

Основы Python

Автор: Сергей Балакирев

Автор практической части: Имада Шерифадзе

Основы Python

Часть I

Батуми – 2023

Основы Python

1. Первое знакомство с Python Установка на компьютер	9
2. Варианты исполнения команд. Переходим в PyCharm	16
3. Переменные, оператор присваивания, функции type и id	21
4. Числовые типы, арифметические операции	28
5. Математические функции и работа с модулем math	35
6. Функции print() и input(). Преобразование строк в числа int() и float()	39
7. Логический тип bool. Операторы сравнения и операторы and, or, not	44
8. Введение в строки. Базовые операции над строками	51
9. Знакомство с индексами и срезами строк	56
10. Основные методы строк	60

Основы Python

11. Спецсимволы, экранирование символов, raw-строки	70
12. Форматирование строк: метод format и F-строки	75
13. Списки - операторы и функции работы с ними	78
14. Срезы списков и сравнение списков	86
15. Основные методы списков	91
16. Вложенные списки, многомерные списки	97
17. Условный оператор if. Конструкция if-else	102
18. Вложенные условия и множественный выбор. Конструкция if-elif-else	106
19. Тернарный условный оператор. Вложенное тернарное условие	111
20. Оператор цикла while	116
21. Операторы циклов break, continue и else	121
22. Оператор цикла for. Функция range()	126
23. Примеры работы оператора цикла for. Функция enumerate()	133

Основы Python

24.Итератор и итерируемые объекты. Функции iter() и next()	137
25.Вложенные циклы. Примеры задач с вложенными циклами	142
26.Треугольник Паскаля как пример работы вложенных циклов	148
27.Генераторы списков (List comprehensions)	150
28.Вложенные генераторы списков	156
29.Введение в словари (dict). Базовые операции над словарями	160
30.Методы словаря, перебор элементов словаря в цикле	165
31.Кортежи (tuple) и их методы	170
32.Множества (set) и их методы	177
33.Операции над множествами, сравнение множеств	183
34.Генераторы множеств и генераторы словарей	190

Основы Python

35.Функции: первое знакомство, определение def и их вызов	193
36.Оператор return в функциях. Функциональное программирование	200
37.Алгоритм Евклида для нахождения НОД	206
38.Именованные аргументы. Фактические и формальные параметры	211
39.Функции с произвольным числом параметров *args и **kwargs	216
40.Операторы * и ** для упаковки и распаковки коллекций	220
41.Рекурсивные функции	224
42.Анонимные (lambda) функции	231
43.Области видимости переменных. Ключевые слова global и nonlocal	235
44.Замыкания в Python	240
45.Введение в декораторы функций	245
46.Декораторы с параметрами. Сохранение	252

Основы Python

свойств декорируемых функций	
47.Импорт стандартных модулей. Команды <code>import</code> и <code>from</code>	256
48.Импорт собственных модулей	261
49.Установка сторонних модулей (<code>pip install</code>). Пакетная установка	266
50.Пакеты (<code>package</code>) в Python. Вложенные пакеты	269
51.Функция <code>open</code> . Чтение данных из файла	276
52.Исключение <code>FileNotFoundError</code> и менеджер контекста (<code>with</code>) для файлов	282
53.Запись данных в файл в текстовом и бинарном режимах	286
54.Выражения генераторы	292
55.Функция-генератор. Оператор <code>yield</code>	296
56.Функция <code>map</code> . Примеры ее использования	300
57.Функция <code>filter</code> для отбора значений итерируемых объектов	305

Основы Python

58.Функция zip. Примеры использования	308
59.Особенности сортировки через sort() и sorted()	312
60.Аргумент key для сортировки коллекций по ключу	316
61.Функции isinstance и type для проверки типов данных	320
62.Функции all и any. Примеры их использования	323
63.Расширенное представление чисел. Системы счисления	328
64.Битовые операции И, ИЛИ, НЕ, XOR. Сдвиговые операторы	333
65.Модуль random стандартной библиотеки	341
66.Аннотация базовыми типами	345
67.Аннотации типов коллекций	353
68.Аннотации типов на уровне классов	361
69.Конструкция match/case. Первое знакомство	370
70.Конструкция match/case с кортежами и списками	379

Основы Python

71.Конструкция match/case со словарями и множествами	384
72.Конструкция match/case. Примеры и особенности использования	390
73. Практические упражнения	399

Введение

В безграничном царстве языков программирования **Python** представляет собой сияющую жемчужину, сияющую простотой и мощностью. Подобно элегантному симфоническому дирижеру, **Python** с изяществом управляет искусством кодирования, что делает его идеальным выбором как для новичков, так и для опытных разработчиков. В этом цифровом путешествии мы приступим к исследованию основополагающих принципов **Python**, раскрывая его универсальность, ясность и удивительную способность превращать абстрактные идеи в осязаемый код. Итак, независимо от того, являетесь ли вы начинающим программистом, желающим впервые познакомиться с миром кодирования, или опытным разработчиком, желающим добавить **Python** в свой арсенал, приготовьтесь быть очарованным очаровательным языком **Basic Python**.

Основы Python

Python — динамичный и универсальный язык программирования, известный своей простотой, читабельностью и надежностью. В этой книге мы отправимся в увлекательное путешествие в мир **Python**, где вы откроете для себя его врожденную элегантность и мощь.

На этих страницах вы изучите фундаментальные концепции **Python**, начиная с таких основ, как переменные, типы данных и структуры управления. По мере нашего продвижения вы углубитесь в более сложные темы, такие как функции, объектно-ориентированное программирование и обработка файлов. Мы также изучим обширную экосистему библиотек и модулей **Python**, которые позволят вам решать широкий спектр задач: от веб-разработки до анализа данных и не только.

Но эта книга посвящена не только изучению синтаксиса **Python**; речь идет о овладении искусством решения проблем посредством программирования. Вы будете участвовать в практических упражнениях и практических примерах, которые отточат ваши навыки программирования и предоставят вам инструменты для решения реальных задач.

Независимо от того, являетесь ли вы новичком или опытным программистом, желающим добавить Python в свой набор навыков, эта книга послужит вашим надежным руководством. К концу вы получите четкое понимание основных концепций Python и уверенность в том, что сможете творчески применять их для создания собственных программных решений. Итак, приготовьтесь открыть двери в мир возможностей программирования, используя Python в качестве ключа.

§1. Первое знакомство с Python Установка на компьютер

Здравствуйтесь, дорогие друзья! Рад, что вы начинаете изучать один из самых моих любимых языков программирования **Python**. Его еще называют **Питон**. **Почему именно Питон?** Это нужно спросить у **Гвидо ван Россума** – главного идеолога и первоначального разработчика этого языка.

Основы Python



Задуман он был еще в **1980**-х, а первая версия вышла в **1991** году. Конечно, с тех пор он претерпел множество улучшений, особенно в версии

Python 3.0

выпущенной в декабре **2008** года. Отличия оказались настолько значительными, что программы предыдущих версий 2.x далеко не всегда можно запустить в интерпретаторе **Python 3.x**. Однако, сейчас это уже не проблема, так как с **2008** года прошло много времени и все важные программы были адаптированы к версии **3**. Именно эту последнюю, современную версию языка **Python** мы с вами и будем изучать.

Язык оказался настолько удачным, что он стал широко применяться при разработке алгоритмов искусственного интеллекта, в частности, в нейронных сетях. При разработке серверной части сайтов, используя известные фреймворки **Django** и **Flask**. Например, на нем разработаны сайты **Youtube**, **Instagram**, поиск от **Google**, **DropBox** и многие другие



Основы Python

Также он используется в многочисленных научных проектах, где требуются сложные математические вычисления или реализация алгоритмов обработки данных, в том числе и больших данных – **Big Data**. В последнее время **Python** стали применять также и для создания игр, обычно мобильных. А если вы еще школьник, то этот язык ввели в **ОГЭ** и **ЕГЭ** по информатике, а также в олимпиадное программирование.

Но, все же, **почему этот язык приобрел такую широкую популярность и завоевал любовь программистов по всему миру?** Считается, что все благодаря его простым, понятным, явным конструкциям, хорошо читаемому тексту программ, богатой библиотеке модулей для программирования самых разных задач и возможность использования языка практически на всех платформах: **Windows, Linux, Mac OS, Android, iOS** и др. Хотя, на самом деле понять, почему один проект «выстреливает», а большинство уходят в небытие, еще никому не удалось. Но **Питон** – это яркая история успеха, правда, не лишенный и своих недостатков. Главные из них – более медленная скорость работы программ и больший объем используемой памяти, по сравнению, например, с аналогичным кодом, написанном на языке **C++**. Но, все же, **Python** позволяет заметно быстрее реализовывать сложные алгоритмы и этим качеством он затмевает большинство современных языков программирования. А скорость работы критического фрагмента кода можно увеличить, если реализовать его на том же **C++**, а затем, вызывать из **Python**-программы. Например, так делается при проектировании и обучении нейронных сетей. Благодаря этому получаем удобство программирования и высокую скорость исполнения.

Установка языка Python

Давайте теперь выполним установку интерпретатора этого языка на свое устройство. У меня – это стационарный компьютер с **ОС Windows 10**. Если я нажму клавиши **Win + Pause**, то появится окно с подробной информацией о системе. В частности, здесь написано название **ОС** и ее разрядность **x64**. Конечно, у вас может быть любое другое устройство и **ОС** – это не существенно, в любом случае, нам нужно перейти на официальный сайт языка **Python**:

Основы Python

<https://www.python.org/>

Затем, выбрать вкладку «**Downloads**» и здесь появится список различных ОС. Вам нужно найти свою ОС и перейти. Так как у меня **Windows**, то я выбираю этот пункт и появляется страница с выбором различных версий языка. Конечно же, следует взять последнюю стабильную для **Python 3**. На момент написания этой книги – это версия **3.11.5**. Я скачаю 64-bit версию установщика:

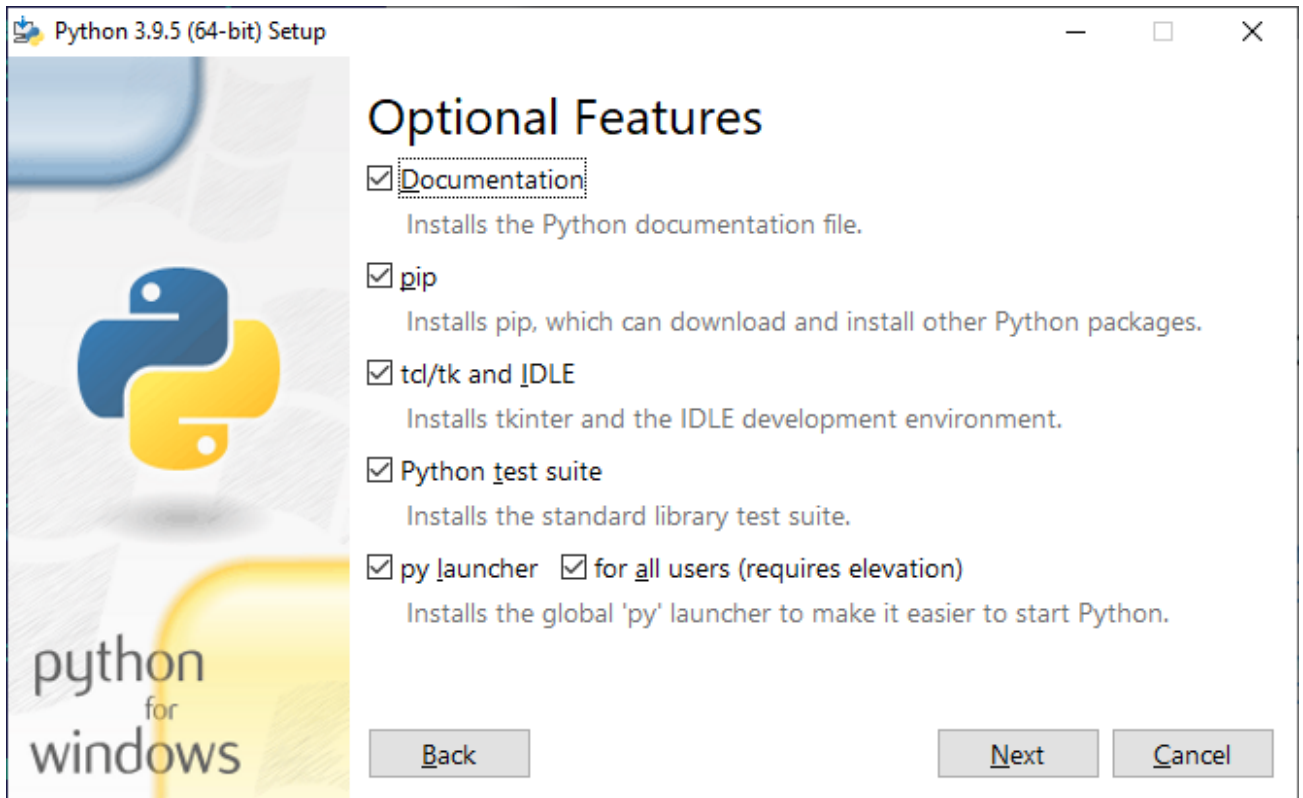
Download Windows installer (64-bit)

так как у меня ОС **Windows 64**-разрядная (обычно, именно такая и установлена на домашнем компьютере). И запущу его. Появится окно, в котором обязательно следует отметить галочкой опцию «**Add Python to PATH**». Она позволит в дальнейшем запускать интерпретатор языка без указания полного пути к нему:



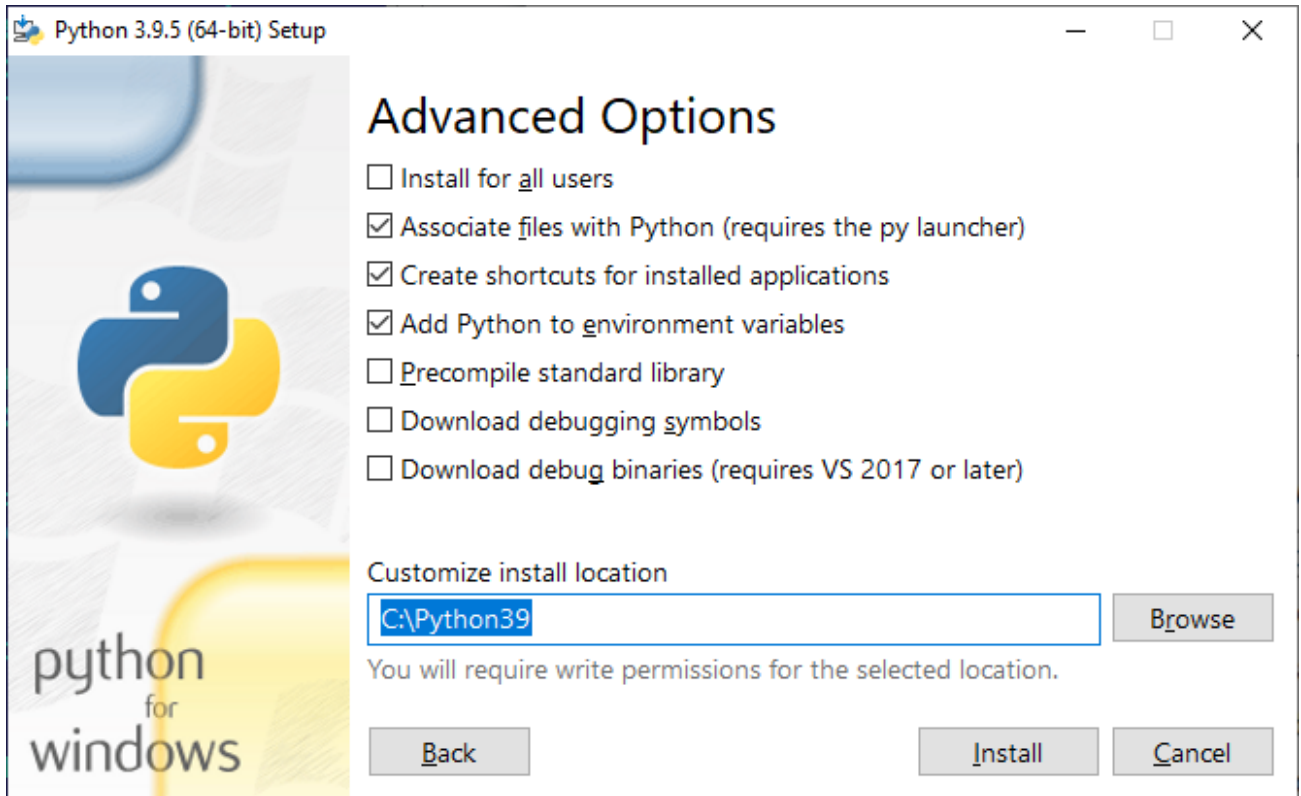
Основы Python

Затем, выбираем режим установки «Customize installation». Здесь все опции должны быть отмечены галочками:



Нажимаем «Next» и в следующем окне я рекомендую путь, который прописан по умолчанию, изменить на другой. А именно, убрать все промежуточные подкаталоги и установить интерпретатор в корень выбранного диска:

Основы Python

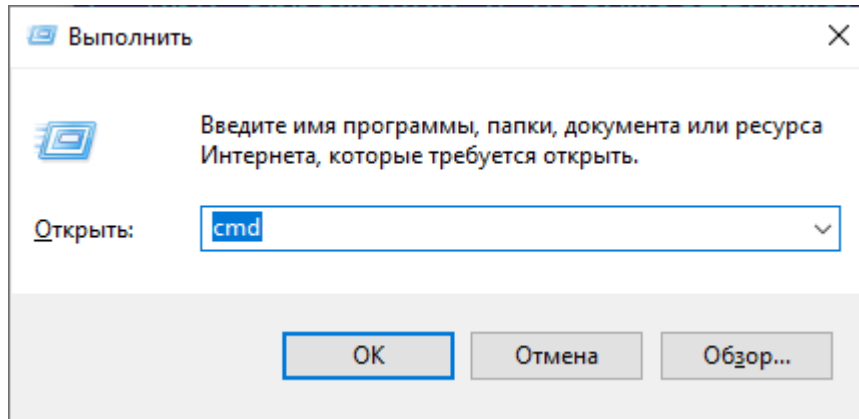


У меня получился путь «C:\3.11». Здесь дополнительно указана версия языка 3.11. Это также рекомендуется делать, так как разных версий на одном устройстве может быть несколько и чтобы переключаться между ними, они должны находиться в разных директориях.

После этого, нажимаем на кнопку «**Install**» и программа устанавливается в указанный каталог.

Осталось проверить работоспособность языка **Python**. Во-первых, мы можем открыть окно выполнения команд в ОС Windows (Win+R) и в появившемся окне набрать «**cmd**»:

Основы Python



Далее, в консольном командном окне набираем «python» и должны увидеть текущую версию интерпретатора этого языка:

```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows [Version 10.0.19042.1052]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\Sergey>python
Python 3.9.5 (tags/v3.9.5:0a7dcdb, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Если вы работаете под **Linux** или **Mac OS**, то следует набрать команду «python3».

Но писать и исполнять команды самого языка лучше через среду **IDLE**, которая устанавливается вместе с интерпретатором. Для ее запуска достаточно нажать на кнопку «Пуск» и выбрать оболочку **IDLE**. Появится окно, в котором можно в интерактивном режиме выполнять любые команды языка. Например, математические

Основы Python

2+4

3*5

8/6

Мы о них еще подробно будем говорить. Здесь я просто привожу пример и показываю, как выполняются простейшие команды языка **Python**.

На этом мы с вами завершим наше первое занятие. В качестве самостоятельного задания вам нужно установить интерпретатор языка на свое устройство (желательно, чтобы оно имело полноценную клавиатуру и манипулятор «мышь»). И попробуйте выполнить в среде **IDLE** те же самые команды, что я привел в этом занятии. До встречи на следующем занятии!

§2. Варианты исполнения команд. Переходим в PyCharm

Здравствуйте, дорогие друзья! Мы продолжаем курс по **Python**. На этом занятии вы узнаете о двух вариантах исполнения команд: **интерактивном** и **файловом**, а также затронем функцию **print()** для отображения данных в консоли **Питона**.

Я, надеюсь, вы все успешно установили интерпретатор языка на свое устройство и смогли повторить команды из прошлого занятия в среде **IDLE**. Так вот, это, как раз интерактивный режим их выполнения, когда каждая конструкция языка отрабатывает сразу же после нажатия на кнопку **Enter**.

Преимущество такого подхода – мгновенное получение требуемого результата. Это бывает полезно при отладке программы, чтобы, например, быстренько посмотреть значения текущих данных и сделать для себя некоторые выводы.

Недостаток интерактивного режима – потеря ранее записанных команд. Например, если перезапустить **IDLE**, то все ранее написанное пропадает, не сохраняется. Поэтому писать полноценные программы в таком режиме не получится.

Основы Python

Для этого следует использовать другой, файловый режим исполнения команд. В оболочке **IDLE** это делается так. Выбираем меню **File**, затем, **New File** и у нас появляется второе окно. Здесь можно написать сразу несколько команд:

```
2+3  
8/3  
5*6
```

Сохранить их в файл и, затем, выполнить программу (**F5**). Результат должен отображаться в командном окне **IDLE**. Но мы ничего не видим. **Почему?** Дело в том, при файловом режиме команды выполняются строчка за строчкой, то есть, последовательно, сверху-вниз, но результаты автоматически не отображаются, как это происходит в интерактивном режиме. Можно подумать, что это плохо. **Возможно, мы хотели увидеть результат действия наших арифметических операций?** Однако, это разумное поведение интерпретатора. Часто целью программ является не вывод всех промежуточных результатов, а только требуемого конечного значения. **Но как его вывести?** Для этого в языке **Python** существует специальная функция **print()**. Если все наши арифметические операции записать в виде:

```
print(2+3)  
print(8/3)  
print(5*6)
```

то в консоли увидим три строчки, три вывода. В интерактивном режиме, кстати, она работает абсолютно также – выводит некоторое сообщение в консоль. На данный момент, просто запомните, то, что находится внутри круглых скобок функции **print()**, та информация и выводится. Причем, эту функцию можно записать и с пустыми скобками:

```
print()
```

Тогда просто отобразится пустая строка. Также можно просто указать число:

```
print(5)
```

Основы Python

```
print(5.6)
```

или строку:

```
print("hello")
```

Пока это просто демонстрация возможностей функции `print()`. Подробнее мы о ней еще поговорим. А до тех пор будем использовать в таком простом виде.

Форматирование текста программ

Давайте вернемся к нашей программе. Как я уже говорил, конструкции выполняются последовательно сверху вниз. Причем, все они должны иметь единый нулевой начальный отступ от левого края. Если добавить перед `print()` хотя бы один пробел, то возникнет ошибка форматирования текста программы. **Питон** очень требователен к формату записи текста программы. И первое, что здесь нужно запомнить, - все начальные конструкции языка должны записываться без отступов слева и каждый `print` следует начинать с новой строки. Это правила руководства **PEP 8** по рекомендациям оформления текста программы на **Python**:

<https://peps.python.org/pep-0008/>

Как видите, это достаточно объемный документ. Я буду стараться придерживаться его указаний и делать оформление программ в соответствии с ним, чтобы вы также приучались к правильному стилю оформления.

На данный момент достаточно запомнить, что все начальные конструкции языка записываются без отступов слева и каждая с новой строки. Конечно, для ясности и лучшей читаемости текста, можно добавлять пустые строки, например, так:

```
print(2+3)
```

```
print(8/3)
```

```
print(5*6)
```

Основы Python

Давайте еще раз подведем итог:

- программа выполняется последовательно сверху-вниз;
- текст программы следует оформлять в соответствии с **PEP 8**;
- начальные отступы слева у команд должны отсутствовать, а каждая команда записана с новой строки.

Интегрированная среда PyCharm

В заключение этого занятия я хочу познакомить вас с интегрированной средой программирования **PyCharm**, разработанной специально для написания программ на **Python**. Она намного удобнее **IDLE**, с которой вы уже немного знакомы, поэтому все дальнейшие действия я буду выполнять в **PyCharm**.

Я рекомендую вам также ее установить, хотя это не обязательное действие, так как все программы курса можно будет выполнять и в **IDLE**. Но, как говорится: лучше один день потерять, чтобы потом за пять минут долететь. Поэтому, лучше затратить немного времени на установку **PyCharm**, чтобы потом стало проще и приятнее писать программы на **Python**.

Вначале нам нужно перейти на официальный сайт программы:

<https://www.jetbrains.com/pycharm/>

и скачать бесплатную версию «**Community**». Для решения большинства задач ее будет вполне достаточно. Затем, скачиваем приложение для установки **PyCharm**. Запускаем его, нажимаем «**Next**», выбираем каталог размещения программы, отмечаем, что хотим создать ярлык на рабочем столе и привязать расширение **py** к данной среде, нажимаем «**Next**» и, затем, «**Install**».

После установки программа предложит запустить **PyCharm**. Запускаем. Появляется окно для импортирования настроек из предыдущей версии (если она была). Я этот шаг пропущу «**Do not import settings**». В следующем окне мы выбираем тему оформления. Я выберу темную. Вы можете выбрать другую – это дело вкуса. Далее, нажимаем на кнопку «**Skip Remaining and Set Defaults**»

Основы Python

пропустить все напоминания и сделать дальнейшие установки по умолчанию.

Здесь при первом запуске необходимо создать новый проект. Нажимаем **«Create New Project»**. В поле **«Location»** указывается расположение проекта и его имя. Пусть проект называется **«stepik»**. Раскрываем вкладку **«Project interpreter»** интерпретатор проекта, здесь укажем существующий интерпретатор. Если его в списке нет, то нажмите вот на это троеточие и в новом окне выберите **«System Interpreter»**. В этом списке отображаются все интерпретаторы, установленные на компьютере. Но я оставлю тот, что был найден по умолчанию. Нажимаем кнопку **«Create»** и создаем проект. Перед нами откроется окно **PyCharm**. Слева отображается структура проекта. Пока он не содержит ни одного файла с текстом программы. Создадим его. Нажимаем правую кнопку мыши, выбираем **«New»** -> **«Python File»**. Вводим имя файла, например, **ex1** и этот файл автоматически добавляется в наш проект. Здесь мы будем писать тексты программ, например, так:

```
print("Hello World!")
```

Далее, для запуска можно выбрать в меню **«Run»** -> **«Run ex1»**, программа начнет выполняться и внизу появится результат ее работы. Однако, удобнее пользоваться **«горячими клавишами»** для выполнения типовых команд. В частности для запуска проекта можно нажать комбинацию клавиш:

- при первом запуске: **Ctrl+Shift+F10**
- при повторных запусках: **Shift+F10**

Пока этого функционала нам будет вполне достаточно. По мере работы с **PyCharm**, вы будете знакомиться с его возможностями и совсем скоро, сами не заметите, как станете с ним одним целым.

На этом мы с вами завершим наше второе занятие. В качестве самостоятельного задания установите среду **PyCharm** и попробуйте выполнить простую программу из этого урока. До встречи на следующем занятии!

Основы Python

§3. Переменные, оператор присваивания, функции type и id

На данный момент, мы с вами установили сам язык и научились запускать программы в средах **IDLE** и **PyCharm**. Пришла пора сделать первый шаг непосредственно в программирование. Вначале, давайте попробуем понять, **что вообще могут делать компьютерные программы?** В самой основе, совсем немного:

- хранить данные;
- выполнять арифметические операции;
- проверять логические условия (операторы ветвления);
- реализовывать циклы.

Фактически, на комбинации этих блоков и выстраивается логика всех программ. Мы с вами последовательно познакомимся с каждым из них. Начнем с самого первого – способа представления и хранения данных в **Python**.

Переменные и оператор присваивания

Как же данные представляются в Python? Условно, это можно представить так. Есть некое хранилище (мы его будем называть объектом) и в нем могут располагаться или числа, или строки, или, какие-либо другие типы данных. Причем в объекте может быть что-то одно: или число, или строка. Одновременно и число и строка находиться в одном хранилище не могут.



Объект с числом



Объект со строкой

Основы Python

Таких хранилищ в программе может быть огромное количество. И **как нам тогда обратиться к нужному и взять оттуда данные?** Все просто. Для этого у нас должна быть ссылка на объект и мы обращаемся к хранилищу по имени этой ссылки. Такие ссылки называются **переменными**.



Как нам создать объект с некоторым содержимым и ссылкой на него? Тоже очень просто. Достаточно придумать имя переменной и через оператор присваивания связать ее с нужным объектом, например, так:

```
a = 7
```

В результате, интерпретатор языка **Python** создаст ссылку с именем **a** и объектом с целым числом **7**. Мы в этом можем легко убедиться, если выведем содержимое объекта по этой ссылке (по переменной **a**):

```
print(a)
```

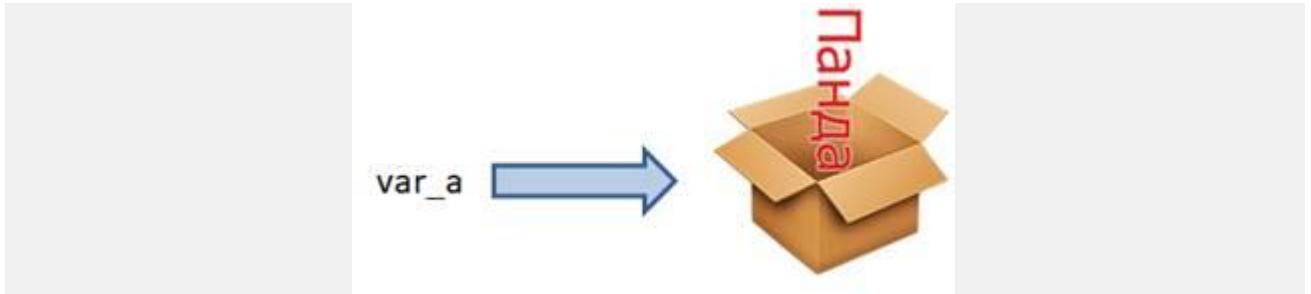
Еще раз обратите внимание на оператор присваивания. В программировании он связывает операнд слева с операндом справа:

операнд слева = операнд справа

Например, строчка:

```
var_a = "Панда"
```

Основы Python

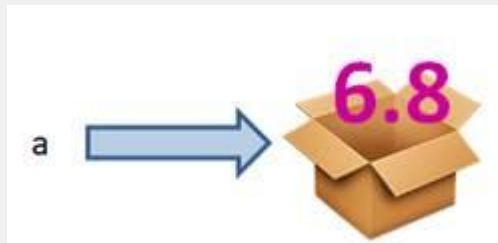


создает переменную с именем **var_a**, которая ссылается на объект со строкой «Панда». И, смотрите, здесь именно создается переменная **var_a**, так как до этого она не существовала. К несуществующим переменным мы обращаться не можем. Например, строчка:

```
print(x)
```

приведет к ошибке, так как переменную с именем **x** мы нигде не создавали. Но если переменная уже была создана и мы снова присваиваем ей какое-либо значение:

```
a = 6.8
```



то она второй раз уже не создается. Но **как будет работать эта строчка?** Здесь создается новый объект со значением **6.8** и на него инициализируется уже существующая переменная. Если на прежний объект **7** нет других ссылок, то он автоматически удаляется из памяти устройства. При этом переменная **a** начинает ссылаться на другой тип данных – вещественное число (до этого было целочисленное значение). То есть, тип переменной определяется в момент присваивания ей того или иного значения. В программировании это называется **динамической типизацией**. В противовес строгой типизации, когда тип переменной указывается в момент ее объявления. Например, так делается в языках **C++** или **Java**.

Основы Python

Но, вернемся к оператору присваивания. Давайте посмотрим, что будет, если связать одну переменную с другой:

```
b = a
```



Теперь обе переменные будут ссылаться на один и тот же объект. То есть, копирования данных (создание нового объекта) не происходит, копируется лишь ссылка на объект. И это важный момент, который хорошо следует запомнить и понимать! В языке **Python** переменные не хранят значения, а лишь ссылаются на них. Как раз, благодаря этому можно одной и той же переменной присваивать самые разные типы данных:

```
b = "Hello"  
b = 0  
b = -8.4
```

Это очень удобно при программировании. Но при этом всегда следует помнить, что переменная – это всего лишь ссылка на данные, а не сами данные.

Иногда, в программах можно встретить вариант **каскадного присваивания**:

```
a = b = c = 0
```



Основы Python

В результате выполнения такой команды, все три переменные будут ссылаться на один и тот же объект со значением 0.

Если же мы хотим, чтобы каждая переменная ссылалась на свой отдельный объект, то можно воспользоваться **множественным присваиванием**:

```
a, b = 1, 2
```



Здесь переменная **a** будет ссылаться на 1, а **b** – на 2. Используя такую команду, можно легко и просто выполнить операцию обмена значениями между двумя переменными:

```
a, b = b, a
```

Функция `type()`

Так как в программе переменные могут иметь самые разные типы данных, то **как можно узнать текущий тип, на который они ссылаются?** Для этого в **Python** имеется специальная встроенная функция **`type()`**, возвращающая тип данных, связанный с указанной переменной:

```
print(type(a))
```

В консоли увидим целочисленный тип данных (**`int`**). Давайте объявим еще две переменные:

```
x = 5.8  
s = "Hello"
```

И выведем для них типы:

Основы Python

```
print(type(x), type(s))
```

Увидим `float` и `str`. Здесь у вас может возникнуть вопрос, а **какие типы данных вообще существуют в Python?** Конечно, я бы мог здесь привести список всех типов, но на данном этапе – это избыточная, справочная информация. По мере прохождения курса, мы с вами познакомимся со всеми встроенными типами и постепенно у вас сложится общее представление. Это будет гораздо эффективнее, чем отображение больших и умных таблиц, которые, все равно, не воспринимаются и быстро забываются.

Правильные имена переменных

Последнее, о чем я хочу вам рассказать на этом занятии – как правильно выбирать имена переменных. Есть несколько простых правил:

1. Имена следует брать существительными (отвечают на вопросы: кто, что).
2. Имена должны быть осмысленными и отражать суть данных.
3. Допустимые символы в именах: первый символ – любая буква латинского алфавита **a-z, A-Z** и символ подчеркивания `_`. В качестве второго и последующих символов еще цифры **0-9**.

Например:

```
msg = "Сообщение"  
count = 0
```

Причем, обратите внимание, переменные:

```
arg = 0  
Arg = 0
```

Это две разные переменные, так как малая буква 'a' и большая 'A' – разные символы, а значит, имена тоже разные. Также нельзя использовать ключевые слова языка **Python** в качестве имен, например, писать:

Основы Python

```
True = 5
```

Полный их список можно посмотреть с помощью вызова в консоли функции:

```
help()
```

а, затем, набрать:

```
keywords
```

Также не следует использовать имена стандартных функций в качестве переменных. Например, если переопределить имя функции:

```
print = 5
```

то оно теперь будет ссылаться не на функцию для печати значений, а на числовой объект **5**. И если, затем, попытаться вызвать функцию:

```
print(6)
```

то возникнет ошибка, так как мы, фактически, пытаемся выполнить объект, содержащий число **5**. Но **Python** делать этого не умеет (по крайней мере, по умолчанию). Поэтому имена функций стандартной библиотеки языка **Python** не следует использовать в качестве имен переменных.

Но **как нам узнать, является ли какое-либо имя встроенной функцией?** Очень просто. Все зарезервированные имена, будь то функции или ключевые слова, автоматически подсвечиваются интегрированными средами и так подсказывают нам, что это имя уже имеет свою функциональность и его лучше не трогать. Со временем, вы запомните все наиболее употребительные имена языка **Python** и будете знать, какие имена лучше не использовать для собственных переменных.

На этом мы завершим наше очередное занятие. Надеюсь, вы теперь понимаете, что из себя представляют переменные в языке **Python**, как они связываются с данными и как работает оператор присваивания. Для

Основы Python

закрепления этого материала обязательно выполните практические задания и переходите к следующему уроку!

§4. Числовые типы, арифметические операции

На этом занятии мы поподробнее поговорим о представлении чисел и арифметических операциях над ними.

В **Питоне** имеются три базовых типа для представления чисел:

- **int** – для целочисленных значений;
- **float** – для вещественных;
- **complex** – для комплексных.

Мы затронем первые два: **int** и **float**. Первый целочисленный тип представляет собой, следующие числа:

0, 1, 2, 100, 6697959484, -1, -2, -7567658

Python поддерживает работу с очень большими числами, поэтому у вас, скорее всего не возникнет проблем с выходом за пределы диапазона.

Вещественные числа, то есть, дробные записываются через точку, например, так:

6.8, -5.567, 345.546, -65467.99

Здесь также довольно широкий диапазон значений, достаточный для большинства практических задач.

Основные арифметические операции

Пока такого понимания чисел будет вполне достаточно. Следующим шагом, нам с вами нужно научиться делать арифметические операции над ними. **Что это за операции?** Базовыми из них являются, следующие:

Оператор	Описание	Приоритет
----------	----------	-----------

Основы Python

+	сложение	2
-	вычитание	2
*	умножение	3
/, //	деление	3
%	остаток деления	3
**	возведение в степень	4

Давайте, я поясню их работу на конкретных примерах. Перейдем в консоль языка **Python**, чтобы выполнять команды в интерактивном режиме. Так будет удобнее для демонстрации возможностей вычислений. В самом простом варианте мы можем просто сложить два целых числа:

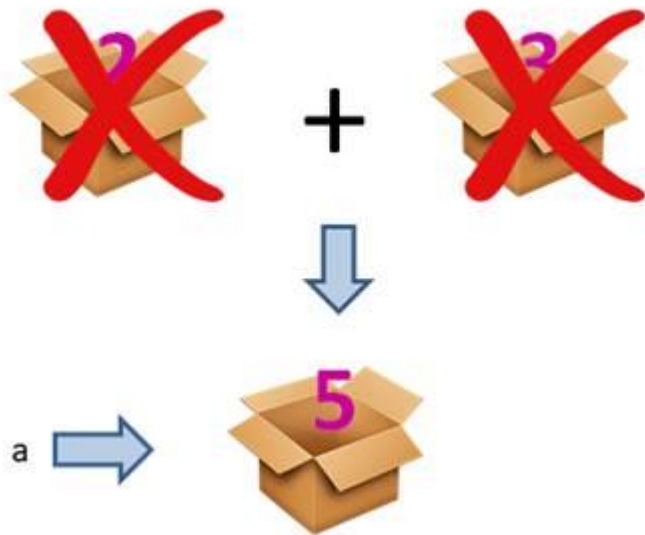
```
2+3
```

Получим результат **5**. Но этот результат у нас нигде не сохраняется. Чтобы иметь возможность делать какие-либо действия с пятеркой, ее следует сохранить через переменную, например, вот так:

```
a = 2+3
```

Теперь **a** ссылается на объект с числом **5**. Давайте разберемся, как работает эта строчка. Сначала в **Python** создаются два объекта со значениями **2** и **3**. Оператор сложения берет эти значения, складывает их и формирует третий объект со значением **5**. А, затем, через оператор присваивания, этот объект связывается с переменной **a**. В конце, если на объекты **2** и **3** не ссылаются никакие другие переменные, они автоматически удаляются из памяти сборщиком мусора.

Основы Python



Возможно, вас удивило, что при такой простой операции сложения двух чисел выполняется столько шагов. Но в **Python** реализовано все именно так. И это справедливо для всех арифметических операций. Мало того, раз операция сложения возвращает объект с результатом, то можно сделать и такое сложение из трех чисел:

```
b = 2+3+4
```

И так далее, можно записать сколько угодно операций сложения в цепочку.

Давайте теперь сложим целое число с вещественным:

```
c = 2 + 3.5
```

Очевидно, что результат получается тоже вещественным. Отсюда можно сделать вывод, что сложение целого числа с вещественным всегда дает вещественное значение.

А вот при делении двух любых чисел, мы всегда будем получать вещественное число (даже если числа можно разделить нацело):

```
d1 = 8 / 2  
d2 = 3 / 6
```

Основы Python

Если же нам нужно выполнить деление с округлением к наименьшему целому, то это делается через оператор:

```
d3 = 7 // 2
```

На выходе получаем значение **3**, так как оно является наименьшим целым по отношению к **3,5**. Обратите внимание, что при делении отрицательных чисел:

```
d3 = -7 // 2
```

получим уже значение **-4**, так как оно наименьшее по отношению к **-3,5**. Вот этот момент следует иметь в виду, применяя данный оператор деления.

Следующий оператор умножения работает очевидным образом:

```
5 * 6  
2 * 4.5
```

Обратите внимание, в последней операции получим вещественное значение **9.0**, а не целое **9**, так как при умножении целого на вещественное получается вещественное число.

Давайте теперь предположим, что мы хотим вычислить целый остаток от деления. **Что это вообще такое?** Например, если делить

10 : 3

то остаток будет равен **1**. Почему так? Все просто, число **3** трижды входит в число **10** и остается значение **10 - 3·3 = 1**. Для вычисления этого значения в **Python** используется оператор:

```
10 % 3
```

Если взять:

```
10 % 4
```

Основы Python

то получим 2. Я думаю, общий принцип понятен. Здесь есть только один нюанс, при использовании отрицательных чисел. Давайте рассмотрим четыре возможные ситуации:

```
9 % 5 # значение 4
-9 % 5 # значение 1
9 % -5 # значение -1
-9 % -5 # значение -4
```

Почему получаются такие значения? Первое, я думаю, понятно. Здесь 5 один раз входит в 9 и остается еще 4. При вычислении $-9 \% 5$ по правилам математики следует взять наименьшее целое, делящееся на 5. Здесь – это значение -10 . А, далее, как и прежде, вычисляем разность между наименьшим, кратным 5 и -9 :

$$-9 - (-10) = 1$$

При вычислении $9 \% -5$, когда делитель отрицательное число, следует выбирать наибольшее целое, кратное 5. Это значение 10. А, далее, также вычисляется разность:

$$9 - 10 = -1$$

В последнем варианте $-9 \% -5$ следует снова выбирать наибольшее целое (так как делитель отрицателен), получаем -5 , а затем, вычислить разность:

$$-9 - (-5) = -4$$

Как видите, в целом, все просто, только нужно запомнить и знать эти правила. Кстати, они вам в дальнейшем пригодятся на курсе математики.

Последняя арифметическая операция – это возведение в степень. Она работает просто:

```
2 ** 3 # возведение в куб
36 ** 0.5 # 36 в степени 1/2 (корень квадратный)
```


Основы Python

```
2 ** 3 ** 2 # 2^3^2 = 512
```

В последней строчке сначала **3** возводится в квадрат (получаем **9**), а затем, **2** возводится в степень **9**, получаем **512**. То есть, оператор возведения в степень выполняется справа-налево. Тогда как все остальные арифметические операции – слева-направо.

Приоритеты арифметических операций

Давайте теперь посмотрим, что будет, если выполнить команду:

```
27 ** 1/3
```

Получим значение **9**. **Почему так произошло?** Ведь кубический корень из **27** – это **3**, а не **9**? Все дело в приоритете арифметических операций (проще говоря, в последовательности их выполнения). Приоритет у оператора возведения в степень ****** - наибольший. Поэтому здесь сначала **27** возводится в степень **1**, а затем, **27** делится на **3**. Получаем искомое значение **9**.

Если нам нужно изменить порядок вычисления, то есть, приоритеты, то следует использовать круглые скобки:

```
27 ** (1/3)
```

Теперь видим значение **3**. То есть, по правилам математики, сначала производятся вычисления в круглых скобках, а затем, все остальное в порядке приоритетов.

Приведу еще один пример, чтобы все было понятно:

```
2 + 3 * 5 # 17  
(2 + 3) * 5 # 25
```

То есть, приоритеты работают так, как нас учили на школьных уроках математики. Я думаю, здесь все должно быть понятно. Также не забывайте, что все арифметические операторы выполняются слева-направо (кроме оператора возведения в степень), поэтому в строчке:

Основы Python

```
32 / 4 * 2
```

сначала будет выполнено деление на 4, а затем, результат умножается на 2.

Дополнительные арифметические операторы

В заключение этого занятия рассмотрим некоторые дополнения к арифметическим операторам. Предположим, что у нас имеются переменные:

```
i = 5  
j = 3
```

И, далее, мы хотим переменную `i` увеличить на 1, а `j` – уменьшить на 2. Используя существующие знания, это можно сделать, следующим образом:

```
i = i + 1  
j = j - 2  
print(i, j)
```

Но можно проще, используя операторы:

```
i += 1  
j -= 2
```

Результат будет прежним, но запись короче. Часто, в таких ситуациях на практике используют именно такие сокращенные операторы.

То же самое можно делать и с умножением, делением:

```
i *= 3  
j /= 4  
print(i, j)
```

и другими арифметическими операторами.

Основы Python

На этом мы завершим наше занятие. Надеюсь, вам были понятны действия с арифметическими операторами. Как всегда, не забывайте проходить практику для закрепления материала и жду вас всех на следующем уроке.

§5. Математические функции и работа с модулем `math`

На этом занятии речь пойдет о наиболее употребительных математических функциях.

Первая встроенная функция `abs()` позволяет вычислять модуль чисел (из отрицательных делает положительные):

```
abs(-5)
```

То есть, для вызова функции нужно записать ее имя и в круглых скобках указать аргумент. Чтобы сохранить модуль того или иного числа, результат следует присвоить переменной:

```
a = abs(-5.6)
```

Если же передать положительное число, то оно просто возвращается данной функцией:

```
abs(1.5)
```

Следующая функция `min()` выбирает минимальное значение среди переданных ей чисел:

```
min(1, 2, 3, 0, -5, 10)
```

А, противоположная ей функция `max()` – ищет максимальное значение:

```
max(1, 2, 3, 0, -5, 10)
```

Число аргументов у этих функций может быть произвольным, но не менее одного. То есть, запись вида:

Основы Python

```
max()
```

приведет к ошибке, т.к. не указан ни один аргумент.

Следующая функция **pow()** возводит числа в указанную степень:

```
pow(6, 2)
```

Это аналог оператора:

```
6 ** 2
```

Или, с дробными значениями:

```
pow(27, 0.5)
```

```
pow(27, 1/3)
```

Последняя встроенная функция, которую мы рассмотрим – это **round()** для округления чисел:

```
round(0.5)
```

```
round(0.51)
```

У этой функции имеется второй необязательный параметр, указывающий точность округления. Если, например, записать:

```
round(7.8756, 2)
```

то число округляется с точностью до сотых (два знака после запятой). Если же указать отрицательное значение:

```
round(7.8756, -1)
```

то округление происходит до десятков. Или округление до сотен и тысяч:

```
round(78756, -2)
```

```
round(78756, -3)
```

Основы Python

Следующая возможность при работе с функциями – это вызов одной из другой. Например, запись вида:

```
max(1, 2, abs(-3), -10)
```

вернет значение **3**, так как вначале вызываются функции в аргументах, а затем, сама функция **max**. Вложенность можно делать любой глубины, например:

```
max(1, 2, abs(min(10, 5, -3)), -10)
```

Модуль math

Некоторые из вас могут заметить, что я привел достаточно ограниченный набор математических функций. Конечно, в **Python** есть и другие. Чтобы ими воспользоваться, нужно импортировать специальный модуль **math**:

```
import math
```

Пока просто запомните эту команду. Об импорте мы еще будем подробнее говорить. Итак, после ее выполнения у нас появляется дополнительный набор общеупотребительных функций, которые можно просмотреть так:

math.

Я отмечу наиболее используемые, а остальные применяются по аналогии.

Для округления до наибольшего целого:

```
math.ceil(5.2)  
math.ceil(-5.2)
```

Для наименьшего целого:

```
math.floor(5.99)  
math.floor(-3.3)
```

Основы Python

Факториал числа:

```
math.factorial(6)
```

Отбрасывание дробной части:

```
math.trunc(5.8)
```

Это аналог встроенной в Python функции:

```
int(5.8)
```

Далее, логарифм по основанию 2, 10 и произвольный:

```
math.log2(4)
math.log10(100)
math.log(2.7)
math.log(27, 3) # по основанию 3
```

Вычисление квадратного корня:

```
math.sqrt(49)
```

Также имеется полный набор тригонометрических функций, например:

```
math.sin(3.14/2)
math.cos(0)
```

Помимо этого есть, следующие константы:

```
math.pi
math.e
```

Этих представленных функций нам пока будет вполне достаточно. Для закрепления материала выполните практические задания и жду вас на следующем уроке.

Основы Python

§6. Функции print() и input(). Преобразование строк в числа int() и float()

На этом занятии речь пойдет о двух распространенных функциях:

- **print()** – вывод данных в консоль;
- **input()** – ввод данных из стандартного входного потока (часто клавиатуры).

Функция print()

О функции **print()** мы с вами уже немного говорили, и вы видели ее использование в самых простых ситуациях, например:

```
print(1)
```

или

```
print(3, 5, 7)
```

выведет указанные значения через пробел. Также можно выводить отдельные переменные:

```
a = -6.84  
print(a)
```

результаты арифметических операций:

```
print(a * 2 + 3)
```

или функций:

```
print(abs(a * 2 + 3))
```

Но у этой функции имеется два необязательных именованных параметра, которые довольно часто используются в практике программирования:

- **sep** – разделитель между данными;

Основы Python

- **end** – завершающий символ или строка.

Давайте я покажу на примере работу этих параметров. Предположим, мы выводим три переменные:

```
a = -6.84
b = 7
c = 25.6
print(a, b, c)
```

Как видите, между ними автоматически добавляется пробел. Но этот символ можно поменять через параметр **sep**, например, так:

```
print(a, b, c, sep=" | ")
```

Соответственно, чтобы параметр **sep** «сработал», необходимо хотя бы два аргумента в функции **print()**. Если указать, например, один:

```
print(a, sep=" | ")
```

то здесь разделять нечего и он будет проигнорирован. А вот для двух переменных уже появится вертикальная черта:

```
print(a, b, sep=" | ")
```

Второй параметр **end** задает окончание строки вывода и по умолчанию:

end = '\n'

переводу на следующую строку. Это такой спецсимвол, о которых мы также еще будем говорить. Так вот, благодаря такому параметру **end** два последовательных вызова функции **print()** напечатают текст с новой строки:

```
print("Hello")
print("World!")
```

Но если в первой функции **print()** добавить этот параметр с пробелом:

Основы Python

```
print("Hello", end=' ')\nprint("World!")
```

то увидим в одной строчке оба слова. Разумеется, здесь параметр **end** с пробелом применяется только к первому **print()**. У второго он уже берется по умолчанию с переносом строки.

Последнее, что я хочу рассказать о функции **print()** – это способ вывода форматированной информации в консоль. Давайте предположим, что у нас имеются две переменные (координаты точки) **x** и **y**:

```
x = 5.76\ny = -8
```

И мы хотим вывести их в формате:

«Координаты точки: x = 5.76; y = -8»

Сделать это можно несколькими способами. Первый, самый очевидный, записать все через запятую:

```
print("Координаты точки: x = ", x, "; y = ", y)
```

Но, начиная с версии **Python 3.11.5**, появилась возможность использовать специальные **F-строки**. О них мы также еще будем говорить, но здесь я приведу простой пример и вы уже сейчас сможете применять этот механизм в своих программах. Запишем функцию **print()**, следующим образом:

```
print(f"Координаты точки: x = {x}; y = {y}")
```

Смотрите, здесь перед строкой ставится специальный символ **f**, указывающий, что это будет **F-строка**. А, в самой строчке внутри фигурных скобок мы можем записывать любые конструкции языка **Python**. В данном случае, я просто указал переменные **x** и **y**. Видите, как это просто, наглядно и удобно. Сейчас практически всегда используются **F-строки** для форматированного вывода информации.

Основы Python

Функция `input()`

Вторая функция `input()` служит для ввода информации, как правило, с клавиатуры. В самом простом варианте ее можно вызвать так:

```
a = input()
```

При этом переменная `a` будет ссылаться на строку:

```
print(type(a))
```

И это важный момент: функция `input()` всегда возвращает строку. **На что это может повлиять?** Например, мы хотим вычислить модуль введенного числа:

```
a = input()
b = abs(a)
```

При вызове функции `abs()` возникнет ошибка, так как в качестве ее аргумента должно быть число, а не строка. **Как решить эту проблему?** Очень просто. Если мы знаем, что пользователь должен ввести число, предположим, целое число, то можно воспользоваться функцией:

```
a = "54"
b = int(a)
```

Теперь `b` будет ссылаться на число `54`, а не строку. И наша программа примет вид:

```
a = input()
a = int(a)
b = abs(a)
print(b)
```

Здесь первые две строчки можно объединить и записать их так:

```
a = int(input())
```

Основы Python

Но функция `int()` преобразовывает только целые числа. Если в строке будет хотя бы один не цифровой символ:

```
int("64.56")
```

возникнет ошибка. То есть, в нашей программе пользователь обязательно должен вводить целые числа. А как тогда преобразовывать вещественные значения? Для этого есть другая функция:

```
float("64.56")
```

Поэтому, когда на входе ожидаются вещественные данные, то следует использовать именно ее:

```
a = float(input())  
b = abs(a)  
print(b)
```

Давайте напишем программу для вычисления периметра прямоугольника. Пользователь будет вводить два числа (стороны прямоугольника), а мы, затем, вычислим периметр:

```
a = float(input())  
b = float(input())  
print("Периметр:", 2 * (a + b))
```

Как видите, все достаточно просто. Мы вводим первое число, нажимаем **Enter** и вводим второе число. Вводить два числа через пробел здесь нельзя, иначе получится строка с двумя числами, разделенные пробелом и ее нельзя будет преобразовать функциями `int()` и `float()`.

Также в этой программе пользователю совершенно непонятно, что нужно вводить. Давайте добавим подсказки. Для этого в функции `input()` первым аргументом передается строка:

```
a = float(input("Введите длину прямоугольника: "))
```

Основы Python

```
b = float(input("Введите ширину прямоугольника: "))  
print("Периметр:", 2 * (a + b))
```

Теперь стало гораздо понятнее, что нужно вводить.

Забегая вперед отмечу, что в **Python** можно выстроить конструкцию для ввода значений через пробел. Это делается так:

```
a, b = map(float, input("Введите две стороны прямоугольника: ").split())
```

Здесь к каждому введенному значению применяется функция **float()** для преобразования в вещественные числа. Пока просто запомните эту конструкцию, в дальнейшем она станет понятной, когда мы изучим строки и функцию **map()**.

Или, для ввода целых чисел, она будет принимать вид:

```
a, b, c = map(int, input("Введите три целых числа: ").split())  
print("Периметр треугольника:", a + b + c)
```

На этом мы завершим с вами очередное занятие. Я надеюсь, вы теперь хорошо понимаете, как использовать функции **print()** и **input()**. Как всегда, закрепите этот материал практическими занятиями и переходите к следующему уроку.

§7. Логический тип bool. Операторы сравнения и операторы and, or, not

На этом занятии вы увидите, как в программах можно делать логические выводы на уровне:

- **True** – истина;
- **False** – ложь.

Предположим, мы бы хотели узнать:

```
4 > 2
```

Основы Python

Для этого достаточно записать такое сравнение и Python возвратит результат: либо **True**, либо **False**. В данном случае получаем **True** (истина), так как 4 действительно больше 2. Если же записать:

```
4 > 7
```

то получим **False**, так как 4 меньше 7. То же самое можно делать и с переменными:

```
a = 5  
b = 7.8  
a <= b
```

Мало того, полученный результат, при необходимости, можно сохранить в переменной, например, так:

```
res = a > b
```

получим значение **False** в переменной **res**.

Давайте теперь посмотрим на тип этой переменной:

```
type(res)
```

да, это новый тип **bool** (булевый тип), с которым мы только что познакомились и он может принимать только два значения: **True** или **False**.

Мы можем присвоить эти значения напрямую переменной, например, так:

```
res = False
```

И здесь, естественно, возникает вопрос, **какие операторы сравнения вообще существуют в Python?** Они, следующие:

<	сравнение на меньше
>	сравнение на больше
<=	сравнение меньше или равно

Основы Python

>=	сравнение больше или равно
==	сравнение на равенство
!=	сравнение на неравенство

Приведу несколько примеров их использования:

```
5 == 7-2
```

Здесь справа записана арифметическая операция. Так тоже можно делать. Вообще, в качестве операндов могут быть любые конструкции языка **Python**: переменные, вычисления, функции и т.п.

Следующий пример, сравнения:

```
8 >= 8
8 > 8
```

В первом случае получаем **True**, а во втором – **False**, так как **8** не больше **8**. Также имейте в виду, что **оператор >= записывать наоборот => нельзя**. Помимо этого знаки:

>=, !=, <=, ==

всегда записываются без пробелов. Если поставить пробел, будет ошибка.

Также частой, нет, очень частой ошибкой начинающих программистов – это запись вместо оператора сравнения **==** оператора присваивания **=**. Сравните две строчки:

```
x = 6
x == 6
```

В первом случае мы присваиваем переменной **x** число **6**, а во втором случае сравниваем переменную **x** со значением **6**. И эти операторы путать нельзя, хотя они и похожи друг на друга.

Основы Python

Еще несколько полезных примеров сравнения. Предположим, мы хотим проверить переменную `x` на четность. Я напомним, что четные числа – это, например, числа:

2, 4, 6, -8, -10

то есть, числа, кратные 2. **Как выполнить такую проверку?** Обычно, это делают, следующим образом:

```
x % 2 == 0
```

Помните **оператор вычисления остатка от деления**? Он здесь будет возвращать 0, если двойка укладывается в `x` без остатка. Это и есть проверка на четность.

Обратную проверку на нечетность, можно записать так:

```
x % 2 != 0
```

По аналогии мы можем проверить, кратна ли переменная какому-либо значению, например, **трем**:

```
a % 3 == 0
```

и так можно делать проверки с любыми числами.

Операторы `not`, `and`, `or` и их приоритеты

Давайте теперь поставим более сложную задачу и определим, **попадает ли число (значение переменной) в диапазон [-2; 5]**? **Какие логические умозаключения здесь нужно провести, чтобы ответить на этот вопрос?** Для простоты представим, что у нас имеется переменная:

```
y = 1.85
```

Очевидно, чтобы она попадала в диапазон [-2; 5], нужно соблюдение двух условий:

Основы Python

```
y >= -2
```

и

```
y <= 5
```

В **Python** записать это можно, следующим образом:

```
y >= -2 and y <= 5
```

Здесь общее условие будет истинно, если истинно каждое из подусловий.

Противоположный пример, вывод о непопадании в этот же диапазон. Его можно сделать проверками, что:

```
y < -2
```

или

```
y > 5
```

В **Python** это записывается так:

```
y < -2 or y > 5
```

Оператор **or** выдает истину, если истинно, хотя бы одно из подусловий.

Понимаете разницу между **and и **or**?** В **and** общее условие истинно, если истинны оба подусловия, а в **or** – хотя бы одно.

Также, если мы используем оператор **and**, то условие:

```
y >= -2 and y <= 5
```

можно записать в краткой форме:

```
-2 <= y <= 5
```


Основы Python

Это будет одно и то же. Какой вариант записи использовать, решать только вам, что удобнее, то и пишете в своих программах.

Приведу еще один пример. Допустим, мы хотим проверить **x** на кратность **2** или **3**. Очевидно, это можно записать так:

```
x % 2 == 0 or x % 3 == 0
```

Также можно инвертировать это условие, то есть, определить, что **x** не кратна **2** и не кратна **3**. Это можно сделать двумя способами:

```
x % 2 != 0 and x % 3 != 0
```

Или, используя оператор **not** (не):

```
not (x % 2 == 0 or x % 3 == 0)
```

Обратите внимание на круглые скобки. У оператора **not** самый высокий приоритет, то есть, он выполняется в первую очередь (как умножение и деление в математике). Поэтому, для инвертирования всего составного условия его необходимо поместить в круглые скобки. Если бы мы записали все в виде:

```
not x % 2 == 0 or x % 3 == 0
```

Это было бы эквивалентно условию:

```
x % 2 != 0 or x % 3 == 0
```

то есть, мы инвертировали бы только первое выражение. Поэтому, при определении составных условий очень важно знать приоритеты операторов:

or	1
and	2
not	3

Основы Python

Приведение к типу bool

В заключение этого занятия я хочу отметить встроенную функцию

`bool()`

которая выполняет преобразование входного аргумента в логический тип: **True** или **False**.

Общее правило здесь такое. Пустое – это **False** (ложь), а не пустое – **True** (истина). **Что значит пустое и не пустое?** Покажу это на примерах. Любое не нулевое число – **не пустое**, то есть, **истина**:

```
bool(1)
bool(-10)
```

А вот **0** – пустое число, дающее **False**:

```
bool(0)
```

Похожим образом и со строками. Пустая строка:

```
bool("") # False
```

а любая не пустая:

```
bool("a") # True
bool(" ") # True
```

Аналогично дело обстоит и с другими типами данных, с которыми мы с вами еще только будем знакомиться.

Итак, из этого занятия вы должны знать операторы сравнения, работу операторов **or**, **and** и **not**, а также уметь пользоваться функцией **bool()**. Для закрепления этого материала вас ждут практические задания, а я вас буду ждать на следующем занятии.

Основы Python

§8. Введение в строки. Базовые операции над строками

Сегодня мы с вами познакомимся с еще одним типом данных – **строками**.

Строки в **Python** задаются очень просто: или в двойных кавычках:

```
s1 = "Панда"
```

или в одинарных (апострофах):

```
s2 = 'Panda'
```

Всегда, когда задаются строки, не забывайте про кавычки, если их не поставить:

```
s2 = Panda
```

то **Panda** будет восприниматься как переменная и возникнет ошибка.

В **Python** есть еще один способ определения многострочных строк. Для этого используются тройные кавычки (**одинарные** или **двойные, неважно**) и в них прописывается текст, например, так:

```
text = """Я Python бы выучил только за то,  
что есть популярные курсы.  
Много хороших курсов!"""
```

Если отобразить содержимое этой строки в консоли Питона, то увидим специальный символ `\n`:

```
'Я Python бы выучил только за то,\nчто есть популярные курсы.\nМного хороших курсов!'
```

Это один символ, отвечающий в тексте за перенос на новую строку и когда функция **print()** встречает его, то осуществляет такой переход:

```
print(text)
```

Основы Python

То есть, просто запомните, что для перехода на новую строку используется спецсимвол, который записывается в виде `'\n'`. Если записать просто `'n'` – это будет символ латинской буквы **n**, а при добавлении слеша – превращается в символ переноса строки.

Далее, строка может вообще не содержать ни одного символа:

```
a = ""
```

Получаем пустую строку. Но если добавить хотя бы один символ, даже если это будет пробел:

```
a = " "
```

то имеем уже не пустую строку, в данном случае содержащей символ пробела.

Базовые операции над строками

Давайте посмотрим, какие базовые операции можно выполнять со строками в **Python**. Например, мы хотим соединить две строки между собой:

```
s1 = "Я люблю"  
s2 = "язык Python"
```

Это можно сделать с помощью оператора `+`, который в случае со строками выполняет их объединение (**конкатенацию**):

```
s3 = s1 + s2  
print(s3)
```

Но мы бы хотели добавить пробел между словами. Сделаем это так:

```
s3 = s1 + " " + s2
```

С помощью первого оператора `+` добавляем **пробел** к первой строке **s1**, а затем, вторым оператором `+` добавляем вторую строку **s2**.

Основы Python

Но при использовании оператора конкатенации следует быть осторожным – он объединяет строки между собой. Например, команда:

```
s3 = s1 + 5
```

приведет к ошибке, так как операнд справа является числом, а не строкой. Если нам все же необходимо соединить строку с числом, то предварительно число нужно преобразовать в строку. Сделать это можно с помощью специальной функции **str()**:

```
s3 = s1 + str(5)
```

Функция **str()** выполняет преобразование в строки разные типы данных, не только числа, например, можно указать булево значение:

```
str(True)
```

а также другие типы данных, о которых мы еще с вами будем говорить.

Следующий оператор *****, применительно к строкам, выполняет их дублирование, указанное число раз:

```
"ха " * 5
```

Причем, здесь мы должны указывать именно целое число, для вещественных получим ошибку:

```
"ха " * 5.5
```

И это понятно, так как продублировать строку 5,5 раз нельзя.

Следующая функция **len()** возвращает длину строки (число символов в строке):

```
a = "hello"  
len(a)
```

Основы Python

Для пустой строки получим значение 0:

```
len("")
```

И, как видите, этой функции можно передавать или переменную на строку, или непосредственно записывать строки:

```
len("Python")
```

Следующий оператор **in** позволяет проверять наличие подстроки в строке, например:

```
'ab' in "abracadabra"  
'abc' in "abracadabra"
```

Следующая важная группа операторов – **сравнения строк**. В самом простом случае, строки можно сравнивать на равенство:

```
a == "hello"
```

Но сравнение:

```
a == "Hello"
```

вернет **False**, так как большая буква **H** и малая **h** – это два разных символа. Для сравнения на неравенство используем оператор не равно:

```
a != "hello"  
a != "hello "
```

Также смотрите строка **"hello"** (без пробела) и строка **"hello "** (с пробелом) – это две разные строки и они не равны между собой.

Наконец, строки можно сравнивать на больше и меньше, например, **кот** больше, чем **кит** с точки зрения строк:

```
'кот' > 'кит'
```

Основы Python

Почему так? Все просто. Здесь используется лексикографический порядок сравнения. Сначала берутся первые символы (**они равны**), затем переходим ко вторым символам. По алфавиту сначала идет символ 'и', а потом – символ 'о', поэтому 'о' больше, чем 'и'. Как только встретились не совпадающие символы, сравнение завершается и последующие символы строк игнорируются.

Если взять равные строки:

```
'кот' > 'кот'
```

то получим **False**, так как ни один символ не больше соответствующего другого из второй строки. Но, добавив пробел в первую строку:

```
'кот ' > 'кот'
```

получим значение **True**, так как при всех прочих равных условиях больше считается более длинная строка. Наконец, если у первой строки первую букву сделать заглавной:

```
'Кот ' > 'кот'
```

то получим **False**. **Почему?** Дело в том, что каждый символ в компьютере связан с определенным числом – **кодом**, в соответствии с кодовой таблицей. Например, в таблице **ASCII** мы видим, что сначала идут символы **заглавных** букв, а затем – **прописных**. Поэтому коды больших букв меньше соответствующих кодов малых букв.

Конечно, в **Python** используется немного другая кодировка **UTF-8**, но в ней этот принцип сохраняется. Мы можем легко посмотреть код любого символа с помощью функции **ord()**:

```
ord('K')  
ord('к')
```

И, как видите, для буквы 'К' код меньше, чем для 'к'.

Основы Python

Итак, из этого занятия вам нужно запомнить, как задавать обычные и многострочные строки. Что из себя представляет символ переноса строки. Знать базовые операции со строками:

- **+** (конкатенация) – соединение строк;
- ***** (дублирование) – размножение строкового фрагмента;
- **str()** – функция для преобразования аргумента в строковое представление;
- **len()** – вычисление длины строки;
- **in** – оператор для проверки вхождения подстроки в строку;
- операторы сравнения: **==** **!=** **>** **<**
- **ord()** – определение кода символа.

Для закрепления этого материала вас ждут практические задания, а я буду ждать на следующем уроке.

§9. Знакомство с индексами и срезами строк

Как вы уже знаете, в **Python** строку можно задавать, например, так:

```
s = "hello python"
```

индексы:	0	1	2	3	4	5	6	7	8	9	10	11
s =	h	e	l	l	o		p	y	t	h	o	n
	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

и она представляет собой упорядоченный набор символов. **Что значит упорядоченный?** Смотрите, каждый символ строки имеет свой уникальный, порядковый номер. Эти номера называются индексами. Первый – **0**, второй – **1** и так до конца. Мы можем использовать эти индексы для обращения к отдельному символу строки. Для этого, записывается имя переменной и в квадратных скобках указывается номер символа:

```
s[0]
```


Основы Python

```
s[1]
```

Но будьте внимательны, если указать несуществующий индекс:

```
s[12]
```

то получим ошибку выхода за пределы диапазона. В нашем случае последний индекс равен **11**, а мы записали **12**. Чтобы взять последний символ нам следовало бы записать:

```
s[11]
```

То есть, последний индекс равен длине строки минус **один**:

```
len(s) - 1
```

И, по идее, мы могли бы его так и указать в квадратных скобках:

```
s[len(s) - 1]
```

Эта конструкция работает для строк произвольной длины (кроме нулевой, когда символов нет). Но в **Python** то же самое можно записать проще, используя отрицательные индексы:

```
s[-1]
```

То есть, длину строки **len(s)** можно опускать и записывать отрицательные значения. Получаем способ индексации с конца строки.

Причем индексацию можно выполнять также и непосредственно над строками, например:

```
"panda"[3]
```

вернет **4**-й символ этой строки. Иногда это тоже может быть полезно.

Основы Python

Срезы

Фактически, когда мы выполняем индексацию, то возвращается новая строка из одного выделенного символа. Но мы можем выделять и сразу несколько символов, используя следующий синтаксис:

`строка[start:stop)`

Выделенная последовательность символов называется срезом строки.

Например, можно сделать так:

```
s[1:3]
```

вернет два символа с индексами 1 и 2. Последнее значение 3 не включается в срез. В срезах можно не указывать последнее значение:

```
s[4:]
```

или первое:

```
s[:5]
```

или оба:

```
s[:]
```

В последнем случае возвращается та же самая строка. Убедиться в этом можно, следующим образом:

```
a = s[:]
id(a)
id(s)
```

В срезах также можно использовать отрицательные индексы, например:

```
s[2:-2]
```

Основы Python

Но если записать:

```
s[-2:2]
```

то получим пустую строку, так как эти индексы не образуют диапазон значений.

Наконец, в срезах дополнительно можно указывать еще и шаг перебора символов, согласно синтаксису:

`строка[start:stop:step)`

Например:

```
s[2:10:2]
```

```
s[2::3]
```

```
s[:5:3]
```

```
s[::-2]
```

И использовать отрицательный шаг:

```
s[::-1]
```

тогда все символы будут перебираться в обратном порядке, начиная с последнего. Если же указать:

```
s[::-2]
```

то будут выбираться символы с конца через один.

Изменение строк

При работе со строками следует иметь в виду, что она относится к неизменяемым типам данных, то есть, существующую строку изменить нельзя. В частности, из-за этого попытка присвоить строке какой-либо символ:

Основы Python

```
s[0] = 'H'
```

приведет к ошибке. Чтобы изменить строку, нужно создать новую с другим содержимым:

```
s2 = 'H' + s[1:]
```

И так происходит каждый раз, когда нужно изменить что-то в уже существующей строке.

Итак, из этого занятия вы должны хорошо себе представлять:

- **строка** – упорядоченный набор символов;
- как выполняется индексация к отдельным символам строки `str[index]`;
- как выделять из строк наборы символов – **срезы**;
- **строка** – неизменяемый объект;
- способ модификации (изменения) строк через индексы и срезы.

Для закрепления этого материала, как всегда пройдите практические задания и переходите к следующему уроку.

§10. Основные методы строк

На этом занятии мы познакомимся с основными методами для строк. **Что такое методы?** Смотрите, когда мы объявляем какую-либо строку:

```
s = "python"
```

то в памяти устройства автоматически создается объект, содержащий указанные символы. Тип данных этого объекта – **строка**:

```
type(s)
```

Так вот, каждый такой объект связан с набором стандартных функций по работе со строками. Эти функции и называются методами.

Основы Python



Чтобы вызвать метод для конкретной строки, необходимо указать объект, поставить точку, записать имя метода и в круглых скобках список аргументов, если они необходимы:

объект.метод(аргументы)

Давайте рассмотрим все на конкретных примерах. Итак, у нас есть объект-строка, на который ссылается переменная `s`. Через эту переменную можно вызывать все методы строк. Чтобы увидеть их полный список, можно записать:

`s.`

и **Pycharm** отобразит полный их список. Мы с вами сейчас увидим работу наиболее употребительных из них. **Первый метод:**

`s.upper()`

возвращает новую строку (**новый объект**) со всеми заглавными буквами. При этом сама строка остается без изменений. И это логично, так как строки относятся к неизменяемым типам данных.

Основы Python

Обратите внимание, для вызова этого метода после его имени обязательно нужно поставить круглые скобки. Без них мы получим просто ссылку на объект-функцию:

```
s.upper
```

но запущена она не будет. Для запуска необходимы круглые скобки в конце – это оператор для выполнения функций и методов.

Если мы хотим сохранить результат преобразования строки в какой-либо переменной, то это делается так:

```
res = s.upper()
```

И теперь **res** ссылается на строку 'PYTHON'.

Второй метод:

```
res.lower()
```

наоборот, переводит все буквенные символы в нижний регистр и возвращает соответствующую строку.

Следующий метод

String.count(sub[, start[, end]])

возвращает число повторений подстроки **sub** в строке **String**. Два необязательных аргумента:

- **start** – индекс, с которого начинается поиск;
- **end** – индекс, которым заканчивается поиск.

В самом простом случае, мы можем для строки

```
msg = "abrakadabra"
```

определить число повторений сочетаний «**ra**»:

Основы Python

```
msg.count("ra")
```

получим значение **2** – именно столько данная подстрока встречается в нашей строке.

Теперь предположим, что мы хотим начинать поиск с буквы **k**, имеющей индекс **4**.

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	k	a	d	a	b	r	a

Тогда метод следует записать со значением **start=4**:

```
msg.count("ra", 4)
```

и мы получим значение **1**. Далее, укажем третий аргумент – **индекс**, до которого будет осуществляться поиск. Предположим, что мы хотим дойти до **10**-го индекса и записываем:

```
msg.count("ra", 4, 10)
```

и получаем значение **0**. **Почему?** Ведь на индексах **9** и **10** как раз идет подстрока «**ra**»? Но здесь, также как и в срезах, последний индекс исключается из рассмотрения. То есть, мы говорим, что нужно дойти до **10**-го, не включая его. А вот если запишем **11**:

```
msg.count("ra", 4, 11)
```

то последнее включение найдется.

Следующий метод

`String.find(sub[, start[, end]])`

Основы Python

возвращает индекс первого найденного вхождения подстроки **sub** в строке **String**. А аргументы **start** и **end** работают также как и в методе **count**.

Например:

```
msg.find("br")
```

возвращает **1**, т.к. первое вхождение «**br**» как раз начинается с индекса **1**.
Поставим теперь значение **start=2**:

```
msg.find("br", 2)
```

и поиск начнется уже со второго индекса. Получим значение **8** – индекс следующего вхождения подстроки «**br**». Если мы укажем подстроку, которой нет в нашей строке:

```
msg.find("brr")
```

то метод **find** возвращает **-1**. Третий аргумент **end** определяет индекс, до которого осуществляется поиск и работает также как и в методе **count**.

Метод **find** ищет первое вхождение слева-направо. Если требуется делать поиск в обратном направлении: справа-налево, то для этого используется метод

String.rfind(sub[, start[, end]])

который во всем остальном работает аналогично **find**. Например:

```
msg.rfind("br")
```

возвратит **8** – первое вхождение справа.

Наконец, третий метод, аналогичный **find** – это:

String.index(sub[, start[, end]])

Основы Python

Он работает абсолютно также как **find**, но с одним отличием: если указанная подстрока **sub** не находится в строке **String**, то метод приводит к ошибке:

```
msg.index("brr")
```

тогда как **find** возвращает -1. Спрашивается: **зачем нужен такой ущербный метод index?** В действительности такие ошибки можно обрабатывать как исключения и это бывает полезно для сохранения архитектуры программы, когда неожиданные ситуации обрабатываются единым образом в блоке исключений. Но, обо всем этом речь пойдет позже.

Следующий метод

String.replace(old, new, count=-1)

Выполняет замену подстрок **old** на строку **new** и возвращает измененную строку. Например, в нашей строке, мы можем заменить все буквы **a** на **o**:

```
msg.replace("a", 'o')
```

на выходе получим строку «obrokodobro». Или, так:

```
msg.replace("ab", "AB")
```

Используя этот метод, можно выполнять удаление заданных фрагментов, например, так:

```
msg.replace("ab", "")
```

Третий необязательный аргумент задает максимальное количество замен. Например:

```
msg.replace("a", 'o', 2)
```

Заменит только первые две буквы a: «msg.replace("a", 'o', 2)». При значении -1 количество замен неограниченно.

Основы Python

Следующие методы позволяют определить, из каких символов состоит наша строка. Например, метод

String.isalpha()

возвращает **True**, если строка целиком состоит из букв и **False** в противном случае. Посмотрим, как он работает:

```
msg.isalpha()
```

вернет **True**, т.к. наша строка содержит только буквенные символы. А вот для такой строки:

```
"hello world".isalpha()
```

мы получим **False**, т.к. имеется символ пробела.

Похожий метод

String.isdigit()

возвращает **True**, если строка целиком состоит из цифр и **False** в противном случае. Например:

```
"5.6".isdigit()
```

т.к. имеется символ точки, а вот так:

```
"56".isdigit()
```

получим значение **True**. Такая проверка полезна, например, перед преобразованием строки в целое число. О проверках мы еще будем говорить.

Следующий метод

String.rjust(width[, fillchar = ' '])

Основы Python

возвращает новую строку с заданным числом символов **width** и при необходимости слева добавляет символы **fillchar**:

```
d="abc"  
d.rjust(5)
```

Получаем строку « **abc**» с двумя добавленными слева пробелами. А сама исходная строка как бы прижимается к правому краю. Этот метод часто используют для добавления незначащих нулей перед цифрами:

```
d = "12"  
d.rjust(3, '0')
```

Получим строку «**012**». Причем вторым аргументом можно писать только один символ. Если записать несколько, то возникнет ошибка:

```
d.rjust(3, "00")
```

Если ширина **width** будет меньше длины строки:

```
d.rjust(1)
```

то вернется исходная строка. Аналогично работает метод

String.ljust(width[, fillchar = ' '])

который возвращает новую строку с заданным числом символов **width**, но добавляет символы **fillchar** уже справа:

```
d.ljust(10, "*")
```

Следующий метод

String.split(sep=None, maxsplit=-1)

возвращает коллекцию строк, на которые разбивается исходная строка **String**. Разбивка осуществляется по указанному сепаратору **sep**. Например:

Основы Python

```
"Иванов Иван Иванович".split(" ")
```

Мы здесь разбиваем строку по пробелам. Получаем коллекцию из **ФИО**. Тот же результат будет и при вызове метода без аргументов, то есть, по умолчанию он разбивает строку по пробелам:

```
"Иванов Иван Иванович".split()
```

А теперь предположим, перед нами такая задача: получить список цифр, которые записаны через запятую. Причем, после запятой может быть пробел, а может и не быть. Программу можно реализовать так:

```
digs = "1, 2,3, 4,5,6"  
digs.replace(" ", "").split(",")
```

мы сначала убираем все пробелы и для полученной строки вызываем **split**, получаем список цифр.

Обратный метод

String.join(список)

возвращает строку из объединенных элементов списка, между которыми будет разделитель **String**. Например:

```
d = digs.replace(" ", "").split(",")  
", ".join(d)
```

получаем строку «1, 2, 3, 4, 5, 6». Или так, изначально была строка:

```
fio = "Иванов Иван Иванович"
```

и мы хотим здесь вместо пробелов поставить запятые:

```
fio2 = ", ".join(fio.split())
```

Теперь **fio2** ссылается на строку с запятыми «Иванов,Иван,Иванович».

Основы Python

Следующий метод

String.strip()

удаляет пробелы и переносы строк в начале и конце строки. Например:

```
" hello world \n".strip()
```

возвращает строку «hello world». Аналогичные методы:

String.rstrip() и **String.lstrip()**

удаляют пробелы и переносы строк только справа или только слева.

В заключение занятия я приведу список всех рассмотренных методов, которые хорошо было бы запомнить и применять по мере необходимости при работе со строками:

Название	Описание
String.upper()	Возвращает строку с заглавными буквами
String.lower()	Возвращает строку с малыми буквами
String.count(sub[, start[, end]])	Определяет число вхождений подстроки в строке
String.find(sub[, start[, end]])	Возвращает индекс первого найденного вхождения
String.rfind(sub[, start[, end]])	Возвращает индекс первого найденного вхождения при поиске справа
String.index(sub[, start[, end]])	Возвращает индекс первого найденного вхождения
String.replace(old, new, count=-1)	Заменяет подстроку old на new
String.isalpha()	Определяет: состоит ли строка целиком из буквенных символов

Основы Python

<code>String.isdigit()</code>	Определяет: состоит ли строка целиком из цифр
<code>String.rjust(width[, fillchar = ' '])</code>	Расширяет строку, добавляя символы слева
<code>String.ljust(width[, fillchar = ' '])</code>	Расширяет строку, добавляя символы справа
<code>String.split(sep=None, maxsplit=-1)</code>	Разбивает строку на подстроки
<code>String.join(список)</code>	Объединяет коллекцию в строку
<code>String.strip()</code>	Удаляет пробелы и переносы строк справа и слева
<code>String.rstrip()</code>	Удаляет пробелы и переносы строк справа
<code>String.lstrip()</code>	Удаляет пробелы и переносы строк слева

Занятие получилось несколько справочным. Но, что поделать, некоторые возможности языка лучше сразу выучить, чтобы потом не изобретать велосипед и подменять уже существующие методы своими фрагментами программного кода.

Запомнить все это также поможет практика, а я буду вас ждать на следующем занятии.

§11. Спецсимволы, экранирование символов, raw-строки

Теперь, когда познакомились со строками и их методами, пришло время узнать, какие специальные символы могут содержать строки в **Питоне**. С одним из них мы уже сталкивались – это символ перевода строки:

`'\n'`

Я напому, например, когда задается многострочная строка:

Основы Python

```
text = """hello  
python"""
```

то в ней автоматически добавляет этот символ перевода между строками:

```
'hello\npython'
```

Причем, это один символ, хотя он и выглядит как два символа: обратный слеш и **n**. Мы в этом легко можем убедиться, если воспользоваться функцией:

```
len(text)
```

Получим значения $12 = 5 + 6 + 1$ – как раз число символов в двух строках плюс один символ перевода строки.

Мало того, мы можем его явно прописывать в любой строке, формируя многострочный текст, например, так:

```
t = "panda needs\npython"
```

и, выводя эту строку с помощью функции:

```
print(t)
```

увидим две строки.

Вообще, в строках языка **Python** можно прописывать следующие спецсимволы:

Обозначение	Описание
<code>\n</code>	Перевод строки
<code>\\</code>	Символ обратного слеша
<code>\'</code>	Символ апострофа
<code>\"</code>	Символ двойной кавычки
<code>\a</code>	Звуковой сигнал
<code>\b</code>	Эмуляция клавиши BackSpace

Основы Python

<code>\f</code>	Перевод формата
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция (размером в 4 пробела)
<code>\v</code>	Вертикальная табуляция
<code>\0</code>	Символ Null (не признак конца строки)
<code>\xhh</code>	Символ с шестнадцатичным кодом hh
<code>\ooo</code>	Символ с восьмиричным кодом ooo
<code>\N{id}</code>	Идентификатор из кодовой таблицы Unicode
<code>\uhhhh</code>	16-битный символ Unicode в шестнадцатичной форме
<code>\Uhhhhhhhh</code>	32-битный символ Unicode в шестнадцатичной форме
<code>\другое</code>	Не является экранированной последовательностью

Все их запоминать совсем не обязательно, на практике используются, в основном:

`\n, \, \', \", \t`

Значительно реже другие варианты. И, обратите внимание, перед каждым спецсимволом записан символ обратного слеша. Это, своего рода, маркер начала спецсимвола. И если после слеша идет одно из обозначений таблицы, то оно будет восприниматься как некая управляющая последовательность.

Давайте я все это продемонстрирую на примерах. Добавим в строку символ табуляции:

```
t = "\tpanda needs\npython"
```

Теперь функция **print()** интерпретирует его, как особый горизонтальный отступ:

```
print(t)
```


Основы Python

Если же мы уберем букву **t**:

```
t = "\panda needs\npython"
```

то при печати увидим просто обратный слеш. В действительности, здесь сработала последняя строчка таблицы: когда не подходит ни одна последовательность, то просто печатается обратный слеш.

Но здесь нужно быть осторожным. Предположим, что мы слово **needs** хотим заключить в обратные слеша:

```
t = "panda \needs\ python"
```

Однако, при печати:

```
print(t)
```

первый слеш пропадет, так как он будет восприниматься началом спецпоследовательности символа переноса строки. Поэтому, для добавления символа обратного слеша в строку, следует записывать два обратных слеша подряд:

```
t = "panda \\needs\\ python"
```

Тогда в строке они будут автоматически заменены на один символ слеша и при выводе мы это и видим. Это называется экранированием, когда мы символы с двойным назначением записываем, добавляя перед ними обратный слеш. В данном случае получаем двойной слеш.

Часто такие символы следует прописывать при определении путей к файлам. Как мы знаем, в ОС Windows маршруты имеют вид:

D:\Python\Projects\stepik\tex1.py

Здесь фигурируют обратные слеша для разделения каталогов. Чтобы правильно описать такой путь, слеша следует экранировать:

Основы Python

```
path = "D:\\Python\\Projects\\stepik\\tex1.py"
```

и при печати:

```
print(path)
```

видим, что маршрут определен верно. Если бы слешы не были экранированы, то получили бы неверный путь к файлу:

```
path = "D:\Python\Projects\stepik\tex1.py"
```

Вот этот момент следует хорошо запомнить.

Кроме обратного слеша экранировать также следует и **кавычки**. Например, мы хотим сформировать строку:

```
s = "Марка вина "Ягодка"
```

Внутри этой строки имеются кавычки. Но эти же самые кавычки определяют начало и конец строки в **Python**. Поэтому такая запись приведет к синтаксической ошибке. Чтобы все работало корректно, нужно выполнить экранирование кавычек:

```
s = "Марка вина \"Ягодка\""
```

Или, в данном случае, можно было бы использовать одинарные кавычки для определения строки, а внутри нее записать двойные:

```
s = 'Марка вина "Ягодка"'
```

Но на практике рекомендуется всегда выполнять экранирование таких символов, чтобы избежать случайных ошибок. Например, в этой строке уже нельзя просто так записать одинарные кавычки:

```
s = 'Марка вина 'Ягодка''
```

Снова получим синтаксическую ошибку, их нужно экранировать:

Основы Python

```
s = 'Марка вина \'Ягодка\''
```

В завершение этого занятия отмечу, что в **Python** можно задавать, так называемые, сырые (**row**) строки. Это строки, в которых игнорируются спецпоследовательности и все символы воспринимаются буквально так, как записаны. Например, если взять строку с путем к файлу:

```
path = "D:\\Python\\Projects\\stepik\\tex1.py"
```

то сейчас, при отображении, мы видим по одному слешу:

```
print(path)
```

Но, если определить эту же строку, как сырую, добавив букву **r** перед ней:

```
path = r"D:\\Python\\Projects\\stepik\\tex1.py"
```

то при печати увидим по два слеша, именно так, как прописали. Поэтому, в таких строках можно убрать спецопределения и записывать строку буквально так, как она должна выглядеть:

```
path = r"D:\Python\Projects\stepik\tex1.py"
```

На этом мы завершим наше очередное занятие по **Python**. Из него вы должны хорошо себе представлять, что такое спецсимволы и экранирование символов. Какие основные спецсимволы для строк существуют и как определяются сырые строки.

§12. Форматирование строк: метод `format` и F-строки

На этом занятии мы подробнее поговорим о способах формирования строк на основе набора различных переменных. Мы уже знаем с вами как создавать строки и соединять их между собой. Предположим, у нас есть две переменные:

```
age=55; name="Имеда"
```

Основы Python

и мы формируем такое сообщение:

```
"Меня зовут "+name+", мне "+str(age)+" и я люблю язык Python."
```

На выходе получим:

```
'Меня зовут Имеда, мне 55 и я люблю язык Python.'
```

Но это не самый удобный и распространенный способ формирования строк по шаблону. Гораздо удобнее использовать специальный метод строки:

str.format(*args)

который в данном конкретном случае можно использовать так:

```
"Меня зовут {0}, мне {1} и я люблю язык Python.".format(name, age)
```

Смотрите, здесь в фигурных скобках мы указываем индекс переменной, значение которой будет подставлено в это место строки. Индекс **0** – первая переменная, указанная в методе **format**, т.е. **name**, а индекс **1** – вторая переменная **age**. То есть, метод формат возвращает новую строку в заданном формате. И чтобы сохранить этот результат в какой-либо переменной, следует записать так:

```
msg = "Меня зовут {0}, мне {1} и я люблю язык Python.".format(name, age)
```

Теперь **msg** ссылается на созданную строку.

Также мы можем одну и ту же переменную указывать несколько раз, например, так:

```
"Меня зовут {0}, мне {1} и я люблю язык Python. {0}".format(name, age)
```

Как видите, это удобнее обычной конкатенации строк. Но эту запись можно сделать еще понятнее, используя именованные параметры. Для этого у каждой переменной пропишем ее имя (**ключ**). Оно придумывается программистом, например:

Основы Python

`format(fio=name, old=age)`

и, далее, в формате строки указываются уже эти имена:

```
"Меня зовут {fio}, мне {old} и я люблю язык Python. {fio}".format(fio=name, old=age)
```

Это уже гораздо нагляднее и понятнее. Глядя на эту строку, мы легко понимаем, что и где будет записано. Конечно, если указать неверное имя, например, так:

```
"Меня зовут {name}, мне {old} и я люблю язык Python. {fio}".format(fio=name, old=age)
```

То возникнет ошибка, говорящая, что ключа **name** не существует. То есть, в фигурных скобках указывается именно ключ, а не переменная. И, кроме того, благодаря использованию ключей, мы можем записывать переменные в методе **format** в любом порядке. Например, поменяем их местами:

```
"Меня зовут {fio}, мне {old} и я люблю язык Python. {fio}".format(old=age, fio=name)
```

На выходе получим то же самое.

Но, начиная с версии **Python 3.11.5**, появился новый, гораздо более удобный способ форматирования строк. Это, так называемые, **F-строки**. Здесь мы рассмотрим лишь основы использования этого подхода, для более глубокого изучения можно обратиться к стандарту **PEP8**:

<https://peps.python.org/pep-0498/>

Итак, чтобы указать **Python** воспринимать строку, как **F-строку**, перед ее литералом ставится символ **'f'** (именно малая буква). И, теперь, в фигурных скобках можно записывать любые конструкции языка **Python**, например, имена переменных:

```
f"Меня зовут {name}, мне {age} и я люблю язык Python."
```

Основы Python

Видите как это удобно?

Далее, внутри фигурных скобок **F-строк** можно записывать, например, и арифметические операции и методы строк:

```
f"Меня зовут {name.upper()}, мне {age*2} и я люблю язык Python."
```

В результате, получим:

```
'Меня зовут ИМЕДА, мне 55 и я люблю язык Python.'
```

Или даже не прописывать переменные, а использовать обычные числовые операции и вызывать функции:

```
f"Меня зовут {len(name)}, мне {10*5+5} и я люблю язык Python."
```

Как видите, все достаточно просто и наглядно. Теперь, в своих программах вы тоже сможете использовать **F-строки**, если ваш интерпретатор языка версии **3.11.5** и выше, либо метод `format()`, если **F-строки** не поддерживаются из-за низкой версии. Обычно, сейчас повсеместно применяют **F-строки**, так как они намного удобнее других способов форматирования строк.

Для закрепления этого материала подготовлено несколько практических заданий и после их выполнения жду всех вас на следующем занятии.

§13. Списки - операторы и функции работы с ними

Мы переходим к новой теме – списки в языке **Python**. **Что такое список и зачем он нужен?** Представьте, что нам в программе нужно хранить и обрабатывать список городов, или список оценок студента, или список булевых значений, или значения функции и многое другое. Часто, когда нам нужно оперировать набором каких-либо данных, используются списки. И на этом занятии мы начнем с ними знакомиться.

Основы Python

Города:	Казань	Тверь	Уфа	Астрахань		
Оценки:	4	3	2	2	4	5
Булевый:	True	False	False	True		
f(x) =	1	2	3	4	5	6

Задать список в программе на **Python** очень просто, ставятся квадратные скобки и внутри них через запятую перечисляются его элементы:

```
["Москва", "Тверь", "Вологда"]
```

Чтобы оперировать списком через переменную, используется оператор присваивания:

```
marks = [2, 3, 4, 3, 5, 2]
```

Давайте теперь посмотрим, **как эту конструкцию, этот список использовать в программировании?** Например, **вычислить средний балл по оценкам?** Для этого нам надо уметь обращаться к отдельным элементам списка. **Как это сделать?** Так как список – это **упорядоченная** коллекция, то все его элементы имеют свой порядковый номер – индекс, начиная с нулевого и заканчивая последним.

индекс:	0	1	2	3	4	5
	2	3	4	3	5	2
	-6	-5	-4	-3	-2	-1

Используя эти индексы и синтаксис:

список[индекс]

мы можем обращаться к отдельным элементам. Например:

Основы Python

```
marks[0]  
marks[2]
```

И для вычисления среднего балла среди шести оценок, получим формулу:

```
avg = (marks[0] + marks[1] + marks[2] + marks[3] + marks[4] + marks[5]) / 6
```

Это простейший пример того, как можно хранить и использовать данные списка. Причем индексирование здесь работает так же, как и со строками. Если указать несуществующий индекс:

```
marks[10]
```

то получим ошибку. А чтобы обратиться к последнему элементу, можно использовать отрицательный индекс:

```
marks[-1]
```

То есть, мы здесь также имеем наборы отрицательных индексов, идущих от конца к началу списка.

Список – изменяемый тип данных

В отличие от строк, списки в **Python** относятся к изменяемым типам данных. То есть, мы можем изменить ранее хранимое значение. Например, студент пересдал первую двойку на тройку и мы хотим внести это изменение в список **marks**. Сделать это можно через оператор присваивания:

```
marks[0] = 3
```

Все, теперь первое значение равно **3**. Как вы помните, со строками такая операция приводила к ошибке, так как строки – это неизменяемый тип. Но со списками мы так делать можем и в этом их кардинальное отличие от строк. **Список** – динамическая структура данных, который может меняться в процессе работы программы.

Основы Python

Мало того, списку, состоящему из чисел, мы легко можем присвоить любой другой тип данных, например, строку:

```
marks[1] = "удовл."
```

Вообще списки могут содержать самые разные типы данных, например:

```
lst = ["Москва", 1320, 5.8, True, "Тверь", False]
```

в том числе и другие, вложенные списки:

```
lst2 = [1, 2.5, [-1, -2, -3], 4]
```

О вложенных списках мы еще будем говорить.

Если нам нужно создать пустой список, то достаточно записать квадратные скобки без элементов:

```
a = []
```

или, воспользовавшись специальной функцией:

```
b = list()
```

которая создает новый пустой список. Если же ей в качестве аргумента указать другой список:

```
a = list([True, False])
```

то будет создан новый список с тем же самым содержимым. Также мы можем передать ей строку:

```
list("Python")
```

тогда получим список, состоящий из отдельных символов этой строки. Вообще на вход функции `list()` можно передавать любой перебираемый объект, на основе которого формируется новый список. Такие перебираемые

Основы Python

объекты еще называются **итерируемыми**, но мы о них будем говорить на будущих занятиях.

Функции работы со списками

Язык **Python** содержит несколько удобных встроенных функций для работы со списками:

- **len()** – определение числа элементов в списке (длина списка);
- **max()** – для нахождения максимального значения;
- **min()** – для нахождения минимального значения;
- **sum()** – для вычисления суммы;
- **sorted()** – для сортировки коллекции.

Для начала воспользуемся уже знакомой нам функцией **len** для определения длины списка:

```
len(marks)
```

Соответственно, для пустого списка:

```
len([])
```

она возвращает **0**.

Две из них **min()** и **max()** нам уже знакомы. Сейчас мы увидим, как их можно применять к спискам. Сформируем список значений температуры по дням города Москвы:

```
t = [23.5, 25.6, 27.3, 26.0, 30.4, 29.5]
```

Теперь, чтобы найти максимальное и минимальное значения, достаточно вызвать функции:

```
max(t)  
min(t)
```

Основы Python

Для подсчета суммы всех значений, запишем функцию **sum**:

```
sum(t)
```

А вычислить среднюю температуру можно следующим образом:

```
sum(t)/len(t)
```

Видите, как просто это делается на уровне функций.

Наконец, последняя функция **sorted**, если ее вызвать с одним аргументом:

```
sorted(t)
```

то она возвратит новый список с отсортированными значениями по неубыванию (или, как часто говорят, по возрастанию). И, обратите внимание, эта функция не меняет прежний список **t**, она именно возвращает новый список с отсортированными значениями. Очевидно, чтобы сохранить результат работы этой функции, следует использовать переменную, например, так:

```
t_sort = sorted(t)
```

Если же нам нужно отсортировать список по невозрастанию (убыванию), то дополнительно прописывается параметр:

```
sorted(t, reverse=True)
```

Функции **min()**, **max()** и **sorted()** работают не только с числовыми типами, но и вообще с любыми, где допустимы операторы сравнения больше и меньше. Например, создадим список из символов (строк):

```
s = list("python")
```

И выполним все те же самые функции:

```
max(s)
```

Основы Python

```
min(s)
sorted(s)
```

А вот функция **sum()** приведет к ошибке:

```
sum(s)
```

Операторы списков

При работе со списками часто используются следующие операторы:

- **+** – соединение двух списков в один;
- ***** – дублирование списка;
- **in** – проверка вхождения элемента в список;
- **del** – удаление элемента списка.

Например:

```
[1, 2, 3] + [4, 5]
```

Но, вот так:

```
[1, 2, 3] + 4
```

работать не будет, так как оператор **+** соединяет именно списки между собой, просто число или какой-либо другой тип данных записывать нельзя. В данном случае правильно будет так:

```
[1, 2, 3] + [4]
```

Или, так:

```
[1, 2, 3] + [True]
```

Как видите, добавлять в список можно самые разные типы данных.

Следующий оператор ***** выполняет дублирование списка указанное число раз:

Основы Python

```
["Я", "люблю", "Python"] * 3
```

Этот оператор работает также как и со строками, здесь можно указывать только целое число (или переменную, ссылающуюся на целое значение). Прописывать дробные числа нельзя:

```
["Я", "люблю", "Python"] * 3.5
```

На практике можно комбинировать операторы и определять, например, такие конструкции:

```
["Я"] + ["люблю"] * 3 + ["Python"]
```

Все достаточно очевидно, гибко, наглядно и просто. Этим и знаменателен этот язык. В нем многое реализуется простыми и понятными методами, в отличие от других языков программирования.

Следующий оператор **in** позволяет определять вхождение некоторого значения в список. Делается это также просто, например:

```
lst = ["Москва", 1320, 5.8, True, "Тверь", False]  
1320 in lst  
120 in lst
```

Или, можно узнать, является ли элементом списка другой список:

```
[1, 2] in lst
```

В данном случае получим **False**, но если его добавить:

```
lst = ["Москва", 1320, 5.8, True, "Тверь", False, [1, 2]]
```

то

```
[1, 2] in lst
```

вернет **True**. И так со всеми типами данных:

Основы Python

```
"Москва" in lst
```

Последний оператор **del** выполняет удаление элемента списка по его индексу, например, так:

```
del lst[2]
```

при этом индексы оставшихся элементов по прежнему идут по порядку от 0 и до конца списка.

На этом мы завершим наше первое занятие по спискам. Из него вы должны себе хорошо представлять, как задавать списки с использованием квадратных скобок и функции **list()**, понимать механизм индексации (обращение к отдельным элементам списка), уметь оперировать функциями **len()**, **max()**, **min()**, **sum()**, **sorted()** и использовать операторы **+**, *****, **in**, **del**. Все это следует закрепить практическими заданиями, после чего, жду всех вас на следующем уроке.

§14. Срезы списков и сравнение списков

Мы продолжаем изучение списков языка **Python**. Это занятие начнем со **срезов**. О срезах мы с вами уже говорили, когда рассматривали строки и с их помощью выделяли наборы символов из строк. Со списками все работает аналогично: срезы позволяют выделять наборы элементов. Например, у нас есть список:

```
lst = ["Москва", "Уфа", "Тверь", "Казань"]
```

и мы хотим выбрать из него 2-й и 3-й элементы. Это можно сделать, так:

```
lst[1:3]
```

Давайте разберемся, как это работает. Синтаксис для срезов имеет вид:

список[start:end]

Основы Python

В данном случае, мы указываем стартовый индекс **1** (второй элемент) и конечный **3** (последний элемент, до которого выделяем срез, не включая его). В итоге, на выходе получаем список из двух элементов:

```
['Уфа', 'Тверь']
```

То есть, это новый список, состоящий из выделенных элементов. Мы в этом можем легко убедиться. Сохраним через переменную результат среза:

```
a = lst[1:3]
```

Затем, изменим его первый элемент:

```
a[0] = "Воронеж"
```

и, как видим, это никак не повлияло на исходный список **lst**.

В срезах можно указывать только начальный индекс:

```
lst[2:]
```

тогда все будет выбрано с **3**-го элемента и до конца списка. Или только конечный индекс:

```
lst[:3]
```

Обратите внимание, последний указанный индекс не включается в срез. Если же не указывать ни первый, ни последний индексы, то получим копию исходного списка:

```
cities = lst[:]
```

Здесь создается именно копия списка. Если мы посмотрим на их **id**:

```
id(lst)  
id(cities)
```

Основы Python

то они будут разными. Также копию списка, можно сделать с помощью рассмотренной ранее функции `list`:

```
c = list(lst)
```

Какой именно вариант использовать для копирования списка, решает сам программист, в зависимости от удобства. Чаще всего используется синтаксис срезов, т.к. это более короткая запись.

А что будет, если мы один список присвоим другому:

```
d = lst
```

будет ли здесь создаваться копия списка? Вспоминая занятие по переменным, мы говорили, что переменные – это ссылки на объекты и оператор присваивания лишь копирует ссылку, но не сам объект. Поэтому переменные `d` и `lst` будут ссылаться на один и тот же список, их `id`:

```
id(d)  
id(lst)
```

равные. Ну а наиболее сомневающимся предлагаю изменить значение списка через `d`:

```
d[0] = "Самара"
```

и посмотреть на список `lst`. В нем произойдет такое же изменение.

Далее, так как у списка имеются отрицательные индексы, то они также могут быть использованы при определении срезов. Например, создадим список оценок:

```
marks = [2, 3, 4, 3, 5, 2]
```


Основы Python

индекс:	0	1	2	3	4	5
marks =	2	3	4	3	5	2
	-6	-5	-4	-3	-2	-1

и запишем следующие срезы:

```
marks[2:-1]  
marks[-3:-1]
```

Дополнительно в срезах можно указывать шаг перебора элементов, согласно синтаксису:

список[start:stop:step]

Например:

```
marks[1:5:2]
```

или без указания границ:

```
marks[:5:2]  
marks[1::2]  
marks[::2]
```

Если шагом является отрицательное число, то перебор элементов осуществляется с конца списка, например:

```
marks[::-1]  
marks[::-2]
```

Вообще, механизм срезов работает абсолютно также как и со строками, только здесь они применяются к спискам.

Но, учитывая, что списки относятся к изменяемым типам данных, то со срезами можно выполнять одну дополнительную операцию – изменение

Основы Python

группы элементов. Например, для списка **marks** мы хотим 3-ю и 4-ю оценки представить в виде строк. Это можно сделать, следующим образом:

```
marks[2:4] = ["хорошо", "удовлетв."]
```

И теперь коллекция содержит данные:

```
[2, 3, 'хорошо', 'удовлетв.', 5, 2]
```

Видите, как легко и просто это можно сделать. Или, другой пример, перебрать все элементы через один и присвоить им 0:

```
marks[::2] = [0, 0, 0]
```

Правда, такая конструкция содержит потенциальную ошибку. Если увеличить размер списка:

```
marks += [3]
```

и повторить операцию:

```
marks[::2] = [0, 0, 0]
```

то получим ошибку, так как число замен здесь уже четыре, а не три. Поэтому, в таких присваиваниях лучше явно указывать границы срезов:

```
marks[:5:2] = [0, 0, 0]
```

Также, для группового присваивания можно использовать и такой синтаксис:

```
marks[2:4] = 10, 20
```

Сравнение списков

В заключение этого занятия рассмотрим возможности сравнения списков между собой с помощью операторов:

>, <, ==, !=

Основы Python

```
[1, 2, 3] == [1, 2, 3] # True  
[1, 2, 3] != [1, 2, 3] # False  
[1, 2, 3] > [1, 2, 3] # False
```

В последнем сравнении получим **False**, т.к. списки равны, но если записать так:

```
[10, 2, 3] > [1, 2, 3] # True
```

то первый список будет больше второго. Здесь сравнение больше, меньше выполняется по тому же принципу, что и у строк: перебираются последовательно элементы, и если текущий элемент первого списка больше соответствующего элемента второго списка, то первый список больше второго. И аналогично, при сравнении меньше:

```
[10, 2, 3] < [1, 2, 3] # False
```

Все эти сравнения работают с однотипными данными:

```
[1, 2, "abc"] > [1, 2, "abc"] # False
```

сработает корректно, а вот так:

```
[1, 2, 3] > [1, 2, "abc"]
```

произойдет ошибка, т.к. число **3** не может быть сравнено со строкой «abc».

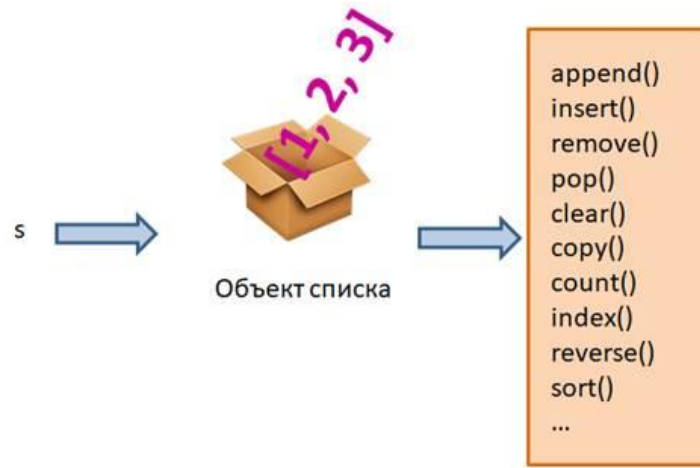
На этом завершим очередное занятие. Надеюсь, вы теперь хорошо себе представляете механизм **срезов**, а также знаете, как выполняется сравнение списков между собой. После закрепления материала практическими заданиями, жду вас всех на следующем уроке.

§15. Основные методы списков

На этом занятии мы сами познакомимся с основными методами, которые есть у списков. Что такое методы вам должно быть уже известно, об этом была речь, когда мы рассматривали методы строк. Кратко напомним, что

Основы Python

список – это объект и с этим объектом связаны функции, которые и называются его методами.



Давайте предположим, что у нас имеется список из чисел:

```
a = [1, -54, 3, 23, 43, -45, 0]
```

и мы хотим в конец этого списка добавить еще одно значение. Это можно сделать с помощью метода:

```
a.append(100)
```

Данный метод **append** ничего не возвращает, а меняет сам список. Поэтому писать здесь конструкцию вида:

```
a = a.append(100)
```

не следует, так как это приведет к потере данных. Этим методы списков отличаются от методов строк, когда мы записывали:

```
string="Hello"  
string = string.upper()
```

Здесь метод **upper** возвращает измененную строку, поэтому все работает как и ожидается. А метод **append** ничего не возвращает, и присваивать значение **None** переменной **a** не имеет смысла.

Основы Python

Учитывая, что список может содержать самые разные данные, то в методе **append** можно прописывать не только число, но, например, строку:

```
a.append("hello")
```

тогда в конец списка будет добавлен этот элемент. Или, булево значение:

```
a.append(True)
```

Или еще один список:

```
a.append([1,2,3])
```

И так далее. Главное, чтобы было указано одно конкретное значение. Например, вот так работать не будет:

```
a.append(1,2)
```

Если нам нужно вставить новый элемент в произвольную позицию, то используется метод:

```
a.insert(3, -1000)
```

Здесь мы указываем индекс вставляемого элемента и далее значение самого элемента.

Следующий метод **remove** удаляет элемент по значению:

```
a.remove(True)  
a.remove('hello')
```

Он находит первый подходящий элемент и удаляет его, остальные не трогает. Если же указывается несуществующий элемент:

```
a.remove('hello2')
```

то возникает ошибка. Еще один метод для удаления

Основы Python

`a.pop()`

выполняет удаление последнего элемента и при этом, возвращает его значение. В самом списке последний элемент пропадает. То есть, с помощью этого метода можно сохранять удаленный элемент в какой-либо переменной:

```
end = a.pop()
```

Также в этом методе можно указывать индекс удаляемого элемента, например:

```
a.pop(3)
```

Если нам нужно очистить весь список – удалить все элементы, то можно воспользоваться методом:

```
a.clear()
```

Получим пустой список. Следующий метод

```
a = [1, -54, 3, 23, 43, -45, 0]  
c = a.copy()
```

возвращает копию списка. Это эквивалентно конструкции:

```
c = list(a)
```

В этом можно убедиться по разным **id** этих объектов:

```
id(c)  
id(a)
```

Следующий метод **count** позволяет найти число элементов с указанным значением:

```
c.count(1)  
c.count(-45)
```

Основы Python

Если же нам нужен индекс определенного значения, то для этого используется метод **index**:

```
c.index(-45)  
c.index(1)
```

возвратит **0**, т.к. берется индекс только первого найденного элемента. Но, мы здесь можем указать стартовое значение для поиска:

```
c.index(1, 1)
```

Здесь поиск будет начинаться с индекса **1**, то есть, со второго элемента. Или, так:

```
c.index(23, 1, 5)
```

Ищем число **23** с **1**-го индекса и по **5**-й не включая его. Если элемент не находится

```
c.index(23, 1, 3)
```

то метод приводит к ошибке. Чтобы этого избежать в своих программах, можно вначале проверить: существует ли такой элемент в нашем срезе:

```
23 in c[1:3]
```

и при значении **True** далее уже определять индекс этого элемента.

Следующий метод

```
c.reverse()
```

меняет порядок следования элементов на обратный.

Ну и последний метод

```
c.sort()
```

Основы Python

выполняет сортировку элементов списка по возрастанию. Для сортировки по убыванию, следует этот метод записать так:

```
c.sort(reverse=True)
```

Причем, этот метод работает и со строками:

```
lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]  
lst.sort()
```

Здесь используется лексикографическое сравнение, о котором мы говорили, когда рассматривали строки.

Отличие метода **sort()** от ранее рассмотренной функции **sorted()** в том, что метод меняет сам список, а функция **sorted()** возвращает новый отсортированный список, не меняя начальный.

Это все основные методы списков, которые вам следует знать и применять в практике программирования:

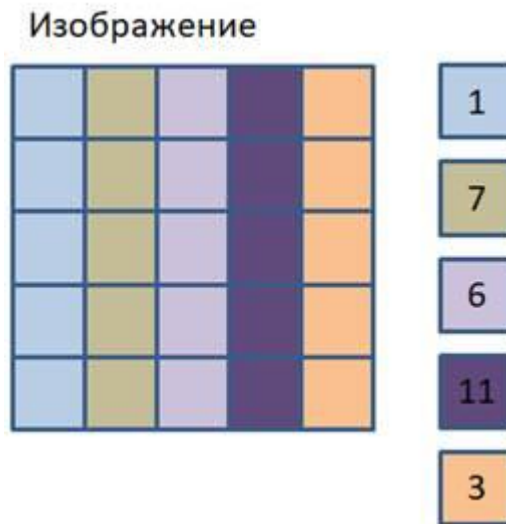
Метод	Описание
append()	Добавляет элемент в конец списка
insert()	Вставляет элемент в указанное место списка
remove()	Удаляет элемент по значению
pop()	Удаляет последний элемент, либо элемент с указанным индексом
clear()	Очищает список (удаляет все элементы)
copy()	Возвращает копию списка
count()	Возвращает число элементов с указанным значением
index()	Возвращает индекс первого найденного элемента
reverse()	Меняет порядок следования элементов на обратный
sort()	Сортирует элементы списка

Основы Python

Как всегда, для закрепления материала обязательно пройдите практические задания и переходите к следующему уроку.

§16. Вложенные списки, многомерные списки

Это заключительное занятие по спискам языка **Python**. Сегодня мы с вами узнаем, как формировать вложенные списки и работать с ними. Но сначала, что это такое и зачем они нужны.



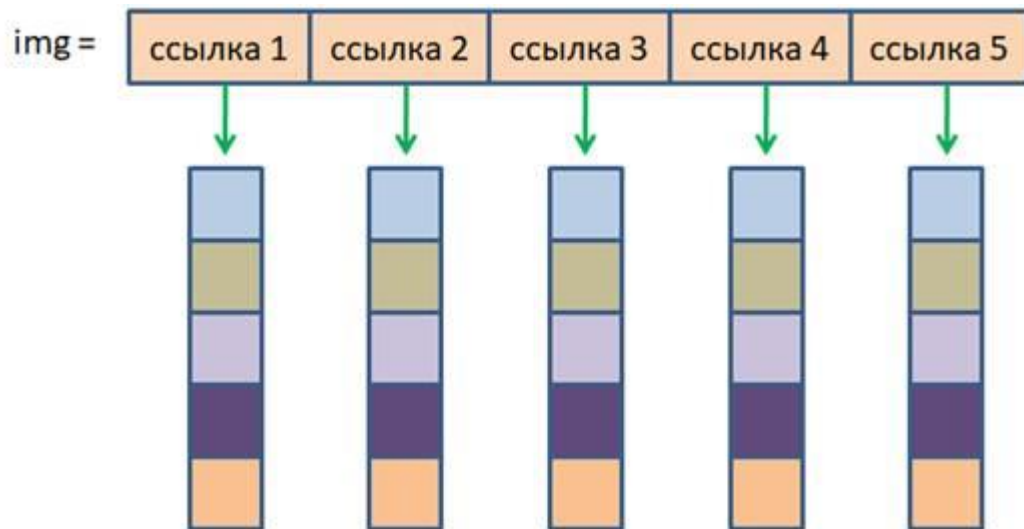
Давайте представим, что нам в программе нужно хранить изображение. Для примера я нарисовал его небольшим, всего **5** на **5** пикселей. Каждый цвет представляется своим уникальным числом. Я, условно, обозначил их **1**, **7**, **6**, **11** и **3**. Значит, для представления этих данных нам нужен двумерный список **5x5** с соответствующими числовыми значениями. Мы уже знаем, как задавать одномерный список:

```
line = [1, 7, 6, 11, 3]
```

Но так он описывает всего лишь одну строку. А нам нужно хранить пять таких строк. Учитывая, что элементом списка может быть другой список, то данное изображение можно задать так:

```
img = [[1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3]]
```

Основы Python



Мы здесь внутри первого списка определили пять вложенных и в результате получили двумерный список. Кстати, его можно было бы сформировать и проще, учитывая, что все вложенные списки одинаковы, на основе списка **line**, следующим образом:

```
img = [line[:], line[:], line[:], line[:], line[:]]
```

В итоге получим такой же список с независимыми строками:

```
[[1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3]]
```

Вот эта последняя запись нам показывает структуру представления многомерных данных на уровне списков. Первый главный список хранит ссылки на вложенные списки. А вложенные списки уже хранят ссылки на соответствующие числа, представляющие тот или иной цвет. Поэтому, если взять первый элемент главного списка:

```
img[0]
```

то мы получим список, представляющий первую строку (или, первый столбец в зависимости от интерпретации программистом этих данных). Главное, что мы получаем доступ к первому вложенному списку. А раз это

Основы Python

так, то можно записать еще одни квадратные скобки и из этого вложенного списка взять, допустим, второй элемент:

```
img[0][1]
```

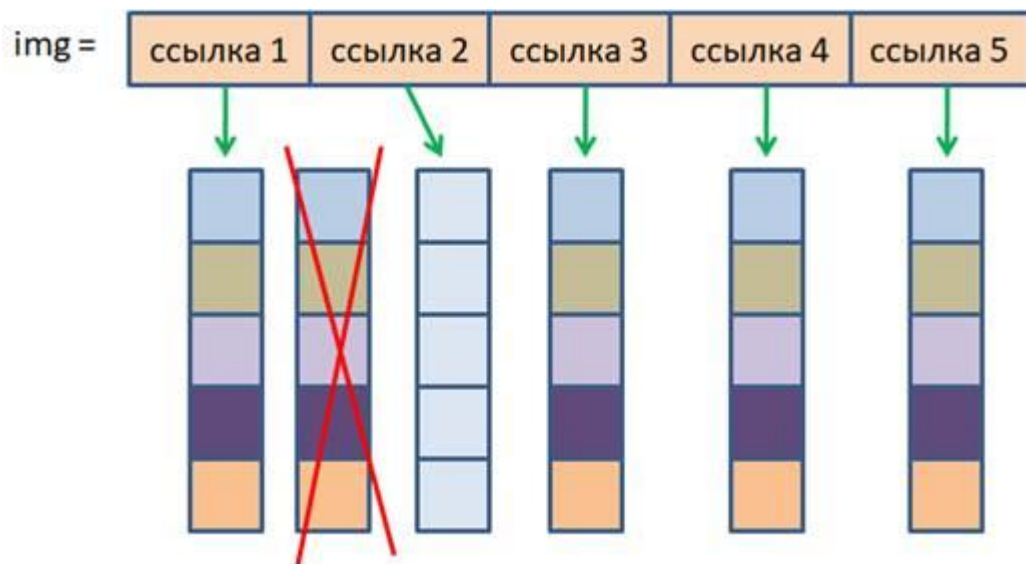
Также, можно заменить, например, вторую строку на новую, допустим, такую:

```
img[1] = [0, 0, 0, 0, 0]
```

или, то же самое, в более краткой форме:

```
img[1] = [0] * 5
```

Что в итоге здесь произошло? Мы сформировали новый объект – список из нулей, связали с ним вторую ссылку главного списка, а прежний список был автоматически удален сборщиком мусора.



Если бы мы хотели изменить значения уже существующего вложенного списка, то следовало бы обратиться к его элементам, например, через механизм срезов:

```
img[1][:] = [0] * 5
```

Основы Python

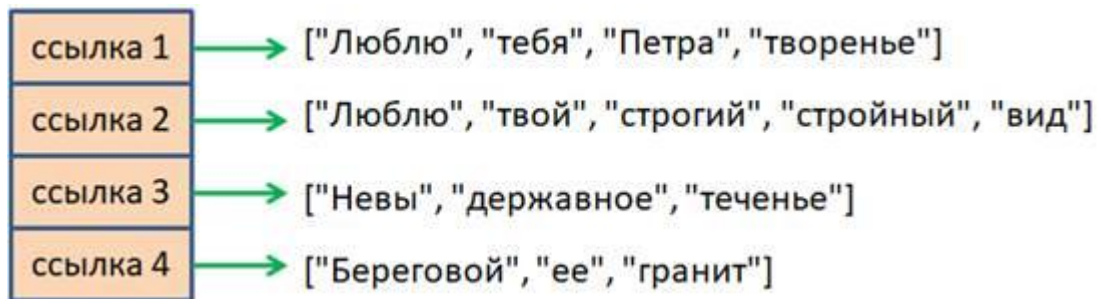
и присвоить его элементам новые числовые значения. Вот так это работает в деталях.

В качестве второго примера мы представим вложенными списками строки известного стихотворения на уровне отдельных слов:

Люблю тебя, Петра творенье,
Люблю твой строгий, стройный вид,
Невы державное течение,
Береговой ее гранит,

Здесь в каждой строке разное число слов, но для вложенных списков – это не проблема. Они могут иметь разное число элементов:

```
t = ["Люблю", "тебя", "Петра", "творенье"],  
    ["Люблю", "твой", "строгий", "стройный", "вид"],  
    ["Невы", "державное", "течение"],  
    ["Береговой", "ее", "гранит"]  
]
```



В результате получаем следующую структуру наших данных. Здесь также для доступа к первой строке достаточно указать первый индекс:

```
t[0]
```

а к отдельному слову этой строки, второй индекс:

```
t[0][2]
```

Основы Python

Если же мы хотим изменить какое-либо слово, то это делается, следующим образом:

```
t[0][2] = "Питон"
```

В итоге, первая строка принимает вид:

```
['Люблю', 'тебя', 'Питон', 'творенье']
```

Мало того, мы можем добавить новую строку, используя известный метод:

```
t.append(["Твоих", "оград", "узор", "чугунный"])
```

Удалять список:

```
del t[1]
```

И так далее, то есть, делать с вложенными списками все те же операции, что и с обычными данными.

В заключение покажу пример многомерного списка с разными уровнями глубины:

```
A = [[[True, False], [1, 2, 3]], ["матрица", "вектор"]]
```

Смотрите, здесь

```
A[0]
```

это двумерный список, а

```
A[1]
```

одномерный вложенный список. Соответственно, для третьего уровня вложенности можем использовать три индекса для доступа к отдельному элементу, например:

```
A[0][1][0]
```

Основы Python

Здесь мы берем первый элемент, затем второй вложенный список и из него выбираем первый элемент.

Вот так в **Python** можно определять многомерные списки разной структуры и разного уровня вложенности.

Для закрепления этого материала подготовлены практические задания, после которых жду вас на следующем уроке.

§17. Условный оператор if. Конструкция if-else

На этом занятии мы с вами познакомимся с работой условного оператора **if**.

Что он делает и для чего нужен? Помните, на одном из прошлых занятий, мы с вами рассматривали операции сравнения, например, проверяли:

```
4 > 2
```

или с переменной:

```
a = 5  
a < 0
```

На выходе получали булевы значения **True** и **False**. Но ничего не говорили, как их дальше использовать в программе. Пришло время восполнить этот пробел. Оператор **if** позволяет выполнить группу операторов, при истинности указанного условия. Давайте я сразу поясню эту конструкцию на конкретном примере. Предположим, мы пишем программу для вычисления модуля числа:

```
x = -4  
if x < 0:  
    x = -x  
print(x)
```

Смотрите, после оператора **if** записываем условие, то есть, определяем, что **x** отрицательное число. И если это так, то оператор меньше вернет значение

Основы Python

True, условие сработает и будет выполнена строка `x = -x`. А вот следующая строка (функция `print()`) стоит уже вне этого условия и будет выполняться всегда. Так как **Python** «понимает», что находится в блоке условного оператора, а **что нет?** Все дело в форматировании текста программы. Когда мы ставим двоеточие, то открывается новый блок, где может располагаться множество операторов. И все конструкции языка **Python**, что имеют отступ (обычно четыре пробела) относительно оператора `if`, будут располагаться внутри этого блока. Именно поэтому строка `x = -x` выполняется только при истинности условия `x < 0`, а функция `print()` стоит уже после оператора `if` и выполняется всегда.

Запустим эту программу и убедимся, что все работает. Видим значение 4, то есть, условие `x < 0` вернуло значение **True**, оператор `if` сработал и выполнялась строка `x = -x`.

А давайте теперь определим:

```
x = 4
```

В этом случае также увидим 4, но строка `x = -x` уже не выполнялась, так как условие для значения **False** не срабатывает. И, смотрите, если функцию `print()` сместить на уровень предыдущего оператора:

```
x = 4
if x < 0:
    x = -x
    print(x)
```

то в консоли мы уже ничего не увидим. Это произошло, как раз, из-за того, что эта функция переместилась внутрь блока оператора `if` и будет теперь выполняться только при истинности условия. Вернем, снова:

```
x = -4
```

и, теперь, видим значение 4 в консоли. Этот пример показывает, что в **Python** форматирование текста программы имеет ключевое значение. И благодаря

Основы Python

этому, кстати, программы становятся более наглядными и читабельными, а также заставляют начинающих программистов правильно их оформлять. На мой взгляд – это большой плюс данного языка программирования.

Чтобы все было понятно, приведу еще пару примеров. Предположим, мы хотим поменять значения двух переменных:

```
a = float(input("a: "))
b = float(input("b: "))
```

Если a меньше b:

```
if a < b:
    a, b = b, a
print(a, b)
```

Также можно прописывать более сложные, составные условия, например, для проверки попадания числа в заданный диапазон:

```
x = int(input())
if x >= -4 and x <= 10:
    print("x в диапазоне [-4; 10]")
```

О составных условиях мы с вами уже говорили на отдельном занятии, поэтому здесь все должно быть понятно.

Или, то же самое условие в **Python** допускается записывать и так:

```
if -4 <= x <= 10:
```

То есть, в качестве условия можно записывать любые конструкции, которые можно интерпретировать как истину (**True**) и ложь (**False**). Даже так, просто указав числовое значение:

```
x = int(input())
if x:
    print("x = True")
```


Основы Python

Мы здесь получим истину, если `x` не равен нулю. Наконец, в условиях можно явно прописывать булево значение:

```
if True:
    print("True")
```

Конечно, это лишь пример и практического смысла в таком операторе особо нет.

Наконец, мы можем использовать операторы проверки для списков. Например, студент имеет следующие оценки по результатам сдачи сессии:

```
marks = [3, 3, 4, 2, 4]
```

и мы хотим узнать, будет ли он отчислен. Выполним для этого, следующую проверку:

```
if 2 in marks:
    print("студент будет отчислен")
```

Вот в этом последнем примере нам бы хотелось в противном случае выдать сообщение, что студент успешно сдал сессию. Это лучше всего реализовать с помощью оператора **else**:

```
marks = [3, 3, 4, 2, 4]
if 2 in marks:
    print("студент будет отчислен")
else:
    print("студент успешно сдал сессию")
```

То есть, смотрите, если срабатывает первое условие, то блок **else** не выполняется. И, наоборот, если условие не срабатывает, то выполняется блок **else**. Это слово можно перевести как «иначе» и получается либо выполнение первого блока, либо второго, но никогда не обоих вместе. То есть, здесь определены два взаимоисключающих случая. Действительно, студент не

Основы Python

может одновременно быть отчисленным и успешно сдавшим сессию (будем полагать, что по собственному желанию он не уходит).

Или другой подобный пример:

```
x = int(input())
if x < 0:
    print("x - отрицательное число")
else:
    print("x - неотрицательное число")
```

Очевидно, что **x** не может быть одновременно отрицательным и неотрицательным, поэтому для проверки такого условия можно использовать оператор **else**.

В последнем примере этого занятия мы проверим введенное число на четность. Сделать это можно так:

```
x = int(input())
if x % 2 == 0:
    print("x - четное число")
else:
    print("x - нечетное число")
```

Надеюсь, из всех этих примеров, вам понятно, что из себя представляет условный оператор **if**, как в нем можно прописывать различные условия и как работает оператор **else**. Вас еще ждет набор практических заданий, а я буду вас ждать на следующем уроке.

§18. Вложенные условия и множественный выбор. Конструкция **if-elif-else**

Мы продолжаем изучение условных операторов языка **Python**. На предыдущем занятии мы познакомились с работой оператора **if** проверки условий. И говорили, что внутри блока этого оператора могут быть любые конструкции языка **Python**, в том числе и другие условия. Давайте посмотрим на примерах, как это работает.

Основы Python



Предположим, что сначала в первом условии нам нужно проверить число на четность, а затем, определить, является ли это число цифрой или каким-либо другим числом:

```
x = int(input())
if x % 2 == 0:
    if 0 <= x <= 9:
        print("x - четная цифра")
    else:
        print("x - четное число")
```

Как видите, это удобно сделать с помощью вложенного условия. И, обратите внимание, оператор **else** здесь относится именно ко второму условию, так как записан на одном уровне с ним. А вот, если бы мы его прописали без отступов:

```
if x % 2 == 0:
    if 0 <= x <= 9:
        print("x - четная цифра")
else:
    print("x - четное число")
```

то он стал бы относиться к первому условию. Программа стала бы работать иначе и, вообще говоря, некорректно для данной задачи.

Основы Python

Я верну отступы для блока **else** и добавлю еще один такой блок для первого условия:

```
x = int(input())
if x % 2 == 0:
    if 0 <= x <= 9:
        print("x - четная цифра")
    else:
        print("x - четное число")
else:
    print("x - нечетное число")
```

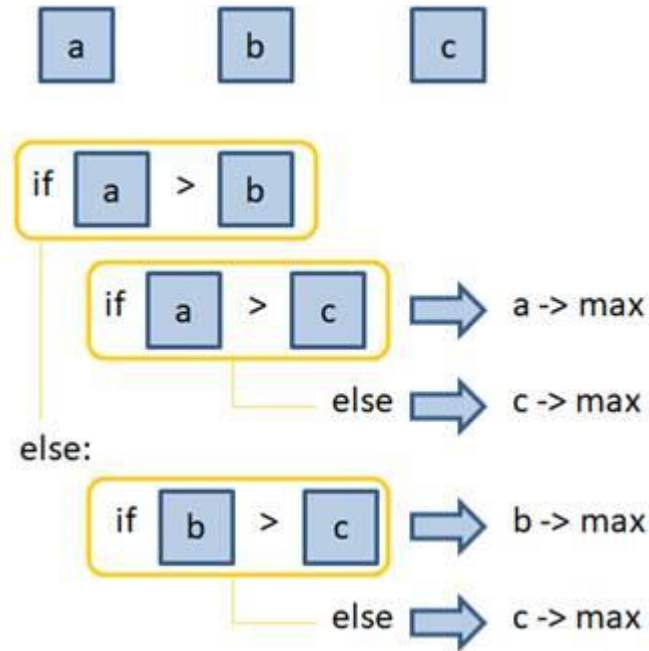
В результате, у нас обрабатываются все возможные исходы для числа **x**.

Другой пример – поиск наибольшего среди трех чисел **a**, **b** и **c**:

```
a = int(input("a: "))
b = int(input("b: "))
c = int(input("c: "))

if a > b:
    if a > c:
        print("a -> max")
    else:
        print("c -> max")
else:
    if b > c:
        print("b -> max")
    else:
        print("c -> max")
```

Основы Python



Думаю, из этих примеров вам стало понятно, как использовать вложенные условия. Конечно, степень вложенности может быть любой, но все будет работать по аналогии. В практике реального программирования лучше избегать больших вложенностей. Нормальным считается до трех вложений. Если у вас выходит больше, то, скорее всего, что-то делаете не так и структуру программы следует пересмотреть.

Оператор elif

Во второй части этого занятия я хочу рассказать вам о способе реализации множественного выбора. **Что это такое?** Представьте, что мы просим пользователя выбрать один из пунктов меню:

1. Курс по Python
2. Курс по C++
3. Курс по Java
4. Курс по JavaScript

Используя текущие знания, мы можем реализовать это так:

```
item = int(input())
```

Основы Python

```
if item == 1:
    print("Выбран курс по Python")
else:
    if item == 2:
        print("Выбран курс по C++")
    else:
        if item == 3:
            print("Выбран курс по Java")
        else:
            if item == 4:
                print("Выбран курс по JavaScript")
            else:
                print("Неверный пункт")
```

Но эту же логику можно реализовать проще и нагляднее, используя еще один условный оператор

elif = else if

То есть, все конструкции с **else if** мы можем просто заменить на **elif**, получим:

```
if item == 1:
    print("Выбран курс по Python")
elif item == 2:
    print("Выбран курс по C++")
elif item == 3:
    print("Выбран курс по Java")
elif item == 4:
    print("Выбран курс по JavaScript")
else:
    print("Неверный пункт")
```

Основы Python

Видите, как это удобнее, когда нам в программе нужно реализовать выбор какого-то одного значения из множества доступных. Причем, последовательность операторов должна быть именно такой:

if – elif – else

После **if** можно опускать любой из них (или оба не прописывать), но порядок следования всегда нужно соблюдать именно таким.

Конечно, помимо проверок на равенство после **elif** можно прописывать любые условия. Например, вводим с клавиатуры целое положительное **x** и для него определяем число десятков:

```
x = int(input())

if x < 0:
    print("x должно быть положительным")
elif 0 <= x <= 9:
    print("x - цифра")
elif 10 <= x <= 99:
    print("x - двузначное число")
elif 100 <= x <= 999:
    print("x - трехзначное число")
```

В целом это все, что я хотел вам рассказать про условный оператор **if**. На следующем занятии мы еще затронем тему тернарных условий. А сейчас вас ждут практические задания по этой теме, после чего переходите к следующему уроку.

§19. Тернарный условный оператор. Вложенное тернарное условие

На этом занятии по курсу языка **Python** мы поговорим о тернарном условном операторе, который появился в версии **3.11.5** Как всегда, вначале поясню, что это за оператор. В простейшем варианте его синтаксис можно представить так:

Основы Python

<значение 1> if <условие> else <значение 2>

Он возвращает значение 1, если условие истинно, а иначе – значение 2. Давайте поясню его работу на конкретном примере. Предположим, у нас имеются две переменные:

```
a = 12  
b = 7
```

и мы хотим для них определить максимальное значение. Используя имеющиеся знания, это можно было бы сделать так:

```
if a > b:  
    res = a  
else:  
    res = b  
  
print(res)
```

А с использованием тернарного оператора это можно реализовать так:

```
res = a if a > b else b
```

Видите, какая простая, понятная и компактная запись в итоге получилась. Это одно из удобств данного оператора. Но, все же, между предыдущей программой с условным оператором **if** и тернарным оператором есть одно принципиальное отличие. Тернарный оператор автоматически возвращает результат. В данном примере – это или переменная **a** или переменная **b**. Тогда как обычный условный оператор предназначен для выполнения блока кода по определенному условию и сам по себе не возвращает никаких значений. То есть, если бы мы попытались записать что-то вроде:

```
d = if a > b:
```


Основы Python

то попросту возникла бы синтаксическая ошибка – так делать нельзя. А вот результат работы тернарного оператора мы, обычно, сохраняем в некой переменной.

Второе важное отличие – в тернарном операторе нет внутренних блоков, где бы мы могли записывать несколько операторов. Вместо **a** и **b** можно прописывать только одну какую-либо конструкцию. Часто это некое значение, или результат работы операторов, например, арифметических:

```
res = a + 2 if a > b else b - 5
```

Так делать вполне допустимо. Или же, можно использовать какие-либо функции:

```
a = -12  
b = -7  
res = abs(a) if a > b else abs(b)
```

То есть, здесь может быть любая конструкция языка **Python**, но только одна. По идее, можно даже записать функцию **print()**:

```
res = print(a) if a > b else print(b)
```

Но тогда переменная **res** будет принимать значение **None**, так как функция **print()** ничего возвращает, то есть, **None**.

Давайте приведу еще один пример. Пусть имеется некая строка и способ преобразования этой строки:

```
s = 'python'  
type = 'upper'
```

Пусть 'upper' означает преобразование строки **s** в верхний регистр. Сделаем это через тернарный оператор, следующим образом:

```
res = s.upper() if type == 'upper' else s
```

Основы Python

Видите, **как элементарно и просто реализуется эта идея?** В этом большой плюс тернарного оператора. Вообще, его можно рассматривать, как некий активный объект, который возвращает определенное значение в зависимости от условия. Благодаря этому, его можно вызывать прямо внутри различных конструкций, например:

```
[1, 2, a if a < b else b, 4, 5]  
"a - " + ("четное" if a % 2 == 0 else "нечетное") + " число"
```

Обратите внимание здесь на круглые скобки. Они необходимы, чтобы следующий оператор + применялся к результирующей строке, а не к тернарному оператору. Если круглые скобки убрать, то последняя добавка пропадет.

Или его можно указать, как один из аргументов функции:

```
max(1, 5, a if a > 0 else b, 4, 5)
```

Я, надеюсь, из этих примеров вам стал понятен принцип использования тернарного условного оператора.

Вложенный тернарный оператор

Так как внутри тернарного оператора можно использовать любые конструкции, то кто нам мешает вложить один тернарный оператор в другой, например, так:

```
<r_1> if <c_1> else <r_2> if <c_2> else <r_3>
```

или так:

```
<r_1> if <c_1> else <r_2> if <c_2> else <r_3> if <c_3> else <r_4>
```

Сразу скажу, что это делается крайне редко и лучше избегать таких вложений, так как восприятие и понимание текста программы резко снижается.

Основы Python

Приведу один пример с такими вложениями. На предыдущем занятии мы с вами записывали вложенные условия для определения максимального из трех чисел. Давайте сделаем то же самое, но через тернарные операторы. Программа будет такой:

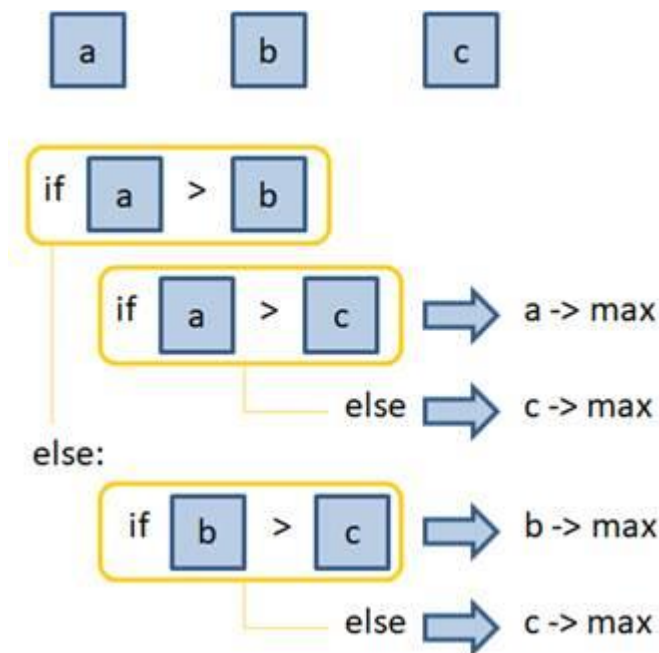
```
a = 2
```

```
b = 3
```

```
c = -4
```

```
d = (a if a > c else c) if a > b else (b if b > c else c)
```

```
print(d)
```



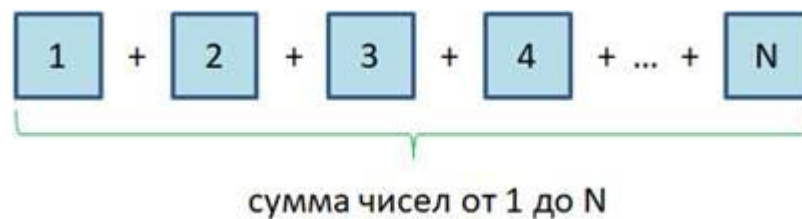
Думаю, общий принцип здесь понятен. Сначала проверяем условие $a > b$, затем, в зависимости от результата проверки, выполняется или левый тернарный оператор, или правый. Причем, опять же здесь обязательно вложенные тернарные операторы следует взять в круглые скобки, чтобы тернарная конструкция **if-else** воспринималась как единое целое во вложениях.

Вам осталось закрепить этот материал практическими заданиями и, затем, переходите к следующему уроку.

Основы Python

§20. Оператор цикла `while`

На этом занятии начнем знакомиться с еще одним ключевым элементом компьютерных программ – **циклами**. Вначале, давайте я на простом примере покажу, о чем идет речь. Представим, что нам нужно вычислить сумму целых чисел от 1 до N . Причем, N может быть сколько угодно большой: тысяча, миллион и так далее. Понятно, что мы не можем здесь просто записать операторы сложения чисел друг за другом для вычисления этой последовательности. Тем более, что на момент написания программы, N может быть неизвестна. Вот в таких ситуациях нам на помощь, как раз, и приходят циклы.



Циклы позволяют реализовывать некие повторяющиеся действия. Например, предположим, что маленькие панды прыгают с горки в течение часа, пока мама-панда не позовет всех к столу – кушать. На уровне текста это можно записать, так:

цикл (пока не прошел час):
 прыгаем с горки

То есть, пока **истинно** условие, цикл работает, как только условие становится **ложным** – прошел час, цикл завершается. Ровно так работает цикл **while**, о котором и пойдет речь на нашем занятии. Он имеет, следующее определение (синтаксис):

Основы Python

```
while <условие> :  
    оператор 1  
    оператор 2  
    ...  
    оператор N
```

заголовок

тело цикла

Вначале записывается ключевое слово **while**, затем, условие работы цикла, ставится двоеточие для определения блока операторов, работающих внутри этого цикла. Такой блок еще называют телом цикла, а ключевое слово **while** с условием – заголовком цикла.

Обратите внимание на форматирование. Здесь также, как и в условных операторах, набор операторов внутри тела цикла должны иметь одинаковые отступы относительно ключевого слова **while**.

Давайте вернемся к исходной задаче – вычисления суммы чисел от 1 до N и посмотрим, как здесь нам поможет цикл **while**. Вначале определим три вспомогательные переменные: N – значение последнего слагаемого; s – для хранения вычисленной суммы (начальное значение 0); i – значение текущего слагаемого (начинается с 1):

```
N = 1000  
s = 0  
i = 1
```

Далее, так как сумму нужно вычислять, пока слагаемое i не достигнет значения N, то условие цикла можно определить, следующим образом:

```
while i <= N:
```

А внутри цикла будем выполнять следующие действия:

```
s += i  
i += 1
```

Основы Python

Вначале i равна 1 и эта единица прибавляется к сумме s . После чего i увеличивается на 1 и становится равной 2. Затем, выполняется проверка условия цикла. Пока оно у нас истинно, поэтому снова попадаем внутрь тела цикла и к s прибавляется уже значение 2, а i опять увеличиваем на 1 и оно становится равным 3. И так до тех пор пока i не станет больше N . К этому моменту мы просуммируем все числа и результат будет храниться в переменной s . Вот принцип работы циклов, причем, во всех языках программирования, не только в Python.

```
s = 0
i = 1
N = 1000
while i <= N:
    s += i
    i += 1
```



итерация цикла

Также однократное выполнение тела цикла в программировании называют **итерацией**. Я буду часто использовать этот термин, поэтому привел его, чтобы вы меня правильно понимали.

Давайте реализуем теперь эту программу на Python и посмотрим как она работает.

Возможно, у вас возник вопрос: а **какие условия можно прописывать в циклах?** В действительности, все те же самые, что и в условных операторах. В том числе и составные. Например, давайте будем вычислять сумму пока не дойдем до слагаемого N или до значения 50. Так как цикл работает, пока **истинно** условие, то его следует записать, так:

```
while i <= N and i <= 50:
    ...
```

Смотрите, мы здесь указали делать цикл пока i меньше или равна N и меньше или равна 50. Если хотя бы одно из этих подусловий окажется ложным, то и все составное условие также станет ложным и цикл завершится. В результате,

Основы Python

`i` достигнет или `N` или `50`. Вот это нужно хорошо себе представлять: в циклах прописываются условия их работы, а не завершения.

Используя этот же цикл, мы легко можем поменять условие задачи и вычислить сумму чисел, которые меняются через один:

1, 3, 5, 7, ...

Для этого достаточно счетчик `i` увеличивать не на `1`, а сразу на два:

```
i += 2
```

При этом, условие цикла можно оставить прежним. Он завершится, как только `i` превысит `N` или `50`. Здесь уже нет гарантии, что последнее слагаемое будет именно `N` или `50`, но нам это и не нужно, мы лишь указываем завершить цикл, когда превысим одно из этих значений.

Конечно, внутри тела цикла можно записывать любые операторы, в том числе и функцию `print()`. Давайте, например, отобразим в консоли цифры от `0` до `9`:

```
i = 0
while i < 10:
    print(i)
    i += 1
```

Я здесь указал условие `i` меньше `10`, а не `i <= 9`, так как оператор `<` работает несколько быстрее оператора `<=`. Поэтому на практике предпочтительно, по возможности, применять не составные, а простые операторы: меньше, больше, равно и не равно. Хотя использование `<=` или `>=` не критично и вполне допустимо. Но, все же, по возможности, лучше прописывать простые операторы в условиях циклов.

Если нам нужно реализовать убывающую последовательность чисел, например:

Основы Python

-1, -2, -3, ..., -N

то это делается аналогично, только с уменьшающимся счетчиком:

```
N = -10
i = -1

while i >= N:
    print(i)
    i -= 1
```

Вообще, счетчик в цикле можно менять произвольным образом, например, умножая на два:

```
i *= 2
```

Здесь нет никаких ограничений.

В заключение этого занятия приведу еще пару примеров с оператором цикла **while**. Предположим, что пользователь вводит пароль, до тех пор, пока не введет верно. Это можно сделать, следующим образом:

```
pass_true = 'password'
ps = ""

while ps != pass_true:
    ps = input("Введите пароль: ")

print("Вход в систему осуществлен")
```

Вначале мы задаем пароль для проверки, затем, переменную **ps**, которая хранит введенный пользователем пароль и делаем цикл, пока пароли не совпадают. Обратите внимание, условие цикла – пока пароли не совпадают. Опять же, всегда следует помнить, что мы прописываем условие работы, а не остановки цикла. Поэтому здесь нам нужно проверять на несовпадение паролей и запрашивать у пользователя пароль, пока они не совпадут.

Основы Python

Наконец, внутри цикла **while** можно прописывать, например, и условные операторы. Давайте выведем все числа, кратные 3, которые нам встретятся при переборе целых значений от 1 до N:

```
N = 20
i = 1

while i <= N:
    if i % 3 == 0:
        print(i)

    i += 1
```

Как видите, все достаточно просто и очевидно. Здесь главное не забывать о правильном форматировании текста программы: функция **print()** находится внутри условия **if**, поэтому перед ней необходимы отступы. А сам блок условия имеет отступы относительно оператора **while**. Так определяются вложенные блоки операторов в языке **Python**. И об этом всегда следует помнить.

На этом мы завершим первое знакомство с оператором цикла **while**. На следующем уроке продолжим эту тему. А для закрепления текущего материала не забудьте выполнить несколько практических заданий. До встречи на следующем уроке.

§21. Операторы циклов **break**, **continue** и **else**

Мы продолжаем курс по языку **Python**. Это занятие начнем со знакомства операторов **break** и **continue** цикла **while**. Суть этих операторов, в следующем:

- **break** – досрочное завершение цикла;
- **continue** – пропуск одной итерации цикла.

Давайте сначала рассмотрим работу оператора **break**. Если в качестве условия цикла записать значение **True**:

Основы Python

```
print("запуск цикла")

i = 0
while True:
    i += 1

print("завершение цикла")
```

То такой цикл будет работать **«вечно»**, он сам никогда не остановится и строчку **«завершение цикла»** мы не увидим. Программу в этом случае можно только прервать и полностью остановить. Однако, если записать в тело цикла оператор **break**, то он прервется, как только встретится:

```
while True:
    i += 1
    break
```

Обратите внимание, оператор **i += 1** один раз будет выполнен, а прерывание сработает только в строчке с оператором **break**.

Но вы скажете, конечно, зачем писать программы с **«вечными»** циклами, **нужно корректно прописывать условие и все будет хорошо?** Часто, именно так и следует поступать. Но бывают ситуации, когда все же требуется досрочно прерывать работу цикла. Например, у нас имеется список с числами:

```
d = [1, 5, 3, 6, 0, -4]
```

И мы хотим определить, есть ли среди них хотя бы одно четное значение. Здесь удобно воспользоваться оператором **break**. Как только встретим первое четное значение, дальнейшую проверку можно не проводить:

```
d = [1, 5, 3, 6, 0, -4]
flFind = False
i = 0
```

Основы Python

```
while i < len(d):
    flFind = d[i] % 2 == 0
    if flFind:
        break

    i += 1

print(flFind)
```

Конечно, эту же программу можно было бы построить и без **break**, используя составное условие:

```
i = 0
while i < len(d) and d[i] % 2 != 0:
    i += 1

flFind = i != len(d)
```

Но, на мой взгляд, первый вариант с **break** красивее и понятнее, чем второй. Хотя, все дело вкуса – кому, как нравится.

Я, думаю, что принцип работы оператора **break** понятен. Он прерывает работу любого оператора цикла, как только выполнение программы перейдет к нему.

Следующий оператор **continue** позволяет пропускать одну итерацию тела цикла. Давайте, опять же, на конкретном примере посмотрим, где и как его целесообразно применять.

Предположим, что мы просим пользователя вводить с клавиатуры числа и все нечетные значения суммируем. Как только пользователь введет **0**, подсчет суммы прекращается. Реализовать эту программу удобно, следующим образом:

```
s = 0
d = 1
```

Основы Python

```
while d != 0:
    d = int(input("Введите целое число: "))
    if d % 2 == 0:
        continue

    s += d
    print("s = " + str(s))
```

Смотрите, мы здесь на каждой итерации цикла проверяем, если текущее значение **d** четное, то последующий подсчет суммы и вывод ее в консоль не выполняется, пропускается. При этом, цикл продолжает свою работу и переходит к следующей итерации, пока пользователь не введет число **0**. Благодаря оператору **continue** мы основную логику программы прописываем непосредственно в теле цикла, а не во вложенном блоке условного оператора. То есть, если программу переписать без **continue**, то она бы выглядела так:

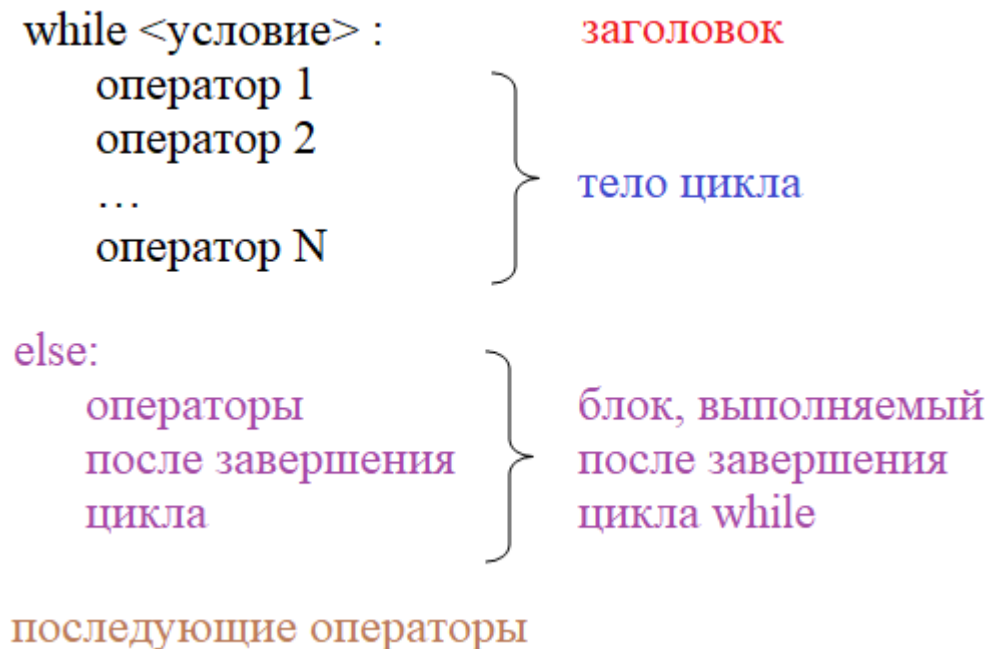
```
s = 0
d = 1
while d != 0:
    d = int(input("Введите целое число: "))
    if d % 2 != 0:
        s += d
    print("s = " + str(s))
```

Здесь получается массивный вложенный блок операторов. Конечно, на производительность это никак не влияет, но, все же такие вложения лучше избегать, так программа становится менее читабельной и удобной для дальнейшего редактирования.

В заключение этого занятия рассмотрим третий оператор циклов **else**. Да, мы уже говорили об этом операторе, когда рассматривали условия. У циклов он тоже есть и сейчас разберем, как работает.

Основы Python

Синтаксис этого оператора, следующий. После тела цикла прописывается необязательный оператор **else**, в котором указываются операторы, исполняющиеся после завершения цикла:



И здесь возникает вопрос: **а чем блок `else` отличается от блока операторов, просто идущих после блока `while`?** Ведь когда цикл `while` завершается, мы так и так переходим к последующим операторам! Но обратите внимание вот на эту фразу «штатное завершение цикла». Штатное завершение – это когда условие цикла стало равно **False** и оператор `while` прекращает свою работу. Только в этом случае мы перейдем в блок `else`. Давайте я покажу это на конкретном примере. Предположим, что мы вычисляем сумму вида:

$$S = 1/2 + 1/3 + 1/4 + 1/10 + \dots + 1/0$$

И если в этой сумме встречается деление на ноль, то вычисления следует прервать. Реализуем эту программу, следующим образом:

```
S=0
i=-10
```

Основы Python

```
while i < 100:
    if i == 0:
        break

    S += 1/i
    i += 1
else:
    print("Сумма вычислена корректно")

print(S)
```

Смотрите, если здесь при вычислении суммы встречается деление на **0**, то срабатывает **break** и цикл досрочно прерывается, то есть, завершается в нештатном режиме. В этом случае блок **else** пропускается и мы не видим сообщения, что сумма вычислена корректно. Если же все проходит штатно и цикл завершается по своему условию, то в консоли появляется сообщение «Сумма вычислена корректно», означающее выполнение блока **else**.

Надеюсь, из этого примера вам стало понятно назначение оператора **else** цикла **while**. Также из этого занятия вы должны себе хорошо представлять работу операторов **break** и **continue**. После закрепления этого материала практическими заданиями, жду всех вас на следующем уроке.

§22. Оператор цикла for. Функция range()

На предыдущих занятиях мы с вами познакомились с оператором цикла **while**, а также вспомогательными операторами **break**, **continue** и **else**. На этом занятии вы узнаете о втором операторе цикла **for**, который довольно часто используется в **Python**.

Он имеет следующий синтаксис:

```
for <переменная> in <итерируемый объект>:
    оператор 1
    оператор 2
```

Основы Python

...
оператор N

С его помощью очень легко реализовывать перебор, так называемых, итерированных объектов. Что это такое, мы будем говорить на одном из следующих занятий, а сейчас, вам достаточно знать, что это объекты, состоящие из множества элементов, которые можно перебирать. Например, списки или строки.

Как всегда, постичь магию работу этого оператора лучше всего на конкретных примерах. Пусть у нас имеется список:

```
d = [1, 2, 3, 4, 5]
```

И мы хотим перебрать все его элементы. Через оператор цикла **for** сделать это можно, следующим образом:

```
for x in d:  
    print(x)
```

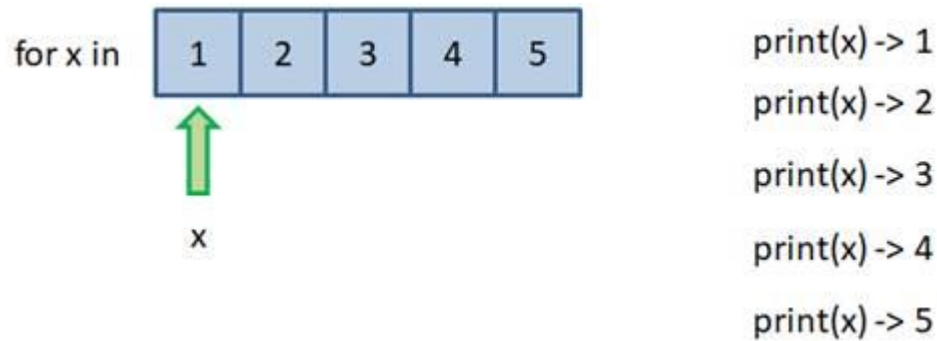
Или, вместо списка, можно взять строку:

```
for x in "python":  
    print(x)
```

Тогда переменная **x** на каждой итерации будет ссылаться на очередной символ этой строки.

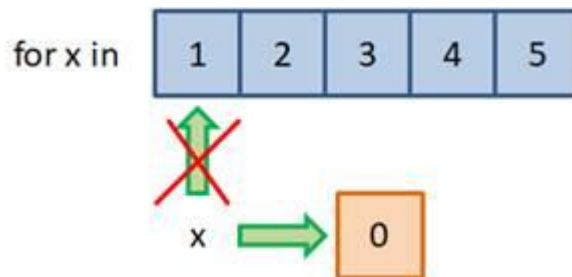
Но давайте посмотрим, как все это работает в деталях. Вернемся к примеру со списком. На первой итерации переменная **x** будет ссылаться на первый элемент со значением **1**. Соответственно, функция **print()** выводит это значение в консоль. На следующей итерации переменная **x** ссылается уже на второй элемент и **print()** выводит значение **2**. И так до тех пор, пока не будет достигнут конец списка.

Основы Python



В этой демонстрации ключевое, что переменная **x** ссылается на элемент списка. То есть, если мы захотим изменить значение в списке, используя переменную **x**, например, вот так:

```
for x in [1, 2, 3, 4, 5]:  
    x = 0
```



то ничего не получится. Здесь **x** просто будет ссылаться на другой объект со значением **0**, но элементы списка это никак не затронет. То есть, в такой реализации оператора цикла **for** мы можем лишь перебирать значения элементов и что-то с ними делать, например, вычислять их произведение:

```
d = [5, 2, 4, 3, 1]
```

```
p = 1  
for x in d:  
    p *= x  
  
print(p)
```

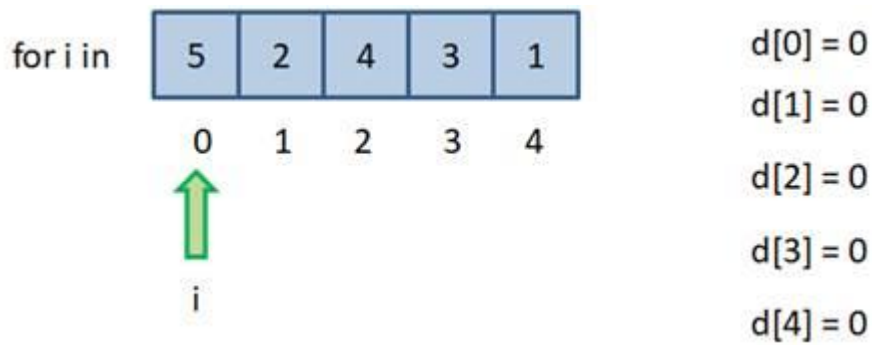

Основы Python

А как тогда менять значения элементов в списке с помощью **for**? Для этого к элементам списка нужно обращаться по индексу. То есть, цикл должен перебирать не элементы списка, а его индексы:

```
d = [5, 2, 4, 3, 1]
```

```
for i in [0, 1, 2, 3, 4]:  
    d[i] = 0
```

```
print(d)
```



В этом случае, мы на каждой итерации цикла, обращаемся сначала к первому элементу списка, присваиваем ему ноль, затем, ко второму элементу, присваиваем ему ноль и так для всех остальных элементов. В итоге меняется сам список.

Однако описывать индексы через еще один список, далеко не лучшая практика. Для подобных целей в **Python** существует специальная функция **range()**, которая генерирует арифметическую последовательность чисел с параметрами:

range(start, stop, step)

Например, для генерации последовательности от **0** до **4**, функцию **range()** можно записать в виде:

```
range(5)
```

Основы Python

```
range(0, 5)  
range(0,5,1)
```

То есть, последнее значение **5** не включается в диапазон. Чтобы увидеть сгенерированные числа, преобразуем их последовательность в список с помощью известной нам функции `list()`:

```
list(range(5))  
list(range(0, 5))  
list(range(0, 5, 1))
```

Если же мы запишем:

```
list(range(0))
```

то получим пустой список, так как значения здесь начинаются с нуля и заканчиваются нулем, при этом, ноль в интервал не входит.

То же самое произойдет при указании любого отрицательного значения, например:

```
list(range(-5))  
list(range(0, -5))
```

Но **как нам тогда формировать последовательность отрицательных значений?** Очень просто, в качестве стартового значения нужно указать число меньше конечного значения:

```
list(range(-10, -5))
```

Или с шагом:

```
list(range(-10, -5, 2))
```

Но если указать отрицательный шаг:

```
list(range(-10, -5, -2))
```

Основы Python

то снова увидим пустой список. Я, думаю, **вы догадались почему?** Теперь, мы начинаем двигаться от **-10** в меньшую сторону и значение **-5** становится недостижимым. В этом случае функция **range()** не выдает никаких значений. Чтобы поправить ситуацию, в качестве конечного значения нужно записать число меньше **-10**, например, **-20**:

```
list(range(-10, -20, -2))
```

Вот так генерируются последовательности в обратном порядке. То же самое можно проделать и в положительной области:

```
list(range(5, 0, -1))
```

Обратите внимание, мы начинаем с **5** и заканчиваем **1**. Здесь конечное значение **0** не включается в диапазон. Если нам нужно дойти до нуля, то в данном случае следует указать **-1** в качестве конечного значения:

```
list(range(5, -1, -1))
```

Вот это всегда следует помнить при работе с функцией **range()** – конечное значение не включается в диапазон.

Итак, теперь, когда мы с вами узнали, как работает функция **range()**, перепишем нашу программу с перебором элементов списка по их индексам, следующим образом:

```
d = [5, 2, 4, 3, 1]
```

```
for i in range(5):  
    d[i] = 0
```

```
print(d)
```

Здесь нам не нужно функцию **range()** превращать в список, оператор цикла **for** умеет перебирать любые итерируемые объекты, а **range()**, как раз

Основы Python

возвращает такой объект, поэтому переменная `i` будет принимать значения от 0 до 4 включительно и мы видим, что все значения списка стали равны нулю.

И последний штрих в этой программе. Число 5 лучше заменить вызовом функции определения длины списка:

```
len(d)
```

Тогда получим универсальную программу, работающую со списком любой длины:

```
for i in range(len(d)):
    d[i] = 0
```

В заключение этого занятия приведу еще один пример использования цикла `for` для вычисления суммы ряда:

$$S = 1/2 + 1/3 + 1/4 + \dots + 1/1000$$

Программу можно реализовать, следующим образом:

```
S = 0

for i in range(2, 1001):
    S += 1 / i

print(S)
```

На этом мы завершим наше первое знакомство с оператором цикла `for` и функцией `range()`. На данный момент вам нужно хорошо понимать, как перебираются списки и строки, как формировать арифметические последовательности функцией `range()` и как ее можно использовать совместно с оператором цикла `for`. Следующий урок я целиком посвящу примерам использования этого оператора цикла со списками и строками, а пока закрепите текущий материал практическими заданиями.

Основы Python

§23. Примеры работы оператора цикла for. Функция enumerate()

Мы продолжаем знакомиться с оператором цикла **for** языка **Python**. Сегодня мы с вами рассмотрим несколько примеров использования цикла **for** при решении разных задач. И **первая задача** – вычислить факториал от натурального числа **n**. Пусть пользователь вводит натуральное целое число, а мы будем выдавать значение его факториала:

```
n = int(input("Введите натуральное число не более 100: "))

if n < 1 or n > 100:
    print("Неверно введено натуральное число")
else:
    p = 1
    for i in range(1, n+1):
        p *= i

    print(f"Факториал {n}! = {p}")
```

Обратите внимание на запись функции **range()**. Чтобы она выдавала последовательность целых чисел от **1** до **n** включительно, необходимо записать **n+1**, так как последнее значение не включается в диапазон.

Следующий пример – отображение символов ***** в виде елочки. Сделать это можно очень просто:

```
for i in range(1, 7):
    print('*' * i)
```

```
*
**
***
****
*****
*****
```

Основы Python

Здесь счетчик `i` будет принимать значения 1, 2, 3, 4, 5 и 6. Соответственно, функция `print()` будет дублировать символ `*` `i` раз. В итоге, в первой строке будет одна звездочка, во второй – две и так до шести звездочек. Видите, как просто можно реализовать такую программу, используя оператор цикла и оператор работы со строками. Именно поэтому, все, что мы проходим, нужно хорошо запоминать и применять по мере необходимости.

Давайте, теперь, соединим все слова списка в одно предложение.

Предположим, что у нас имеется список со словами:

```
words = ["Python", "дай", "мне", "силы", "пройти", "этот", "курс", "до", "конца"]
```

Которые объединим через цикл `for`:

```
s = ""
for w in words:
    s += ' ' + w

print(s)
```

У нас здесь получается вначале строки пробел. Убрать его можно двумя способами. Либо вызвать метод строки:

```
s = s.lstrip()
```

Либо, сделать так, чтобы он и не появлялся:

```
s = ""
flFirst = True
for w in words:
    s += (' ' if flFirst else '') + w
    flFirst = False
```

Но, вот этот второй вариант получается более корявым и медленным по скорости работы. Поэтому, лучше воспользоваться методом `lstrip()`.

Основы Python

А вот внимательный ученик с упреком заявил бы, что все это можно сделать еще проще – с помощью уже знакомого нам метода `join()`:

```
print(" ".join(words))
```

И будет абсолютно прав! Те из вас, кто думал также – просто красавчики! Так держать!

Следующая задача. В списке:

```
digs = [4, 3, 100, -53, -30, 1, 34, -8]
```

все двузначные числа заменить нулями. Используя те знания, что у нас сейчас есть, это можно сделать так:

```
for i in range(len(digs)):
    if 10 <= abs(digs[i]) <= 99:
        digs[i] = 0
```

```
print(digs)
```

Обратите внимание, что мы здесь в цикле перебираем индексы элементов списка, а не сами элементы. Благодаря этому, в цикле получаем возможность изменять значение *i*-го элемента.

Как видите, в этой задаче нам потребовалось в цикле знать и индекс элемента и его значение. Именно для таких случаев в **Python** существует специальная функция:

индекс, значение = enumerate(объект)

которая возвращает пару (индекс, значение). Используя эту функцию, можно переписать наш пример, следующим образом:

```
for i, d in enumerate(digs):
    if 10 <= abs(d) <= 99:
        digs[i] = 0
```

Основы Python

Для меня такой вариант выглядит более читабельным и удобным. Во всем остальном эта программа подобна предыдущему варианту.

Последний пример, который я приведу на этом занятии – преобразование кириллицы в латиницу. Такая задача иногда возникает при создании сайтов. Вначале определим список замен для соответствующих русских букв (по алфавиту от **а** до **я**):

```
t = ['a', 'b', 'v', 'g', 'd', 'e', 'zh',  
     'z', 'i', 'y', 'k', 'l', 'm', 'n', 'o', 'p',  
     'r', 's', 't', 'u', 'f', 'h', 'c', 'ch', 'sh',  
     'shch', ' ', 'y', ' ', 'e', 'yu', 'ya'  
]
```

Затем, определим кодовое значение для первой буквы этого списка:

```
start_index = ord('a')
```

Далее, зададим строку и переменную **slug**, где будем формировать строку на латинице:

```
title = "Программирование на Python - лучший курс"  
slug = ""
```

Затем, само преобразование в цикле, перебирая каждый символ исходной строки (причем, предварительно, строку мы переводим в нижний регистр – все буквы становятся малыми):

```
for s in title.lower():  
    if 'a' <= s <= 'я':  
        slug += t[ord(s) - start_index]  
    elif s == 'ё':  
        slug += 'yo'  
    elif s in ' !?,:.':  
        slug += '-'  
    else:
```


Основы Python

```
slug += s
```

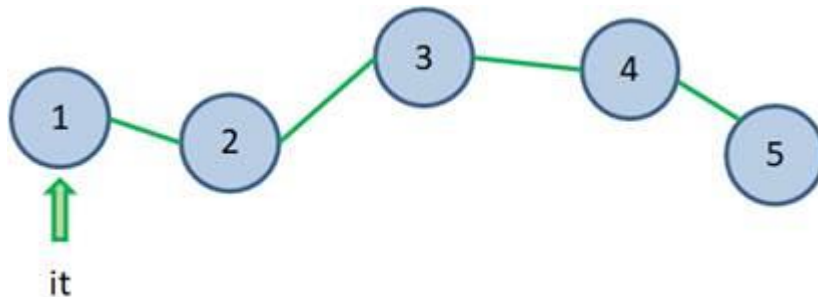
После преобразования следует удалить все подряд идущие символы дефиса:

```
while slug.count('--'):
    slug = slug.replace('-', '-')
print(slug)
```

Конечно, это не самая оптимальная реализация данной задачи. Лучше ее делать с использованием, так называемых, регулярных выражений. Но, используя только наши текущие знания и как пример она очень хорошо вписывается в этот урок.

§24. Итератор и итерируемые объекты. Функции `iter()` и `next()`

На этом занятии мы узнаем, что такое итерируемые объекты и познакомимся со способами их перебора. Мы уже знаем, что в **Python** существуют объекты, содержащие последовательность некоторых элементов. Например, строки и списки. Так вот, существует универсальный механизм для перебора элементов этих и других подобных им объектов. Реализуется он через специальную конструкцию под названием **итератор**. То есть, каждый итерируемый объект предоставляет доступ к своим элементам через итератор. С помощью этого итератора можно один раз пройти по всем элементам коллекции от начала до конца.



Основы Python

Чтобы получить доступ к итератору объекта, например, списка, нужно вызвать специальную функцию `iter()` и первым аргументом указать итерируемый объект. Например, возьмем список:

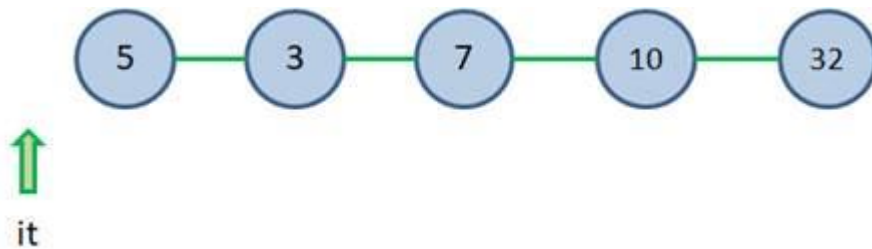
```
d = [5, 3, 7, 10, 32]
```

и для него вызовем функцию `iter()`:

```
iter(d)
```

Смотрите, нам был возвращен объект-итератор для списка. Сохраним его через переменную `it`:

```
it = iter(d)
```



Все, теперь у нас есть итератор для однократного перебора элементов списка `d`. Далее, чтобы перебрать значения итерируемого объекта, используется функция `next()`, которой следует передать объект-итератор:

```
next(it)
```

Она осуществляет переход к следующему элементу (при первом вызове переходим к первому элементу) и возвращает значение этого элемента. Видим значение **5**. Вызовем эту функцию еще раз:

```
next(it)
```

Итератор был перемещен на следующий элемент и возвратилось значение **3**. И так можно пройти до конца нашего списка:

```
next(it)
```

Основы Python

```
next(it)  
next(it)
```

Если для последнего элемента вызвать функцию еще раз:

```
next(it)
```

то получим ошибку **StopIteration**.

Когда итератор дошел до конца коллекции, его уже нельзя вернуть назад и пройти все элементы заново. Для этого придется создавать новый итератор с помощью функции **iter()**:

```
it = iter(d)
```

и заново перебирать все элементы:

```
next(it)
```

Так работает этот механизм перебора элементов итерируемого объекта. Причем, он универсален и не зависит от типа объекта: это может быть и список и строка и любой другой перебираемый объект. Давайте я покажу, как это будет работать со строкой:

```
s = "python"
```

Также создаем итератор:

```
it_s = iter(s)
```

и перебираем символы функцией **next()**:

```
next(it_s)  
next(it_s)
```

Основы Python

Как видите, итератору все равно что перебирать, главное, чтобы сам объект поддерживал этот механизм, то есть, был итерируемым. Также следует иметь в виду, что доступ к элементам через итератор и по индексу:

```
s[2]
```

это совершенно разные способы обращения к элементам. К тому же не у всех итерируемых объектов есть возможность указания индексов.

Помимо строк и списков к итерируемым объектам также относится ранее рассмотренная функция:

```
r = range(5)
```

Для нее мы можем получить итератор:

```
it = iter(r)
```

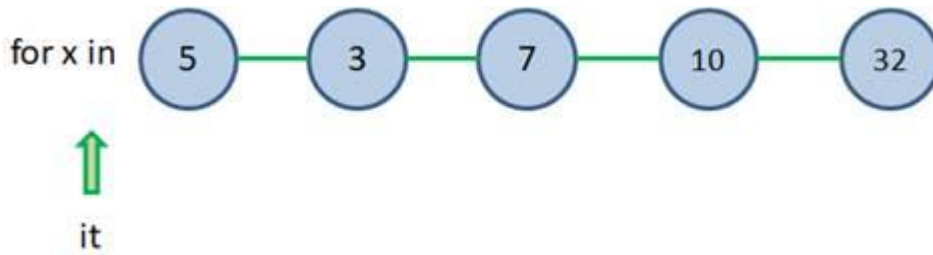
и перебирать значения функцией `next()`:

```
next(it)  
next(it)
```

Когда я вам все это рассказываю, у вас, наверное, постоянно крутится вопрос: а **зачем это надо?** Мы и без всяких итераторов можем обращаться к элементам строк, списков и даже объекта `range()`? Да, все верно, если нам нужно извлечь какое-либо значение, скажем, из списка, то следует использовать индексы или срезы. То же самое и со строками. Но если нам в программе нужно перебирать итерируемые объекты самых разных типов, то единственный универсальный и безопасный способ это сделать – использовать итераторы. Например, так происходит в операторе цикла `for`. Мы можем ему указать перебрать любой итерируемый объект и он должен «**уметь**» это делать вне зависимости от типа этого объекта. Поэтому он обращается к итератору и перебирает элементы через этот универсальный механизм, пока не возникнет исключение `StopIteration`. Именно поэтому мы в цикле можем с легкостью перебирать и списки:

Основы Python

```
for x in [5, 3, 7, 10, 32]:  
    print(x)
```



И строки:

```
for x in "python":  
    print(x)
```

И объект `range()`:

```
for x in range(1, 6):  
    print(x)
```

И вообще любой другой тип объектов, которые поддерживают итератор, то есть, являются итерируемыми.

А что будет, если мы попробуем получить итератор для не итерированного объекта, например, числа:

```
iter(5)
```

В этом случае получим ошибку, что объект типа `int` не является итерируемым, то есть, не поддерживает итератор и перебор элементов. Действительно, число – это одно значение и перебирать тут нечего, поэтому оно относится к неитерируемым объектам. Соответственно, и в цикле `for` мы не можем указывать такие элементы:

```
for x in 5:  
    print(x)
```

Основы Python

Перебрать одно число не получится. Здесь можно указывать только итерируемые объекты.

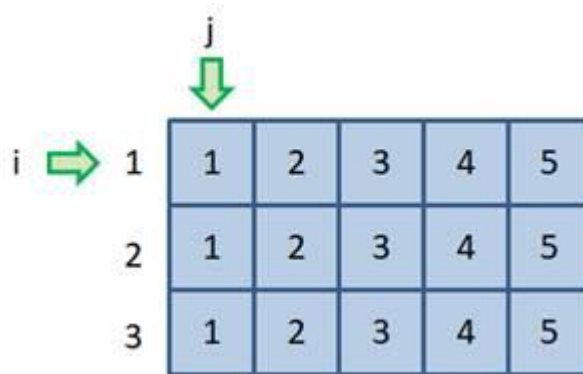
Надеюсь, из этого занятия вам стало понятно, что такое итератор и итерируемые объекты, как получать итератор с помощью функции `iter()` и как перебирать элементы функцией `next()`. Закрепите этот материал практическими заданиями, после чего жду всех вас на следующем уроке.

§25. Вложенные циклы. Примеры задач с вложенными циклами

На предыдущих занятиях мы с вами в деталях познакомились с работой операторов циклов `for` и `while`. На этом занятии сделаем следующий важный шаг и узнаем, как реализуются и работают вложенные циклы.

Само это название уже говорит, что один оператор цикла можно вложить в другой. Это могут быть два цикла `for` или два цикла `while` или смешанные варианты. Давайте вначале разберемся, как работают эти конструкции. Принцип у всех един, поэтому, для простоты, я возьму два цикла `for` (один вложен в другой). Эти циклы будут просто пробегать диапазоны чисел, первый от 1 до 3, а второй – от 1 до 5:

```
for i in range(1, 4):  
    for j in range(1, 6):  
        print(f'i = {i}, j = {j}', end=' ')  
    print()
```



	j ↓	1	2	3	4	5
i → 1		1	2	3	4	5
2		1	2	3	4	5
3		1	2	3	4	5

Основы Python

Во втором вложенном цикле мы будем выводить значения `i` и `j` в строку без перехода на новую строку. А после завершения работы вложенного цикла вызовем функцию `print()`, как раз, для перевода курсора на новую строку. В результате выполнения этой программы, мы получим таблицу значений переменных `i` и `j`.

Почему получились именно такие значения? Вначале у нас счетчик `i` принимает значение `1`, а счетчик `j` пробегает числа от `1` до `5`, в итоге получаем первую строку. После завершения вложенного цикла, срабатывает функция `print()` и курсор переходит на новую строку. После этого переходим ко второй итерации первого цикла и `i = 2`. Счетчик `j` снова проходит значения от `1` до `5` и получаем вторую строку. На следующей итерации первого цикла `i = 3`, `j` проходит от `1` до `5` и получаем третью строку. То есть, у нас вложенный цикл `for` трижды запускался заново и каждый раз `j` изменялось от `1` до `5`. Это и есть принцип работы вложенного цикла – на каждой итерации он обрабатывает снова и снова, пока не завершится первый цикл.

Теперь **второй вопрос** – **зачем все это нужно?** Давайте представим, что у нас есть вложенный (двумерный) список:

```
a = [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]
```

(О таких списках мы с вами уже говорили и вам здесь все должно быть понятно). Так вот, если мы будем перебирать его элементы с помощью одного оператора цикла `for`:

```
for row in a:  
    print(row, type(row))
```

То переменная `row` будет ссылаться сначала на первый вложенный список, затем, на второй и потом на третий. Но, так как `row` ссылается на список, то есть, на итерируемый объект, то нам ничто не мешает перебрать его элементы с помощью второго, вложенного цикла `for`:

```
for row in a:  
    for x in row:
```

Основы Python

```
print(x, type(x), end=' ')\nprint()
```

Как видите, теперь в консоль выводятся числа типа **int**, то есть, мы обращаемся непосредственно к элементам этого двумерного списка.

Ну, хорошо, а все-таки, **зачем это может быть нужно?** Например, так можно выполнить сложение значений из двух одинаковых двумерных списков:

```
a = [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]\nb = [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]]
```

И сформировать на их основе третий список:

```
c = []
```

следующим образом:

```
for i, row in enumerate(a):\n    r = []\n    for j, x in enumerate(row):\n        r.append(x + b[i][j])\n\n    c.append(r)\n\nprint(c)
```

1	2	3	4		1	1	1	1	
2	3	4	5	+	2	2	2	2	=
3	4	5	6		3	3	3	3	

Я здесь воспользовался еще одной уже знакомой нам функцией **enumerate()**, которая возвращает индекс и значение текущего элемента. Это удобно для реализации данной программы. Внутри первого цикла мы каждый раз

Основы Python

создаем новый пустой список и с помощью метода **append()** добавляем в его конец новый элемент как сумму значений из списков **a** и **b**. Полученную строку (список **r**) мы, затем, добавляем в основной список **c**. Так вычисляется сумма значений элементов двух одинаковых списков **a** и **b**.

Как видите, для реализации данной программы нам потребовался вложенный оператор цикла **for**. И это лишь один маленький пример. Другой пример, пусть у нас имеется текст, представленный в виде списка:

```
t = ["– Скажи-ка, дядя, ведь не даром",  
     "Я Python выучил с каналом",  
     "Балакирев что раздавал?",  
     "Ведь были ж задания боевые",  
     "Да, говорят, еще какие!",  
     "Недаром помнит вся Россия",  
     "Как мы рубили их тогда!"  
]
```

Здесь в строках присутствуют два и более пробелов. Наша задача удалить их и оставить только один. Сделаем это с помощью вложенных циклов. В первом цикле **for** будем перебирать строки – элементы списка, а во втором (вложенном) цикле **while** удалять лишние пробелы:

```
for i, line in enumerate(t):  
    while line.count(' '):  
        line = line.replace(' ', ' ')  
  
    t[i] = line  
  
print(t)
```

В качестве условия цикла мы здесь вызываем метод **count()**, который подсчитывает число фрагментов из двух пробелов подряд. Как только их станет **0** – это будет означать **False** и цикл завершится. Преобразованная

Основы Python

строка становится новым i -м элементом списка и в конце результат выводим в консоль.

Следующий пример. Предположим, вначале мы формируем вложенный список размером $M \times N$ элементов. Причем, M , N вводим с клавиатуры. Вначале сформируем список, состоящий из всех нулей:

```
M, N = list(map(int, input("Введите M и N: ").split()))

zeros = []
for i in range(M):
    zeros.append([0]*N)

print(zeros)
```

А после этого все элементы заменим на единицы, используя вложенные циклы:

```
for i in range(M):
    for j in range(N):
        zeros[i][j] = 1
```

Как видите, в целом, все достаточно просто.

И последний пример. Пусть у нас имеется квадратный список (размерности совпадают):

```
A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

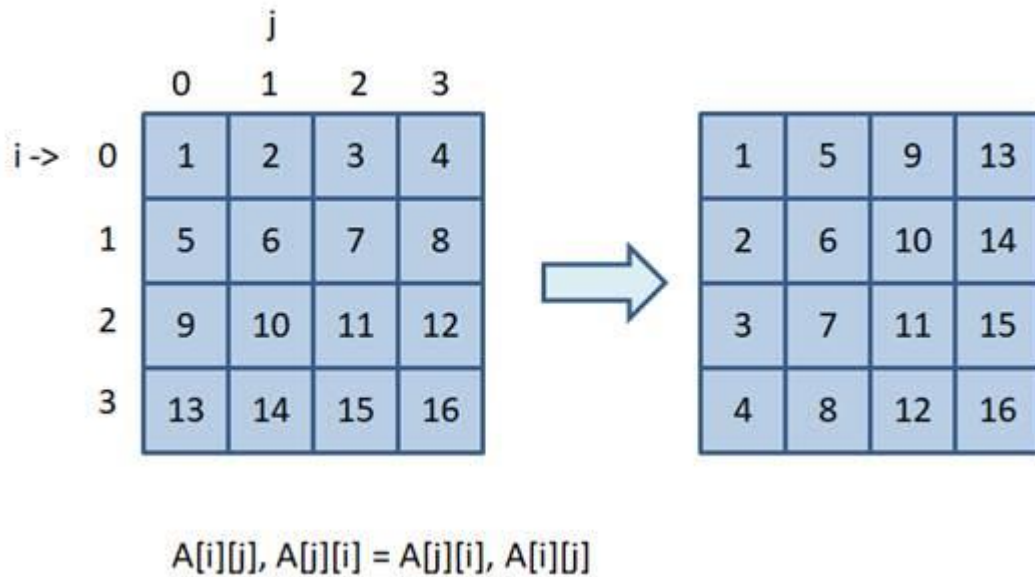
Необходимо поменять строки на столбцы и получить новое представление этого же списка. Для этого достаточно поменять местами элементы, стоящие выше главной диагонали с элементами, стоящими ниже главной диагонали. То есть, у нас счетчик i будет меняться от 0 до 3, а счетчик j от $i+1$ до 3. Затем, соответствующие элементы будем менять между собой:

```
for i in range(len(A)):
```

Основы Python

```
for j in range(i+1, len(A)):
    A[i][j], A[j][i] = A[j][i], A[i][j]
```

```
for r in A:
    for x in r:
        print(x, end='\t')
    print()
```



Как видите, у нас получилось нужно преобразование. В математике это называется транспонированием матрицы.

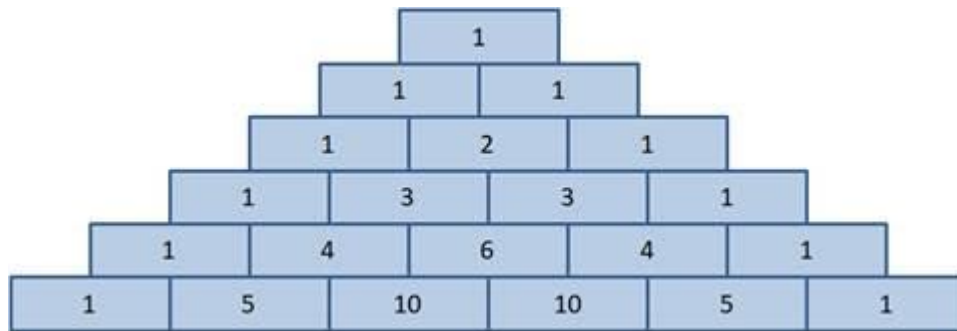
Конечно, уровень вложенности операторов циклов может быть еще больше и два и три и четыре. Однако, на практике слишком большого вложения следует избегать, так как программа становится менее читаемой и гибкой для дальнейшего изменения. Лучше ограничиваться тремя подряд идущими во вложении операторами циклов, не более.

Для закрепления, как всегда, пройдите практические задания и, затем, переходите к следующему уроку.

Основы Python

§26. Треугольник Паскаля как пример работы вложенных циклов

На этом занятии мы с вами напишем программу для построения **треугольника Паскаля**. Этот треугольник состоит из особых чисел, которые вычисляются по правилу: по краям каждой строки стоят единицы, а каждое из остальных чисел равно сумме двух стоящих над ним из предыдущей строки. Именно в таком виде его привел французский математик Блез Паскаль в «Трактате об арифметическом треугольнике». Популярность этого треугольника возникла из-за того, что эти числа довольно часто встречаются в самых разных задачах математики: **алгебре, комбинаторике, теории вероятностей** и многих других. Мы сейчас не будем вдаваться в их анализ, так как перед нами стоит несколько другая задача – **написать программу формирования такого треугольника**.



Первый вопрос, на который, обычно, требуется ответить при разработке новой программы – **как хранить данные?** В данном случае, **как хранить эти числа?** Я это сделаю с помощью неравномерного вложенного списка. Смотрите, весь этот треугольник можно представить как набор вложенных списков постоянно увеличивающейся длины:

```
P = [ [1],  
      [1, 1],  
      [1, 2, 1],  
      ...  
    ]
```

Основы Python

Далее, в программе мы запишем два вложенных цикла **for** со счетчиками: **i** – для перебора строк треугольника Паскаля; **j** – для перебора элементов формируемой строки. Причем, **j** будет меняться в диапазоне **[0; i]**. То есть, для первой строки принимает только одно значение **0**, для второй – два значения **[0; 1]**, для третьей – три **[0; 1; 2]** и так далее.

А формировать значения строк будем следующим образом. Для каждой новой (**i**-й) строки мы создадим список длиной **i+1**, состоящий из всех единиц:

```
row = [1] * (i+1)
```

Затем, все крайние элементы с индексами:

$$j == 0 \text{ or } j == i$$

мы трогать не будем, так как там уже стоят нужные нам единицы. А все другие значения формировать на основе суммы двух предыдущих:

```
row[j] = P[i-1][j-1] + P[i-1][j]
```

В итоге, всю программу можно записать, так:

```
N = 7
P = []

for i in range(0, N):
    row = [1] * (i + 1)
    for j in range(i + 1):
        if j != 0 and j != i:
            row[j] = P[i-1][j-1] + P[i-1][j]

    P.append(row)

for r in P:
    print(r)
```

Основы Python

Мы вначале определяем общее число строк, затем коллекцию **P**, которая будет хранить значения треугольника Паскаля и, далее, вложенные циклы. Первый цикл со счетчиком **i** перебирает строки, а второй – со счетчиком **j** перебирает элементы текущей строки **row**. Внутри вложенного цикла проверяем, если индекс **j** ссылается не на крайние элементы строки, то формируем их значения на основе суммы двух чисел из предыдущей строки. После этого добавляем строку в список **P** и проделываем эту операцию **N** раз. В конце программы выводим результат и давайте посмотрим, что у нас получится.

Как видите, у нас сформировался треугольник Паскаля, состоящий из семи строк. Вот так, с помощью вложенных циклов можно решить эту, относительно несложную задачу.

Опять же, все программы, что я привожу на учебном курсе не являются истиной в последней инстанции. У вас могут получаться другие программные реализации той же самой задачи. Здесь главное помнить, что лучшая реализация будет та, которая расходует меньше памяти и работает быстрее. Это два самых главных критерия построения любых программ. Ну и, разумеется, программы должны приводить к корректным результатам, иначе их можно считать неработоспособными.

Надеюсь, из этого занятия вы хорошо себе представили, как можно запрограммировать задачу формирования треугольника Паскаля с произвольным числом строк. Если здесь никаких вопросов не возникает, то смело переходите к следующему уроку.

§27. Генераторы списков (List comprehensions)

Мы продолжаем курс по языку программирования **Python**. На этом занятии мы с вами познакомимся с еще одной довольно полезной и популярной конструкцией – **генератором списков**. По-английски это записывается как:

List comprehension

Основы Python

Для лучшего понимания, давайте я начну сразу с конкретного примера. Предположим, мы хотели бы сформировать список из квадратов целых чисел от 0 до N. Используя наши текущие знания, очевидно, это можно было бы сделать, следующим образом:

```
N = 6
a = [0] * N

for i in range(N):
    a[i] = i ** 2

print(a)
```

Но это не лучший вариант. Все то же самое можно реализовать буквально в одну строчку:

```
a = [x ** 2 for x in range(N)]
```

И, кроме того, этот вариант будет работать быстрее предыдущей программы, т.к. **Python** оптимизирует работу таких конструкций.

Давайте теперь разберемся в этом синтаксисе. Вначале мы указываем, что будем делать с переменной **x**. Казалось бы, переменная нигде не задана, а мы уже говорим что с ней делать. Да, это так, это такой элемент синтаксиса **list comprehensions**. Далее, через пробел мы записываем цикл **for** и уже там указываем эту переменную **x** и говорим как она будет меняться. То есть, эта временная переменная **x** существует только внутри списка и пропадает после его создания.

Такой подход и называется генератором списков. Вначале указываем, как формируются значения списка, а затем, описываем изменение параметра **x** через ключевое слово **for**. Причем, здесь можно указывать только его. К примеру, ключевое слово **while** прописывать нельзя.

Итак, для генерации списков следует придерживаться следующего синтаксиса (определения):

Основы Python

[<способ формирования значения> for <переменная> in <итерируемый объект>]

В самом простом варианте мы можем сформировать список и так:

```
a = [1 for x in range(N)]
```

Хотя, в таких случаях, когда все значения одинаковы, обычно, используют оператор *:

```
b = [1] * N
```

Но, в отличие от простого дублирования с помощью **List comprehension** можно формировать более сложные последовательности, например, состоящих из остатков от деления на 4:

```
a = [x % 4 for x in range(N)]
```

То есть, мы можем использовать любые доступные нам операторы для формирования текущего значения. Если, например, взять оператор определения четного и нечетного значений:

```
a = [x % 2 == 0 for x in range(N)]
```

то получим список из булевых величин **True** и **False**. Или же сформировать значения линейной функции:

```
a = [0.5*x+1 for x in range(N)]
```

И так далее. Кроме того, мы здесь можем вызывать и функции. Пусть в программе пользователь вводит несколько целых чисел через пробел:

```
d_inp = input("Целые числа через пробел: ")
```

Тогда с помощью генератора списков мы легко можем преобразовать их в набор чисел:

Основы Python

```
a = [int(d) for d in d_inp.split()]
```

Здесь конструкция `d_inp.split()` возвращает список строк из чисел, а функция `int(d)` преобразовывает каждую строку в целое число. Если убрать функцию `int()` и записать просто `d`:

```
a = [d for d in d_inp.split()]
```

то у нас получится уже список из строк, то есть, функция `int()` действительно делает нужное нам преобразование.

Работа этой реализации очень похожа на функцию `map()`, которую мы с вами использовали в подобных задачах:

```
d_inp = list(map(int, input("Целые числа через пробел: ").split()))
```

(О функции `map()` подробнее мы еще будем говорить).

Вообще, в генераторе списков можно использовать любые итерируемые объекты, в том числе и строки. Например, вполне допустима такая реализация:

```
a = [d for d in "python"]
```

получим список из отдельных символов. Или, можно каждый символ превратить в его код:

```
a = [ord(d) for d in "python"]
```

Имеем список из соответствующих кодов. Если же взять список из слов:

```
t = ["Я", "б", "Python", "выучил", "только", "за", "то", "что", "есть", "он", "на",  
"этом", "канале"]
```

то генератор списка:

```
a = [d for d in t]
```

Основы Python

сформирует точно такой же список. То есть, строки здесь воспринимаются уже как отдельные элементы и `d` ссылается на каждую из них по порядку. Я добавлю функцию `len()` для определения длины строк:

```
a = [len(d) for d in t]
```

и теперь мы имеем список из длин.

Условия в генераторах списков

В генераторах дополнительно можно прописывать условия. Например, выберем все числа меньше нуля:

```
a = [x for x in range(-5, 5) if x < 0]
```

Мы здесь после оператора `for` записали необязательный оператор `if`. В результате, в список будут попадать только отрицательные значения `x`. Или, можно сделать проверку на нечетность:

```
a = [x for x in range(-5, 5) if x % 2 != 0]
```

Получаем только нечетные числа из диапазона `[-5; 5)`. Также можно прописывать составные условия, например:

```
a = [x for x in range(-6, 7) if x % 2 == 0 and x % 3 == 0]
```

Выберем все числа, которые кратны 2 и 3 одновременно. Или, такой пример, выберем из списка городов только те, длина которых меньше семи:

```
cities = ["Москва", "Тверь", "Рязань", "Ярославль", "Владимир"]  
a = [city for city in cities if len(city) < 7]
```

Видите, как легко и просто можно реализовать такую задачу с помощью **List comprehension**.

Ну и последнее, что я хочу отметить на этом занятии – это возможность использования тернарного условного оператора внутри генераторов списков.

