GeekBrains

Основы Python

Урок 1. Знакомство с Python

Оглавление

Как будет проходить обучение

Уроки

Самое важное условие успешного прохождения курса — присутствие на вебинарах в **реальном** времени. Поэтому постарайтесь скорректировать свой график. Возможно, некоторые занятия придётся пересмотреть в записи, и не раз. Часто информация обрабатывается в подсознании, и при втором или третьем просмотре вы начинаете замечать много новых нюансов.

Настройка среды

Для написания кода на курсе и выполнения домашних заданий необходимо установить интерпретатор Python версии 3.6 или новее (3.7, 3.8 и т.д.). Это задача тривиальная, но если будут затруднения — смотрите видеоматериалы, приложенные к уроку. В некоторых ОС, например в ОС Ubuntu, интерпретатор уже установлен. Для старта подходит любая операционная система.

Среда разработки

Python-программы можно писать в **любом** текстовом редакторе, главное — сохранять в кодировке `UTF-8`. На курсе будем использовать среду разработчика PyCharm. Для начинающих разработчиков большим плюсом будут возможности по подсветке синтаксиса и контекстные подсказки. Для ускорения старта рекомендуется ставить на SSD-диск. При работе может занимать больше 1 Гб в оперативной памяти.

Домашние задания

Нужно сразу понять, что ДЗ нужны **только** вам. Если вы их не делаете или делаете поверхностно, вы просто отдаляете дату вашего первого оффера (предложения работодателя). Вам неизбежно придётся проходить собеседования, и только колоссальный объем работы по написанию кода поможет достойно отвечать на вопросы. Плохая практика откладывать выполнение ДЗ на следующий урок. Как правило, это приводит к серьезному отставанию в понимании материала. Это как на работе сказать, что вы сегодня не будете работать. Да, бывают обстоятельства, и разово можно не успеть, но в ближайшие выходные необходимо всё наверстать.

Важно правильно относиться к оценке преподавателя. Если вам поставили 3 или «не сдано» — это сигнал, что материал не усвоен и нужно приложить двойные усилия. Если вместо этого вы будете обижаться — это тупик. Задача преподавателя — показать реальный прогресс, а не его иллюзию. Представьте, что это ваш работодатель. Его оценка — это метрика, насколько вы интересны как потенциальный сотрудник.

Текст домашнего задания нужно воспринимать как тестовое задание работодателя. Любое отклонение от него недопустимо. Очень важно внимательно несколько раз перечитать формулировку задачи перед её решением. На собеседованиях очень часто проверяют навык точно понять задание.

Если вам что-то непонятно, задайте вопрос преподавателю, наставнику, одногруппникам или в чате Телеграм. Но перед этим лучше ещё раз подумать — формулировки сделаны однозначно.

Если не сказано иного, при решении задач нельзя использовать внешние библиотеки и модули. На данном этапе обучения главная задача — реализовать алгоритм своими силами, а не путем импорта готового решения. Например, если на собеседовании вас попросят вычислить факториал числа и вы попытаетесь использовать функцию `math.factorial()`, — это будет ошибкой. Вместо этого работодатель попросит написать эту функцию «голыми руками». Именно этого мы хотим от вас при выполнении ДЗ.

Старайтесь при выполнении ДЗ думать на шаг вперёд: из-за чего может сломаться код? Какие данные могут быть недопустимы? Что будет, если масштаб задачи увеличится на порядок или на несколько порядков? Помните, что если в ДЗ был список, состоящий из 10 элементов, то в реальных условиях размер списка может быть десять тысяч, сто тысяч элементов и больше. Это должно предостеречь вас от некоторых заведомо неэффективных способов решения задачи.

Рекомендуем работать по принципу «Make It Work Make It Right Make It Fast». То есть сначала надо добиться работоспособности кода, и только потом что-то улучшать.

Всегда проверяйте код перед отправкой ДЗ. Прислать нерабочий код — очень грубая ошибка.

Рекомендуем создавать к каждому ДЗ файл `readme.txt`, с кратким изложением итогов выполнения ДЗ по пунктам (выполнено, сложно, легко). Если что-то не получилось — напишите, что сделали, сколько времени ушло на попытки решить задачу. Это хорошая обратная связь для преподавателя и повод что-то дополнительно рассказать в начале урока. А вам это поможет следить за собственным прогрессом и оценивать пробелы в знаниях.

Именуйте файлы в ДЗ по схеме `task_<dz_number>_<task_number>.py`, например, `task_5_1.py`. Сами файлы должны быть в папке, имя которой <Last_name>_<First_name>_dz_<dz_number>, например, `Ivanov_Ivan_dz_1`. Это часть технического задания, а не наш каприз. Здесь мы учимся быть аккуратными. Много тестовых заданий отклоняются работодателями по формальному признаку: человек оформил не по заданию.

Как отправлять ДЗ? Хорошей практикой было бы отправлять ссылку на `pull-request` или репозиторий и параллельно прикреплять архив с папкой, содержащей ДЗ. Однако каждый преподаватель может уточнить удобный для него способ приема ДЗ.

