# Приветствие

## О блоке

Этот блок включает в себя:

- вступительный тест по Python, позволяющий определить свой уровень;
- описание процесса установки среды разработки для прохождения курса;
- описание синтаксиса языка программирования Python;
- разбор типовых синтаксических конструкций при программировании на Python;
- примеры анализа возникающих ошибок, помогающие в дальнейшем прохождении курса.

## Вводная лекция про Python

Добро пожаловать во вводный блок курса! Если у вас есть опыт программирования на `Python`, рекомендуем пройти тест по ссылке – так вы сможете понять, какие разделы стоит изучить или освежить в памяти перед прохождением курса. Тест можно игнорировать, если вы только начинаете свое знакомство с `Python`.

Итак, `Python` – это такой язык программирования, который позволяет сообщить компьютеру о том, что нужно сделать, чтобы достичь некоего результата. За последнее десятилетие он получил быстрое распространение и сейчас является одним из самых популярных языков программирования в мире. Входной порог для его использования достаточно низок: вы можете использовать `Python` для решения своих задач даже если никогда не имели дела с программированием.

### Что такое `Python`?

`Python` – это язык программирования *общего назначения*, используемый во многих приложениях. Например:

- разработка веб-приложений;
- создание игр;
- продвинутая аналитика данных, в том числе с использованием нейронных сетей;
- компьютерная графика;
- геофизика;
- психология;
- химия;
- теория графов.

`Python` используют практически все крупные компании, о которых вы слышите каждый день: Google, Yandex, YouTube, Dropbox, Amazon, Facebook … список можно продолжать часами.

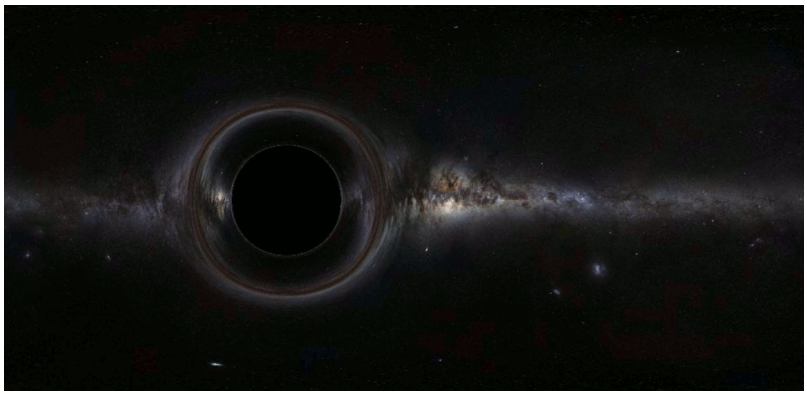


**Fig. 1** Рассчитанная в Python симуляция преломлений света черной дырой

### Чем примечателен `Python`?

В основе разностороннего применения и популярности лежит **простота изучения**: все чаще люди начинают свой путь в программировании с Python, поскольку он очень **дружелюбен к новичкам** и позволяет максимально быстро перейти к решению целевой задачи.

Сюда же можно отнести **многообразие библиотек** (*или расширений функциональности*, то есть кода, написанного другими людьми, который вы можете переиспользовать). Хотите изучить физику небесных тел и симулировать их взаимодействия? Можно найти и скачать библиотеку, позволяющую за один вечер провести вычисления, о которых в прошлом веке можно было лишь мечтать. Хотите создать прототип мобильного приложения? И на этот случай есть библиотека. Вам нравится квантовая физика и вы хотите использовать ее вместе с умными компьютерными алгоритмами? Что ж, тогда вы снова по адресу.

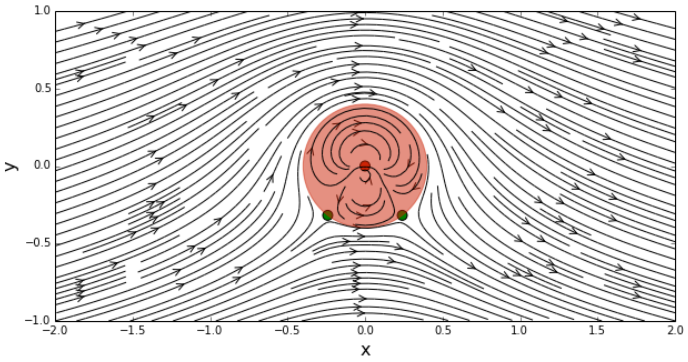



**Fig. 2** Пример моделирования аэродинамики в Python с помощью библиотеки AeroPython

Python – это **высокоуровневый язык для быстрой разработки и/или прототипирования**, на нем очень удобно проверять гипотезы и идеи. "*Высокоуровневый*" означает, что вам не нужно вникать в устройство компьютера и тонкости взаимодействия с ним, чтобы перейти к задаче. Многое «сделано за вас»: вы работаете с простыми *абстракциями* (или удобными представлениями), а не боретесь с компьютером из-за непонимания сложностей его устройства.

Еще один плюс в копилку популярности языка – это **элегантность и краткость синтаксиса** (принципов написания кода, как будто это абзацы в тексте или колонки в газете). Вместе с вышеупомянутым обилием библиотек вы можете буквально за 5 минут и 10 строк кода – а это меньше половины листа А4 – воспроизвести научную статью, в которую вложено несколько человеко-лет. А еще такой синтаксис делает **код легким для чтения, запоминания и понимания**.

Стоит отметить, что Python – это **интерпретируемый** язык, а значит, компьютер каждый раз перед выполнением программы читает код строчку за строчкой и определяет (интерпретирует), что нужно сделать дальше, не проводя никаких оптимизаций и предварительных расчетов. Это негативно влияет на общую скорость работы: Python является одним из самых медленных языков. Тем не менее он отлично подходит для академических целей, например, исследовательской работы или других задач, где скорость работы не является критически важной. Настоящая сила Python заключается в том, что это **"язык-клей"**: он обеспечивает удобный доступ к различным библиотекам, написанным на высокоэффективных языках, например, на C/C++, Fortran, CUDA C и других.

### И в чем подвох?

В простоте языка и его доступности для быстрого старта таится одна из проблем: вы *можете не понимать, что происходит внутри*, поэтому иногда бывает сложно разобраться в причинах ошибок и неточностей, возникающих по ходу работы над задачей. В целом к Python применим следующий принцип: "**Easy to learn, hard to master**". Возвращаясь к примеру элегантности кода, когда 10 строк кода выполняют всю работу: важно понимать, что за ними стоят еще *сотни* или даже *тысячи строк кода*, а это может приводить к ситуациям, когда поиск ошибки в минимальном наборе команд растягивается на несколько дней.

## Но не пугайтесь!

В данном блоке мы постараемся дать вам всю необходимую интуицию и теорию для успешного прохождения настоящего курса, ответим на основные вопросы, покажем типовые примеры использования Python и разберем классические ошибки.

Дополнительно отметим, что `Python` хорош и для квантового машинного обучения (`QML`), ради которого весь курс и затеян, в особенности – для классического машинного обучения (`ML`). В области `ML` этот язык программирования стал де-факто стандартом, который используют практически все специалисты.

### Интересные факты

Еще немного дополнительной информации про `Python`:

- Разработчик языка Гвидо ван Россум назвал его в честь популярного британского комедийного телешоу 1970-х «Летающий цирк Монти Пайтона».
- Актуальной версией `Python` считаются версии `3.6` и выше (`3.7`, `3.8.12`…). Долгое время (до 2020) года существовал `Python 2`, который ныне не поддерживается и не обновляется. Если вы видите кусок кода на `Python 2` и вам предстоит работать с ним, возможно, сначала придется его переписать, хотя большая часть кода имеет совместимость и работает корректно. В этом курсе мы не будем изучать `Python 2`.
- У `Python` огромное сообщество: большинство проблем, с которыми вы можете столкнуться, уже было озвучено и даже решено. Это означает, что используя поисковик, вы можете решить практически все проблемы в течение 10-30 минут. Главное – научиться правильно формулировать свои вопросы.
- При работе с `Python` следует придерживаться принципа “должен существовать один и, желательно, только один очевидный способ сделать это”. Другие принципы (`Дзен Питона`) на русском языке – по ссылке.
- `Python` – это *открытый проект*, в который каждый может внести изменения (но они должны быть предварительно одобрены), например, тут.
- Есть целый набор рекомендаций и предложений по улучшению кода (`PEP`, Python Enhancement Proposals). Они содержат указания на то, как следует писать код и чего стоит избегать, а также дискуссии о будущих изменениях в языке.
- Язык постоянно развивается, в нем появляются новые возможности, улучшается производительность (скорость выполнения).
- Сборник всех существующих в открытом доступе библиотек находится тут.
- Если вы столкнетесь с багом (системной ошибкой, вызванной внутренним механизмом языка), то сообщить об этом можно на специальном сайте.

**Fig. 3** Кот для привлечения внимания и в благодарность за то, что вы начали проходить курс и сделали самый сложный шаг – прошли первую лекцию!

# Знакомство с инструментарием: Jupyter

## Описание лекции

Эта лекция расскажет:

- что такое `Jupyter` и чем он хорош;
- о видах ячеек и режимах работы в `Jupyter`;
- о самых необходимых горячих клавишах;
- что такое ядро и почему вас это должно волновать.