Основы Python

Так как первым элементом мы можем прописывать любые конструкции языка **Python**, то кто нам мешает сделать следующее. Пусть имеется список чисел:

```
d = [4, 3, -5, 0, 2, 11, 122, -8, 9]
```

и мы хотим преобразовать его в последовательность со словами «четное» и «нечетное». Сделать это можно как раз с помощью тернарного условного оператора:

```
a = ["четное" if x % 2 == 0 else "нечетное" for x in d]
```

Обратите еще раз внимание, что этот оператор не является какой-то особой частью синтаксиса генератора списка. Мы здесь всего лишь используем его как обычный оператор языка **Python**. Это такой же оператор как сложение, умножение и другие:

```
a = [x + 2 for x in d]
```

которые мы также можем использовать при формировании списков.

Полученная запись генератора списка с тернарным оператором получилась не очень читаемой. Это можно исправить, если записать все в отдельных строчках:

```
a = ["четное" if x % 2 == 0 else "нечетное"  
     for x in d  
     if x > 0  
    ]
```

Кстати, здесь можно добавить еще и условие.

На этом мы завершим первое знакомство с генераторами списков. Для закрепления материала обязательно пройдите практические задания, после чего переходите к следующему уроку, где мы продолжим знакомиться с этой темой.

Основы Python

§28. Вложенные генераторы списков

Мы продолжаем тему генераторов списков языка **Python**. На прошлом занятии мы увидели, как можно формировать списки, используя конструкции:

```
[<способ формирования значения> for <счетчик> in <итерируемый объект>]
```

и

```
[<способ формирования значения>  
  for <счетчик> in <итерируемый объект>  
  if <условие>  
]
```

Но, в действительности, **Python** позволяет записывать любое число циклов **for** в генераторах списков. Здесь операторы **if** после циклов являются необязательными (мы их можем прописывать, а можем и пропускать).

Как всегда, работу таких вложенных циклов лучше всего увидеть на примере. В самом простом варианте, мы сформируем просто пары чисел в списке, следующим образом:

```
a = [(i, j) for i in range(3) for j in range(4)]  
print(a)
```

Для большей наглядности запишем генератор списка в несколько строк:

```
a = [(i, j)  
      for i in range(3)  
      for j in range(4)  
]
```

Вот отсюда уже гораздо лучше видно, что второй цикл вложен в первый. То есть, сначала при $i = 0$ отработывает внутренний цикл по j от 0 до 3. Затем,

Основы Python

переходим к первому циклу, i увеличивается на 1 и внутренний цикл повторяется. В результате, получаем пары чисел:

[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3)]

То есть, здесь у нас происходит работа двух вложенных циклов.

Дополнительно, мы можем указывать условия для каждого из них, например, так:

```
a = [(i, j)
      for i in range(3) if i % 3 == 0
      for j in range(4)
    ]
```

Или, у обоих вместе:

```
a = [(i, j)
      for i in range(3) if i % 3 == 0
      for j in range(4) if j % 2 != 0
    ]
```

Используя этот подход, мы можем, например, сформировать таблицу умножения:

```
a = [f'{i}*{j} = {i*j}'
      for i in range(1, 4)
      for j in range(1, 4)
    ]
```

Или, двумерный список превратить в одномерный:

```
matrix = [[0, 1, 2, 3],
           [10, 11, 12, 13],
           [20, 21, 22, 23]]
```

```
a = [x
```

Основы Python

```
for row in matrix
    for x in row
]
```

Обратите внимание, мы во втором цикле используем переменную **row** из первого цикла. Это вполне допустимая операция, т.к. второй цикл вложен в первый и в нем доступны все переменные, объявленные ранее.

Вложенные генераторы списков

Давайте еще раз посмотрим на исходное определение генератора списка:

```
[<оператор> for <счетчик> in <итерируемый объект>]
```

Как я уже отмечал, в качестве оператора можно записывать любую конструкцию языка **Python**. А раз так, то кто нам мешает прописать здесь еще один генератор:

```
[[генератор списка]
 for <переменная> in <итерируемый объект>
]
```

В результате, получим вложенный генератор списка. Давайте посмотрим на примере, как это будет работать:

```
M, N = 3, 4
matrix = [[a for a in range(M)] for b in range(N)]

print(matrix)
```

Получаем двумерный список, вида:

```
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

Результат вполне очевиден. Смотрите, сначала отработывает первый (внешний) генератор списка и переменная **b = 0**. Затем, выполнение переходит к вложенному генератору, который выдает список **[0, 1, 2]**. Этот

Основы Python

список помещается как первый элемент основного списка. Далее, снова отработывает первый генератор и **b** принимает значение 1. После этого переходим к вложенному генератору, который возвращает такой же список [0, 1, 2]. И так пока не закончится работа первого генератора. В итоге, видим список, в который вложены четыре других списка.

Где может пригодиться такой подход? Например, для изменения значений двумерного списка. Давайте предположим, что у нас есть вот такой список:

```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

И мы хотим возвести все его значения в квадрат. Лучше всего это сделать именно через генератор, следующим образом:

```
A = [[x ** 2 for x in row] for row in A]
```

В первом генераторе происходит перебор строк (вложенных списков) матрицы **A**, а во вложенном генераторе – обход элементов строк матрицы. Каждое значение возводится в квадрат и на основе этого формируется текущая строка. Обратите внимание, что во вложении мы можем использовать переменные из внешнего генератора списка, в частности, переменную **row**, ссылающуюся на текущую строку матрицы **A**.

Другой пример – это транспонирование матрицы **A** (то есть, замена строк на столбцы) с использованием вложенных генераторов. Сделать это можно, следующим образом:

```
A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]  
A = [[row[i] for row in A] for i in range(len(A[0]))]
```

Поясню работу этой конструкции. Сначала значение **i = 0**, а переменная **row[i]** пробегает первые значения строк матрицы **A**. В результате формируется первая строка транспонированной матрицы. Далее, **i** увеличивается на 1 и **row[i]** пробегает уже вторые элементы строк матрицы **A**. Получаем вторую строку транспонированной матрицы. И так делаем для всех столбцов. На выходе формируется транспонированная матрица.

Основы Python

С таким типом вложений, я думаю, в целом все должно быть понятно. Другой вариант, когда мы список помещаем в качестве итерируемого объекта. Да, сам по себе генератор списка поддерживает механизм итерирования – перебора элементов через итератор, поэтому может быть использован в операторе **for**, например, так:

```
g = [u ** 2 for u in [x+1 for x in range(5)]]
```

Здесь сначала отработывает вложенный генератор списка, получаем список `[1, 2, 3, 4, 5]`, а затем, эти значения перебираются первым генератором и возводятся в квадрат, получаем результат:

`[1, 4, 9, 16, 25]`

Это очень похоже на вычисление значений сложной (вложенной) функции:

$$g(u(x+1)) = (x+1) ^ 2$$

Вот такие варианты вложений генераторов списков, а также их комбинации, можно использовать в **Python**. Для закрепления этого материала пройдите практические задания и жду всех вас на следующем уроке.

§29. Введение в словари (dict). Базовые операции над словарями

Мы продолжаем курс по языку **Python**. На этом занятии мы с вами познакомимся с новой коллекцией данных – словарем. Давайте представим, что мы в программе хотели бы описать, следующие зависимости:

house -> дом
car -> машина
tree -> дерево
road -> дорога
river -> река

Если определить значения через список:

```
d = ["дом", "машина", "дерево", "дорога", "река"]
```


Основы Python



то мы получим коллекцию, где каждое значение (**русское слово**) будет связано с числом – индексом. И изменить эти индексы невозможно – они создаются автоматически самим списком. А нам бы хотелось обращаться к этим значениям по английским словам. Как раз это позволяет делать словарь. Он формирует коллекцию в виде ассоциативного массива с доступом к элементу по ключу.

Для создания словаря используется следующий синтаксис:

`{key1: value1, key2: value2, ..., keyN: valueN}`

и он определяет неупорядоченную коллекцию данных, то есть, данные в словаре не имеют строгого порядка следования. Располагаться они могут произвольным образом, но всегда связаны с одним, строго определенным ключом.

В нашем конкретном случае словарь можно определить, так:

```
d = {"house": "дом", "car": "машина",  
     "tree": "дерево", "road": "дорога",  
     "river": "река"}
```



Как видите, мы можем записывать значения в несколько строчек, не обязательно все писать в одну.

Основы Python

После создания словаря, мы можем по ключу получать нужное нам значение. Для этого записываются квадратные скобки и в них указывается ключ:

```
d["house"]
```

возвращается значение «дом», которое связано с этим ключом. Если же указать не существующий ключ:

```
d[100]
```

то получим ошибку. Разумеется, ключи в словарях всегда уникальны. Если записать два одинаковых:

```
d = {"house": "дом", "house": "дом 2", "car": "машина"}
```

то ключ «house» будет ассоциирован с последним значением – «дом 2».

Также для определения словаря в **Python** существует специальная встроенная функция **dict()**. Ей, в качестве аргументов, через запятую перечисляются пары в формате **ключ=значение**:

```
d2 = dict(one = 1, two = 2, three = "3", four = "4")
```

Здесь ключи преобразовываются в строки и должны определяться как и имена переменных. Например, использовать числа уже не получится:

```
d2 = dict(1 = "one", two = 2, three = "3", four = "4")
```

Это неверное имя переменной.

Один из плюсов этой функции – возможность создать словарь из списка специального вида. **Что это за вид?** Например, такой:

```
lst = [[2, "неудовлетворительно"], [3, "удовлетворительно"], [4, "хорошо"], [5, "отлично"]]
```

Основы Python

Здесь вложенные списки состоят из двух элементов, которые функцией `dict()` интерпретируются как ключ и значение. Если мы сформируем словарь:

```
d3 = dict(lst)
```

то, как раз, увидим такую структуру данных. Причем, в этом случае, в качестве ключей можно уже выбирать и другие типы данных, не только строки, например, числа.

Если вызвать эту функцию без параметров:

```
d = dict()
```

то получим пустой словарь. Это эквивалентно такой записи:

```
d = {}
```

и, чаще всего, она используется на практике, так как короче.

Давайте теперь ответим на вопрос: а **что вообще можно использовать в качестве ключей? Какие типы данных?** В наших примерах это было или число, или строка. **Что можно брать еще?** На самом деле любые неизменяемые типы. Обратите внимание – **неизменяемые!** Например, можно написать так:

```
d[True] = "Истина"  
d[False] = "Ложь"
```

Здесь ключи – булевы значения. В результате, получим словарь:

```
{True: 'Истина', False: 'Ложь'}
```

Также этот пример показывает, что присваивая словарию значение с новым ключом, оно автоматически добавляется в словарь. В результате, наш изначально пустой словарь стал содержать две записи. Если же мы существующему ключу присваиваем другое значение:

Основы Python

```
d[True] = 1
```

то он просто поменяет свое значение в словаре. Вот так можно добавлять и менять значения словаря. То есть, словарь относится к изменяемым типам.

А вот если ключом указать список:

```
d[[1,2]] = 1
```

то возникнет ошибка, т.к. список – это изменяемый объект. Вообще, чаще всего в качестве ключей используются строки или числа.

Это то, что касается ключей, а вот на значения словаря никаких ограничений не накладывается – там могут самые разные данные:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
```

Операторы и функции работы со словарем

В заключение этого занятия рассмотрим некоторые операторы и функции работы со словарем. С помощью функции:

```
len(d)
```

можно определить число элементов в словаре. **Оператор:**

```
del d[True]
```

выполняет удаление пары **ключ=значение** для указанного ключа. Если записать несуществующий ключ:

```
del d["abc"]
```

то оператор возвращает ошибку. Поэтому перед удалением лучше проверить существует ли данный ключ в словаре с помощью оператора **in**:

```
"abc" in d
```

Основы Python

Обратите внимание, оператор **in** проверяет именно наличие ключа, а не значения. Например, если добавить это значение:

```
d[1] = "abc"
```

и повторно выполнить команду:

```
"abc" in d
```

то увидим значение **False**.

Также можно делать противоположную проверку на отсутствие ключа, прописывая дополнительно оператор **not**:

```
"abc" not in d
```

Надеюсь, из этого занятия вы узнали, что такое словарь, как можно его задавать и обращаться к элементам этой коллекции по ключам.

Познакомились с функцией **dict()** для создания словарей, функцией **len()** – для определения числа элементов и операторами **in** и **del**. Для закрепления этого материала подготовлены практические задания, после которых жду вас на следующем уроке.

§30. Методы словаря, перебор элементов словаря в цикле

Мы продолжаем изучение словарей языка **Python**. Это занятие начнем с ознакомления основных методов, которые есть у этой коллекции. Начнем с метода

`dict.fromkeys(список[, значение по умолчанию])`

который формирует словарь с ключами, указанными в списке и некоторым значением. Например, передадим методу список с кодами стран:

```
a = dict.fromkeys(["+7", "+6", "+5", "+4"])
```

в результате, получим следующий словарь:

Основы Python

```
{'+7': None, '+6': None, '+5': None, '+4': None}
```

Обратите внимание, все значения у ключей равны **None**. Это значение по умолчанию. Чтобы его изменить используется второй необязательный аргумент, например:

```
a = dict.fromkeys(['+7', '+6', '+5', '+4'], "код страны")
```

на выходе получаем словарь с этим новым указанным значением:

```
{'+7': 'код страны', '+6': 'код страны', '+5': 'код страны', '+4': 'код страны'}
```

Следующий метод

```
d.clear()
```

служит для очистки словаря, то есть, удаления всех его элементов.

Далее, для создания копии словаря используется метод `copy()`:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
d2 = d.copy()
d2["list"] = [5,6,7]
print(d)
print(d2)
```

Также копию можно создавать и с помощью функции **dict()**, о которой мы с вами говорили на предыдущем занятии:

```
d2 = dict(d)
```

Результат будет абсолютно таким же, что и при вызове метода `copy()`.

Следующий метод **get** позволяет получать значение словаря по ключу:

```
d.get("list")
```

Его отличие от оператора

Основы Python

```
d["list"]
```

в том, что при указании неверного ключа не возникает ошибки, а выдается по умолчанию значение **None**:

```
print(d.get(3))
```

Это значение можно изменить, указав его вторым аргументом:

```
print( d.get(3, False) )
```

Похожий метод

dict.setdefault(key[, default])

возвращает значение, ассоциированное с ключом **key** и если его нет, то добавляет в словарь со значением **None**, либо **default** – если оно указано:

```
d.setdefault("list")  
d.setdefault(3)
```

Добавит ключ **3** со значением **None**. Удалим его:

```
del d[3]  
d.setdefault(3, "three")
```

тогда добавится этот ключ со значением **«three»**. То есть, этот метод способен создать новую запись, но только в том случае, если ключ отсутствует в словаре.

Следующий метод

```
d.pop(3)
```

удаляет указанный ключ и возвращает его значение. Если в нем указывается несуществующий ключ, то возникает ошибка:

```
d.pop("abc")
```

Основы Python

Но мы можем легко исправить ситуацию, если в качестве второго аргумента указать значение, возвращаемое при отсутствии ключа:

```
d.pop("abc", False)
```

Здесь возвратится **False**. Если же ключ присутствует, то возвращается его значение.

Следующий метод

```
d.popitem()
```

выполняет удаление произвольной записи из словаря. Если словарь пуст, то возникает ошибка:

```
d2 = {}  
d2.popitem()
```

Следующий метод

```
d.keys()
```

возвращает коллекцию ключей. По умолчанию цикл **for** обходит именно эту коллекцию, при указании словаря:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}  
for x in d:  
    print(x)
```

то есть, эта запись эквивалента такой:

```
for x in d.keys():  
    ...
```

Если же вместо **keys** записать метод **values**:

```
for x in d.values():
```


Основы Python

...

то обход будет происходить по значениям, то есть, метод **values** возвращает коллекцию из значений словаря.

Последний подобный метод **items**

```
for x in d.items():
```

...

возвращает записи в виде кортежей: ключ, значение. О кортежах мы будем говорить позже, здесь лишь отмечу, что к элементу кортежа можно обратиться по индексу и вывести отдельно ключи и значения:

```
print(x[0], x[1])
```

Или, используя синтаксис множественного присваивания:

```
x, y = (1, 2)
```

можно записать цикл **for** в таком виде:

```
for key, value in d.items():  
    print(key, value)
```

что гораздо удобнее и нагляднее.

Следующий метод **update()** позволяет обновлять словарь содержимым другого словаря. Например, есть два словаря:

```
d = dict(one = 1, two = 2, three = "3", four = "4")  
d2 = {2: "неудовлетворительно", 3: "удовлетворительно", 'four': "хорошо", 5:  
"отлично"}
```

И мы хотим первый обновить содержимым второго:

```
d.update(d2)
```

Основы Python

Смотрите, ключ **four** был заменен строкой «хорошо», а остальные, не существующие ключи были добавлены.

Если же нам нужно создать новый словарь, который бы объединял содержимое обоих, то для этого можно воспользоваться конструкцией:

```
d3 = {**d, **d2}
```

Однако, детально как она работает будет понятно позже, когда мы познакомимся с операторами распаковки коллекций.

Итак, на этом занятии мы с вами познакомились с основными методами словарей, а также способами их перебора и объединения. Для закрепления этого материала выполните практические задания и переходите к следующему уроку.

§31. Кортежи (tuple) и их методы

На этом занятии мы с вами познакомимся с еще одной новой коллекцией данных – кортежами. **Что такое кортеж?** Это упорядоченная, но неизменяемая коллекция произвольных данных. Они, в целом, аналогичны спискам, но в отличие от них, элементы кортежей менять нельзя – это неизменяемый тип данных.

0	1	2	3
-5	'hello'	True	[1, 2, 3]
-4	-3	-2	-1

Для задания кортежа достаточно перечислить значения через запятую:

```
a = 1,2
```

или, что то же самое, в круглых скобках:

```
a = (1, 2, 3)
```

Основы Python

Но, обратите внимание, при определении кортежа с одним значение, следует использовать такой синтаксис:

```
b = 1,    или    b = (1,)
```

Здесь висячая запятая указывает, что единицу следует воспринимать как первый элемент кортежа, а не как число 1. Без запятой – это будет уже просто число один. Вот это нужно очень хорошо запомнить, начинающие программисты здесь часто делают ошибку, не прописывая висячую запятую.

Далее, мы можем переменным сразу присвоить соответствующие значения из кортежа, перечисляя их через запятую:

```
x, y = (1, 2)
```

но, если число переменных не будет совпадать с числом элементов кортежа:

```
x, y = (1, 2, 3)
```

то возникнет ошибка.

Длину кортежа (**число его элементов**) можно определить с помощью известной уже вам функции:

```
len(a)
```

А доступ к его элементам осуществляется также как и к элементам списков:

- **по индексу:**

```
a[0]    a[2]        a[-1]
```

- **через срезы:**

```
a[1:2]    a[0:-1]    a[:3]        a[1:]    a[:]
```

О срезах мы с вами уже подробно говорили, когда рассматривали списки, здесь все абсолютно также, кроме одной операции – взятия полного среза:

Основы Python

```
b = a[:]
```

В случае с кортежем здесь не создается его копии – это будет ссылка на тот же самый объект:

```
print(id(a), id(b))
```

Напомню, что для списков эта операция приводит к их копированию. Этот момент нужно также очень хорошо запомнить: **для кортежей** – возвращается тот же самый объект, а **для списков** – создается копия.

Некоторые из вас сейчас могут задаваться вопросом: **зачем было придумывать кортежи, если списки способны выполнять тот же самый функционал? И даже больше – у них можно менять значения элементов, в отличие от кортежей?** Да, все верно, по функциональности списки шире кортежей, но у последних все же есть свои преимущества.

Во-первых, то, что кортеж относится к неизменяемому типу данных, то мы можем использовать его, когда необходимо **«запретить»** программисту менять значения элементов списка. Например, вот такая операция:

```
a[1] = 100
```

приведет к ошибке. Менять значения кортежей нельзя. **Во-вторых**, кортежи можно использовать в качестве ключей у словарей, например, так:

```
d = {}  
d[a] = "кортеж"
```

Напомню, что списки как ключи применять недопустимо, так как список – это изменяемый тип, а ключами могут быть только неизменяемые типы и кортежи здесь имеют преимущество перед списками.

В-третьих, кортеж занимает меньше памяти, чем такой же список, например:

```
lst=[1,2,3]  
t=(1,2,3)
```

Основы Python

```
print(lst.__sizeof__())  
print(t.__sizeof__())
```

Как видите, размер кортежа меньше, чем списка. Здесь использован метод `__sizeof__`, который возвращает размер данных в байтах.

Из всего этого можно сделать такой общий вывод: **если мы работаем с неизменяемым упорядоченным списком, то предпочтительнее использовать кортеж.**

Теперь, когда мы поняли, что кортежи играют свою значимую роль в программах на **Python**, вернемся к их рассмотрению. Чтобы создать пустой кортеж можно просто записать круглые скобки:

```
a = ()
```

или воспользоваться специальной встроенной функцией:

```
b = tuple()  
print(type(a), type(b))
```

Но здесь сразу может возникнуть вопрос: **зачем создавать пустой кортеж, если он относится к неизменяемым типам данных?** Слово **неизменяемый** наводит на мысль, что вид кортежа остается неизменным. Но это не совсем так. В действительности, мы лишь не можем менять уже существующие элементы в кортеже, но можем создавать новые, используя оператор `+`, например:

```
a = ()  
a = a + (1,)
```

или для добавления данных в начало кортежа:

```
a = (2, 3) + a
```

Также можно добавить вложенный кортеж:

```
a += (("a", "hello"),)
```

Основы Python

или дублировать элементы кортежа через оператор `*`, подобно спискам:

```
b = (0,)*10  
b = ("hello", "world") * 5
```

Как видите, все эти операции вполне допустимы. А вот удалять отдельные его элементы уже нельзя. Если мы попытаемся это сделать:

```
del a[1]
```

то возникнет ошибка. Правда, можно удалить объект целиком:

```
del a
```

тогда кортеж просто перестанет существовать, не будет даже пустого кортежа, здесь объект удаляется целиком.

Далее, с помощью функции `tuple()` можно создавать кортежи на основе любого итерируемого объекта, например, списков и строк:

```
a = tuple([1,2,3])  
a = tuple("Привет мир")
```

А если нам нужно превратить кортеж в список, то подойдет известная нам функция `list()`, которая формирует список также из любого итерируемого объекта:

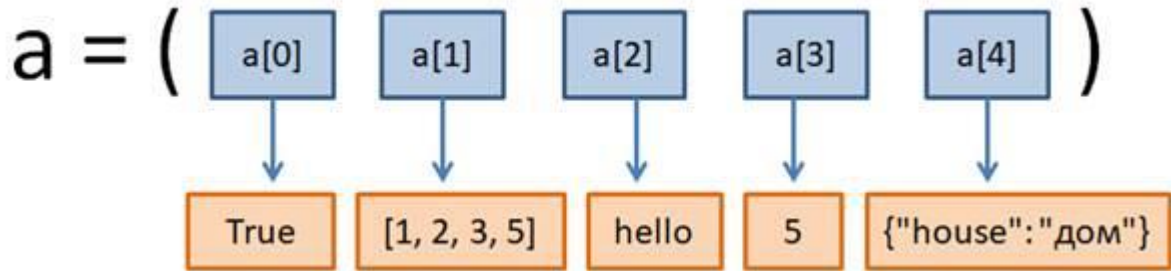
```
t = (1,2,3)  
lst = list(t)
```

Так как кортеж – это перебираемый, то есть, итерируемый объект, то никто не мешает нам превращать его в список.

Как я отмечал в самом начале, кортежи могут хранить самые разные данные:

```
a = (True, [1,2,3], "hello", 5, {"house": "дом"})
```

Основы Python



Причем, смотрите, если обратиться, например, к списку:

```
a[1]
```

то это изменяемый объект, следовательно, его значение даже в кортеже мы можем спокойно менять:

```
a[1].append("5")
```

То есть, неизменность кортежа относится к его структуре элементов и значениям ссылок на объекты. Но, если сами объекты, при этом, могут меняться, то мы можем легко это делать. То есть, обращаясь ко второму элементу кортежа, мы, фактически, имеем список, с которым возможны все его операции без каких-либо ограничений. Я, думаю, **это должно быть понятно?**

В заключение этого занятия рассмотрим два метода, которые часто используются в кортежах – это:

- **tuple.count(значение)** – возвращает число найденных элементов с указанным значением;
- **tuple.index(значение[, start[, stop]])** – возвращает индекс первого найденного элемента с указанным значением (**start** и **stop** – необязательные параметры, индексы начала и конца поиска).

Эти методы работают так же, как и у списков. Пусть у нас есть кортеж:

```
a = ("abc", 2, [1,2], True, 2, 5)
```

Основы Python

И мы хотим определить число элементов со значениями:

```
a.count("abc")  
a.count(2)
```

Как видите, метод **count()** возвращает число таких элементов в кортеже. Если же элемент не будет найден:

```
a.count("a")
```

то возвращается **0**. Следующий метод:

```
a.index(2)
```

возвращает индекс первого найденного элемента с указанным значением. Если значение не найдено:

```
a.index(3)
```

то этот метод приводит к ошибке. Поэтому, прежде чем его использовать, лучше проверить, а есть ли такое значение вообще в кортеже:

```
3 in a
```

Второй необязательный параметр

```
a.index(2, 3)
```

позволяет задавать индекс начала поиска. В данном случае поиск будет начинаться с третьего индекса. А последний третий аргумент:

```
a.index(2, 3, 5)
```

определяет индекс, до которого идет поиск (не включая его). Если же записать вот так:

```
a.index(2, 3, 4)
```


Основы Python

то возникнет ошибка, т.к. в поиск будет включен только 3-й индекс и там нет значения 2.

Надеюсь, из этого занятия вы узнали, что такое кортежи, как с ними можно работать и зачем они нужны. В целом, вы должны запомнить следующее:

операторы работы с кортежами, а также функции и методы кортежей.

Закрепите этот материал практическими заданиями и переходите к следующему уроку.

§32. Множества (set) и их методы

На этом занятии мы с вами познакомимся с последней базовой коллекцией – множествами.

Формально, множество (set) – это неупорядоченная коллекция уникальных элементов. Уникальных, то есть, в ней отсутствуют дублирующие значения.

Для задания множества используются фигурные скобки, в которых через запятую перечисляются значения элементов:

```
a = {1, 2, 3, "hello"}
```

Это немного похоже на определение словаря, но в отличие от него, здесь не прописываются ключи – только значения. В результате получаем совершенно другой тип объекта – **множество**:

```
type(a)
```

Так как множество состоит только из уникальных значений, то попытка в него поместить несколько одинаковых значений:

```
a = {1, 2, 3, "hello", 2, 3, "hello"}
```

приведет к тому, что останутся только не повторяющиеся. Это ключевая особенность работы данной коллекции – она автоматически отбрасывает все дубли.

Основы Python

Во множество мы можем записывать только неизменяемые данные, такие как:

- числа;
- булевы значения;
- строки;
- кортежи

```
a = {1, 4.5, "hi", "hi", (4, True), ("a", False)}
```

А вот изменяемые типы данных:

- списки;
- словари;
- множества

добавлять нельзя, будет ошибка:

```
b = {[1, 2], 3, 4}
b = {[1: 1, 2: 2], 3, 4}
b = {[1, 2], 3, 4}
```

Для создания множества используется встроенная функция `set()`. Если ее записать без аргументов:

```
set()
```

то получим пустое множество. Но, обратите внимание, если попытаться создать пустое множество вот так:

```
b = {} # словарь
```

то получим не пустое множество, а пустой словарь! Пустое множество создается именно с помощью функции `set()`. Вот этот момент нужно запомнить.

Если указать какой-либо итерируемый объект, например, список:

Основы Python

```
set([1, 2, 1, 0, -1, 2])
```

то получим множество, состоящее из уникальных значений этого списка. Или, если передать строку:

```
set("abracadabra")
```

то увидим множество из уникальных символов этой строки. И так для любого итерируемого объекта, даже можно прописать функцию **range()**:

```
b = set(range(7))
```

Обратите внимание, множества представляют собой неупорядоченную коллекцию, то есть, значения элементов могут располагаться в них в любом порядке. Это также означает, что с множествами нельзя выполнять операции доступа по индексу или срезам:

```
b[0]
```

приведет к ошибке.

Для чего вообще может потребоваться такая коллекция в программировании?

Я, думаю, большинство из вас уже догадались – с их помощью, например, можно легко и быстро убирать дубли из данных. Представим, что некоторая логистическая фирма составила список городов, куда доставляли товары:

```
cities = ["Калуга", "Краснодар", "Тюмень", "Ульяновск", "Москва", "Тюмень",  
"Калуга", "Ульяновск"]
```

И отдел статистики хотел бы получить список уникальных городов, с которыми было сотрудничество. Очевидно, это можно сделать с помощью множества:

```
set(cities)
```

Основы Python

Обратите внимание, порядок городов может быть любым, так как множество – неупорядоченная коллекция данных. Мало того, мы можем этот полученный перечень снова преобразовать в список, используя функцию:

```
list(set(cities))
```

Все, мы отобрали уникальные города и представили их в виде списка. Видите, как это легко и просто можно сделать с помощью множеств.

А что если нам не нужен список уникальных городов, а достаточно лишь перебрать все элементы полученного множества:

```
q = set(cities)
```

Учитывая, что множество – это итерируемый объект, то пройти по всем его элементам можно с помощью цикла **for**:

```
for c in q:  
    print(c)
```

Здесь переменная **c** последовательно ссылается на значения элементов множества **q** и соответствующий город выводится в консоль. Или, то же самое можно сделать через механизм итераторов:

```
it = iter(q)  
next(it)
```

Наконец, функция:

```
a = {"abc", (1, 2), 5, 4, True}  
len(a)
```

возвратит нам число элементов в множестве. А для проверки наличия значения в множестве можно воспользоваться, уже знакомым нам оператором **in**:

```
"abc" in a
```

Основы Python

Он возвращает **True**, если значение имеется и **False** в противном случае. Или можно проверить на непринадлежность какого-либо значения:

```
7 not in a
```

Методы добавления/удаления элементов множества

В заключение этого занятия рассмотрим методы для добавления и удаления элементов в множествах. Первый метод **add()** позволяет добавлять новое значение в множество:

```
b.add(7)
```

Значение будет добавлено только в том случае, если оно отсутствует в множестве **b**. Если попробовать, например, еще раз добавить семерку:

```
b.add(7)
```

то множество не изменится, но и ошибок никаких не будет.

Если в множество требуется добавить сразу несколько значений, то для этого хорошо подходит метод **update()**:

```
b.update(["a", "b", (1, 2)])
```

В качестве аргумента указывается любой итерируемый объект, в данном случае, список, значения которого добавляются в множество с проверкой уникальности. Или, можно передать строку:

```
b.update("abracadabra")
```

Получим добавку в виде уникальных символов этой строки. И так далее, в качестве аргумента метода **update()** можно указывать любой перебираемый объект.

Далее, для удаления элемента по значению используется метод **discard()**:

```
b.discard(2)
```

Основы Python

Если еще раз попытаться удалить двойку:

```
b.discard(2)
```

то ничего не произойдет и множество не изменится.

Другой метод для удаления элемента по значению – **remove()**:

```
b.remove(4)
```

но при повторном таком вызове:

```
b.remove(4)
```

возникнет ошибка, т.к. значение **4** в множестве уже нет. То есть, данный метод возвращает ошибку при попытке удаления несуществующего значения. Это единственное отличие в работе этих двух методов.

Также удалять элементы можно и с помощью метода **pop()**:

```
b.pop()
```

При этом он возвращает удаляемое значение, а сам удаляемый элемент оказывается, в общем-то, случайным, т.к. **множество** – это неупорядоченная коллекция. Если вызвать этот метод для пустого множества, то возникнет ошибка:

```
c=set()  
c.pop()
```

Наконец, если нужно просто удалить все элементы из множества, то используется метод:

```
b.clear()
```

Основы Python

На этом мы с вами завершим первое знакомство с множествами. Для закрепления материала, как всегда, вас ждут практические задания, а я буду всех вас ждать на следующем уроке.

§33. Операции над множествами, сравнение множеств

На этом занятии мы с вами будем говорить об операциях над ними – это:

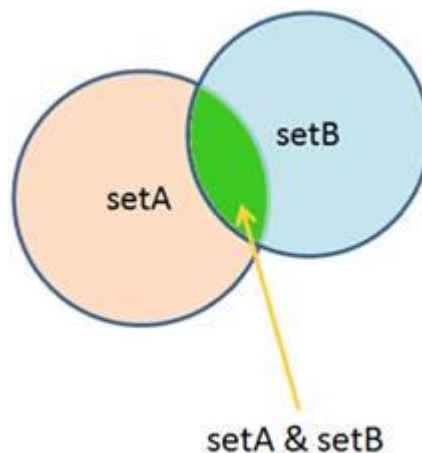
- пересечение множеств;
- объединение множеств;
- вычитание множеств;
- вычисление симметричной разности.

Для простоты зададим два множества с числами:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
```

На их основе можно построить третье множество, как результат пересечения этих двух:

setA & setB



На выходе получаем новое множество, состоящее из значений, общих для множеств **setA** и **setB**. Это и есть операция пересечения двух множеств. При

Основы Python

этом, исходные множества остаются без изменений. Здесь именно создается новое множество с результатом выполнения этой операции.

Разумеется, чтобы в дальнейшем в программе работать с полученным результатом, его нужно сохранить через какую-либо переменную, например, так:

```
res = setA & setB
```

Также допустима запись в виде:

```
setA &= setB
```

это будет эквивалент строчки:

```
setA = setA & setB
```

То есть, мы вычисляем результат пересечения и сохраняем его в переменной **setA** и, как бы, меняем само множество **setA**.

А вот если взять множество:

```
setC = {9, 10, 11}
```

которое не имеет общих значений с другим пересекающимся множеством, то результатом:

```
setA & setC
```

будет пустое множество.

Обычно, на практике используют оператор пересечения **&**, но вместо него можно использовать специальный метод:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
setA.intersection(setB)
```


Основы Python

Результат будет тем же, на выходе получим третье множество, состоящее из общих значений множеств **setA** и **setB**. Если же мы хотим выполнить аналог операции:

```
setA &= setB
```

то для этого следует использовать другой метод:

```
setA.intersection_update(setB)
```

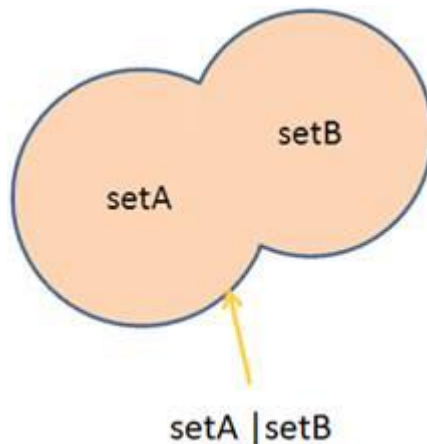
В результате, меняется само множество **setA**, в котором хранятся значения пересечения двух множеств.

Вот такие способы вычислений пересечения множеств существуют в **Python**.

Объединение множеств

Следующая операция – это объединение двух множеств. Она записывается, так:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
setA | setB
```



На выходе также получаем новое множество, состоящее из всех значений обоих множеств, разумеется, без их дублирования.

Основы Python

Или же можно воспользоваться эквивалентным методом:

```
setA.union(setB)
```

который возвращает множество из объединенных значений. При этом, сами множества остаются без изменений.

Операцию объединения можно записать также в виде:

```
setA |= setB
```

Тогда уже само множество **setA** будет хранить результат объединения двух множеств, то есть, оно изменится. Множество **setB** останется без изменений.

Вычитание множеств

Следующая операция – это вычитание множеств. Например, для множеств:

```
setA = {1,2,3,4}  
setB = {3,4,5,6,7}
```

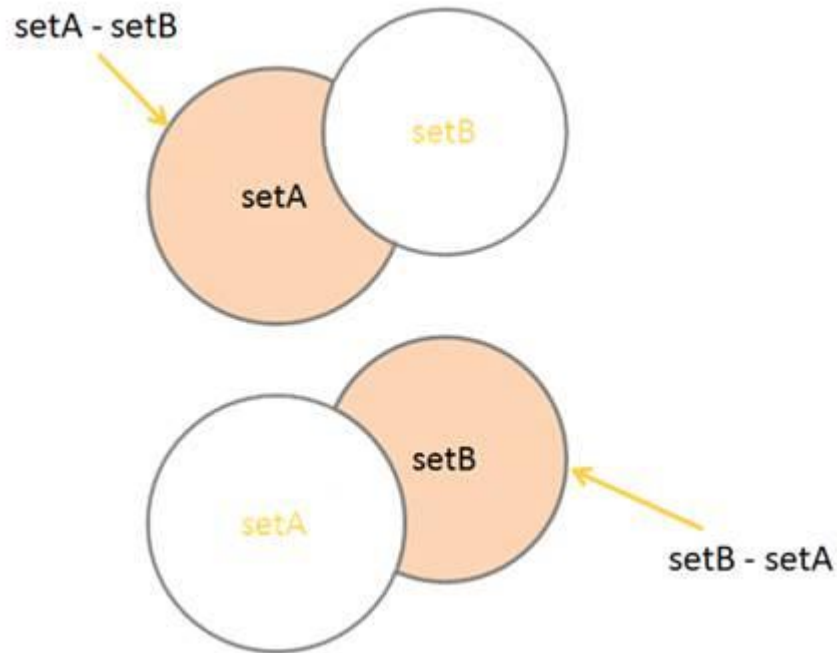
операция

```
setA - setB
```

возвратит новое множество, в котором из множества **setA** будут удалены все значения, повторяющиеся в множестве **setB**:

```
{1, 2}
```

Основы Python



Или, наоборот, если из множества **setB** вычесть множество **setA**:

setB - setA

то получим значения из которых исключены величины, входящие в множество **setA**.

Также здесь можно выполнять эквивалентные операции:

```
setA -= setB  
setB -= setA
```

В этом случае переменные **setA** и **setB** будут ссылаться на соответствующие результаты вычитаний.

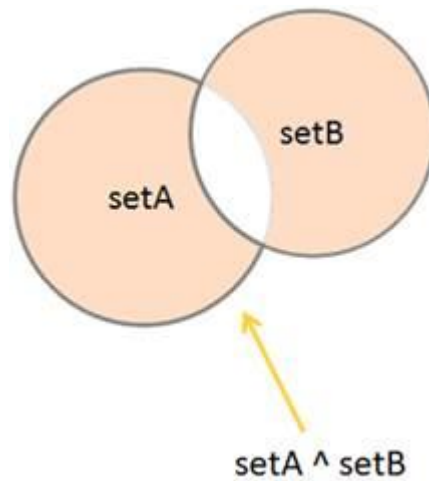
Симметричная разность

Наконец, последняя операции над множествами – симметричная разность, возвращает множество, состоящее из неповторяющихся значений обоих множеств. Реализуется она следующим образом:

Основы Python

```
setA = {1,2,3,4}
setB = {3,4,5,6,7}
setA ^ setB
```

На выходе получим третье, новое множество, состоящее из значений, не входящих одновременно в оба множества.