Когда отправлять ДЗ? Лучше вечером (ночью) за день перед уроком. Официально — за 4 часа до начала урока, но не все преподаватели могут успеть проверить за такое короткое время. **Обязательно** закладывайте запас в несколько часов на отладку возможных багов и поиск ошибок в коде. Были случаи, когда поиск одной неправильной запятой занимал несколько часов. Программирование требует большой концентрации внимания, и этот навык только вы сами можете выработать во время поиска опечаток в коде.

Совет: никогда не копируйте чужой код (в том числе и наши исходники с урока), вместо этого располагайте его в соседнем окне и печатайте **своими** руками. Это важно.

Язык программирования Python и его преимущества

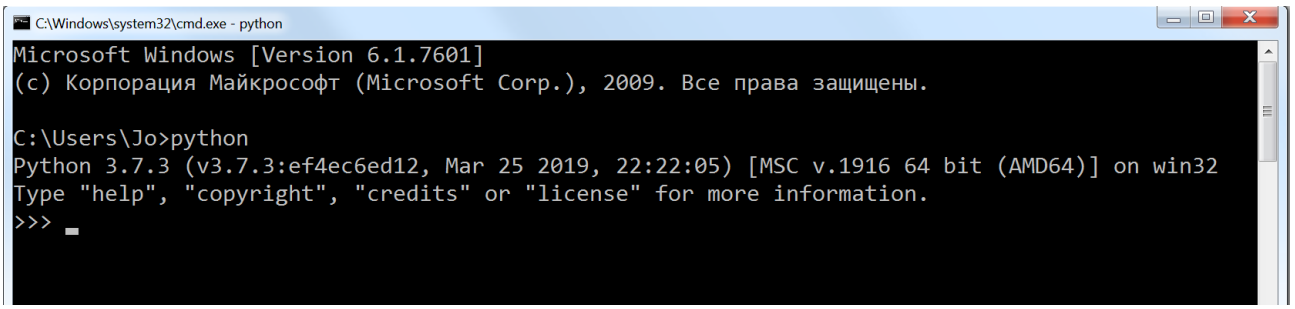
Название языка программирования происходит от телевизионного шоу «Летающий цирк Монти Пайтона», а не от имени семейства чешуйчатых. Главные его характеристики:

- высокоуровневый;
- интерпретируемый;
- строгая динамическая типизация;
- объектно-ориентированный;
- имеет свою философию The Zen of Python.

Динамическая типизация означает, что мы можем одной и той же переменной присваивать значения различных типов:

```
>>> a = 14
>>> type(a)
<class 'int'>
>>> a = '14'
>>> type(a)
<class 'str'>
>>> a = 14.0
>>> type(a)
<class 'float'>
```

Этот код был выполнен при запуске интерпретатора интерактивно в командной строке (терминале). В ОС `Windows` нужно нажать сочетание клавиш `Win + R` и написать `cmd`. А затем в появившемся черном окне написать `python`. В *nix системах нужно нажать `Ctrl + Alt + T` и написать `python3`. Если интерпретатор в системе установлен, вы должны увидеть приглашение `>>>`:



Если этого не случилось, приобретаем дополнительный опыт по правильной установке `python`-интерпретатора в систему. Чаще всего при установке забывают поставить галочку рядом с

пунктом `Add to PATH`. Есть два пути решения проблемы: красивый — добавить путь к интерпретатору вручную, быстрый — удалить python и поставить заново. Если первый способ для вас сложен, не надо тратить на него время, делайте вторым.

Для выхода из интерактивной оболочки выполняем `exit()` или нажимаем `Ctrl+D` (в ОС `Windows` `Ctrl+Z`, затем `Enter`).

Вернёмся к коду. Здесь мы использовали встроенную функцию `type()` — интуитивно понятно, что она возвращает тип переменной. В результате выполнения кода увидели, что одна и та же переменная `a` имеет тип `int`, `str` или `float` в разные моменты времени. Это и есть динамическая типизация. С одной стороны, это большое преимущество языка `Python` — повышается скорость разработки. С другой стороны, она предъявляет более высокие требования к культуре программирования в команде. В некоторых проектах используют аннотации или специальные инструменты для контроля типов, например pydantic.

Вы должны были заметить ещё одну особенность — слово `class` рядом с именем каждого типа. Важно с первых шагов знать, что в `Python` **всё** является объектом, даже числа и строки. Такой подход, как и динамическая типизация, повышает скорость разработки, но увеличивает потребляемые ресурсы вычислительной системы.

Работа интерактивно в интерпретаторе бывает чрезвычайно полезна, но мы на курсе, как правило, будем запускать написанные в среде разработки программы.

К преимуществам `Python` можно отнести лаконичный синтаксис: нет `;` в конце строк, нет операторных скобок { }, часто можно избежать обёртывания выражений в обычные скобки `()`, ключевые слова разумно сокращены (например, `def` вместо `function`). Всё это делает код, написанный на Python, выразительным и легко читающимся. Если это не так — надо рефакторить.