# Введение

Как любой мастер должен знать свой инструмент, так и любой человек, решивший пройти курс QML, должен понимать тонкости рабочей среды. Как вы уже могли понять по прошлому занятию, всю (или большую часть) работы мы будем делать в `Jupyter Notebook`. Но бояться нечего: по сути, `Jupyter` – это *продвинутый текстовый редактор* с функцией запуска кода и получения результатов вычислений. Настолько продвинутый, что позволяет вам не только рисовать картинки и писать формулы, но даже строить целые интерактивные карты:

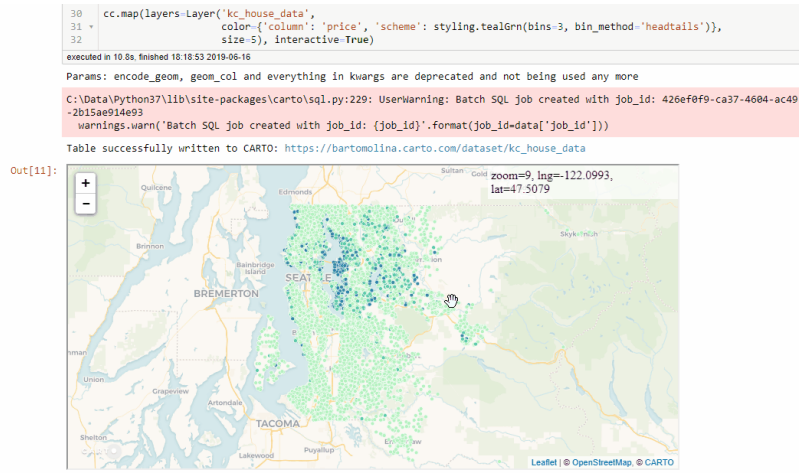



**Fig. 4** Пример интерактивной визуализации прямо внутри рабочего файла с кодом

На курсе, конечно, работы с гео-данными не будет, однако очень пригодятся вывод информации в виде *таблиц*, *создание и отрисовка простых графиков*, а главное, все это происходит в **браузере**. Редактирование кода в браузере не вызывает лишних проблем со средами разработки и в то же время оно доступно максимально широкому кругу людей. В этом редакторе можно запускать `Python`-код, что очень похоже на интерактивный редактор MATLAB, если вы с ним знакомы.

Благодаря удобству использования и доступности `Jupyter` в настоящее время стал крупным игроком в нише научных вычислений и быстрого прототипирования. Вдобавок он безумно удобен для обучения и передачи знаний. Почему? Давайте разбираться.

# Типы ячеек

В `Jupyter` существует несколько типов **ячеек**, мы поговорим о двух основных: **Code** и **Markdown**. В прошлом уроке мы создали пустой **ноутбук**, чтобы проверить установку `Jupyter`. **Ноутбуком** это называется потому, что в переводе с английского **notebook** – это тетрадка (альтернативное название на русском языке). В тетрадке можно писать что-то осмысленное, черкаться, оставлять пометки. Сейчас вы должны видеть вот такой экран:

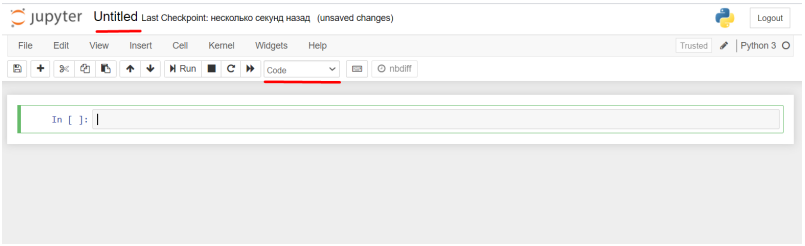



**Fig. 5** Пример пустой только что созданной тетрадки.

Здесь верхней красной чертой выделено поле с **названием** ноутбука. Можете кликнуть по нему, переименовать во что-то осмысленное и нажать `Enter`, чтобы применить изменения. Нижней же чертой обозначен выпадающий список переключений **типов ячеек**. По умолчанию создана одна **code** ячейка – в ней в будущем мы будем писать `Python`-код. Попробуйте кликнуть по списку и выбрать **Markdown** – визуально ячейка немного изменится.

> **Что такое Markdown?**
>
> **Markdown** (произносится маркда́ун) – облегченный язык разметки, созданный с целью обозначения **форматирования** в простом тексте с максимальным сохранением его читаемости человеком.

Пример: `Text attributes _italic_,**bold**, `monospace`.`

С помощью Markdown можно разнообразить код, вставить формулы (в том числе в LaTeX формате, если он вам знаком), ссылки на статьи и многое другое. Попробуйте скопировать код из примера выше в Markdown-ячейку и нажать кнопку Run:

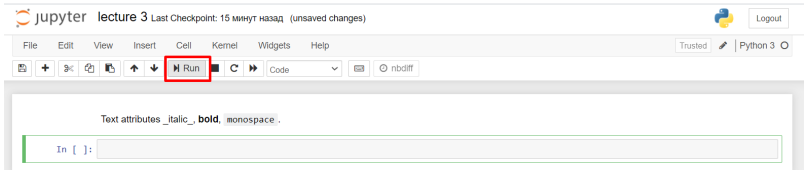
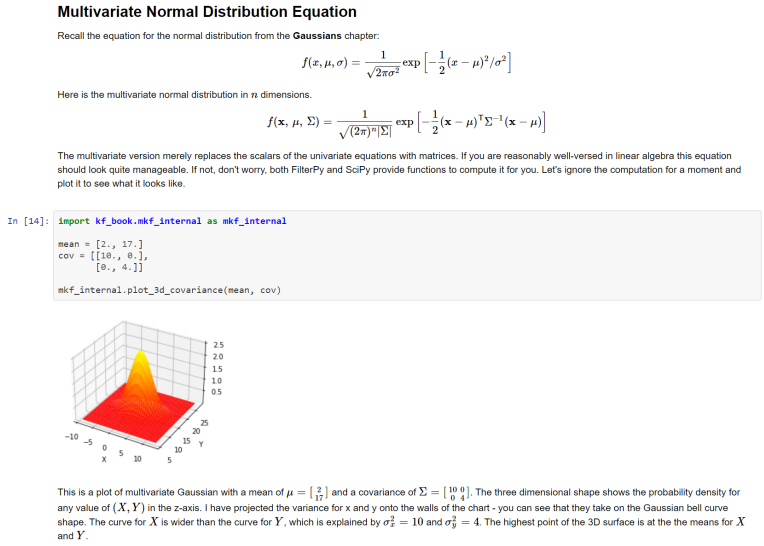



**Fig. 6** Пример выполненной ячейки. Красным квадратом выделена кнопка Run.

Произошло следующее: ваша ячейка выполнилась и `Jupyter` отобразил ее содержимое. С помощью такого форматирования можно писать целые статьи с выкладками, формулами, графиками, то есть сопроводительной информацией. Поэтому, как уже было сказано, тетрадки очень удобны, особенно если соблюдать структуру, то есть писать сверху вниз с разделением на логические блоки. Также стоит отметить, что создалась новая Code-ячейка прямо под первой.



**Multivariate Normal Distribution Equation**

Recall the equation for the normal distribution from the **Gaussians** chapter:

$$f(x,\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left[-\frac{1}{2}(x-\mu)^2/\sigma^2\right]$$

Here is the multivariate normal distribution in $n$ dimensions.

$$f(\mathbf{x},\,\mu,\,\Sigma) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}}\exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^\mathsf{T}\Sigma^{-1}(\mathbf{x}-\mu)\right]$$

The multivariate version merely replaces the scalars of the univariate equations with matrices. If you are reasonably well-versed in linear algebra this equation should look quite manageable. If not, don't worry, both FilterPy and SciPy provide functions to compute it for you. Let's ignore the computation for a moment and plot it to see what it looks like.

```
In [14]: import kf_book.mkf_internal as mkf_internal

         mean = [2., 17.]
         cov = [[10., 0.],
                [0., 4.]]

         mkf_internal.plot_3d_covariance(mean, cov)
```



This is a plot of multivariate Gaussian with a mean of $\mu = \begin{bmatrix}2\\17\end{bmatrix}$ and a covariance of $\Sigma = \begin{bmatrix}10&0\\0&4\end{bmatrix}$. The three dimensional shape shows the probability density for any value of $(X, Y)$ in the z-axis. I have projected the variance for x and y onto the walls of the chart - you can see that they take on the Gaussian bell curve shape. The curve for $X$ is wider than the curve for $Y$, which is explained by $\sigma_x^2 = 10$ and $\sigma_y^2 = 4$. The highest point of the 3D surface is at the the means for $X$ and $Y$.

**Fig. 7** Пример грамотно оформленной Jupter-тетрадки. Такую можно скинуть коллегам – и всем все будет понятно!

> **Note**
>
> Самая прекрасная часть тетрадок: **ячейки разных типов можно смешивать по порядку**, таким образом сначала описывая какую-то логику, а затем непосредственно реализовывая ее в коде и выполняя.

Вам не так часто придется писать Markdown-заметки самостоятельно, основная причина их создания – ваше желание и дальнейшее удобство использования ноутбука. Есть также другая причина, по которой мы акцентируем на них внимание. Может так произойти, что вы *случайно* изменили тип ячейки и не заметили этого. Теперь, если в Markdown-ячейку вставить `Python`-код, то ничего не произойдет или возникнет ошибка. Если вы заметили что-то странное при выполнении кода в тетрадке – **проверьте, корректен ли тип ячейки**. Для выполнения кода нужно выставить тип **Code**.