Сравнение множеств

В заключение этого занятия я хочу показать вам, как можно сравнивать множества между собой.

Предположим, имеются два таких множества:

```
setA = {7, 6, 5, 4, 3}
setB = {3, 4, 5, 6, 7}
```

И мы хотим узнать, равны они или нет. Для этого, используется уже знакомый нам оператор сравнения на равенство:

```
setA == setB
```

Увидим значение **True** (истина). **Почему?** Дело в том, что множества считаются равными, если все элементы, входящие в одно из них, также принадлежат другому и мощности этих множеств равны (то есть они

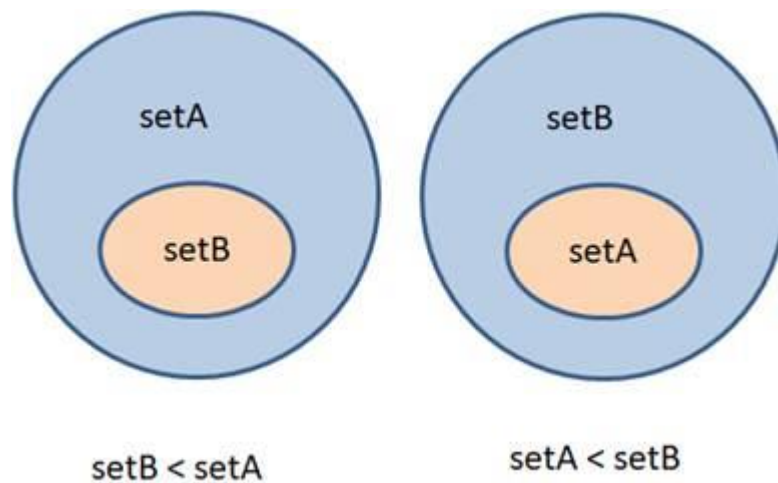
Основы Python

содержат одинаковое число элементов). Наши множества `setA` и `setB` удовлетворяют этим условиям.

Соответственно, противоположное сравнение на не равенство, выполняется, следующим образом:

```
setA != setB
```

Следующие два оператора сравнения на **больше** и **меньше**, фактически, определяют, входит ли одно множество в другое или нет.



Например, возьмем множества

```
setA = {7, 6, 5, 4, 3}
setB = {3, 4, 5}
```

тогда операция

```
setB < setA
```

вернет **True**, а операция

```
setB > setA
```

значение False. Но, если хотя бы один элемент множества `setB` не будет принадлежать множеству `setA`:

Основы Python

```
setB.add(22)
```

то обе операции вернут **False**. Для равных множеств

```
setA = {7, 6, 5, 4, 3}
setB = {3, 4, 5, 6, 7}
```

обе операции также вернут **False**. Но вот такие операторы:

```
setA <= setB
setA >= setB
```

вернут **True**.

Я, думаю, из этих примеров понятно, как выполняется сравнение множеств между собой.

Для закрепления этого материала, как всегда, пройдите практические задания и, затем, дальше – на покорение новых вершин языка **Python**. Буду вас ждать на следующем уроке.

§34. Генераторы множеств и генераторы словарей

На этом занятии мы с вами поговорим о генераторах множеств и словарей. Ранее, я вам уже рассказывал о генераторах списков, когда мы их создавали, используя синтаксис:

[<способ формирования значения> for <счетчик> in <итерируемый объект>]

Например:

```
a = [x ** 2 for x in range(1, 5)]
print(a)
```

Далее, я буду полагать, что вы помните и знаете этот материал. Так вот, те же самые конструкции можно определять и для множеств со словарями. Для этого достаточно вместо квадратных скобок прописать фигурные:

Основы Python

```
a = {x ** 2 for x in range(1, 5)}
```

и вместо списка получим уже множество. **Почему было сформировано именно множество, а не словарь?** Как мы помним, множество представляет собой набор отдельных значений, а в словаре дополнительно еще прописываются ключи. Здесь же, при генерации мы получаем серию значений, поэтому, такая коллекция в **Python** воспринимается именно как множество.

А вот если мы будем генерировать последовательность с ключами, например, так:

```
a = {x: x ** 2 for x in range(1, 5)}
```

то получим уже словарь.

Где такие генераторы могут использоваться? Давайте представим, что нам нужно выделить уникальные значения из списка:

```
d = [1, 2, '1', '2', -4, 3, 4]
```

Причем, все элементы следует привести к целым числам перед записью в множество. Для решения этой задачи удобно воспользоваться генератором множества:

```
a = {int(x) for x in d}
```

Видите, как легко и просто, в одну строчку мы это сделали. Мало того, генераторы работают быстрее циклов. Поэтому реализация задачи в виде:

```
set_d = set()
for x in d:
    set_d.add(int(x))
print(set_d)
```

будет и более громоздкой и более медленной. Поэтому там, где это возможно, лучше использовать соответствующие генераторы.

Основы Python

То же самое и со словарем. Допустим, нам нужно все его ключи записать заглавными буквами, а значения представить целыми числами:

```
m = {"неудовл.": 2, "удовл.": 3, "хорошо": '4', "отлично": '5'}
```

Опять же используем генератор:

```
a = {key.upper(): int(value) for key, value in m.items()}
```

и на выходе получаем искомый словарь. Видите, как элегантно это можно сделать с использованием генераторов.

Также как и со списками, в генераторах множеств и словарей можно использовать условия. Например, мы хотим в множество добавить только положительные числа из списка:

```
d = [1, 2, '1', '2', -4, 3, 4]
```

Также предварительно их преобразовываем в числа и делаем проверку:

```
a = {int(x) for x in d if int(x) > 0}
```

Или, со словарем. Поменяем в нем местами ключи и значения и поместим в него только отметки от 2 до 5:

```
m = {"безнадежно": 0, "убого": 1, "неудовл.": 2, "удовл.": 3, "хорошо": '4',  
"отлично": '5'}  
a = {int(value): key for key, value in m.items() if int(value) >= 2 and int(value) <= 5}
```

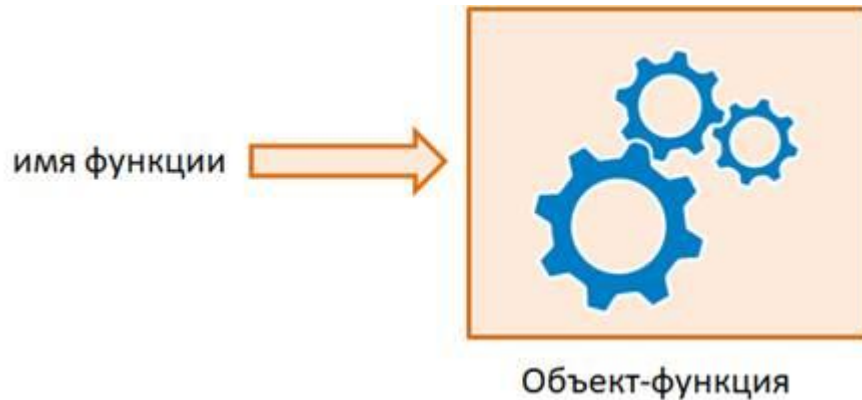
Вот так просто и легко можно использовать генераторы множеств и словарей в своих программах. Основное их преимущество перед обычными циклами – более высокая скорость работы и наглядность текста программы. Поэтому, если циклы можно относительно легко заменить генераторами, то стоит это делать. Также в генераторах множеств и словарей можно применять и вложенные схемы, когда один генератор вложен в другой. Причем, можно комбинировать генераторы списков с генераторами множеств и словарей.

Основы Python

Делается это все по аналогии с вложенными генераторами списков, о которых мы с вами уже говорили.

§35. Функции: первое знакомство, определение `def` и их вызов

Этим занятием мы открываем с вами одну из ключевых тем в программировании – **функции**.



Что это такое? Условно, их можно представить как активный объект, который не просто хранит какие-либо данные, а выполняет заданный фрагмент программы. Ссылка на такой объект называется именем функции. Например, знакомая нам функция **print** выполняет вывод в консоль переданных ей данных. Имя этой функции так и можно воспринимать, как ссылку на свой объект-функцию. Действительно, если в интерактивном режиме напечатать имя **print**, то увидим, что оно ссылается на встроенную функцию. Чтобы активизировать программу, заложенную в объект-функцию, нужно поставить круглые скобки после имени функции:

```
print()
```

Теперь был исполнен определенный код и в консоль была выведена пустая строка. Эти круглые скобки являются оператором вызова функций. То есть, само по себе имя – это ссылка на функцию, а имя функции с круглыми скобками – это уже вызов функции. Вот этот момент нужно очень хорошо и четко понимать, чтобы уметь правильно обращаться с функциями при построении программ.

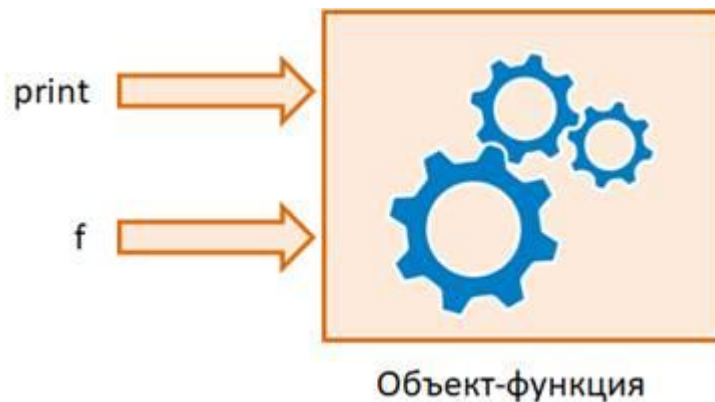
Основы Python

Итак, раз **print** – это ссылка на функцию, то что нам тогда мешает создать еще одну ссылку на этот же объект-функцию, через оператор присваивания:

```
f = print
```

И вызвать ее через второе имя **f**:

```
f("hello")
```



Как видите, все работает. Теперь функцию **print** можно вызывать и через **f**. А можно ли тогда исходному имени функции **print** присвоить другой объект, например, строку:

```
print = "это была функция print"
```

Да, и это нам никто не мешает делать. Теперь прежнюю функцию можно вызвать только через имя **f**:

```
f(print)
```

То есть, имя функции – это просто ссылка на объект-функцию и не более того. Конечно, заменять стандартные имена функций на другие, без острой необходимости не стоит. А вначале лучше совсем этого не делать. Именно этого правила и следует придерживаться.

Хорошо, мы теперь знаем, как правильно воспринимать функцию в **Python**. Но зачем они нужны? Разве нельзя писать программы и без них? Конечно

Основы Python

можно. И самые первые языки программирования их даже не имели. Но писать программы без них, это все равно, что поехать в другой город на лошади, а не на автомобиле. С функциями программы становятся более модульными, а процесс программирования значительно упрощается. **Так за счет чего это происходит?**

Допустим, мы пишем программу, в которой постоянно нужно делать отправку электронных писем по одному и тому же алгоритму. В результате, у нас в программе будут постоянно появляться одни и те же строчки кода для отправки писем. Такое дублирование программного кода негативно влияет на надежность программ, возможность их быстрого редактирования, читабельности и т.п. В программировании это называется кратко:

DRY – Don't Repeat Yourself

Как только, вы замечаете в программе повторение (дублирование) кода, значит, неверно строите ее архитектуру и структуру кода следует пересмотреть. Это очень грубая ошибка в программировании.

Так вот, чтобы нам устранить дублирование при отправке электронных писем, повторяющийся фрагмент можно поместить в функцию, а затем, вызывать ее в нужных местах программы. Это первое, для чего были введены функции в языки программирования – устранять дублирование кода.

В **Python** существует множество стандартных, встроенных функций, но на все случаи жизни их не придумаешь. Поэтому, программист может сам создавать свои собственные по мере необходимости. Для этого используется следующий синтаксис:

```
def <имя функции>([список аргументов]):  
    оператор 1  
    оператор 2  
    ...  
    оператор N
```

Основы Python

Здесь имя функции придумывается программистом подобно именам переменных и, так как функция – это определенное действие, то ее имя следует выбирать как глагол, например:

go, show, get, set и т.п.

Далее, идет набор операторов, которые образуют **тело функции**. Именно они начинают выполняться при ее вызове.

```
def <имя функции>([список аргументов]):  
    оператор 1  
    оператор 2  
    ...  
    оператор N
```

тело функции

Давайте в качестве примера, зададим простую функцию, которая будет имитировать отправку письма:

```
def send_mail():  
    text = "Уважаемый, Имеда Шерифадзе! Я так и не понял, что такое  
    функция. Объясните лучше!"  
    print(text)
```

Обратите внимание, имя функции **send_mail** – фраза-глагол, действие. Кроме того, два разных слова **send** и **mail** отделены символом подчеркивания. Я его записал для лучшей читаемости текста программы. И, глядя на название функции, сразу понятно, что она должна делать. Это рекомендуемый подход к программированию – называть переменные и функции понятными именами.

Тело этой функции состоит из двух команд: переменной **text** со строкой и функции **print()**. Так как вы дошли до этого урока, то должны хорошо понимать, почему эти две строчки образуют единый блок операторов тела функции. Да, у них у всех единый отступ от левого края. Здесь все также, как

Основы Python

и в условных операторах или операторах циклов. Поэтому повторяться не буду.

Итак, мы с вами определили функцию. Но, если сейчас запустить эту программу, то в консоли ничего не увидим, то есть, функция не сработала. Все верно, на данном этапе мы лишь определили, объявили функцию, но еще не вызывали. Как я говорил вначале, для вызова функции нужно записать ее имя и поставить круглые скобки – оператор вызова функции:

```
send_mail()
```

Теперь, при запуске программы, мы видим заветное сообщение. Обратите внимание, я записал вызов функции через две пустые строки после объявления функции. Это оформление по рекомендации стандарта **PEP8**. В интегрированной среде **PyCharm**, чтобы привести текст программы в соответствие с этим стандартом, достаточно нажать комбинацию клавиш **Alt + Ctrl + L**.

Эту же функцию мы можем вызвать и дважды:

```
send_mail()  
send_mail()
```

Соответственно, она сработает два раза подряд. И еще, обратите внимание, мы можем вызывать функцию только после ее объявления. То есть, сначала сделать ее вызов, а потом объявить не получится, возникнет ошибка, что имя **send_mail** не определено. Нужно сначала объявлять функции и только потом их вызывать.

Вернем два подряд идущих вызова функции **send_mail()** и посмотрим в режиме отладки, как будет выполняться эта программа (**показываем**). Если нажать **F8**, то мы сразу выполним функцию. Чтобы зайти внутрь ее нужно нажать клавишу **F7**. А в теле функции уже можно **F8**. Так будет понятнее, как выполняется эта программа.

Основы Python

Сейчас у нашей функции нет никаких параметров. Давайте добавим один с именем отправителя:

```
def send_mail(from_name):  
    text = f"""Уважаемый, Имеда Шерифадзе!  
Я так и не понял, что такое функция.  
Объясните лучше!  
Ваш, навсегда {from_name}!"""  
    print(text)
```

Для этого в круглых скобках записываем параметр с именем **from_name** (это имя мы придумываем сами, я решил назвать так) и, затем, в многострочной F-строке мы добавим это имя в конце сообщения.

Теперь функцию **send_mail()** следует вызывать с одним аргументом и по нашей задумке этот аргумент должен быть строкой, поэтому, запишем вызов, так:

```
send_mail("Иван Иванович")
```

а второй вызов поставим в комментарий. После запуска программы увидим сформированное сообщение. Здесь важно понять, как все это работает.

Во-первых, обратите внимание на терминологию. Определение внутри функции называется параметром, а передаваемое значение при вызове функции – аргументом. В дальнейшем, я буду пользоваться этими словами именно в этом смысле. **Во-вторых**, когда происходит вызов функции, то параметр ссылается на переданный аргумент. В итоге, в теле функции **from_name** принимает значение "Иван Иванович", которое и подставляется в шаблон сообщения.

Давайте посмотрим, как это происходит в режиме отладки (**показываем**).

А **что будет, если теперь попытаться вызвать эту же функцию без аргументов, так как мы это делали вначале?** (Убираем комментарий). Здесь уже возникает

Основы Python

ошибка, что функций ожидает один аргумент, а мы ничего не передаем. То же самое будет, если передать больше аргументов, например, **два**:

```
send_mail("Иван Иванович", "Иван Петрович")
```

То есть, функции нужно передавать ровно столько аргументов, сколько параметров в ней определено. Давайте пропишем второй параметр – возраст отправителя:

```
def send_mail(from_name, old):  
    text = f"""Уважаемый, Имеда Шерифадзе!  
    Я так и не понял, что такое функция.  
    Объясните лучше!  
    Ваш, навсегда {from_name}! И не судите строго, мне всего {old} лет."""  
  
    print(text)
```

Как видите, параметры записываются в круглых скобках через запятую и принимают значения аргументов при вызове функции:

```
send_mail("Иван Иванович", 7)  
send_mail("Иван Иванович", "Иван Петрович")
```

Причем, на первый аргумент ссылается первый параметр, а на второй – второй. То есть, порядок здесь имеет значение. Также, смотрите, при втором вызове у нас вместо числа указана строка. Параметру все равно на какой тип данных ссылаться, так как в **Python** используется динамическая типизация (мы об этом ранее говорили). В итоге, второе сообщение несколько бессмысленно, но, тем не менее, все сработало без ошибок.

Итак, на этом первом занятии по функциям, мы с вами узнали, что это такое и зачем они нужны, как объявляются функции и как они вызываются с разным числом аргументов. Для закрепления этого материала пройдите практические задания и переходите к следующему уроку, где мы продолжим изучение этой темы.

Основы Python

§36. Оператор return в функциях. Функциональное программирование

На прошлом занятии мы увидели, как их можно объявлять и вызывать. Сегодня сделаем следующий шаг и разберемся, как функции могут возвращать результаты своей деятельности.

Я открою программу, которую мы делали на предыдущем занятии с функцией `send_mail()`. И давайте здесь объявим еще одну функцию, которая бы вычисляла квадратный корень из положительных чисел:

```
def get_sqrt(x):  
    res = None if x < 0 else x ** 0.5
```

Обратите внимание, что в соответствии со стандартом **PEP8** каждое объявление функции разделяется двумя пустыми строчками. Рекомендуются так делать, чтобы текст программы был хорошо читаем.

Итак, у нас здесь функция с именем `get_sqrt()`, которое я сам придумал и одним параметром `x`. Внутри функции я воспользовался тернарным условным оператором, так что переменная `res` будет принимать `None` для отрицательных значений и квадратный корень – для неотрицательных.

Давайте попробуем ее вызвать и посмотрим, что она возвратит:

```
a = get_sqrt(49)  
print(a)
```

Во-первых, мы здесь видим новую конструкцию: переменной `a` присваиваем вызов функции. Это следует воспринимать так, что переменная `a` будет ссылаться на результат работы этой функции. Что такое результат работы функции, мы сейчас узнаем. В данном случае, значение `a` равно `None`. Фактически, это означает, что функция ничего не возвращает, никакого результата! **Но почему? Мы же в теле этой функции делаем вычисление квадратного корня?** И аргумент передаем положительный. Все дело в том, что внутри функции нужно явно указать, что именно она должна вернуть. Делается это с помощью специального оператора **return**, после которого через

Основы Python

пробел указываются данные, которые будут возвращены. В данном случае, пропишем переменную **res**.

Снова запустим программу и, смотрите, теперь наша переменная **a** ссылается на вычисленное значение. **Как это произошло?** Оператор **res** вернул объект с вычисленным значением **7.0** и это означает, что функция возвращает этот объект. Затем, с помощью оператора присваивания, создалась переменная **a** со ссылкой на этот объект. Поэтому при выводе этой переменной, мы видим значение **7.0**

Причем, вот этот вот параметр **x** и переменная **res** существуют только в момент вызова функции **get_sqrt()** и пропадают за ее пределами. Мы об этом еще поговорим. Но, пока имейте в виду, что за пределами функции вывести напрямую переменную **res**, например, не получится:

```
print(a, res)
```

возникнет ошибка, что имя **res** не определено.

Внутри тела функции можно указывать только один оператор **return**, либо ни одного, тогда функция будет возвращать значение **None**, как мы только что видели. Но, что если все же прописать в функции два оператора **return**, **что будет?** Давайте посмотрим:

```
def get_sqrt(x):  
    res = None if x < 0 else x ** 0.5  
    return res  
    return x
```

Нам интегрированная среда здесь сразу указывает писать этот оператор без отступов. Сейчас вы поймете почему. Запускаем программу и видим, что никаких ошибок нет и отображается прежнее значение **7.0**. Дело в том, что функция завершает свою работу сразу, как только встретит оператор **return**. Поэтому, как только была выполнена строка **return res**, все остальные команды в теле функции просто игнорируются. Убедимся в этом на отладке (**убеждаемся**). Поэтому второй **return** здесь просто бессмысленен – он

Основы Python

никогда не будет выполнен. Равно, как и любая другая команда, например, функция `print()`:

```
def get_sqrt(x):  
    res = None if x < 0 else x ** 0.5  
    return res  
    print(x)
```

А вот если ее записать до этого оператора, то увидим вывод `x`, а затем, вычисленного значения `res`.

Но все же, что делать, если нам нужно вернуть оба значения и `res` и `x`? Для этого следует поместить эти переменные в коллекцию, обычно, кортеж и вернуть ее:

```
def get_sqrt(x):  
    res = None if x < 0 else x ** 0.5  
    return (res, x)
```

На практике часто не пишут круглые скобки, а просто записывают элементы через запятую:

```
return res, x
```

В дальнейшем, я именно так и буду делать. Причем, теперь, мы можем принимать данные, используя множественное присваивание:

```
a, b = get_sqrt(49)  
print(a, b)
```

Видите, как это удобно, легко и наглядно делается в языке **Python**. Далеко не во всех языках программирования можно вот так запросто вернуть несколько значений через функцию.

Давайте зададим в программе еще одну функцию для определения максимального значения среди двух чисел:

Основы Python

```
def get_max2(a, b):  
    return a if a > b else b
```

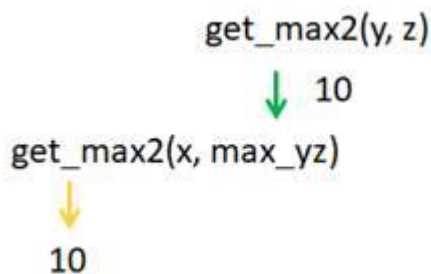
Казалось бы, что может быть примечательного в такой простой функции. Но, не торопитесь, сейчас кое-какая магия откроется перед вами. Вызвать эту функцию можно, очевидным образом, например, так:

```
x, y = 5, 7  
print(get_max2(x, y))
```

Я здесь специально определил дополнительно две переменные **x**, **y**, чтобы показать, что их также можно спокойно передавать в качестве аргументов. После запуска в консоли увидим максимальное значение **7**.

Давайте теперь усложним задачу и будем искать максимум среди трех чисел. Помните, для этого мы использовали вложенные условия. Здесь же, поступим иначе. Определим три переменные и воспользуемся все той же функцией **get_max2()**:

```
x, y, z = 5, 7, 10  
print(get_max2(x, get_max2(y, z)))
```



Вот это и есть обещанная магия. Смотрите, как это работает. Сначала будет вызвана функция, записанная в качестве аргумента, которая возвратит максимальное среди чисел **7** и **10**, то есть, значение **10**, а затем, вызывается первая функция, которая определяет максимум из чисел **5** и **10**. Соответственно, на выходе получаем результат **10**, который и выводится в консоль. Вот так функции можно записывать в аргументах и использовать в

Основы Python

своих программах. Мало того, мы можем ту же самую конструкцию использовать и при объявлении новой функции, определяющей максимум среди трех чисел:

```
def get_max3(a, b, c):  
    return get_max2(a, get_max2(b, c))
```

А, затем, просто вызвать ее с тремя аргументами:

```
print(get_max3(x, y, z))
```

Видите, как элегантно можно решить эту задачу, используя функциональный подход в программировании. Разумеется, здесь функция `get_max3()` должна объявляться после функции `get_max2()`, так как использует ее, иначе, мы бы получили ошибку.

Вообще, определение (**объявление**) функций в **Python** очень похоже на объявление переменных. Например, в программе, нам никто не запрещает объявить некую переменную с именем `get_rect` и присвоить ей значение `1` или `2` в зависимости от значения другой переменной `PERIMETR`:

```
PERIMETR = True
```

```
if PERIMETR:  
    get_rect = 1  
else:  
    get_rect = 2
```

Но оказывается, то же самое можно делать и на уровне функций, вместо переменной `get_rect` будем объявлять разные функции:

```
if PERIMETR:  
    def get_rect(a, b):  
        return 2 * (a + b)  
else:  
    def get_rect(a, b):
```

Основы Python

```
return a * b
```

И, если сейчас вызвать ее:

```
print(get_rect(1.5, 3.8))
```

то получим вычисление периметра прямоугольника. Если же переменную **PERIMETR** изменить на **False**:

```
PERIMETR = False
```

то функция с именем **get_rect()** теперь будет вычислять **площадь** прямоугольника. Здорово, да! Какая гибкость определения функций в языке **Python**! Такого мало где встретишь.

Наконец, мы можем вообще убрать условие и дважды определить одну и ту же функцию:

```
def get_rect(a, b):  
    return 2 * (a + b)
```

```
def get_rect(a, b):  
    return a * b
```

К ошибкам это не приведет, а просто ссылка **get_rect** будет инициализирована на второй объект-функцию и, соответственно, она и будет вызвана.

В заключение занятия приведу еще один пример использования функций. Объявим функцию для определения четности числа:

```
def even(x):  
    return x % 2 == 0
```

А, затем, вызовем ее в цикле для определения: является ли текущее число четным:

Основы Python

```
for i in range(1, 20):  
    if even(i):  
        print(i)
```

После запуска в консоли увидим отображение только четных чисел.

На этом мы завершим наше второе занятие по функциям. Из него вам должно быть понятно, как и для чего используется оператор **return**, как вызываются функции, записанные в аргументах другой функции, а также объявление функций, подобно переменным. Если все это, действительно, понятно, то переходите к практическим заданиям, пока я собираюсь с мыслями для следующего урока.

§37. Алгоритм Евклида для нахождения НОД

На этом занятии я хочу показать вам пример использования функций для решения одной частной задачи – нахождения наибольшего общего делителя (НОД) для двух натуральных чисел **a** и **b**. Причем, мы не просто напишем алгоритм, а еще выполним его тестирование с применением тестирующей функции. То есть, это будет полноценный пример, показывающий принцип разработки программ с использованием функций и тестов.

Но, вначале пару слов о самом **алгоритме Евклида**, о принципе его работы. Сначала рассмотрим его медленный, но простой вариант.

Например, пусть даны два натуральных числа: **a = 18** и **b = 24**. Чтобы определить для них НОД, будем действовать, следующим образом. Из большего значения вычтем меньшее и результат сохраним в переменной с большим значением, то есть, в **b**. Фактически, это означает, что мы выполняем операцию: **b = b - a**. Теперь у нас два значения **a = 18**, **b = 6**. Для них повторяем тот же самый процесс. Здесь большее уже переменная **a**, поэтому, корректируем ее значение, вычитая меньшее. Получаем новую пару **a = 12**, **b = 6**. Опять повторяем этот процесс и видим, что **a = 6**, **b = 6** – переменные равны. В этом случае останавливаем алгоритм и получаем, что **НОД(18, 24) = 6**, что, в общем то, верно.

Основы Python

Весь этот алгоритм можно представить следующим псевдокодом:

```
пока a != b
    находим большее среди a и b
    уменьшаем большее на величину меньшего

выводим полученное значение величины a (или b)
```

Давайте его опишем с помощью, следующей функции:

```
def get_nod(a, b):
    """Вычисляется НОД для натуральных чисел a и b
    по алгоритму Евклида.
    Возвращает вычисленный НОД.
    """
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a

    return a
```

Смотрите, здесь вначале идет многострочная строка с описанием работы функции. Так рекомендуется делать для ключевых функций программы, чтобы другие программисты могли быстро понимать, как их применять на практике. А, далее, после описания следует сам **алгоритм Евклида**.

Выполним эту функцию со значениями аргументов **18** и **24**:

```
res = get_nod(18, 24)
print(res)
```

Видим в консоли верное значение **6**. Вот пример правильного оформления ключевых функций программы. Мало того, встроенная функция:

Основы Python

```
help(get_nod)
```

позволяет выводить описание указанных функций. И мы видим в консоли наше сформированное сообщение. Это очень удобно, особенно при групповой работе над проектом.

После того, как функция определена, ее следует протестировать и убедиться в корректности возвращаемых результатов. Для этого тестировщик создает свою вспомогательную функцию. Используя наши текущие знания, мы ее опишем, следующим образом:

```
def test_nod(func):  
    # -- тест №1 -----  
    a = 28  
    b = 35  
    res = func(a, b)  
    if res == 7:  
        print("#test1 - ok")  
    else:  
        print("#test1 - fail")  
  
    # -- тест №2 -----  
    a = 100  
    b = 1  
    res = func(a, b)  
    if res == 1:  
        print("#test2 - ok")  
    else:  
        print("#test2 - fail")  
  
    # -- тест №3 -----  
    a = 2  
    b = 10000000  
  
    st = time.time()
```


Основы Python

```
res = func(a, b)
et = time.time()
dt = et - st
if res == 2 and dt < 1:
    print("#test3 - ok")
else:
    print("#test3 - fail")
```

В первых двух тестах мы проверяем корректность вычислений, а в третьем – еще и скорость работы. Конечно, это довольно примитивное тестирование, показывающее лишь принцип разработки программы, но для учебных целей вполне достаточно.

Далее, выполним импорт нужного нам модуля **time** для вызова функции **time()**:

```
import time
```

и в конце вызовем тестирующую функцию для тестирования **get_nod**:

```
test_nod(get_nod)
```

Смотрите, у нас первые два теста прошли, а третий – не прошел, так как функция слишком долго вычисляла результат.

Давайте поправим ее и ускорим алгоритм Евклида. **Как это можно сделать?** Смотрите, если взять два числа **a = 2** и **b = 100**, то по изначальному алгоритму мы будем делать многочисленные вычитания из **b a**, пока значения не сравняются. То есть, мы здесь, фактически, вычисляем остаток от вхождения двойки в сотню, а это есть не что иное, как операция:

$$b = b \% a = 0$$

И никаких циклических вычитаний! Это, очевидно, будет работать много быстрее. При этом, как только получаем остаток равный нулю, то **НОД** – это значение меньшей переменной, то есть, в нашем примере – **a = 2**.

Основы Python

То же самое для предыдущих значений $a = 18$, $b = 24$. Получаем серию таких вычислений:

$$b = 24 \% 18 = 6$$

$$a = 18 \% 6 = 0$$

Значит, $\text{НОД}(18, 24) = 6$. Видите, как это быстро и просто! На уровне псевдокода быстрый **алгоритм Евклида** можно описать так:

пока меньшее число больше 0

 большому числу присваиваем остаток от деления на меньшее число
выводим большее число

Реализуем его в виде функции:

```
def get_fast_nod(a, b):
    """Вычисляется НОД для натуральных чисел a и b
    по быстрому алгоритму Евклида.
    Возвращает вычисленный НОД.
    """
    if a < b:
        a, b = b, a

    while b != 0:
        a, b = b, a % b

    return a
```

Предлагаю, в качестве самостоятельного задания, вам самим в деталях разобраться, как она работает. А мы ее сразу протестируем:

```
test_nod(get_fast_nod)
```

Как видите, она проходит все три наших теста.

Основы Python

Надеюсь, из этого занятия мне удалось донести до вас общий принцип разработки и тестирования ключевых программных функций. А также объяснить работу **алгоритма Евклида**. Если все это понятно, то смело переходите к следующему уроку.

§38. Именованные аргументы. Фактические и формальные параметры

Сейчас мы с вами будем говорить о способах определения параметров в функциях и передачи им аргументов. Казалось бы, мы подробно разобрали эту тему и каждый из вас уже знает, что при объявлении функции в круглых скобках через запятую можно указать сколько угодно параметров. Например, простая функция вычисления объема прямоугольного параллелепипеда, очевидно должна принимать, как минимум, три параметра (ширину, высоту и глубину):

```
def get_V(a, b, c):  
    print(f'a = {a}, b = {b}, c = {c}')  
    return a * b * c
```

А, затем, может быть вызвана с конкретными числовыми значениями:

```
v = get_V(1, 2, 3)  
print(v)
```

Причем, параметру **a** будет соответствовать число **1**, параметру **b** – число **2**, а **c** – число **3**. Это так, потому что здесь используется позиционная запись аргументов при вызове функции, то есть, значения параметров **a**, **b**, **c** определяются порядком записи аргументов. **А можно ли, не меняя порядка, параметру b присвоить 1, параметру c – 2, а a – 3?** Оказывается да, в языке **Python** такое возможно, если явно указывать имена параметров при вызове функции:

```
v = get_V(b=1, a=2, c=3)
```

Такие аргументы называются именованными. Теперь, при запуске программы, мы видим, указанные значения у параметров **a**, **b** и **c**.

Основы Python

А можем ли мы комбинировать позиционные и именованные аргументы? Да и такое тоже возможно. Только вначале следует указывать позиционные, а в конце – именованные, например, так:

```
v = get_V(1, c=2, b=3)
```

Если же, мы не будем следовать этому правилу и позиционные аргументы запишем после именованного:

```
v = get_V(a=1, 2, 3)
```

то возникнет синтаксическая ошибка – так делать нельзя. Сначала всегда позиционные и только потом – именованные:

```
v = get_V(1, 2, c=3)
```

Причем, обратите внимание, последним именованным аргументом здесь может быть только имя параметра **c**. Если записать любой другой из трех, например **b**:

```
v = get_V(1, 2, b=3)
```

то у нас получится дублирование передаваемых данных. Второй позиционный аргумент уже присваивается параметру **b**, а далее, мы снова этому же параметру присваиваем значение **3**. Так делать нельзя.

Вернемся теперь к параметрам самой функции. Мы их объявили просто через запятую с именами **a**, **b**, **c**. Однако, можно задавать параметры со значениями по умолчанию, например, так:

```
def get_V(a, b, c, verbose=True):  
    if verbose:  
        print(f'a = {a}, b = {b}, c = {c}')  
  
    return a * b * c
```

Основы Python

Такие параметры называются формальными, а обычные – фактическими. **В чем отличие формальных параметров от фактических, помимо значений по умолчанию?** Их не обязательно прописывать при вызове функции. Например, наш прежний вызов:

```
v = get_V(1, 2, 3)
```

сработает без каких-либо проблем. Мы не указали аргумент для последнего формального параметра **verbose**. В этом случае он принимает значение по умолчанию **True**. Если же указать его:

```
v = get_V(1, 2, 3, False)
```

то функция **print()** внутри функции вызвана уже не будет. Разумеется, можно использовать и соответствующий именованный аргумент:

```
v = get_V(1, 2, 3, verbose=False)
```

Все будет работать также.

Зачем вообще нужны формальные параметры и когда их следует использовать? Я, думаю, ответ здесь очевиден – для удобства использования функций. Как мы только что видели, аргументы формальным параметрам можно не передавать, если нас устраивает поведение функции по умолчанию. В других, как полагается, редких ситуациях, всегда можно поменять значение такого параметра на другое и скорректировать работу функции.

Чтобы это было понятнее, приведу такой простой пример. Допустим, мы собираемся сравнивать строки в разных режимах: с учетом и без учета регистра, а также учитывать или не учитывать пробелы в начале и в конце. Для этого объявим следующую функцию:

```
def compare_str(s1, s2, reg=False, trim=True):  
    if reg:  
        s1 = s1.lower()  
        s2 = s2.lower()
```

Основы Python

```
if trim:
    s1 = s1.strip()
    s2 = s2.strip()

return s1 == s2
```

Формальные параметры **reg** и **trim** определяют наиболее частый вариант использования операции сравнения строк: с учетом регистра и с удалением пробелов.

Теперь мы можем вызывать эту функцию, просто с двумя аргументами:

```
print(compare_str("Python ", " Python"))
```

или менять ее поведение, указывая другие значения формальных параметров:

```
print(compare_str("Python ", " Python", trim=False))
print(compare_str("Python", "PYTHON", True, False))
```

В последнем варианте записаны обычные позиционные аргументы. В этом случае параметр **reg** примет значение **True**, а параметр **trim** – **False**. Конечно, всегда можно воспользоваться и именованными аргументами, здесь полная свобода выбора:

```
print(compare_str("Python", "PYTHON", reg=True, trim=False))
```

В заключение этого занятия хочу показать вам один нюанс при объявлении функции с формальными параметрами. Предположим, мы определяем вот такую очень простую функцию:

```
def add_value(value, lst=[]):
    lst.append(value)
    return lst
```

У нее один фактический и один формальный параметр, причем, второй параметр **lst** по умолчанию ссылается на изменяемый тип данных – список.

Основы Python

Конечно, нам никто не запрещает этого делать, но давайте посмотрим, как она будет работать. Вызовем ее два раза:

```
l = add_value(1)
l = add_value(2)
print(l)
```

В консоли видим два значения в списке **1** и **2**. Возможно, объявляя таким образом функцию, мы ожидали, что при каждом вызове параметр **lst** будет ссылаться на пустой список и функция будет каждый раз возвращать один элемент в этом списке. Но, получилось, что она сохраняет прежнее состояние списка при повторном вызове. **Почему так произошло?**

На самом деле, все просто. Когда мы объявляем функцию, то создается объект-функция и объекты для формальных параметров, в данном случае пустой список. Когда, затем, мы вызываем функцию, то (опуская некоторые детали) формальный параметр **lst** ссылается на этот список и добавление элемента происходит в него. При повторном вызове функции **lst** продолжает ссылаться на этот же список и в него добавляется следующее значение. Это все показывает, что при вызовах функции список не инициализируется повторно, а используется один и тот же объект.

Конечно, мы можем при вызовах функции каждый раз передавать пустой список:

```
l = add_value(1, [])
l = add_value(2, [])
```

И тогда формальный параметр **lst** будет при каждом вызове ссылаться уже на новый пустой список. А **как сделать так, чтобы такое поведение было по умолчанию, чтобы нам явно не приходилось передавать пустой список?** Исправить ситуацию можно, например, так:

```
def add_value(value, lst=None):
    if lst is None:
        lst = []
```

Основы Python

```
lst.append(value)
return lst
```

Значение формального параметра определим неизменяемым значением **None**, а в самой функции сделаем проверку, если этот параметр равен **None**, то есть, функция вызвана с одним первым аргументом, то создаем новый пустой список и в него помещаем значение. В этом случае, при каждом вызове функции:

```
l = add_value(1)
l = add_value(2)
```

будет автоматически создаваться новый список. То, что мы и хотели. А если хотим продолжить предыдущий, то достаточно его указать вторым аргументом:

```
l = add_value(2, l)
```

Вот такой нюанс существует при определении формальных параметров функции на изменяемые объекты, такие как списки, словари и множества.

