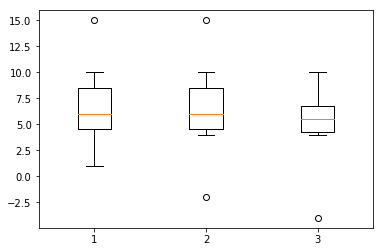
import matplotlib.pyplot as plt

plt.boxplot([[1, 5, 7, 4, 6, 10, 15],

[-2, 5, 7, 4, 6, 10, 15],

[-4, 5, 7, 4, 6, 10]])

plt.show()



Как не трудно догадаться, в основе данного типа графиков находится статистическая подоплека и наиболее полезен он именно в этой области.

**Объектно-ориентированное программирование**

Наиболее классическое определение, к которому прибегают при необходимости объяснить что такое ООП, это — «способ моделирования реального мира».‎ Можно предположить, что ООП делает код более простым и наглядным, однако такая формулировка слишком размыта и уклончива, она не открывает самой сути ООП.

ООП стоит на трёх китах:

Инкапсуляция — способ спрятать сложную логику внутри класса, предоставив программисту лаконичный и понятный интерфейс для взаимодействия с сущностью.

Наследование — способ легко и просто расширить существующий класс, дополнив его функциональностью.

Полиморфизм — принцип «один интерфейс — множество реализаций». Например, метод print может вывести текст на экран, распечатать его на бумаге или вовсе записать в файл.

Если резюмировать: ООП даёт контроль над зависимостями в коде. Это способ сделать так, чтобы высокоуровневый код не зависел от низкоуровневой реализации. ООП позволяет вести разработку раздельно, поскольку взаимодействие между сущностями определено интерфейсами.

-

Самый простой способ объяснить и понять ООП — воспользоваться метафорой. Метафорой объекта в ООП является объект реального мира, например, человек. Объекты надо отличать между собой и у них есть что-то, что их определяет. Например, для человека это может быть имя, когда мы говорим про нашего знакомого Васю, и все понимают о ком речь. Люди неким образом похожи друг на друга. Подмножество людей, обладающих одинаковым набором свойств (имя, фамилия, возраст и т.д.) и общим поведением, будет называться класс. Возьмем для примера сотрудников нашей компании. Для каждого из нас определен департамент (я, например, в департаменте разработки ПО числюсь, ДРПО), должность, уровень зарплаты и т.д. Эти свойства обычно определяют в момент, когда в компанию приходит новый сотрудник. У человека можно запросить информацию по его навыкам или попросить помочь коллеге — это общее поведение для всех сотрудников.

Зарплату сотрудника знает он сам, его руководитель и бухгалтер, остальные — нет. Такое сокрытие данных называется инкапсуляция. Какие свойства и поведение будет доступно другим объектам обычно определяется на уровне класса. Руководитель отдела также является сотрудником, но он обладает рядом дополнительных свойств, например, у него есть подчиненные. Таким образом класс «руководитель», расширяет класс «сотрудник» или, другими словами, происходит наследование. При этом между классами устанавливается отношение «является» — то есть любой руководитель является сотрудником, но не наоборот — не каждый сотрудник является руководителем. Если у класса больше одного наследника, то образуется иерархия. Классы, которые являются родственниками в иерархии не связаны отношением «является», например, бухгалтер является сотрудником, но бухгалтер не является руководителем.

При помощи этих правил иерархию можно проверить на корректность. Если взять ведомость со списком всех сотрудников, в нее очевидным образом попадут и руководители, и бухгалтеры, но в общем списке они не будут отличаться от других сотрудников. Если мы захотим уточнить список подчиненных у каждого руководителя, то нам понадобится подготовить отдельную ведомость со свойствами, специфичными для класса «руководитель». Такое свойство объектов называется полиморфизмом, где состав свойств и поведение будет определяться классом, через который мы смотрим на объект: мы можем обращаться к объекту, как и к любому из предков его класса, но это не верно для потомков или других родственников.

Так мы рассмотрели, как связаны объекты и классы, и такие понятия, как: инкапсуляция, наследование и полиморфизм. Все это — базовые понятия ООП.

-

Методология объектно-ориентированного программирования (ООП) подразумевает представление всей программы или ее частей объектами. У каждого объекта есть тип — в ООП он называется классом. Классы можно объявлять или наследовать и создавать из них экземпляры. Собственно, объект — это и есть экземпляр класса.

Обычно объект объединяет в себе данные и методы для работы с ними. Представим, что у нас есть тип «Позвоночное существо», у которого есть свойство «Класс». У каждого из Позвоночных существ это свойство равно одному из пяти значений: Рыба, Земноводное, Птица, Пресмыкающееся, Млекопитающее. Добавим метод получить\_класс — он будет возвращать это значение. Далее объявим тип «Человек», который наследует типу «Позвоночное существо». Создадим несколько экземпляров: Иван Иванов, Марина Иванова, Антон Антонов. Добавим присущие только Человеку свойства: Имя и Фамилию. У каждого из них будет метод получить\_имя/получить\_фамилию, а также перешедший от Позвоночного существа метод получить\_класс.

Так можно продолжать сколь угодно долго: повторно описывать методы родительских классов нам не нужно, и любой экземпляр класса будет обладать заявленными свойствами.

Основные задачи ООП — структурировать код, повысить его читабельность и ускорить понимание логики программы. Косвенно выполняются и другие задачи: например, повышается безопасность кода и сокращается его дублирование.

Дело в том, что человеку гораздо удобнее работать с реальными объектами, чем отдельно с набором данных и функциями. Представляя данные в программе как свойства объекта, а функции по обработке данных — как возможные методы объекта, мы приближаем процесс программирования к процессу описания метода решения задачи. Это достигается за счет добавления знакомой человеку структуры абстракций: ведь даже язык, на котором мы говорим, следует принципам ООП. У каждой буквы есть произношение и написание, каждое слово включает буквы и имеет свое произношение и написание, то же верно и для предложений, и для более крупных конструкций. Все в этом мире — объект!

Главное, о чем не стоит забывать: ООП — это не единственная парадигма. У нее есть свои плюсы и минусы, для каких-то задач она подходит, для каких-то — нет. Например, ООП не даст особых преимуществ, если вы пишете «однострочники» и простые скрипты. Однако в больших проектах неразделенный на отдельные сущности код быстро превратится в «лапшу» и перестанет читаться, и ООП здесь сильно упростит работу.