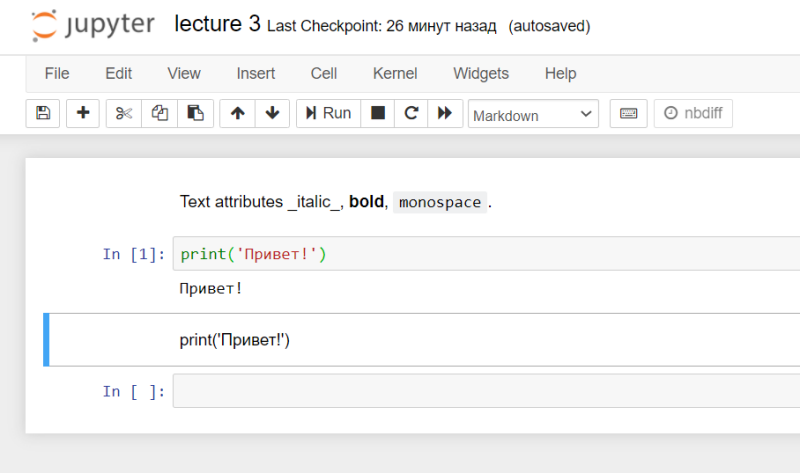



**Fig. 8** Верхняя ячейка с `print` написана в Code-режиме и корректно выполняется, печатая строчку с приветствием. Нижняя ячейка же содержит **текст, а не код**, поэтому ничего работать не будет (точнее, код отобразится как текст, но не будет выполнен).

> **Tip**
>
> Обратите внимание, как визуально отличаются эти две ячейки. Одна из них имеет прозрачный фон, другая – серый. У **Code** ячейки также есть странная надпись слева (про нее еще поговорим).

Каждый раз вручную запускать код (или Markdown) через кнопку `Run` не очень-то удобно, поэтому можно запомнить две комбинации клавиш. `CTRL+Enter` **выполнит** текущую ячейку и оставит “курсор” (указатель на ячейку) **на том же месте**, не создавая лишнюю строчку в ноутбуке. `Shift+Enter` повторит функциональность кнопки `Run`: выполнит ячейку, а затем **перейдет на следующую** (или **создаст новую**, если текущая ячейка является **последней**).

Первая комбинация (`CTRL+Enter`) будет полезна в том случае, если вы что-то написали и знаете, что будете вносить изменения (например, менять цвет линии на графике в попытках добиться визуальной красоты), а значит, придется менять код в этой же ячейке.

Вторая (`Shift+Enter`) пригодится тогда, когда вы хотите запустить много-много идущих подряд ячеек (можете представить, что коллега скинул вам свою тетрадку с 30 клетками и вы хотите ее запустить, чтобы получить данные).

Не беспокойтесь, буквально к концу первого блока лекций у вас выработается мышечная память и вы будете использовать сочетания клавиш на автомате.

## Режимы работы

Пришло время разобраться с цветом курсора, выделяющего ячейки. Он может быть **синим** или **зеленым**.

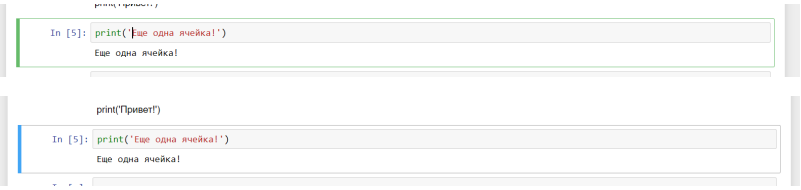



**Fig. 9** Пример разного цвета указателя клетки.

Никакой тайны за этим нет, это два режима: **режим редактирования и командный режим**. Зеленый цвет сигнализирует о том, что вы работаете **с текстовым содержимым ячейки**, то есть редактируете его! Можете писать код, вставлять формулы, что угодно. Но как только вы нажмете `ESC` на клавиатуре, цвет сменится на синий, что означает **возможность редактирования всего ноутбука, а не отдельных ячеек в нем**. Можно передвигать ячейки, удалять их (полностью, а не только текст в них), добавлять новые. Стрелочками на клавиатуре можно выбирать ячейки (скакать вверх и вниз). Как только доберетесь до нужной (а вместо этого можно просто кликнуть по ней мышкой, что полезно в ситуации, когда клетка *очень* далеко, в самом низу страницы) – жмите `Enter`, чтобы вернуться к редактированию.

> **Tip**
>
> Можно осуществлять переходы между режимами кликом мышки (внутри блока кода либо где-нибудь в стороне, слева или справа от ячейки, где ничего нет).

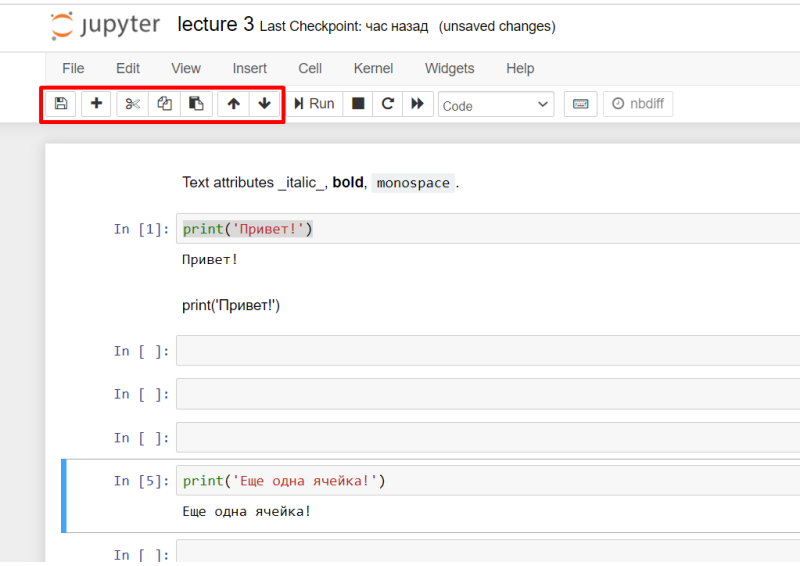



**Fig. 10** Кнопки управления ноутбуком.

Выполнять описанные выше операции можно с помощью горячих клавиш (или хоткеев), либо через интерфейс. Описание выделенного блока кнопок для картинки выше (в порядке слева направо, с указанием сочетаний клавиш):

1. *Сохранение ноутбука* (`CTRL+S`) – делайте его **почаще**, чтобы не потерять результаты работы!
2. *Создание ячейки ниже текущей* (`B`) – `B` потому, что создается клетка снизу, то есть `Below`. Логика для `A` и `Above` аналогична.
3. *Вырезать ячейку* (`X`) – применимо и к целому блоку ячеек (можно выделить с зажатой клавишей `Shift`). Функциональность как и в Excel/Word: убрать в одном месте, чтобы вставить в другом.
4. *Копировать ячейку* (`C`).
5. *Вставить ячейку из буфера* (`V`) – после вырезания или копирования ячейки.
6. *Переместить текущую выделенную ячейку **вверх***.
7. *Переместить текущую выделенную ячейку **вниз***.

Описание всех доступных команд (и соответствующих им хоткеев) доступно при нажатии на кнопку с клавиатурой в правой части выделенного блока меню (вне красного прямоугольника).

> **Attention**
>
> Попробуйте потратить 5-7 минут на практику использования этих кнопок и сочетаний клавиш.

Первое время можете пользоваться только элементами UI-интерфейса – это нормально, главное, сопоставить кнопки и стоящую за ними функциональность.

## Оставшиеся кнопки на панели

Про кнопку `Run` (и хоткей `Shift+Enter`) мы уже поговорили, а что с остальными?

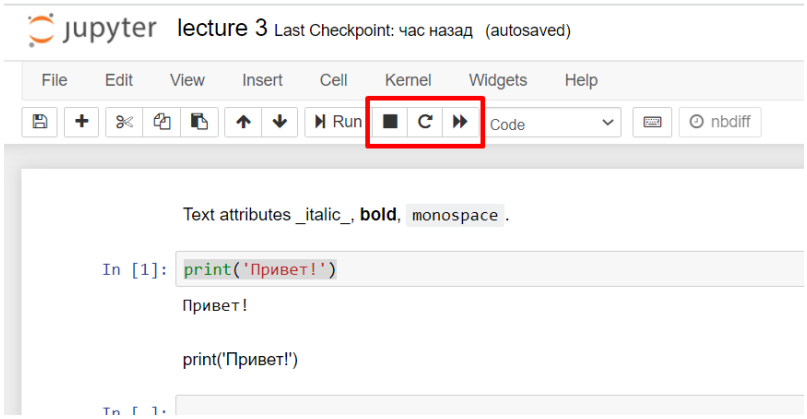



**Fig. 11** Кнопки управления **ЯДРОМ** ноутбуком.

Для того, чтобы вы могли запускать код `Python`, запускается так называемое **“ядро” (или kernel)**, то есть приложение, которое непосредственно выполняет (запускает) ваш код и передает результаты обратно в `Jupyter`-ноутбук. За это отвечает как раз `Run`.

Справа от нее расположен `Stop`, который **прерывает выполнение программы**. Он может быть полезен в случаях, когда вы запустили расчеты на час, но заметили ошибку – и поэтому нужно и код переписать, и ячейку с кодом снова запустить. И в этой ситуации вы сначала **останавливаете** выполнение, редактируете код, затем жмете `Run` – и все готово!

Но случается беда и код не останавливается, потому что ядро `Python` зависает. В таких случаях нужно **перезапустить ядро** – и кнопка с закругленной стрелочкой `Restart` поможет осуществить задуманное. Будьте аккуратны – **вы потеряете ВСЕ несохраненные данные** (значения переменных, результаты расчетов, данные для построения графиков). Сама тетрадка останется без изменений, то есть **написанное сохранится**. Концепция “ядра” и запуска кода станет более понятна, когда мы перейдем к практике.

> **Tip**
>
> Пока стоит держаться правила: “Накосячил? Попробуй остановить (`Stop`) ядро. Не получается? Тогда перезапускай (`Restart`) его!”

Нужно понимать, что ядро “помнит” все предыдущие выполненные ячейки (пока не будет перезагружено или выключено), а значит, вы можете позже в коде переиспользовать те части, которые были описаны ранее (например, переменные или физические константы). Иными словами, состояние ядра сохраняется во времени и между ячейками – **оно относится к документу в целом**, а не к отдельным ячейкам.