Ещё одно серьёзное преимущество — поддержка принципа модульного программирования и колоссальное количество уже готовых библиотек (иногда их называют «батарейками»). Их можно посмотреть на сайте https://pypi.org/.

Кроссплатформенность программ, написанных на `Python`, — это тоже большой плюс.

В отличие от некоторых языков программирования, Python дружелюбен и к программистам, пишущим код в парадигме структурного (функционального) программирования, и к программистам, предпочитающим объектно-ориентированное программирование (ООП).

К недостаткам Python можно отнести меньшую по сравнению с компилируемыми языками программирования скорость и большее потребление памяти.

Сферы применения и проекты, в которых используется Python

Области применения Python

1. Разработка сайтов — Flask, Django.
2. Разработка API — DRF.
3. Машинное обучение — TensorFlow, scikit-learn.
4. Работа с данными — Scrapy, pandas.
5. Системные утилиты — docker-compose.
6. Работа с базами данных — sqlalchemy.
7. Сложные вычисления — numpy.

Проекты, реализованные с применением Python

1. YouTube.
2. Instagram.
3. BitTorrent (до версии 6.0).
4. Blender.
5. Ubuntu Software Center.
6. DropBox.
7. Civilization IV, Battlefield 2, World of Tanks.

Кроме IT-гигантов, `Python` используют в своих разработках гиганты финансовой сферы: UBS, JPMorgan, Citadel.

Введение в стандарты программирования на Python

Одно из важнейших требований к коду Python-разработчика — следование стандарту PEP-8, который представляет собой описание рекомендованного стиля кода. Причем PEP-8 действует для основного текста программы, а для строк документации разработчику рекомендуется придерживаться положений PEP-257. Документ содержит достаточно объёмное описание стандарта. На этом курсе мы познакомимся только с частью его положений, необходимых для отработки учебных примеров и выполнения практических заданий.

1. **Необходимо избегать дополнительных пробелов в скобках (круглых, квадратных, фигурных)**

Некорректно:

```
x = [ '2', 4 ]
y = ( x [ 1 ] , x [0] )
```

```
z = { 'key' : y [ 0 ] }
```

Корректно:

```
x = ['2', 4]
y = (x[1], x[0])
z = {'key': y[0]}
```

2. Необходимо использовать пробелы вокруг арифметических операций

Некорректно:

```
a = b+c
```

Корректно:

```
a = b + c
```

Внимание: если используются операторы с разными приоритетами, правила расстановки пробелов усложняются.

3. Имена переменных и функций, атрибутов и методов класса задавать в нижнем регистре, разделяя символами подчёркивания входящие в имена слова

Некорректно:

```
MyVar, myVar, Var, VAR, MyFunc, myFunc, Func, FUNC
```

Корректно:

```
var, my_var, func, my_func
```

4. При оформлении блоков кода в Python необходимо позаботиться об отступах

Рекомендуемый отступ составляет четыре пробельных символа. Знаки табуляции применять не рекомендуется. В популярных IDE не требуется ставить пробелы вручную. При переходе на очередную строку программного кода число необходимых пробельных символов определяется автоматически или выставляется четыре пробела при нажатии клавиши Tab (удаляется по Shift+Tab).

Некорректно:

```
Главная инструкция:
    Вложенная инструкция
```

Корректно:

```
Главная инструкция:
    Вложенная инструкция
```

Визуально данные фрагменты кода идентичны, но в первом отступ выполнен с помощью табуляции, а во втором — с помощью четырёх пробельных символов.

5. Будьте внимательнее при комбинировании апострофов и кавычек

При определении строк кавычки и апострофы равнозначны. Но при их комбинировании возможны ошибки:

Некорректно:

```
print("This is my string - "text"")
print('This is my string - 'text'')
```

Корректно:

```
print("This is my string - 'text'")
print('This is my string - "text"')
```

Из чего состоит программа

Как вы думаете, чем занимается разработчик? Да, пишет программы. Зачем нужны программы? Для разных целей: решение бизнес-задачи, обработка данных, автоматизация, реализация некоторого алгоритма работы оборудования и т.д.

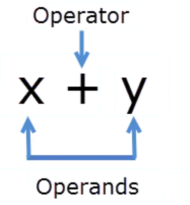
Главная способность программиста — умение абстрагироваться. Умение увидеть общее в частном. Умение формализовать порой нечёткие исходные данные. Зачастую необходимо уточнить задачу, чтобы получить релевантное решение. Пример из жизни: «Сходи в магазин!» Что здесь не так с точки зрения программиста? Есть задача, но не хватает исходных данных для её реализации. «Но ведь её реально решить», — скажете вы. Да, потому что есть контекст, в рамках которого она задана: вы из

предыдущего опыта знаете, в какой магазин идти и что там купить. Программист в такой ситуации должен задать вопросы — получить входные данные для реализации алгоритма и результата на выходе.

Любая программа на языке `Python` состоит из **строк**. В строке обычно записана какая-то операция или вызов функции.

Операция (англ. statement) — наименьшая автономная часть языка программирования; команда. Для обозначения операций используют операторы. Исходные данные задаются операндами. Операторы бывают бинарными (два операнда) и унарными (один операнд).