Последняя кнопка из выделенного блока имеет говорящее название: `Re-start and run all`. Ядро будет **перезапущено** (все переменные и данные удалятся), а затем **каждая ячейка будет выполнена в порядке сверху вниз**. Поэтому рекомендуется соблюдать структуру, чтобы запускать код с нуля (после возвращения к ноутбуку на следующий день, но с новым ядром, так как компьютер был выключен) – и он отрабатывал.

## Что это за In [*]?

Та самая надпись слева от запущенной Code-ячейки. Это вспомогательная информация о том, что происходит с **кодовой** ячейкой (`In` означает **Input**, то есть ввод кода). Возможно несколько вариантов заполнения.

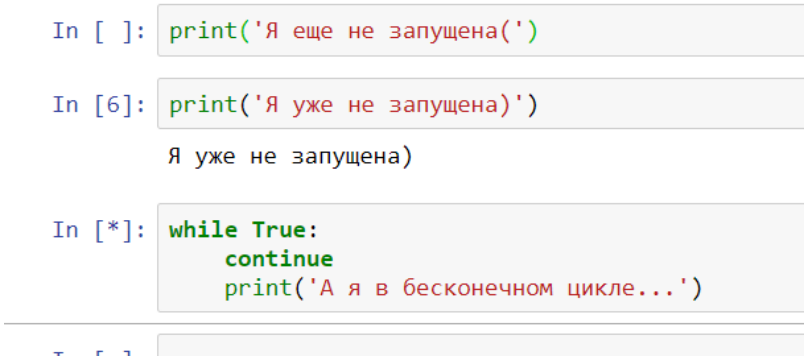



**Fig. 12** Пример трех видов информации о статусе ячейки.

В первом случае в квадратных скобках **ничего нет** – это значит, что **ячейка еще не была запущена**. Возможно, вы забыли, а быть может, она просто ждет своего часа.

Во втором случае **ячейка была запущена** шестой по счету (да-да, **ячейки выполняются по порядку, который задаете вы сами**!) и она **успешно отработала и завершилась**.

В последней строчке умышленно был сделан бесконечный цикл. Это означает, что *код никогда не сможет выполниться* и будет висеть до тех пор, пока вы не остановите (`Stop`ните) ядро. Поэтому там выведен **индикатор выполнения ячейки** – в скобках указана звездочка `*`. Обратите внимание: **это не всегда плохой сигнал**. Если ваш код должен выполняться 2-3 минуты, то все это время будет выводиться `[*]`. Когда код отработает и результат будет получен, отрисуется цифра (например, `[7]` ).

## Самая полезная клавиша

Пришло время программировать! Скопируйте себе в ноутбук кусок кода ниже и попробуйте его запустить. Не переживайте, он может показаться сложным и непонятным, но сейчас не требуется понимание всех деталей.

```
import numpy as np
import mat.pyplot as plt
%matplotlib inline

# Fixing random state for reproducibility
np.random.seed(19680801)

# Compute pie slices
N = 20
θ = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
colors = plt.cm.viridis(radii / 10.)

ax = plt.subplot(111, projection='polar')
ax.bar(θ, radii, width=width, bottom=0.0, color=colors, alpha=0.5)
plt.show();
```

Вы увидите ошибку. По сообщению видно (стрелочка в левой части указывает на проблемное место), что во второй строчке используется слово `mat`, при этом `Python` жалуется на отсутствие такого модуля. Все дело в том, что в коде выше производится попытка рисования графика и для этого используется библиотека `matplotlib`. Но в одной из строк написано только `mat`. Это не дело, давайте исправлять. Однако всех библиотек не запомнишь – и это не нужно. Попробуйте поставить курсор после буквы `t` (и перед точкой) и нажать `TAB`. Вы должны увидеть **список подсказок** и из него выбрать нужный вариант. Этот список не только сокращает время написания кода (за счет автоматического дополнения), но и позволяет избежать ошибок в написании. Обязательно пользуйтесь этим инструментом.

Если вы все сделали правильно, воспользовавшись подсказкой, то после очередного запуска (`Run`) кода появится рисунок.

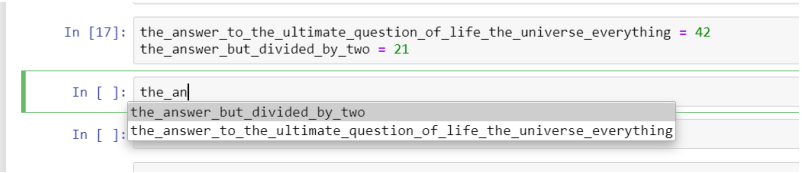



**Fig. 13** Другой пример удачного использования: есть несколько переменных со сложным, но очень похожим названием. Не стоит их перепечатывать – достаточно нажать TAB!

Что ж, большое количество новой информации позади, давайте подведем итоги!

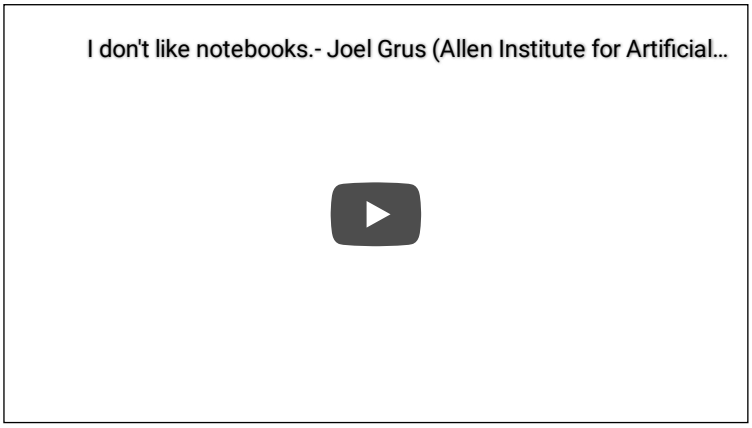
## Что мы узнали из лекции

- Существует два основных типа ячеек, один предназначен для программирования (написания кода), другой – для описания (формул, определений).
- Существует два режима работы: `edit` – редактирование конкретной ячейки и `control` – работа со всей структурой тетрадки/ноутбука.
- Режимы работы можно различить по цвету рамки вокруг активной ячейки, а тип ячейки – по прозрачности фона и наличию надписи `In[ ]` слева.
- Между режимами можно переключаться с помощью `Esc` и `Enter`.
- Чтобы запустить ячейку (с кодом или текстом), нужно нажать на кнопку `Run` сверху, либо воспользоваться сочетаниями клавиш: `Shift+Enter`, `CTRL+Enter`.
- Нужно не забывать сохранять ноутбук (`CTRL+S`), а быстро добавить ячейку кода можно с помощью плюсика слева сверху (или клавиши `B`).
- Клавиши `C` / `V` позволяют копировать и вставлять ячейки в `control`-режиме.
- `In[3]` указывает на порядок выполнения ячеек с кодом, `In[*]` – на процесс выполнения.
- Если вы долго ждете выполнения Code-блока, можно `Stop`нуть ядро, если не помогает – `Restart`нуть.
- Ядро – это процесс, выполняющий код, и после перезагрузки оно не сохраняет переменные.
- `TAB` – ваш друг, позволяющий избегать опечаток, а также реже пользоваться документацией.

## Бонус-материал

Полная официальная документация по Jupyter находится по ссылке тут.

Если вам захочется узнать больше о трюках в ноутбуках, о недостатках и преимуществах по сравнению с альтернативами, предлагаем посмотреть выступление Joel Grus:




# Переменные и вывод информации в Python

## Описание лекции

Из этой лекции вы узнаете о:

- базовых типах данных и переменных;
- простейших операциях с числами и строками;
- функции print;

- выводе информации с подстановкой значений.

## Суть переменных в `Python`

Настало время приступить к изучению непосредственно `Python`, ведь прошло три лекции, а мы об языке программирования и не говорили вовсе! И поскольку наш курс посвящен физике, то начнем со знакомой всем по школьным карандашам формулы $E=mc^2$. По ней можно вычислить полную энергию физического объекта $E$ с помощью известной массы объекта $m$ и константы $c$. Эта постоянная, указывающая на скорость света в вакууме, используется настолько часто, что для нее выделили **отдельное обозначение в виде буквы латинского алфавита**, как и для многих других аналогичных величин. Если в формуле встречается $c$ (в известном контексте), то вы всегда уверены, что именно нужно подставить при расчетах.

Этот пример полностью описывает концепцию **переменных** в языках программирования, и `Python` не исключение. Запись $x = 3$ означает, что везде по тексту далее под иксом подразумевается именно тройка, и ничего другого (пока не будет введено новое определение). Этой же логике подчиняется `Python`. Сначала указывается **имя переменной**, а затем – ассоциируемое с ней значение.

```
c = 299_792_458 # запишем константу, м/с
m = 0.5 # масса некоторого абстрактного объекта, кг
E = m * (c ** 2) # вычисляем энергию, Дж

some_variable_1 = 10.2 # какая-то другая переменная
m = 12
```

Пример кода выше иллюстрирует сразу несколько базовых концепций, которые нужно запомнить:

1. В объявлении переменной нет ничего сложного. Синтаксис и правила интуитивно понятны: это можно делать как в физике/математике, как в учебниках и статьях.
2. `#` означает комментарий, то есть произвольный текст, который не воспринимается `Python` (все **до конца строки** кода полностью игнорируется). Служит исключительно для создания подсказок в коде, объяснения происходящего, то есть для удобства.
3. Числа могут быть **целыми и вещественными**. Разряды в целых числах для удобства визуального восприятия можно разделять нижней чертой.
4. **Значение переменной может быть вычислимым**, то есть являться производной от других переменных (как $E$), ведь это результат перемножения). На самом деле значение вычисляется в момент объявления переменной (при сложной формуле расчета процесс может занимать некоторое время).
5. Операция возведения в квадрат реализуется с помощью `**`.
6. В качестве названия переменных можно использовать **буквы и цифры**, а также некоторые символы. Однако **имя переменной не может начинаться с цифры**.
7. Переменные можно переопределять (и даже менять тип). Однако **старое значение в этом случае будет безвозвратно утрачено**. В данном примере после выполнения последней строчки нельзя установить, чему было равно $m$ до того, как переменной было присвоено значение дюжины.

Если говорить менее строго и более абстрактно, то **переменная – это контейнер** (или коробка), в котором что-то лежит, и на самой коробке на приклеенном листочке бумаги указано содержимое. Чем понятнее надпись, тем легче найти и использовать объект (поэтому переменные с названием из одной буквы воспринимаются плохо, особенно если таких переменных очень много).

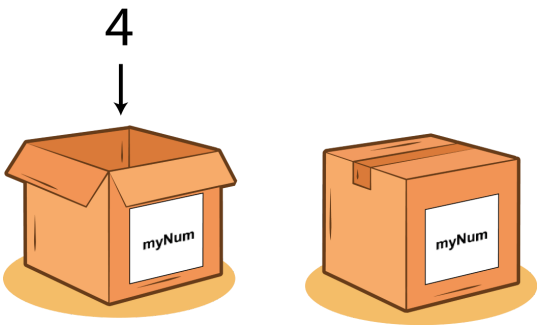



**Fig. 14** Объявить переменную – значит положить объект в коробку с подписью.

## Типы переменных

В листинге кода выше важно заметить, что существует разница между двумя типами численных переменных: **целые и вещественные**. При сугубо математических расчетах и арифметических операциях тип переменной не имеет значения. Однако для некоторого функционала в `Python` нужно быть аккуратным. Мы поговорим подробно об этом в следующих лекциях, а пока стоит запомнить, что вещи, которые необходимо посчитать – в том числе и **длину** чего-то *счетного* – должны быть целочисленными (как и в жизни: первый, второй, третий…).

> **Attention**
>
> Целочисленный тип называется **int** (от `Integer`), вещественный – **float**. Эти типы можно переводить из одного в другой. При переводе вещественного числа в целое теряется часть информации.

**Тип переменной** – и это относится не только к числам, но и к **любому** объекту – можно узнать с помощью функции `type`. Для вывода информации в `Python` используется функция `print`. Что именно представляет собой функция мы рассмотрим в более поздних лекциях, пока стоит думать об этом как о некотором объекте, который зависит (рассчитывается) от других объектов и выдает некоторый результат. Для передачи аргументов используются круглые скобки (аналогично математике: \(y = F(x)\)). Давайте скомбинируем эти знания и рассмотрим пример:

```
first_variable = 10
second_variable = 10.0

# запишем в переменные значения типов данных
type_of_first_variable = type(first_variable)
type_of_second_variable = type(second_variable)

# и распечатаем сами типы, чтобы посмотреть глазами и сравнить
print(type_of_first_variable)
print(type_of_second_variable)

# перезапишем переменные
first_variable = 12.9
second_variable = int(first_variable)
third_variable = float(second_variable)

# в print() можно передавать несколько переменных
print(first_variable, second_variable, third_variable)
```

```
<class 'int'>
<class 'float'>
12.9 12 12.0
```

Внимательно проанализируйте код выше – в нем продемонстрирован базовый синтаксис **преобразования типов** и **вывода информации**. Легко увидеть подтверждение высказанных ранее тезисов: `second_variable` действительно потеряла часть информации (дробную часть числа), которую нельзя вернуть, если преобразовать переменную обратно во `float`. Преобразование типов в языках программирования называется **приведением** (типов, то есть привести одно к другому, а не из-за страшилок про духов).

## Арифметические операции с числами

Математика `Python` максимально близка к естественной: `+-*` и `**` (рассмотренное ранее возведение в степень) работают в точности как ожидается. С делением `/` есть нюанс: **возвращаемое значение всегда вещественное**.

```
a = 3
b = 12.1

c = a + b

# можно объединять вызовы функций print и type
# без создания лишней переменной
print(type(c))

# и даже трёх функций, включая приведение типа
print(type(int(c)))

# деление числа на само себя даёт единицу, но..
print(a / a)
print(b / b)
print(c / c)
print(12 / 4)
```

```
<class 'float'>
<class 'int'>
1.0
1.0
1.0
3.0
```

> **Note**
>
> Обратите внимание, что операции не изменяют переменную саму по себе (то есть операция `a + b` не меняет ни `a`, ни `b`). Чтобы сохранить получаемое значение, нужно присвоить его некоторой переменной (в примере выше это `c`). Если вы хотите изменить непосредственно саму переменную, то можно переприсвоить ей значение на основе расчёта: `a = a + b` или `c = c + 12`.

Даже несмотря на то, что кейс с делением числа на само себя очевиден (всегда получается единица, кроме деления на нуль), будет выведено вещественное значение. Сами же вещественные значения можно складывать, вычитать, умножать и возводить в степень как с целыми, так и с вещественными числами (и наоборот). Если в таком выражении используется хотя бы одна `float`-переменная, то и результат будет не целочисленным. Однако:

```
a = 3
b = 2

print(a + b, type(a + b))
print(a * b, type(a * b))
print(a ** b, type(a ** b))
```

```
5 <class 'int'>
6 <class 'int'>
9 <class 'int'>
```

Это *практически* все тонкости, которые необходимо знать, чтобы не совершать базовые ошибки.

### Примечание

Возможно, у вас родился вопрос относительно расстановки пробелов в коде выше. Обязательно ли соблюдать такой синтаксис? Нужно ли ставить пробелы до и после знаков операций? На самом деле нет: это делается исключительно для удобства чтения кода и **настоятельно рекомендуется не удаляться от стандартов языка**. Код ниже выполнится без ошибок, однако ухудшается читаемость:

```
a=           3
b    =2

print(a +b, type(a+ b))
print(a     * b, type(a *b))
print(a**b, type(a        ** b))
```

```
5 <class 'int'>
6 <class 'int'>
9 <class 'int'>
```

## Строковые переменные

Мы разобрались в том, как описывать и хранить числа, как производить арифметические расчеты. Базовый математический язык освоен, но мы же люди, и хочется общаться словами! Конечно, `Python` позволяет это делать. Благодаря **строковым переменным** можно хранить и соединять текстовую информацию:

```
text_variable = 'тут что-то написано'
another_text_variable = "Вася, впиши сюда что-нибудь перед публикацией курса!"

long_text = '''
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt
in culpa qui officia deserunt mollit anim id est laborum.
'''

print(another_text_variable)
```

```
Вася, впиши сюда что-нибудь перед публикацией курса!
```

В примере выше рассмотрено три способа создания текстовых переменных. Первые два не отличаются между собой с точки зрения `Python`, то есть неважно, используете ли вы одинарные кавычки `'` или двойные `"`. Однако стоит понимать, что если ваша строка содержит в себе такой символ, то кавычка должна быть изменена:

```
error_string = 'Chillin' kid'
another_error_string = "И тут он мне говорит: "у тебя нет ног!""

print(error_string)
print(another_error_string)
```

Механизм ошибки таков, что `Python` неясно: это вы закончили строчку и дальше идет какая-то команда, или же строчка продолжается. В обоих случаях **нужно сменить способ создания строки** – и тогда все будет хорошо:

```
error_string = "Chillin' kid"
another_error_string = 'И тут он мне говорит: "у тебя нет ног!"'

print(error_string)
print(another_error_string)
```

```
Chillin' kid
И тут он мне говорит: "у тебя нет ног!"
```

Если необходимо сохранить какой-либо объемный текст или сообщение, можно воспользоваться мультистрочным объявлением переменной, как в первом примере блока.

Строки можно объединять для удобства вывода информации:

```
first_string = 'Результат вычислений: '
second_string = ". Это не так много!"

a = 12
b = 2
result = a * b

# два способа вывода:
print(first_string, result, second_string)

# либо через склейку строк вручную
# обратите внимание на приведение типа int к str
result_string = first_string + str(result) + second_string
print(result_string)
```

```
Результат вычислений:  24 . Это не так много!
Результат вычислений: 24. Это не так много!
```

**⚠ Warning**

Будьте аккуратны со сложением строк. Объединение строк `"3"` и `"5"` даст результат `"35"`, а не `8` – и тип результирующего значения **будет строковый**. **Сложить строку и число нельзя**: вы получите ошибку и никакого приведения типов не произойдет. Здесь возникнет двусмысленность – нужно привести число к строке и затем сконкатенировать или же строку к числу (а вдруг это невозможно?), после чего сложить.

Склеивание строк называется **конкатенацией**. Попробуйте в `Jupyter`-ноутбуке объединить строковые, целочисленные и вещественные переменные в разных комбинациях. Разберитесь, что означает ошибка, которая будет выведена в случае, если не делать приведение типов (то есть без `str` в `str(result)`).

> **Note**
>
> Обратите внимание на пробел между числом и точкой в первом случае. Они добавлены автоматически функцией `print` – это сделано для того, чтобы разные объекты при последовательном выводе не “склеивались” друг с другом. Во втором случае этого не происходит, так как мы напрямую склеиваем строки и только затем передаем результат конкатенации на печать в `print`.

Но на практике это не совсем удобно, поэтому в `Python` придумали `F`-строки. Их суть в том, что переменная из кода напрямую подставляется (с автоматическим приведением типа к строке) в саму строку! Вот:

```
a = 12
b = 2
result = a * b

result_string = f'Результат вычислений: {result}. Это не так много!'

# и без f
wrong_result_string = 'Результат вычислений: {result}. Это не так много!'

print(result_string)
print(wrong_result_string)
```

```
Результат вычислений: 24. Это не так много!
Результат вычислений: {result}. Это не так много!
```

Для объявления `F`-строки нужно, во-первых, использовать одинаковые кавычки на концах текста. Во-вторых, нужно указать литеру `f` перед самой строкой. И последнее – нужно обрамить название конкретной переменной (`result` в данном случае) в фигурные скобки.