Пример бинарной операции:



Это было абстрактно, для начинающих изучать программирование может быть полезен конкретный пример:

```
a = 2 + 4  # здесь сразу два оператора
print(a)  # вызов функции print()
```

Если у вас возник вопрос, в каком порядке будут выполняться команды (операторы), это очень хороший признак. Можете прямо сейчас посмотреть таблицу или для начала запомнить: оператор присваивания `=` выполняется последним; в остальном порядок выполнения, как учили в школе на уроках математики. Мы еще коснёмся этого вопроса сегодня.

Первый шаг ˜— работа оператора «+» над операндами `2` и `4`, результат — значение `6`.

Второй шаг — работа оператора «=» , результат — переменной `a` (левый операнд) присвоено значение 6 (правый операнд, который является результатом вычислений на первом шаге).

Третий шаг — вызываем функцию `print` и передаём ей в качестве входных данных переменную `a`. Подробнее о функциях поговорим позже. Пока достаточно понимать, что любая функция что-то принимает в качестве исходных данных (аналогия со списком продуктов, которые нужно купить), затем она что-то делает с этими данными (аналогия с посещением магазина) и может вернуть что-нибудь в качестве результата (покупки, которые вы принесли).

Обратите внимание! Если в качестве операнда в выражении используется имя переменной, то вместо имени подставляется её значение. Прежде чем использовать переменную, её нужно объявить.

Итак, мы поговорили немного о строках. Из строк складываются модули — файлы с расширением `.py`. Модули можно объединить в пакет — папку, в которой есть файл `__init__.py`.

Постепенно небольшой проект может вырасти в библиотеку или фреймворк.

Переменные в Python

Теперь подробнее о переменных. Если вы поймёте, что такое переменная в `Python`, — автоматически узнаете, как реализована динамическая типизация. Давайте подумаем, что такое по сути `https://geekbrains.ru`: это сайт или ссылка, при помощи которой можно попасть на сайт? Правильно: ссылка, то есть способ попасть на сайт.

Попробуйте выполнить код:

```
a = 6
print(a, id(a))  # 6 8791193445168
```

Вы увидели два значения: первое — это число 6, а второе — результат работы функции id(), возвращающей адрес объекта, с которым связана переменная (в нашем случае это переменная `a`). Если присвоить новое значение переменной `a`, функция `id()` вернёт другой адрес — теперь та же переменная ссылается на другой объект. Для подтверждения модифицируем выполненный ранее в оболочке Python код и запустим в PyCharm:

```
a = 14
print(a, id(a))  # 14 8791280280624
a = '14'
print(a, id(a))  # 14 30828112
a = 14.0
print(a, id(a))  # 14.0 195568840
```

Так как переменная в Python — это по сути ссылка, информация о типе и остальные данные хранятся в самом объекте. Получаем динамическую типизацию.

Однако всё не так просто. Нюансы возникают, когда мы начинаем присваивать одну переменную другой, но об этом поговорим на втором уроке.

Как вы уже заметили, у каждой переменной в Python должно быть уникальное имя. Можно использовать буквы a–z, цифры 0–9, символ «_». В Python принят стиль **`snake_case`** (змеиная нотация), когда имя переменной пишем в нижнем регистре, разделяя семантические части символом подчеркивания.

Правильно:

```
first_name, user_1, request, _tmp_name, min_step_shift
```

Как и в других языках программирования, первым символом не может быть цифра. Python — язык регистрозависимый (большие и маленькие буквы в имени переменной различаются).

Неправильно:

```
1_name, User_1, 0request, tmpName, minStep_shift
```

Отдельного типа для констант в Python нет, но, как и во многих других языках, есть соглашение между разработчиками — писать константы в верхнем регистре, например:

```
MAX_ATTEMPTS, MEDIA_URL, DEBUG
```

Очень важно: давайте переменным осмысленные имена, избегайте коротких переменных вида `a`, `b`, `x`, `y`, `z`. Не рекомендуется использовать переменные `i` и `l` — они похожи друг на друга и цифру `1`. И, конечно, **никакого транслита**: `skorost`, `imya`. Только английские слова. Если с английским не очень, надо отложить обучение программированию и подтянуть его.

Как правильно именовать собственные классы в Python, вы узнаете позже в курсе, посвящённом ООП.

Важно! Обязательно распечатайте на отдельном листе зарезервированные слова и имена, используемые в стандартной библиотеке Python (встроенные функции, встроенные типы и т.д.), — их **ни в коем случае** нельзя использовать в качестве имен переменных.

Очень грубые ошибки:

```
list = [1, 2, 3, 4]
str = 'hello world'
pass = 'my password'
```

Ввод/вывод данных в Python

Как мы уже говорили, для работы любой программы нужны исходные данные. Когда мы пишем простые программы, эти данные можно «захардкодить» (написать в коде). Но в какой-то момент возникнет необходимость получить их извне. Какие есть варианты решения этой задачи в Python?

1. С клавиатуры: функция input().

2. Из файла: функция open().
3. Из базы данных: например, модуль sqlite3.
4. Из web API: например, внешняя библиотека requests.
5. Запуск скрипта с параметрами.