Когда переменная одна, а также нет текста после ее использования, то выгода `F`-строк не так очевидна (относительно простого `print(some_string, some_variable)`). Однако представьте, что вам нужно вывести координаты точки в трехмерном пространстве, значение времени, параметры системы и значение некоторой функции от всех переменных выше!

```
# так тоже можно!
x, y, z = 12.1, 0, 13
# скобки, как и в математике, задают порядок выполнения вычислений
func_val = (x * y) ** z

current_time = 30.113412

# а вот так можно писать длинные f-строки (но работает и для обычных)
out_string = (f'В точке с координатами X={x}, Y={y}, Z={z} значение функции '
              f'равно {func_val}. Состояние системы указано на момент '
              ' времени t=' + str(current_time))

print(out_string)
```

```
В точке с координатами X=12.1, Y=0, Z=13 значение функции равно 0.0. Состояние
системы указано на момент  времени t=30.113412
```

## Что мы узнали из лекции

- **Переменные – это “контейнеры”**, в которые можно что-то положить и дать название.
- Математика в `Python` не имеет сложных правил, процесс вычислений максимально интуитивен.
- Арифметические операции могут **менять тип** результирующей переменной.
- `type()`, `print()` – базовые функции, с помощью которых можно делать **самопроверки** по ходу написания кода.
- **Сменить тип** переменной можно вызовом функций `int()`, `float()`, `str()`.
- Строки могут обрамляться как `'`, так и `"` (но этих символов **не должно быть внутри** текста).
- `F`-строки облегчают комплексный вывод, содержащий как текст, так и переменные `Python` (и автоматически приводит типы).

# Условные конструкции, булева логика и сравнения

## Описание лекции

В этой лекции мы расскажем про:

- `if`/`else`-конструкции и условия;
- тип `bool`;
- операторы сравнения;
- блоки кода и отступы.

## Ветвление логики

В прошлых лекциях мы рассмотрели программы с линейной структурой: сначала выполнялась первая конструкция (например, объявление переменной), затем вторая (преобразование переменной или расчет по формуле), после – третья (`print` для вывода результатов). Можно сказать, что происходило последовательное исполнение команд, причем каждая инструкция выполнялась **обязательно**. Но что делать, если хочется опираться на обстоятельства и принимать решения о том, выполнять одну часть кода или другую?

Допустим, по числу `x` нужно определить его абсолютную величину, то есть модуль. Программа должна напечатать значение переменной `x`, если `x > 0` или же величину `-x` в противном случае (`-(-5) = 5`). Эту логику можно записать следующим образом:

```
x = -3  # попробуйте поменять значение переменной

if x > 0:
    print("Исходный x больше нуля")
    print(x)
else:
    print("Исходный x меньше или равен нулю")
    print(-x)
```

```
Исходный x меньше или равен нулю
3
```

В этой программе используется условная инструкция `if` (в переводе с английского *“если”*). `if` – это ключевое зарезервированное слово (так *нельзя назвать свою переменную*), указывающее на **условную конструкцию**. После `if` следует указать вычислимое выражение, которое **можно проверить на истинность** (то есть можно сказать, правда это или нет). Общий вид конструкции следующий:

```
if (Условие):
    <Блок инструкций 1>
else:
    <Блок инструкций 2>
```

`else` – тоже ключевое слово (в переводе – *“иначе”*). Таким образом, можно в голове придерживаться такой интерпретации: “**если** условие верно (истинно), **то** выполни первый блок команд, **иначе** выполни второй блок”.

Условная инструкция содержит как минимум ключевое слово `if` (единожды), затем может идти любое количество (включая ноль) блоков с условием `else if <условие>` (*иначе если*, то есть будет выполнена проверка нового условия в случае, если первая проверка в `if` не прошла), затем – опционально – конструкция `else`. Логика чтения и выполнения кода сохраняет порядок **сверху вниз**. Как только одно из условий будет выполнено, выполнится соответствующая инструкция (или набор инструкций), а все последующие блоки будут проигнорированы. Это проиллюстрировано в коде:

```
x = -3.8  # попробуйте поменять значение переменной

if x > 0:
    print('x больше нуля')
elif x < 0:  # можно написать "else if x < 0:"
    print('x меньше нуля')
else:
    print('x в точности равен нулю')
print('Такие дела!')
```

```
x меньше нуля
Такие дела!
```

Понятно, что `x` не может одновременно быть и больше нуля, и меньше (или равен ему). Среди всех трех `print`-блоков будет выполнен **только один**. Если `x` действительно больше нуля, то второе условие (`x < 0`) даже не будет проверяться – `Python` сразу же перейдет к последней строке и выведет надпись “Такие дела!”.

Чтобы лучше разобраться в том, как работает код, можно использовать **визуализаторы** – например, такой. Прогоняйте через него весь код (даже в несколько строк) и сверяйте со своими ожиданиями от его работы.

## А что вообще такое эти ваши условия?

Выше было указано, что после конструкций `if`/`else if` необходимо указать **условие**, которое еще и должно быть истинным или ложным (“правда или нет”). Давайте попробуем определить необходимый **тип** переменной.

```
x = -3.8

condition_1 = x > 0
condition_2 = x < 0

print(condition_1, type(condition_1))
print(condition_2, type(condition_2))
```

```
False <class 'bool'>
True <class 'bool'>
```

Видно, что оба условия имеют один и тот же тип - `bool`, то есть `boolean`. По определению:

> Boolean (Булев, Логический тип данных) – примитивный тип данных в информатике, которые могут принимать **два возможных значения**, иногда называемых истиной (True) и ложью (False).

Оказывается, что в коде выше мы получили **ВСЕ** возможные варианты булевой переменной – это истина (`True`, пишется только с заглавной буквы) и ложь (`False`, аналогично). Никаких других значений быть для условия не может. Вот такой это простой тип данных.

## Способы получения `bool`

Какими вообще могут быть условия? Как с ними можно обращаться? Согласно официальной документации, в `Python` есть такие операторы сравнения:

| Операция | Значение |
|---|---|
| `<` | строго меньше чем |
| `<=` | меньше или равно |
| `>` | строго больше, чем |
| `>=` | больше или равно |
| `==` | равный |
| `!=` | не равный |

**Fig. 15** Все операции сравнения работают нативно (так же, как и в математике)

```
print(3.0 > 3)
print(3.0 == 3)
```

```
False
True
```

Здесь практически нечего рассматривать, операторы сравнения они и в `Python` операторы. Куда интереснее принцип **объединения различных условий в одно** – для создания комплексной логики.

Пусть стоит задача определения четверти точки по ее координатам на двумерной плоскости. Решение такой задачи может быть записано следующим образом:

```
x = -3.6
y = 2.5432

if x > 0:
    if y > 0:
        # x > 0, y > 0
        print("Первая четверть")
    else:
        # x > 0, y < 0
        print("Четвертая четверть")
else:
    if y > 0:
        # x < 0, y > 0
        print("Вторая четверть")
    else:
        # x < 0, y < 0
        print("Третья четверть")
```

```
Вторая четверть
```

Пример показывает, что выполняемым блоком кода может быть любой блок `Python`, включая новый логический блок с `if-else` конструкцией. Однако его можно сократить с помощью **логических операторов** `and`, `or` и `not`. Это стандартные логические операторы Булевой алгебры.

Логическое **И** является бинарным оператором (то есть оператором с двумя операндами: левым и правым) и имеет вид `and`. Оператор `and` возвращает `True` тогда и только тогда, когда **оба его операнда имеют значение** `True`.

Логическое **ИЛИ** является бинарным оператором и возвращает `True` тогда и только тогда, когда **хотя бы один операнд равен** `True`. Оператор “логическое ИЛИ” имеет вид `or`.

Логическое **НЕ** (отрицание) является унарным (то есть **с одним операндом**) оператором и имеет вид `not`, за которым следует единственный операнд. Логическое НЕ возвращает `True`, **если операнд равен** `False` и **наоборот**.

Эти правила необходимо запомнить для успешного создания сложных условий с целью разделения логики, заложенной в `Python`-коде.

Проиллюстрируем правила в коде на простых примерах. Обратите внимание на то, как можно объявлять `bool`-переменные – это не сложнее, чем создание целочисленного значения:

```
true_value = True
false_value = False

# False потому, что один из операндов является False
some_value = true_value and false_value
print(some_value)

# True потому, что хотя бы один из операндов равен True
some_value = true_value or false_value
print(some_value)

# отрицание True (истины) есть False (ложь)
some_value = not true_value
print(some_value == false_value)

# пример сложного условия - порядок лучше в явном виде задавать скобками
hard_condition = (not true_value or false_value) or (true_value != false_value)
print(hard_condition)
```

```
False
True
True
True
```

Теперь попробуем их применить на приближенных к практике примерах:

```
x = -3.6
y = 2.5432

if x > 0 and y > 0:  # конструкция заменяет два вложенных if
    print("Первая четверть")
elif x > 0 and y < 0:
    print("Четвертая четверть")
elif y > 0:
    print("Вторая четверть")
else:
    print("Третья четверть")

# определим, большое ли число x (в терминах модуля)
x_is_small = (x < 3) and (x > -3)
# число большое, если оно не маленькое (по модулю)
x_is_large = not x_is_small  # можно отрицать факт малости x

print('Is x small? ', x_is_small)
print('Is x large? ', x_is_large)

# так тоже можно писать - на манер неравенств в математике
another_x_is_small = -3 < x < 3
print(another_x_is_small)
print(another_x_is_small == x_is_small)
```

```
Вторая четверть
Is x small?  False
Is x large?  True
False
True
```

Так как вторая переменная `x_is_large` – это отрицание (`not`) первой (`x_is_small`), то они **никогда** не будут равны.

## Блоки кода и отступы

В примерах выше вы наверняка заметили упоминание термина “блок кода”, а также откуда-то взявшиеся отступы после условий, и это не случайно. Во-первых, давайте признаем, что так условные конструкции (особенно вложенные!) читать куда легче, и глаза не разбегаются. Во-вторых, это особенность языка `Python` – здесь не используются скобки `{}` для указания блоков, все форматирование происходит с помощью отступов. Отступы **всегда** добавляются в строки кода **после двоеточия**.

Для выделения блока инструкций (строк кода, выполняющихся подряд при любых условиях), относящихся к инструкциям `if`, `else` или другим, изучаемым далее, в языке `Python` используются **отступы**. Все инструкции, которые относятся к одному блоку, должны иметь **равную величину отступа**, то есть одинаковое число **пробелов в начале строки**. В качестве отступа PEP 8 рекомендует использовать **отступ в четыре пробела** и не рекомендует использовать символ табуляции. Если нужно сделать еще одно вложение блока инструкций, достаточно добавить еще четыре пробела (см. пример выше с поиском четверти на плоскости).

> **Tip**
>
> Хоть и не рекомендуется использовать символ табуляции для создания отступов, кнопка `Tab` на вашей клавиатуре в `Jupyter`-ноутбуке (при курсоре, указывающим на начало строки кода) создаст отступ в четыре пробела. Пользуйтесь этим, чтобы не перегружать клавишу пробела лишними постукиваниями :).

## Что мы узнали из лекции

- Для задания логики выполнения кода и создания нелинейности используются **условные инструкции**, поскольку они следуют некоторым условиям.
- Условная инструкция задается ключевым словом `if`, после которого может следовать несколько (от нуля) блоков `else if`/`elif`, и – опционально – в конце добавляется `else`, если ни один из блоков выше не сработал.
- Условия должны быть **булевого типа** (`bool`) и могут принимать **всего два значения** – `True` и `False`. Выполнится тот блок кода, который задан истинным (`True`) условием (и только первый!).
- Условные конструкции можно вкладывать друг в друга, а также объединять с помощью **логических операторов** `and`, `or` и `not`.
- **Блок кода** – это несколько подряд идущих команд, которые будут выполнены последовательно.
- Чтобы выделить блок кода после условия, используйте **отступы** – четыре пробела.
- Чтобы создать отступ в `Jupter`, нужно нажать `Tab` в начале строки кода.

# Списки и циклы в Python

## Описание лекции

На этом занятии мы разберем следующие темы:

- списки (`list`) и их методы;
- индексация списков;
- что такое срезы и зачем они нужны;
- цикл `for` и функция `range`;
- итерация по спискам, list comprehensions.

## Введение в списки объектов

В предыдущих лекциях мы оперировали малым количеством переменных. Для каждого блока логики или примера кода вводилось 3-5 объектов, над которыми осуществлялись некоторые операции. Но что делать, если объектов куда больше? Скажем, вам необходимо хранить информацию об учащихся класса – пусть это будет рост, оценка по математике или что-либо другое. Не знаю, как вы, но я нахожу крайне неудобным создание 30 отдельных переменных. А если еще и нужно посчитать среднюю оценку в классе!

```
average_grade = petrov_math + kosareva_math + zinchenko_math + kotenkov_math + ...
average_grade = average_grade / 30
```

Такой код к тому же получается крайне негибким: если количество студентов, как и их состав, изменится, то нужно и формулу переписать, так еще и делитель – в нашем случае 30 – изменять.

Часто в программах – даже в (квантовом) машинном обучении – приходится работать с большим количеством **однотипных** переменных. Специально для этого придуманы **массивы** (по-английски array). В Python их еще называют **списками** (list). В некоторых языках программирования эти понятия отличаются, но не в Python. Список может хранить переменные **разного** типа. Также списки называют “(контейнерами)[https://ru.wikipedia.org/wiki/Контейнер_(программирование)]”, так как они хранят какой-то набор данных. Для создания простого списка необходимо указать квадратные скобки или вызвать конструктор типа (`list` – это отдельный тип, фактически такой же, как int или str), а затем перечислить **объекты через запятую**:

```
# разные способы объявления списков
first_list = []
second_list = list()
third_list = list([1,2, "stroka", 3.14])
fourth_lust = [15, 2.2, ["another_list", False]]

print(type(second_list), type(fourth_lust))
print(first_list, fourth_lust)
```

```
<class 'list'> <class 'list'>
[] [15, 2.2, ['another_list', False]]
```

> **Tip**
>
> Хоть список и хранит переменные разного типа, но так делать без особой необходимости не рекомендуется – вы сами скорее запутаетесь и ошибетесь в обработке объектов списка. В большинстве других языков прогрммирования массив может хранить только объекты одного типа.
>
> Для хранения сложных структур (скажем, описание студента – это не только оценка по математике, но и фамилия, имя, адрес, рост и так далее) лучше использовать классы – с ними мы познакомимся в будущем. А еще могут пригодиться **кортежи**, или `tuple`. Про них в лекции не рассказано, самостоятельно можно ознакомиться по ссылке.

Теперь можно один раз создать список и работать с ним как с единым целым. Да, по прежнему для заведения оценок студентов придется разово их зафиксировать, но потом куда проще исправлять и добавлять! Рассмотрим пример нахождения средней оценки группы, в которой всего 3 учащихся, но к ним присоединили еще 2, а затем – целых 5:

```
# базовый журнал с тремя оценками
math_journal = [3, 3, 5]

# добавим новопришедших студентов
math_journal.append(4)
math_journal.append(5)

# и сразу большую группу новых студентов
math_journal.extend([2,3,4,5,5])

print(f"{math_journal = }")

# найдем среднюю оценку как сумму всех оценок, деленную на их количество
avg_grade = sum(math_journal) / len(math_journal)
print(f"{avg_grade = }")
```

```
math_journal = [3, 3, 5, 4, 5, 2, 3, 4, 5, 5]
avg_grade = 3.9
```

В коде выше продемонстрировано сразу несколько важных аспектов:

1. Добавлять по одному объекту в конец списка можно с помощью метода списка `append`;
2. Метод `append` принимает в качестве аргумента один Python-объект;
3. Слияние списков (конкатенация, прямо как при работе со строками) нескольких осуществляется командой `extend` (расширить в переводе с английского);
4. Для списков определена функция `len`, которая возвращает целое число `int` – количество объектов в списке;
5. Функция `sum` может применяться к спискам для суммирования всех объектов (если позволяет тип – то есть для `float`, `int` и `bool`. Попробуйте разобраться самостоятельно, как функция работает с последним указанным типом);
6. Для методов `append` и `extend` не нужно приравнивать результат выполнения какой-то переменной – изменится сам объект, у которого был вызван метод (в данном случае это `math_journal`);
7. Списки в Python **упорядочены**, то есть объекты сами по себе места не меняют, и помнят, в каком порядке были добавлены в массив.

> **Tip**
>
> В тексте выше встречается термин **метод**, который, быть может, вам не знаком. По сути метод – это такая же **функция**, о которых мы говорили раньше, но она принадлежит какому-то объекту с определенным типом. Не переживайте, если что-то непонятно – про функции и методы мы поговорим подробно в ближайших лекциях!
>
> `print`, `sum` – функции, они существуют сами по себе; `append`, `extend` – методы объектов класса `list`, не могут использоваться без них.

## Индексация списков

Теперь, когда стало понятно, с чем предстоит иметь дело, попробуем усложнить пример. Как узнать, какая оценка у третьего студента? Все просто – нужно воспользоваться **индексацией** списка:

```
# базовый журнал с пятью оценками
math_journal = [1, 2, 3, 4, 5]

third_student_grade = math_journal[3]
print(third_student_grade)
```

```
4
```

И снова непонятный пример! Давайте разбираться:

1. Для обращения к `i`-тому объекту нужно в квадратных скобках указать его индекс;
2. **Индекс** в Python начинается **С НУЛЯ** – это самое важное и неочевидное, здесь чаще всего случаются ошибки;
3. Поэтому `[3]` обозначает взятие **четвертой** оценки (и потому выводится четверка, а не тройка);
4. Всего оценок 5, но так как индексация начинается с нуля, то строчка `math_journal[5]` выведет ошибку – нам доступны лишь индексы `[0, 1, 2, 3, 4]` для взятия (так называется процедура обращения к элементу списка по индексу – взятие по индексу).