Нам пока будет достаточно ввода с клавиатуры. Всё просто: функция `input()` принимает единственный необязательный аргумент — приглашение (подсказку) для ввода и всегда возвращает объект класса `str` (строку):

```
user_name = input('введите имя: ')
```

В результате выполнения этого кода мы увидим в консоли приглашение с текстом 'введите имя: '. После того, как пользователь введёт имя и нажмет клавишу `<Enter>`, введённые данные будут присвоены переменной `user_name`. Дальше можно что-то делать с ними. Например, вывести приветствие:

```
user_name = input('введите имя: ')  # Иван
print('Добрый вечер, ', user_name, '!')  # Добрый вечер,  Иван !
```

Трюк! Если вы работаете в PyCharm, то можете, удерживая клавишу `<Ctrl>`, подвести курсор мыши к имени функции `input()` в коде и дождаться появления контекстной справки. Можно даже кликнуть — окажетесь в исходниках функции. Там будет много незнакомого кода, поэтому рекомендуем вернуться к этой манипуляции позже.

Арифметические операции в Python

Мы уже говорили об операциях сегодня. Теперь конкретнее про арифметику. Есть классические операторы:

- +
- -
- *
- /

Вы же не забыли, что такое операнды? Что значит «бинарная операция» или «унарная операция»? Если помните, идем дальше и узнаем, как в Python можно возвести число `x` в степень `y`: **`x ** y`**.

```
print(5 ** 2)  # 25
x = 2
y = x ** 3
print(y)  # 8
```

Во многих алгоритмах необходимо поделить нацело. Возможно вы слышали про оператор `div` — именно про него и идет речь. В Python его роль играет оператор `//`. В качестве примера сравним целочисленное деление и обычное:

```
x = 5 // 2
y = 5 / 2
print(x, type(x))  # 2 <class 'int'>
print(y, type(y))  # 2.5 <class 'float'>
```

Важно! Оператор `//` всегда возвращает целое число.

Спутником оператора `div` в программировании является оператор `mod` — остаток от деления. В Python это будет оператор `%`. Рассмотрим на примере:

```
a = 17 % 5
b = 5 % 5
c = 3 % 5
d = 0 % 5
print(a)  # 2
print(b)  # 0
print(c)  # 3
print(d)  # 0
```

Совет! Вспомните, как в школе находили остаток от деления. Подумайте, почему получились такие результаты. Не всё так просто, как может показаться на первый взгляд

Случай из реальности. Есть адрес ячейки памяти, который вычисляется по определённому алгоритму. Всего существует 128 ячеек. Как гарантировать, что адрес никогда не превысит 127? Почему 127? Ведь адресация в информатике начинается с 0! Решение — использовать оператор `%`:

```
MEMORY_ROWS = 128

_address = 148
address = _address % MEMORY_ROWS
print(address)  # 20

_address = 127
address = _address % MEMORY_ROWS
print(address)  # 127

_address = 128
address = _address % MEMORY_ROWS
print(address)  # 0

_address = 20
address = _address % MEMORY_ROWS
print(address)  # 20
```

Увидели в этом коде константу? Отлично. Увидели переменную, имя которой начинается с «_»? Так обычно именуют вспомогательные или временные переменные. Задайте себе вопрос: «Все ли коварные случаи рассмотрены в этом примере?» Важно всегда задумываться, «когда может что-то сломаться в алгоритме».

Логические операции в Python и операции сравнения

Если вы ещё не устали, поговорим о логике. Основным отличием логических выражений от привычных нам со школы арифметических является то, что их результатом является либо True, либо False. Обычно в таких выражениях используют три оператора:

- `and` — логическое умножение «И»;
- `or` — логическое сложение «ИЛИ»;
- `not` — логическое отрицание «НЕ».

Логическое умножение работает по сути как обычное:

```
print(True and True)  # True
print(1 * 1)  # 1
print(True and False)  # False
print(1 * 0)  # 0
```

С логическим сложением немного интереснее:

```
print(True or True)  # True
print(1 + 1)  # 2
print(True or False)  # True
print(1 + 0)  # 1
print(False or False)  # False
print(0 + 0)  # 0
```

Видим, что есть отличие для случая «`1 + 1`». По сути, в логическом выражении произошло «округление» до ближайшего значения — `True`. Это особенности двоичной системы счисления, которая представлена значениями `False` (аналог `0`) и `True` (аналог `1`). Для представления десятичного числа 2 в двоичной системе необходимо добавлять разряд, как мы это делаем, переходя от 9 к 10. Если вам интересен этот момент, почитайте ещё об одной логической операции: исключающее или.

Логическое отрицание меняет значение на противоположное:

```
print(not True)  # False
print(not False)  # True
```

Операндами логических выражений могут быть операции сравнения:

- «==» — равно;
- «!=» — не равно;
- «>» — больше;
- «>=» — больше или равно;
- «<» — меньше;
- «<=» — меньше или равно.

Пример из реальности:

```
age = 41
gender = 'F'
do_mammography = gender == 'F' and age > 40
print(do_mammography)  # True
```

Что здесь происходит? Сначала вычисляем выражение `gender == 'F'`. Оно равно `True`. Потом вычисляем выражение `age > 40` — оно тоже `True`. И уже потом вычисляем выражение `True and True` и видим результат — `True`. Почему именно в таком порядке идут вычисления, узнаем позже.

Зачем нужны логические выражения? Обычно это принятие решений: делать или не делать что-либо в ходе реализации алгоритма.

Важно! В современных языках программирования используется трюк, связанный с ленью логических операторов: если левый операнд `and` равен `False` или левый операнд `or` равен `True` — правый операнд не вычисляется. Говорят, что «логическое умножение лениво по лжи» и «логическое сложение лениво по правде». Подумайте, почему ввели такую оптимизацию?

Продвинутый пример:

```
gender = 'M'
do_mammography = gender == 'F' and age > 40
print(do_mammography)  # False

gender = 'F'
do_mammography = gender == 'F' and age > 40
print(do_mammography)  # NameError: name 'age' is not defined
```

Почему первый раз не было ошибки, хотя переменная `age` не задана вообще в этом коде? Это лень.

Порядок выполнения операций в Python

Выражения, которые мы до этого рассматривали, были очень простыми. Приведём более сложный пример:

```
room_number = 78
do_cleaning = room_number and room_number // 10 > 7 or room_number % 10 == 3
print(do_cleaning)  # False

room_number = 73
do_cleaning = room_number and room_number // 10 > 7 or room_number % 10 == 3
print(do_cleaning)  # True

room_number = 80
do_cleaning = room_number and room_number // 10 > 7 or room_number % 10 == 3
print(do_cleaning)  # True
```

На собеседовании вас вполне могут попросить в уме вычислить такое выражение. Часто программисты ставят скобки, чтобы однозначно задать порядок вычислений, как это делалось в школе. Но такой подход — следствие незнания порядка выполнения операций. Обязательно распечатайте эту таблицу и держите первое время перед глазами:

Приоритет выполнения операций в Python от самого низкого к самому высокому
`=`
`if – else`
`or`
`and`
`not x`
`in, not in, is, is not, <, <=, >, >=, !=, ==`
`+, -`
`*, /, //, %`
`+x, -x`
`**`
`x[index], x[index:index], x(arguments...), x.attribute`

`(expressions...),`
`[expressions...], {key: value...}, {expressions...}`

Как пользоваться таблицей? Чем ниже операция в таблице, тем раньше она будет выполняться (у неё более высокий приоритет). Видно, что присваивание «=» будет всегда происходить последним. Логическое умножение всегда раньше логического сложения. А отрицание всегда раньше умножения. Приоритет скобок выше любой из логических или арифметических операций.

Ветвления в Python

Рано или поздно вам понадобится менять поведение программы в зависимости от некоторых условий. Для этого придумали ветвления. Попробуем вывести информацию о чётности числа:

```
number = 157
if not number % 2:
   print('число четное')
else:
   print('число нечетное')  # число нечетное
```

Здесь мы знакомимся с очень важным аспектом Python — indentation. При написании алгоритмов часто бывает необходимо выполнить несколько действий как нечто цельное — их называют блоками кода. Например, если вы решили выйти из дома, то должны обуться, открыть дверь, пройти по подъезду, открыть входную дверь. Эта цепочка событий выполняется как нечто целокупное. Во многих языках блоки кода обособляются при помощи фигурных скобок `{...}`. С течением времени приобрёл популярность способ обособления при помощи отступов, например, в `*nix` системах конфигурационные файлы `.yaml`. Именно он и используется в Python: **единица отступа,** или **`indent`** принят равным **4 пробелам**. Любое другое значение, например любимый многими символ табуляции, не поощряется в python-сообществе. В среде Pycharm при нажатии клавиши `Tab` автоматически происходит замена на 4 пробела. Следует отметить, что блоку кода должно предшествовать двоеточие в конце предыдущей строки.

С отступами перед функциями `print()` разобрались. Теперь о двух ключевых словах: `if` и `else`. Если выражение, которое стоит после `if`, равно `True`, выполняется код, размещённый в следующем после двоеточия блоке кода. Если необходимо выполнить некоторый код, когда выражение, которое стоит после `if`, равно `False`, пишем его в блоке после ключевого слова `else` (от англ. «иначе»). Если вы планируете быстро прогрессировать — почитайте про elif.

Пояснение к примеру. Остаток от деления `157` на `2` равен `1`, что эквивалентно `True`. После отрицания получаем `False`, значит, выполнится код после ключевого слова `else` и мы видим фразу «`число нечетное`».

Важно! Если вам что-то непонятно в этом примере — разбирайтесь с ним. Дальше идти смысла нет.

Списки в Python. Введение

Что делать, если нам необходимо в программе хранить информацию о продажах офиса за текущий день? Или треки плейлиста? Или оценки ученика за четверть? Для таких целей в языках программирования существуют контейнерные типы данных. В Python список (list) — один из них.