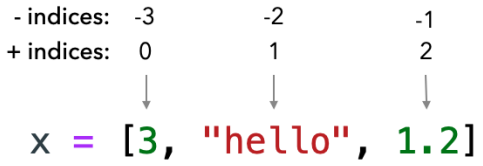



**Fig. 16** Пример списка из трех объектов. Сверху показаны их индексы, включая отрицательные

Также в Python существуют отрицательные индексы (-1, -2 …). Они отсчитывают объекты списка, начиная с конца. Так как нуль уже занят (под первый объект), то он не используется.

```
# базовый журнал с пятью оценками
math_journal = [1, 2, 3, 4, 5]

# возьмем последнюю оценку
last_grade = math_journal[-1]
print(f"Последняя оценка: {last_grade}")

# а теперь -- предпоследнюю
prev = math_journal[-2]
print(f"Предпоследняя оценка: {prev}")

# конечно, взятие по индексам можно использовать в ранее разобранном синтаксисе

if math_journal[-1] < math_journal[-2]:
    math_journal[-1] += 1
    print("Последняя оценка меньше предпоследней. Натянем студенту?")
else:
    math_journal[-2] = 2
    print("Последний студент сдал очень хорошо, на его фоне предпоследний просто
двоечник!")
```

```
Последняя оценка: 5
Предпоследняя оценка: 4
Последний студент сдал очень хорошо, на его фоне предпоследний просто двоечник!
```

Все это важно не только для грамотного оперирования конкретными объектами, но и следующей темы -

## Срезы

Срезы, или slices – это механизм обращения сразу к нескольким объектам списка. Для создания среза нужно в квадратных скобках указать двоеточие, слева от него – индекс начала среза (по умолчанию 0, можно не выставлять) **включительно**, справа – границу среза **не включительно** (пустота означает “до конца списка”). Может показаться нелогичной такая разнородность указания границ, но на самом деле она безумно удобна – особенно вместе с тем, что индексация начинается с нуля. Быстрее объяснить на примере:

```
# базовый журнал с пятью оценками
math_journal = [1, 2, 3, 4, 5]

# как взять первые 3 оценки?
first_3_grades = math_journal[:3]
print(f"{first_3_grades = }")

# как взять последние две оценки?
last_2_grades = math_journal[-2:]
print(f"{last_2_grades = }")

# сделаем срез на 4 оценки, начиная со второй (с индексом 1)
start_index = 1
some_slice = math_journal[start_index : start_index + 4]
print(f"{some_slice = }")

# возьмем столько объектов из начала, сколько объектов в some_slice
yet_another_slice = math_journal[:len(some_slice)]

# а вот так можно проверить, попадает ли объект в список
print("Верно ли, что единица входит в some_slice? {1 in some_slice}")
print("Верно ли, что единица входит в yet_another_slice? {1 in yet_another_slice}")
```

```
first_3_grades = [1, 2, 3]
last_2_grades = [4, 5]
some_slice = [2, 3, 4, 5]
Верно ли, что единица входит в some_slice? {1 in some_slice}
Верно ли, что единица входит в yet_another_slice? {1 in yet_another_slice}
```

> **Tip**
>
> Можно сделать пустой срез, и тогда Python вернет пустой список без объектов. Можете проверить сами: `["1", "2", "3"][10:20]`

Давайте проговорим основные моменты, которые **крайне важно понять**:

1. Так как индексация начинается с нуля (значение по умолчанию) и правая граница не включается в срез, то берутся объекты с индексами `[0,1,2]`, что в точности равняется трем первым объектам;
2. Срез `[-2:]` указывает на то, что нужно взять все объекты до конца, начиная с предпоследнего
3. Значения в срезе могут быть **вычислимы** (и задаваться сколь угодно сложной формулой), но должны оставаться **целочисленными**;
4. Если нужно взять `k` объектов, начиная с `i`-го индекса, то достаточно в качестве конца среза указать `k+i`;
5. Для проверки вхождения какого-либо объекта в список нужно использовать конструкцию `x_obj in some_list`, которая вернет `True`, если массив содержит `x_obj`, и `False` в ином случае;
6. Самый простой способ сделать копию списка - это сделать срез по всему объекту: `my_list[:]`. Однако будьте внимательны – в одних случаях копирование происходит полностью (по значению), а в некоторых сохраняются ссылки (то есть изменив один объект в скопированном списке вы измените объект в исходном). Связано это с типом объектов (mutable/immutable), подробнее об этом будет рассказано в следующей лекции. В общем, если вы работаете с простыми типами (`int`/`str`), то срез вернет копию, и её изменение не затронит исходный список. Однако для хранения новых данных нужна память, поэтому при копировании десятков миллионов объектов можно получить ошибку, связанную с нехваткой памяти.

## Циклы

До сих пор в примерах мы хоть и обращались к нескольким объектам, добавляли и меняли их, все еще не было рассмотрено взаимодействие сразу с несколькими. Давайте попробуем посчитать, сколько студентов получили оценку от 4 и выше. Для этого интуитивно кажется, что нужно **пройтись по всем оценкам от первой до последней**, сравнить каждую с четверкой. Для прохода по списку, или **итерации**, используются **циклы**. Общий синтаксис таков:

```
example_list = list(...)
for item in example_list:
    <> блок кода внутри цикла (аналогично блоку в if)
    ... что-то сделать с item
    <>
```

Здесь `example_list` – это некоторый итерируемый объект. Помимо списка в Python существуют и другие итерируемые объекты, но пока будем говорить о массивах.

Этот цикл работает так: указанной **переменной `item` присваивается первое значение из списка**, и выполняется **блок кода** внутри цикла (этот блок, напомним, определяется отступом. Он выполняется весь от начала отступа и до конца, как и было объяснено в пятой лекции). Этот блок еще иногда называют **телом цикла**. Потом переменной `item` присваивается следующее значение (второе), и так далее. Переменную, кстати, можно называть как угодно, не обязательно `item`.

**Итерацией** называется каждый **отдельный проход** по телу цикла. Цикл всегда повторяет команды из тела цикла несколько раз. Два примера кода ниже аналогичны:

```
math_journal = [3, 4, 5]
counter = 0

for cur_grade in math_journal:
    if cur_grade >= 4:
        counter += 1

print(f"Всего хорошистов и отличников по математике {counter} человека")
```

```
Всего хорошистов и отличников по математике 2 человека
```

```
math_journal = [3, 4, 5]
counter = 0

cur_grade = math_journal[0]
if cur_grade >= 4:
    counter += 1

# не забываем менять индекс с 0 на 1, так как каждый раз берется следующий элемент
cur_grade = math_journal[1]
if cur_grade >= 4:
    counter += 1

# и с единицы на двойку
cur_grade = math_journal[2]
if cur_grade >= 4:
    counter += 1

print(f"Всего хорошистов и отличников по математике {counter} человека")
```

```
Всего хорошистов и отличников по математике 2 человека
```

Понятно, что первый кусок кода обобщается на любой случай – хоть оценок десять, хоть тысяча. Второе решение не масштабируется, появляется **много одинакового кода, в котором легко ошибиться** (не поменять индекс, к примеру).

Движемся дальше. Так как каждый элемент списка закреплен за конкретным индексом, то в практике часто возникают задачи, логика которых завязана на индексах. Это привело к тому, что появилась альтернатива для итерации по списку. Функция `range` принимает аргументы, аналогичные срезу в списке, и возвращает итерируемый объект, в котором содержатся целые числа (индексы). Так как аргументы являются аргументами функции, а не среза, то они соединяются запятой (как `print(a, b)` нескольких объектов). Если подан всего один аргумент, то нижняя граница приравнивается к нулю. Посмотрим на практике, как сохранить номера (индексы) всех хорошо учащихся студентов:

```
math_journal = [4, 3, 4, 5, 5, 2, 3, 4]
good_student_indexes = []

for student_index in range(len(math_journal)):
    curent_student_grade = math_journal[student_index]
    if curent_student_grade >= 4:
        good_student_indexes.append(student_index)

print(f"Преуспевающие по математике студенты находятся на позициях:
{good_student_indexes}")
```

```
Преуспевающие по математике студенты находятся на позициях: [0, 2, 3, 4, 7]
```

В примере `student_index` принимает последовательно все значения от `0` до `7` включительно. `len(math_journal)` равняется `8`, а значит, восьмерка сама не будет включена в набор индексов для перебора. На каждой итерации `curent_student_grade` меняет свое значение, после чего происходит проверка. Если бы была необходимость пробежаться только по студентам, начиная с третьего, то нужно было бы указать `range(2, len(math_journal))` (двойка вместо тройки потому, что индексация с нуля, ведь мы перебираем индексы массива).

Выше описаны основные концепции обращения со списками. Их крайне важно понять и хорошо усвоить, без этого писать любой код будет безумно сложно. Скопируйте примеры к себе в `.ipynb`-ноутбук, поиграйтесь, поменяйте параметры цикла и проанализируйте изменения.

## List comprehensions

Некоторые циклы настолько просты, что занимают 2 или 3 строчки. Как пример – привести список чисел к списку строк:

```
# грубый вариант
inp_list = [1,4,6,8]
out_list = []

for item in inp_list:
    out_list.append(str(item))

# list comprehension
out_list = [str(item) for item in inp_list]
print(out_list)
```

```
['1', '4', '6', '8']
```

Две части кода идентичны за вычетом того, что нижняя – с непонятной конструкцией в скобках – короче. Python позволяет в рамках одной строки произвести какие-либо простые преобразования (помним, что `str()` – это вызов функции, а значит если у вас есть сложная функция, которая делает квантовые вычисления, то ее также можно применить!). Фактически самый частый пример использования – это паттерн "**применение функции к каждому объекту списка**".

## Что мы узнали из лекции

- `list` – это **объект-контейнер, который хранит другие объекты разных типов**. Запись происходит упорядочено и последовательно, а каждому объекту присвоен **целочисленный номер, начиная с нуля**;
- для добавления одного объекта в `list` нужно использовать метод объекта `list` – `append`, а для расширения списка сразу на несколько позиций пригодится `extend`;
- проверить, сходит ли объект в список, можно с помощью конструкции `obj in some_list`;
- индексы **могут быть отрицательными**: `-1`, `-2` … В таком случае нумерация начинается от последнего объекта;
- можно получить часть списка, сделав **срез** с помощью конструкции `list[start_index : end_index]`, при этом объект на позиции `end_idnex` не будет включен в возвращаемый список (т.е. **срез работает не включительно по правую границу**);
- часто со списками используют **циклы, которые позволяют итерироваться по объектам массива** и выполнять произвольную логику в рамках отделенного отступом блока кода;
- для итерации по индексам можно использовать `range()`;
- простые циклы можно свернуть в **list comprehension**, и самый частый паттерн для такого преобразования – это **применение некоторой функции к каждому объекту** списка (если `x` это функция, то синтаксис будет таков: `[x(item) for item in list])`).

---