Пример:

```
half_year_months = ['январь', 'февраль', 'март', 'апрель', 'май', 'июнь']
print(type(half_year_months))  # <class 'list'>
```

Синтаксически списки похожи на массивы — квадратные скобки и перечисление элементов через запятую. Но есть и отличия: размер списка может изменяться (можно добавлять и удалять элементы), список может хранить элементы различных типов.

Можно провести параллель между списком и почтовыми ящиками. Списки гарантируют сохранение порядка элементов — если какой-то элемент был добавлен в список третьим, он останется на этом месте, пока мы сами что-то не поменяем. Все ячейки списка, как и почтовые ящики, пронумерованы. Как вы думаете, какой номер у первой ячейки? Правильно — номер ноль! Для доступа к ячейкам списка используем имя переменной и квадратные скобки:

```
...
print(half_year_months[2])  # март
```

Здесь мы прочитали данные из 3-й ячейки. Аналогичным образом можно сохранить новые данные в любую существующую ячейку:

```
...
half_year_months[2] = 'Март'
print(half_year_months)  # ['январь', 'февраль', 'Март', 'апрель', ...]
```

Можно добавить элемент **в конец списка** при помощи метода `.append()`:

```
...
half_year_months.append('March')
print(half_year_months)  # [..., 'май', 'июнь', 'March']
```

Примечание: метод - это действие, обычно глагол, подробнее будем говорить на следующем уроке.

Фактически мы можем работать с любой ячейкой списка, как с обычной переменной: можем получить значение, можем присвоить. Пока этих знаний про списки будет достаточно.

Циклы в Python

Зачем нужны циклы

Давайте представим, что у нас есть список с данными о продажах супермаркета за сутки, состоящий из 10 000 ячеек. Нам нужно вычислить среднюю сумму продаж за один товар. Сможем ли мы решить эту задачу, опираясь на текущие знания? Да. Строк кода получится 10 000+! Как вы думаете, это хорошая идея? Разумеется, нет. Важно заметить, что нам нужно 10 000 раз повторить одно и то же действие: добавить стоимость текущего товара в общую сумму. Для таких задач придумали циклы.

Как у всех — цикл while

Это классика в любом языке программирования — цикл с предусловием. Конкретный пример:

```
day_sales = [1589.5, 2687.5, 5478.2, 1236.5, 4756.5]
idx = 0
total_sales = 0
while idx < len(day_sales):  # pre-condition
   total_sales = total_sales + day_sales[idx]
   idx = idx + 1
price_per_product = total_sales / len(day_sales)
print(price_per_product)  # 3149.6400000000003
```

Если вам этот код кажется сложным, нужно досконально его разобрать. Дальше идти нельзя.

Переменная `idx` здесь — это счётчик или индекс. Суммарные продажи храним в `total_sales`. Перед каждым шагом цикла проверяем, не вышло ли значение счётчика за пределы допустимого. Подумайте, почему здесь сравнение «меньше», а не «меньше, либо равно»? Это важный нюанс.

Если предусловие верно, выполняется тело (шаг) цикла — блок кода, который надо повторить многократно. В нашем примере тело цикла состоит всего из двух строк: добавляем к суммарным продажам стоимость текущего продукта и увеличиваем счетчик на единицу. Сколько раз, по-вашему,

оно будет выполняться? Здесь важно вспомнить роль индентов в Python — ведь именно четыре пробела определяют, принадлежит очередная строка телу цикла или нет.

Мы использовали в примере новую функцию len(). Надеемся, что вы догадались о её роли: в данном случае она определяет длину списка.

Проведем рефакторинг кода:

```
day_sales = [1589.5, 2687.5, 5478.2, 1236.5, 4756.5]
idx = 0
total_sales = 0
while idx < len(day_sales):
   total_sales += day_sales[idx]
   idx += 1
price_per_product = total_sales / len(day_sales)
print(price_per_product)  # 3149.6400000000003
```

Использовали оператор `+=`, который добавляет к существующему значению величину выражения справа. Надеемся, что вы всегда будете использовать этот оператор в подобных ситуациях.

Не как у всех — цикл for in

Можно ли решить задачу проще? Да:

```
day_sales = [1589.5, 2687.5, 5478.2, 1236.5, 4756.5]
total_sales = 0
for product_price in day_sales:
   total_sales += product_price
price_per_product = total_sales / len(day_sales)
print(price_per_product)  # 3149.6400000000003
```

Здесь мы не использовали счётчик. Вместо этого просто перебрали все значения списка `day_sales` в цикле-итераторе `for in`. Это один из важнейших моментов изучения Python. Нам не нужно обращаться по номерам к ячейкам списка. Просто говорим «следующее значение», до тех пор пока эти значения есть. Текущее значение хранится в переменной, которая написана после ключевого слова `for`. Имя этой переменной может быть любым, в нашем случае читабельность кода выиграет от имени `product_price`. Справа от ключевого слова `in` должно быть нечто, называемое «итерируемым» (iterable) — то, для чего можно сказать «следующий в …».

Примеры из реальности: очередь в супермаркете (покупатель), книга (страница), лента новостей (новость).

Особенность: цикл `for in` используется только для обхода последовательностей, но не для изменения значений в этих последовательностях. Мы не сможем, например, применить скидку и поменять цены в списке `day_sales`, используя этот цикл.

Меняем значения — range() + цикл for in

Решим задачу о скидке на товар. Чтобы изменить значение элемента списка, необходимо обратиться к нему по номеру. Мы делали так в цикле `while`, используя счетчик `idx`. Но в Python есть способ намного эффективнее — функция `range(n)`, которая генерирует (запомните это слово!) числа от `0` до `n-1`. В сочетании с циклом `for in` получается красивый способ изменения значений в последовательности:

```
day_sales = [1589.5, 2687.5, 5478.2, 1236.5]
discount = 5
for idx in range(len(day_sales)):
   day_sales[idx] *= (100 - discount) / 100
print(day_sales)  # [1510.0249999999999, 2553.125, 5204.29, 1174.675]
```

Мы уменьшили значение каждой цены в исходном списке на 5%. **Важно:** новый список не создавали! Можно было решить эту задачу через цикл `while`, но это будет сложнее выглядеть — не соответствует Zen of Python.

Узнаём номер элемента в цикле for in

Бывают ситуации, когда необходимо при обходе последовательности знать номер элемента. Можно по аналогии с предыдущим примером использовать связку `range()` и `for in`, но есть более красивое решение — функция `enumerate()`:

```
day_sales = [1589.5, 2687.5, 5478.2, 1236.5, 4756.5]
for idx, item in enumerate(day_sales, start=1):
   print('товар №', idx, '-', item)
```

Этот пример посложнее предыдущих. Пока мы его приводим для ознакомления. Внешне логика должна быть понятна: функция `enumerate()` что-то делает, и в результате появляются пары значений `<номер элемента>` и `<элемент>`. Значения этой пары неким образом попадают в переменные `idx` и `item`. Дополнительный аргумент `start=1` для функции `enumerate()` — это число, с которого начинается нумерация (по умолчанию `0`).

На сегодня с теорией закончили.

Задача. Написать программу, определяющую, что число трёхзначное и средняя цифра равна 5.

Никогда не решайте задачу целиком сразу. Постарайтесь декомпозировать ее на несколько простых подзадач. В нашем случае сначала можно сделать проверку, что число трехзначное:

```
num = 457
is_three_digits = 0 < num // 100 <= 9
print(is_three_digits)  # True
```

Если число меньше 100, остаток от целочисленного деления будет всегда 0. Если число больше 999, он будет больше 9. Да, в Python можно писать двойные неравенства!

Как теперь выяснить, что средняя цифра равна 5? По сути, это десятки. Доберемся до них:

```
...
tens = num % 100 // 10
print(tens)  # 5
```

Над этим кодом советуем подумать минут 10–15. Он непрост. Советуем потренироваться извлекать единицы, сотни из чисел.

Остался финал:

```
...
print(tens == 5)  # True
```

Многие пытаются решать эту задачу читерским методом через строки. Огорчим вас: на собеседовании читерство запрещено. Всех интересует, умеете ли вы придумывать и реализовывать алгоритмы.

Практическое задание

1. Реализовать вывод информации о промежутке времени в зависимости от его продолжительности `duration` в секундах:
    a. до минуты: <s> сек;
    b. до часа: <m> мин <s> сек;
    c. до суток: <h> час <m> мин <s> сек;
    d. * в остальных случаях: <d> дн <h> час <m> мин <s> сек.

    Примеры:

```
duration = 53
53 сек
duration = 153
2 мин 33 сек
duration = 4153
1 час 9 мин 13 сек
duration = 400153
4 дн 15 час 9 мин 13 сек
```

Примечание: можете проверить себя здесь, подумайте, можно ли использовать цикл для проверки работы кода сразу для нескольких значений продолжительности, будет ли тут полезен список?

2. Создать список, состоящий из кубов нечётных чисел от 1 до 1000 (куб X - третья степень числа X):
    a. Вычислить сумму тех чисел из этого списка, сумма цифр которых делится нацело на 7. Например, число «19 ^ 3 = 6859» будем включать в сумму, так как 6 + 8 + 5 + 9 = 28 – делится нацело на 7. **Внимание:** использовать только арифметические операции!
    b. К каждому элементу списка добавить 17 и заново вычислить сумму тех чисел из этого списка, сумма цифр которых делится нацело на 7.
    c. * Решить задачу под пунктом `b`, не создавая новый список.

3. Склонение слова

    Реализовать склонение слова «процент» во фразе «N процентов». Вывести эту фразу на экран отдельной строкой для каждого из чисел в интервале от 1 до 100:

    1 процент

    2 процента

    3 процента

    4 процента

    5 процентов

    6 процентов

    ...

    100 процентов

Задачи со * предназначены для продвинутых учеников, которым мало сделать обычное задание. Пробуйте их решать, если **уверены** в своих силах.

Дополнительные материалы

1. Лутц Марк. Изучаем Python.
2. enumerate().

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://realpython.com/python-for-loop/.
2. https://docs.python.org/3/.