Итак, из этого занятия вам нужно запомнить, что такое позиционные и именованные аргументы и как их можно записывать и комбинировать. Что такое фактические и формальные параметры, зачем нужны и как используются на практике, ну и, конечно же, как работают формальные параметры с изменяемыми типами данных. Закрепляйте материал практическими заданиями и переходите к следующему уроку.

§39. Функции с произвольным числом параметров ***args** и ****kwargs**

На этом занятии узнаем, как функциям можно передавать произвольное число аргументов. **Где это используется, я думаю, вы понимаете?** Например, известная нам функция

```
max(1, 2, 3, -4)
```


Основы Python

может принимать разное число аргументов и возвращает максимальное значение. **Как можно самим определять такие функции?** Делается это очень просто. Допустим, мы с вами хотим задать функцию для формирования маршрута к файлу:

`F:\~stepik.org\Добрый, добрый Python (Питон)\39\p39. Функции.docx`

И этот маршрут может состоять из нескольких частей. Причем, число фрагментов может быть произвольным. Опишем такую функцию. Я назову ее `os_path()`, а вместо списка параметров запишу звездочку и одну переменную `args`:

```
def os_path(*args):  
    print(args)
```

Для начала, посмотрим, как это будет работать. Вызовем функцию с тремя строковыми аргументами:

```
os_path("F:\~stepik.org", "Добрый, добрый Python (Питон)", "39\p39.  
Функции.docx")
```

Не забываем здесь про экранирование обратных слешей. Выполним эту программу и в консоли видим, что переменная `args` ссылается на кортеж со значениями переданных трех аргументов. **Здорово, да?!** Чтобы функция принимала произвольное число аргументов, в ее объявлении достаточно у параметра прописать оператор `*`. Это оператор упаковки аргументов в кортеж и через переменную `args` мы сможем с ним работать.

Давайте теперь довершим нашу функцию и сформируем полный путь на основе его фрагментов. Сделать это можно с помощью знакомого нам метода `join()`, следующим образом:

```
def os_path(*args):  
    path = "\\".join(args)  
    return path
```

Основы Python

И, далее, вызвать эту функцию:

```
p = os_path("F:\\~stepik.org", "Добрый, добрый Python (Питон)", "39\\p39. Функции.docx")
print(p)
```

Надеюсь, из этого примера, вам понятно, как объявлять функцию с произвольным числом фактических параметров. Хорошо, а что если мы передадим этой функции дополнительно один именованный аргумент:

```
p = os_path("F:\\~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx",
            sep='/')
)
```

При запуске программы увидим ошибку, что функция не имеет такого формального параметра. Дело в том, что записывая объявление ***args** мы определяем лишь произвольное число фактических параметров, но не формальных. **Как это можно поправить?** Здесь есть, по крайней мере, два способа. В самом простом варианте, достаточно прописать этот формальный параметр в объявлении функции:

```
def os_path(*args, sep='\\'):
    path = sep.join(args)
    return path
```

И теперь никаких проблем с вызовом нет. Но, конечно, указать, какой-либо другой именованный аргумент мы не можем:

```
p = os_path("F:\\~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx",
            sep='/', trim=True)
)
```

Основы Python

Снова получим ту же самую ошибку. Так **как же определить в функции произвольное число формальных параметров?** Делается это с помощью следующего синтаксиса:

```
def os_path(*args, **kwargs):  
    print(kwargs)  
    path = kwargs['sep'].join(args)  
    return path
```

Мы прописываем уже две звездочки, а затем, имя переменной, которая будет ссылаться на упакованные значения в виде словаря. Убедимся в этом, выполним программу и смотрите, в консоли коллекция **kwargs** действительно представляет собой словарь, ключами которого являются имена аргументов, а значениями – значения аргументов. Все очень удобно и просто, как всегда в **Python!**

Причем, коллекция ****kwargs** обязательно должна быть записана после коллекции ***args**, наоборот нельзя, так как вначале должны идти фактические параметры и только потом – формальные. Мало того, мы можем некоторые параметры указывать явно, например:

```
def os_path(*args, sep='\\', **kwargs):  
    path = sep.join(args)  
    return path
```

И, тем самым, гарантировать их существование внутри функции. А другие, передаваемые именованные аргументы, следует проверять, прежде чем использовать, например, для параметра **trim** сначала делаем проверку его существования в словаре **kwargs**, а затем, смотрим, чему равно это значение:

```
def os_path(*args, sep='\\', **kwargs):  
    if 'trim' in kwargs and kwargs['trim']:  
        args = [x.strip() for x in args]  
  
    path = sep.join(args)
```

Основы Python

```
return path
```

Если условие выполняется, то удаляем пробелы до и после фрагментов путей к файлу.

То же самое и с фактическими параметрами. Некоторые из них можно явно указать, при объявлении функции:

```
def os_path(disk, *args, sep='\\', **kwargs):
    args = (disk,) + args

    if 'trim' in kwargs and kwargs['trim']:
        args = [x.strip() for x in args]

    path = sep.join(args)
    return path
```

И тогда на первый аргумент будет ссылаться параметр **disk**, а остальные позиционные аргументы упаковываться в коллекцию **args**:

```
p = os_path("F:", "~stepik.org",
            "Добрый, добрый Python (Питон)",
            "39\\p39. Функции.docx",
            sep='/', trim=True
)
```

Вот принцип, по которому объявляются функции с произвольным числом фактических и формальных параметров.

Вам осталось закрепить этот материал практическими заданиями и жду всех вас на следующем уроке.

§40. Операторы * и ** для упаковки и распаковки коллекций

На этом занятии я хочу немного отступить от темы функций и рассказать об операторах * и **, которые мы затронули на предыдущем уроке.

Основы Python

Мы знаем, что они позволяют упаковывать аргументы в **кортеж** и **словарь**. Но их можно использовать не только в объявлении функций, но и при работе с разными коллекциями. Например, если взять кортеж из двух значений:

```
x, y = (1, 2)
```

то его можно распаковать в две переменные. Но, если мы пропишем там больше значений, например, четыре:

```
x, y = (1, 2, 3, 4)
```

то уже получим ошибку, так как элементов четыре, а переменных всего две. Но, используя оператор `*`, мы можем упаковать оставшиеся значения во вторую переменную:

```
x, *y = (1, 2, 3, 4)
```

или, в первую, без разницы:

```
*x, y = (1, 2, 3, 4)
```

То же самое можно проделывать и со списками:

```
x, *y = [1, "a", True, 4]
```

и строками:

```
*x, y, z = "Hello Python!"
```

И вообще с любыми итерируемыми объектами. То есть, оператор `*` упаковывает оставшиеся значения в список. Правда, мы не можем упаковывать уже упакованные данные, например, так:

```
*y = 1, 2, 3
```

произойдет ошибка, но вот так:

```
x, *y = 1, 2, 3
```

Основы Python

уже будет работать.

Этот же оператор может выполнять и обратную операцию – распаковывать коллекции в набор данных. Пусть у нас имеется список:

```
a = [1, 2, 3]
```

И на его основе мы хотим сформировать кортеж. Если просто записать переменную в круглых скобках:

```
(a,)
```

то увидим кортеж со списком внутри. Но, если прописать оператор `*` перед списком:

```
(*a,)
```

то произойдет распаковка его элементов и список превратится в кортеж. То же самое можно сделать и при вызове функций. Допустим, определим кортеж из двух значений:

```
d = -5, 5
```

и вызовем с этими значениями функцию:

```
range(d)
```

Возникнет ошибка, так как функция ожидает числа в качестве аргументов, а не коллекции. Но, мы можем распаковать кортеж `d` в два числа, поставив перед ним оператор `*`:

```
range(*d)
```

и теперь никаких ошибок нет. Давайте посмотрим, что вернет эта функция, преобразуем все к списку:

```
list(range(*d))
```

Основы Python

Да, получаем ожидаемые значения от -5 до 5. И, теперь, это же преобразование генератора `range()` в список мы можем сделать и так:

```
[*range(*d)]
```

Здорово, да?! Мы оператором `*` распаковали итерируемый объект и составили из его значений список.

Мало того, мы таким образом можем делать объединение разных коллекций в одну коллекцию, например, список:

```
[*range(*d), *(True, False), *a]
```

Как видите, оператор `*` - невероятно удобный инструмент. И те же самые действия можно делать и со словарем. Зададим, следующий словарь с расшифровкой оценок:

```
d = {0: "безнадежно", 1: "убого", 2: "неуд.", 3: "удовл.", 4: "хорошо", 5: "отлично"}
```

Распаковать его можно двумя способами. Если прописать один оператор `*`:

```
{*d}
```

То получим множество, состоящее из ключей этого словаря. Или, можно сформировать список из ключей:

```
[*d]
```

То есть, оператор `*` перебирает словарь как обычный итерируемый объект и по умолчанию, перебираются именно ключи. Если нам нужно перебрать значения, то следует вызвать дополнительно метод:

```
[*d.values()]
```

и получим список из значений. Соответственно, метод:

Основы Python

```
[*d.items()]
```

вернет кортежи из пары ключ-значение.

Если же требуется распаковать словарь как словарь, то перед ним следует прописать две звездочки:

```
{**d}
```

Теперь вместо множества мы получаем словарь. **Где это нам может пригодиться?** Например, для объединения нескольких словарей в один. Создадим еще один словарь:

```
d2 = {6: "превосходно", 7: "элитарно", 8: "божественно"}
```

И соединим их через распаковку данных:

```
{**d, **d2}
```

На выходе получаем новый словарь с объединенными данными. Такой прием часто используется на практике, когда нужно объединить сразу несколько словарей. А вот для упаковки оператор ****** не используется, если прописать, что то вроде:

```
a, b, **c = d
```

то получим синтаксическую ошибку. Можно указывать только одну звездочку. Исключение только параметр ****kwargs** при определении функций.

Надеюсь, теперь вы знаете, как работают операторы ***** и ****** для упаковки и распаковки коллекций. Закрепляйте все практическими заданиями и переходите к следующему уроку.

§41. Рекурсивные функции

На этом занятии речь будет идти о рекурсивных функциях. То есть, о функциях, которые вызывают сами себя.

Основы Python

На первый взгляд может показаться невероятным, что функция может вызывать саму себя. Но это так. И, кстати, так можно делать во всех современных языках программирования, а не только в **Python**. Поэтому, изучая рекурсивные функции, вы, фактически, знакомитесь с фундаментальным материалом, который применим и в других языках программирования.

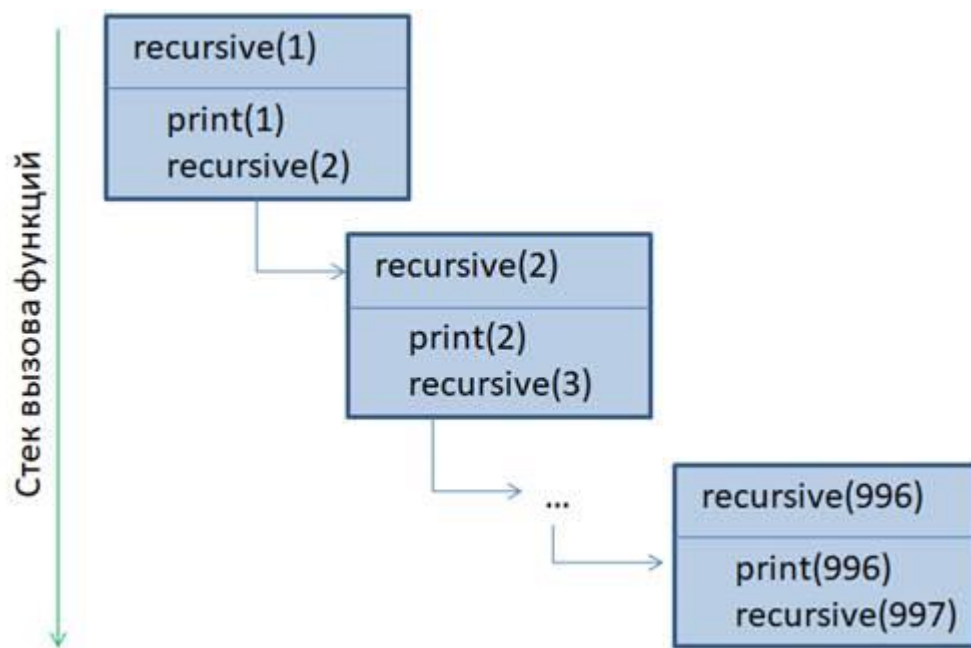
Итак, сразу к примеру. Вот простая рекурсивная функция:

```
def recursive(value):  
    print(value)  
    recursive(value+1)
```

Давайте ее вызовем с начальным значением один и посмотрим, что получится:

```
recursive(1)
```

Смотрите, функция **996** раз вызвала саму себя и произошла ошибка – достигнута максимальная глубина рекурсии. **Что это за глубина и как все это работает?** Сейчас все узнаете.

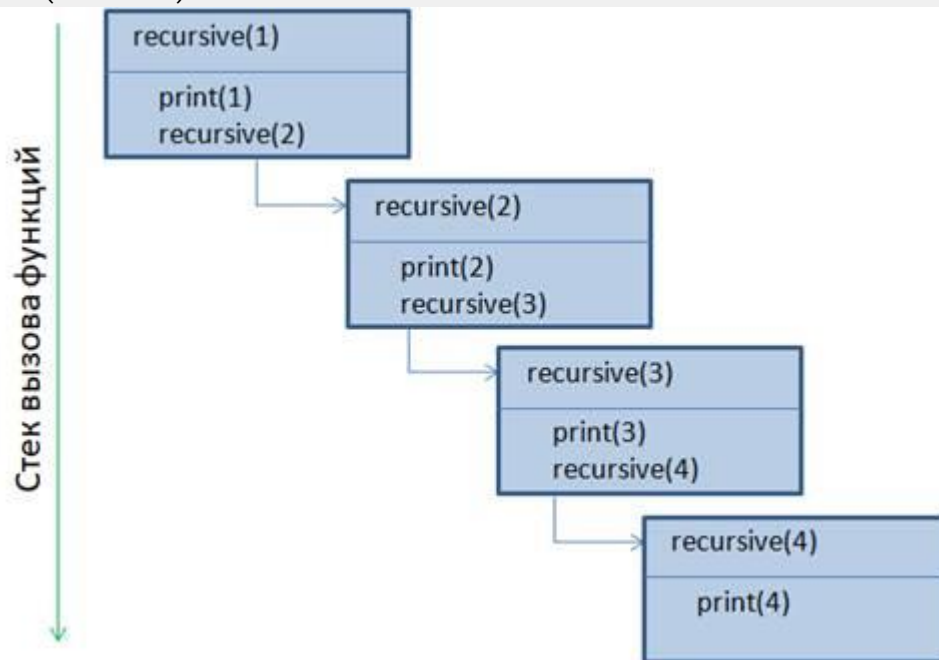


Основы Python

Когда вызывается какая-либо функция, то она помещается в стек вызова функций, в котором хранится порядок вызова различных функций. Затем, выполняется тело функции и здесь у нас снова встречается вызов той же самой функции, но с аргументом на единицу больше. Соответственно, в стеке появляется новая запись. Далее, все повторяется до тех пор, пока этот стек полностью не заполнится и не произойдет уже известная нам ошибка.

Что же сделать, чтобы такой ошибки не возникало? Очевидно, нужно ограничить глубину рекурсии. То есть, у любой рекурсивной функции должно быть условие останова, чтобы она не продолжалась вечно. В нашем случае условие для прерывания дальнейшей рекурсии можно записать, следующим образом:

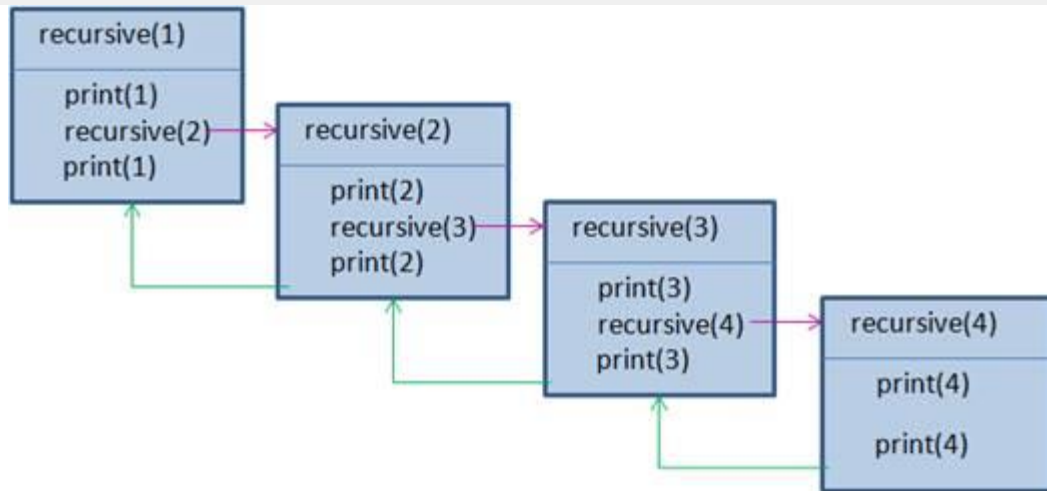
```
def recursive(value):  
    print(value)  
    if value < 4:  
        recursive(value+1)
```



В этом случае функция будет вызвана ровно четыре раза. И в консоли увидим значения от 1 до 4. А теперь последний штрих для истинного понимания работы рекурсии. После условия запишем еще один вызов функции **print()**:

Основы Python

```
def recursive(value):  
    print(value)  
    if value < 4:  
        recursive(value+1)  
    print(value)
```



Запускаем программу и видим сначала увеличение переменной **value**, а затем, уменьшение. **Что за магия? Откуда взялось это уменьшение?** Как всегда, все просто. Вначале вызывается **recursive(1)** и печатается значение **1**, затем, по рекурсии переходим к **recursive(2)**, выводится **2** и так до **4**. Когда значение **value** становится равным **4**, условие становится ложным и рекурсия прекращается. Но после условия записана функция **print()**, которая выведет текущее значение переменной **value**, то есть, **4**. **Что будет дальше?** А дальше текущая функция завершает свою работу, она извлекается из стека и управление передается предыдущей функции. А предыдущая функция – это **recursive(3)**, в которой первые две команды уже выполнены и осталась одна – последний **print(3)**, который печатает значение **3**. Эта функция также завершается, извлекается из стека и мы попадаем в функцию **recursive(2)**, где также осталось выполнить последнюю строчку **print(2)** – напечатать **2**. И так до самой вершины, где печатаем **1**. Вот теперь, вы знаете, как работают рекурсивные функции.

Основы Python

Чтобы это все лучше улеглось, приведу два классических примера использования рекуррентных функций. **Первый** – это вычисление факториала натурального числа n . Можно заметить, что:

$$n! = (n-1)! * n$$

и так далее:

$$(n-1)! = (n-2)! * (n-1)$$

...

$$2! = 1 * 2$$

$$1! = 1$$

Чем нам это может помочь? Если ввести функцию:

$$\text{fact}(n) = n!$$

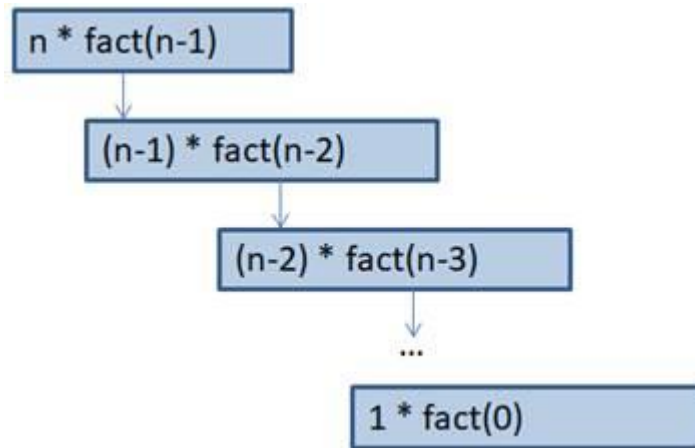
То этот факториал можно вычислить по рекурсии:

$$\text{fact}(n) = n * \text{fact}(n-1)$$

Именно ее мы сейчас и реализуем:

```
def fact(n):  
    if n <= 0:  
        return 1  
    else:  
        return n * fact(n-1)
```

Основы Python



Вначале проверяем, если параметр **n** меньше или равен нулю, то возвращаем единицу, тогда **fact(0)** будет равен 1. А иначе идем по рекурсии, уменьшая значение **n**, то есть, мы будем устремляться к нулю. В итоге, у нас будет формироваться последовательность:

$$n * (n-1) * (n-2) * \dots * 1$$

Запустим эту функцию при **n = 6** и видим в консоли значение **720**, что верно:

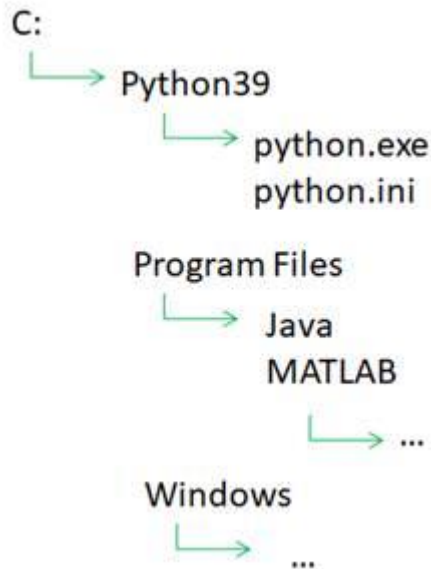
```
res = fact(6)
print(res)
```

Второй пример будет связан с рекурсивным обходом каталогов и файлов. Так как пока мы с вами не работали с файловой системой, то для имитации структуры файлов, я создам следующий словарь:

```
F = {
    'C:': {
        'Python39': ['python.exe', 'python.ini'],
        'Program Files': {
            'Java': ['README.txt', 'Welcome.html', 'java.exe'],
            'MATLAB': ['matlab.bat', 'matlab.exe', 'mcc.bat']
        },
    },
    'Windows': {
        'System32': ['acledit.dll', 'aclui.dll', 'zipfldr.dll']
    }
}
```

Основы Python

```
}  
}  
}
```



А для обхода этой коллекции запишем следующую рекурсивную функцию:

```
def get_files(path, depth=0):  
    for f in path:  
        print(" "*depth, f)  
        if type(path[f]) == dict:  
            get_files(path[f], depth+1)  
        else:  
            print(" "*(depth+1), ", ".join(path[f]))
```

Вначале, при запуске, мы ей передадим исходный словарь:

```
get_files(F)
```

Затем, в цикле будем перебирать ключи этого словаря. В данном случае, он один – это строка 'C:'. Отображаем этот ключ с отступом в **depth** пробелов от левого края и проверяем, является ли значение этого ключа словарем. Если так, то имеем набор вложенных каталогов, которые также перебираются этой же функцией, вызванной по рекурсии и с отступом на один больше. В

Основы Python

рекурсиях отображаем все вложенные каталоги (ключи вложенных словарей), а если встречается список, то рекурсия останавливается, а список файлов отображается через запятую.

После запуска программы, видим в консоли структуру каталогов и файлов в соответствии со словарем **F**. Вот так, достаточно просто можно реализовать перебор иерархических данных с помощью рекурсивных функций. В подобных задачах они незаменимы и часто используются на практике.

На этом мы завершим наше занятие. Из него вы должны хорошо себе представлять работу рекурсивных функций. Выполните практические задания для закрепления материала и жду всех вас на следующем уроке.

§42. Анонимные (lambda) функции

Это занятие будет посвящено, так называемым, анонимным или, их еще называют, **lambda** функциям. **Что это за функции и почему их называют анонимными?** Сейчас вы все узнаете!

Лямбда функция определяется по очень простому синтаксису:

lambda param_1, param_2, ...: команда

и записывается в программе, как оператор. Например, вот так можно определить лямбда-функцию для сложения двух значений:

```
lambda a, b: a + b
```

У нее два параметра **a** и **b**, а затем, через двоеточие написано, что с ними нужно сделать. Полученное значение будет автоматически возвращено этой функцией. И, обратите внимание, у этой функции нет имени, поэтому она и называется анонимной. **Но как тогда ее вызывать?** Для этого объект-функцию, который создает лямбда-функция, нужно присвоить какой-либо переменной:

```
s = lambda a, b: a + b
```

Основы Python

И уже через нее вызывать саму функцию:

```
s(1, 2)
```

В чем особенность такого определения функции? Зачем ее придумали и почему бы не пользоваться обычными функциями? У нее есть одно принципиальное отличие от ранее рассматриваемых нами функций – она может быть записана как элемент любой конструкции языка **Python**. Например, прямо как элемент списка:

```
a = [4, 5, lambda: print("lambda"), 7, 8]
```

Мы здесь описываем лямбда-функцию и сразу же передаем ее в список. С обычными функциями так не получится. Они должны быть объявлены заранее и только потом мы могли бы передать ссылку на нее в список. Если мы сейчас обратимся к третьему элементу этого списка:

```
a[2]
```

то увидим ссылку на объект-функцию. И, как мы уже знаем, чтобы ее запустить на выполнение нужно прописать оператор – круглые скобки:

```
a[2]()
```

И, действительно, функция была выполнена и в консоли отобразилось сообщение.

Конечно, анонимные функции, обычно, не используются для вывода какой-либо информации. Как правило, они выполняют определенную команду и возвращают полученный результат. Для ясности, приведу такой пример. Предположим, у нас имеется список:

```
lst = [5, 3, 0, -6, 8, 10, 1]
```

и мы хотим написать функцию, которая бы выбирала значения из этого списка по определенному критерию (**фильтру**):

Основы Python

```
def get_filter(a, filter=None):
    if filter is None:
        return a

    res = []
    for x in a:
        if filter(x):
            res.append(x)

    return res
```

Здесь второй параметр **filter** – это ссылка на другую функцию, которая будет отбирать значения из списка **lst**. Если вызвать ее только с первым аргументом:

```
lst = [5, 3, 0, -6, 8, 10, 1]
r = get_filter(lst)
print(r)
```

То увидим в консоли тот же самый список без изменений. Но, если вторым аргументом передать вот такую лямбда-функцию:

```
r = get_filter(lst, lambda p: p % 2 == 0)
```

то возвратится новый список, состоящий только из четных значений. **Как это произошло?** В этом примере второй параметр **filter** функции **get_filter** стал ссылаться на лямбда-функцию. Эта функция возвращает **True** для четных значений и **False** – для нечетных. Соответственно, в цикле при переборе элементов списка, условие будет срабатывать только для четных значений и только они будут добавляться в список **res**. **Видите, как удобно и наглядно можно использовать лямбда-функции в программировании?** Если бы их не существовало, нам бы пришлось объявлять отдельную функцию:

```
def even(p):
    return p % 2 == 0
```

А, затем, передавать ссылку на нее:

Основы Python

```
r = get_filter(lst, even)
```

Согласитесь, это несколько более трудоемкий и менее удобный вариант? Мало того, если нам понадобится быстро изменить фильтр, то достаточно поправить реализацию анонимной функции, например, для выделения только положительных чисел:

```
r = get_filter(lst, lambda p: p > 0)
```

Быстро и просто! В этом одно из удобств этих анонимных функций – их можно сразу прописать в нужном месте программы.

Однако, у таких функций есть одно существенное ограничение – в них можно прописать только одну конструкцию языка **Python**, то есть, выполнить только одну какую-либо команду. Также нельзя объявлять анонимные функции в несколько строк:

```
lambda a:  
    print(a)
```

вызовет синтаксическую ошибку. Наконец, в лямбда-функциях нельзя использовать оператор присваивания:

```
lambda a: a = 1
```

Это также вызовет синтаксическую ошибку. Здесь мы можем лишь создавать новый объект на основе входных параметров (или глобальных, общих переменных программы):

```
s = lambda a: a + 1  
s(1)  
p = lambda: "hello python"  
p()
```

либо просто возвращать ссылки на уже существующие объекты:

```
lambda a: a
```

Основы Python

На этом мы завершим текущее занятие. Для закрепления материала пройдите практические задания и переходите к следующему уроку.

§43. Области видимости переменных. Ключевые слова `global` и `nonlocal`

На этом занятии мы поговорим о работе с **глобальными** и **локальными** данными, то есть, фактически, об области видимости переменных.

Когда мы начинаем писать какую-либо программу и сохраняем ее в файле, например, с названием:

`myprog.py`

то в **Python** автоматически создается пространство имен с названием `myprog` (слайд). Изначально оно пустое, но если там задать какие-либо переменные:

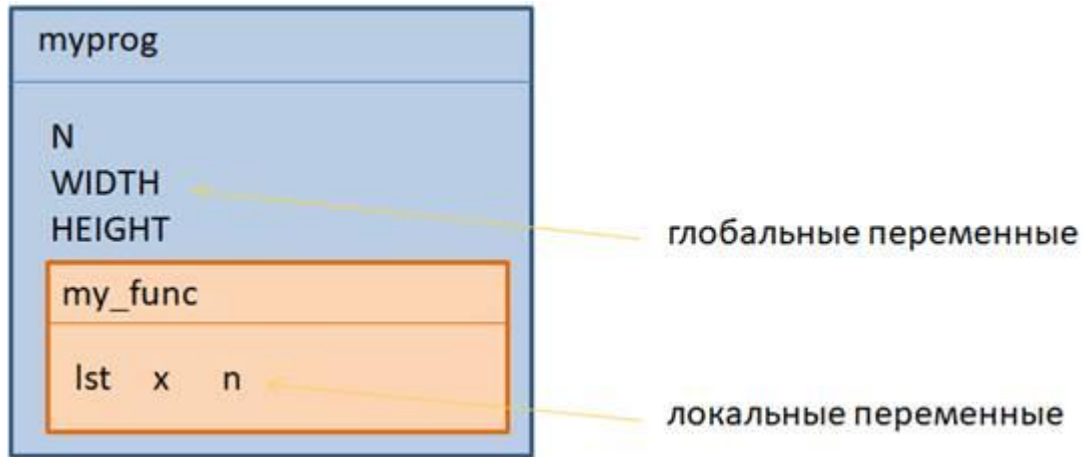
```
N = 100
WIDTH, HEIGHT = 1000, 500
```

то в этом пространстве появятся эти имена (**ссылки**). Так вот, все переменные, которые находятся непосредственно в этом пространстве имен, являются глобальными в пределах текущего программного модуля. То есть, они доступны в любом месте данной программы, после их объявления.

Добавим в эту программу еще и функцию, пусть она будет такой:

```
def my_func(lst):
    for x in lst:
        n = x + 1
        print(n)
```

Основы Python



В этом случае в пространстве имен появится переменная **my_func** – ссылка на объект-функцию, причем эта функция образует свое, локальное пространство имен с названием **my_func**. И внутри этого локального пространства автоматически появляются переменные **lst**, **x** и **n**. Обратите внимание, цикл **for** не образует своего локального пространства, только функция и все переменные внутри цикла находятся в области видимости функции.

Как вы уже догадались, переменные внутри локальной области видимости, называются локальными и доступны только в ее пределах. Например, вне функции мы не можем обращаться к локальным переменным **lst**, **x** и **n**.
Строчка кода:

```
print(x)
```

приведет к ошибке – **x** не определена. А вот внутри функции все три переменные существуют:

```
print(lst, n, x)
```

Если теперь вызвать функцию:

```
my_func([1, 2, 3])
```

то мы увидим значения этих локальных переменных. Также внутри функции доступны все переменные из внешней области видимости, в данном случае, глобальной. Если записать:

Основы Python

```
print(N, WIDTH)
```

то значения этих переменных будут отображены.

А что будет, если мы внутри функции определим локальную переменную **N**, то есть, с тем же именем, что и глобальная переменная? Давайте попробуем:

```
def my_func(lst):  
    N = 10  
    ...
```

В этом случае будет использоваться именно локальная переменная **N**, созданная в области видимости функции. То есть, алгоритм поиска переменных здесь такой. Сначала идет обращение к локальной области видимости и если переменная с нужным именем находится, то она и используется. Если же требуемая переменная не существует в локальной области, то поиск переходит на следующий уровень к внешней области и продолжается там. Поэтому, при отсутствии локальной переменной **N** внутри функции, берется глобальная переменная **N** из внешней области видимости. При этом, значение глобальной переменной никак не меняется, так как мы ее не используем:

```
my_func([1, 2, 3])  
print(N)
```

Хорошо, а как нам тогда изменять значения глобальных переменных внутри функции, если попытка присвоить ей какое-либо значение приводит к созданию аналогичной локальной переменной? Для этого в программе нужно явно указать, что мы хотим работать именно с глобальной переменной. Это делается с помощью ключевого слова **global**, за которым перечисляются имена глобальных переменных, например, так:

```
def my_func(lst):  
    global N  
    ...
```

Основы Python

Теперь имя **N** – это имя глобальной переменной и ее изменение внутри функции приведет к изменению и глобального значения.

Однако, здесь следует быть аккуратным, если в функции уже была создана локальная переменная **N**, то конструкция **global** приведет к ошибке:

```
N = 5
global N
```

Так можно делать только в отсутствии соответствующих локальных переменных.

Интересно, что если сейчас глобальную переменную **N** поставить в комментарий:

```
# N = 100
```

то есть, она не будет существовать изначально в программе, то после выполнения функции с конструкцией **global N**, эта глобальная переменная будет создана.

В **Python** имеется еще один интересный режим работы с локальными переменными с использованием ключевого слова **nonlocal**. Давайте предположим, что у нас имеется объявление одной функции внутри другой (так тоже можно делать):

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)
```

Основы Python

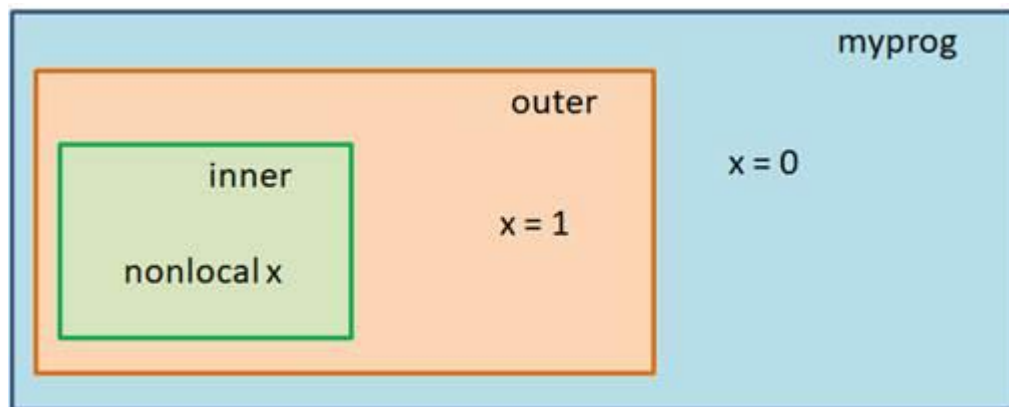
```
outer()  
print("global:", x)
```

При запуске программы мы увидим ожидаемые результаты, так как берутся локальные переменные внутри каждой функции:

```
inner: 2  
outer: 1  
global: 0
```

А теперь представим, что внутри функции **inner** мы хотели бы работать с переменной **x**, объявленной уровнем выше, то есть, в функции **outer**. Для этого в локальной области этой функции следует прописать ключевое слово **nonlocal** и указать переменную, которую следует взять из внешней области видимости:

```
nonlocal x
```



Теперь строка **x = 2** будет означать изменение переменной **x** в функции **outer** и при запуске программы получим результаты:

```
inner: 2  
outer: 2  
global: 0
```

Основы Python

Но так можно делать только с локальными переменными. С глобальной работать не будет. Если мы пропишем строчку

```
nonlocal x
```

в функции **outer**, то возникнет ошибка, т.к. уровнем выше находится уже глобальная область. Здесь, вместо **nonlocal** следует уже использовать **global**:

```
global x
```

а **nonlocal** в **inner** убрать, иначе опять же получится ссылка на глобальную переменную.

Теперь вы знаете, что такое глобальные и локальные переменные, как к ним обращаться и как использовать ключевые слова **global** и **nonlocal**. Для закрепления материала вас ждут практические задания, а я уйду в следующий урок.

§44. Замыкания в Python

На этом занятии затронем новую тему – **замыкания**. Это один из любимых вопросов на собеседовании в области программирования – рассказать, что такое замыкания. И сейчас вы узнаете подробный ответ на него.

Как мы с вами видели на предыдущем занятии, функции можно объявлять внутри других функций. Например, так:

```
def say_name(name):  
    def say_goodbye():  
        print("Don't say me goodbye, " + name + "!")  
  
    say_goodbye()
```

А, затем вызвать:

```
say_name("Sergey")
```


Основы Python

Мы здесь сначала вызываем внешнюю функцию, затем, в ней формируется вложенная функция `say_goodbye()` и вызывается. В результате, в консоли видим сообщение «Don't say me goodbye, Sergey!» - (“Не прощайся, Сергей!”).

Я, думаю, здесь вам все должно быть понятно. А теперь сделаем, следующее. Вместо вызова внутренней функции возвратим ссылку на нее с помощью оператора `return`:

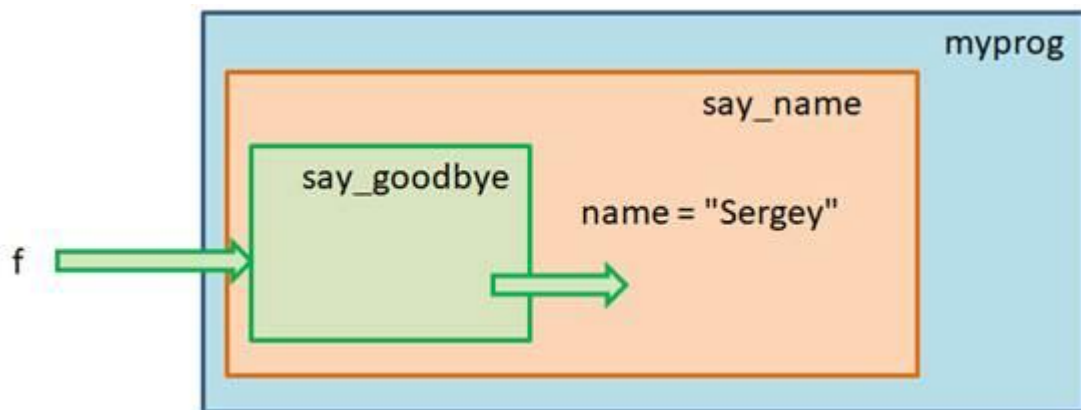
```
def say_name(name):  
    def say_goodbye():  
        print("Don't say me goodbye, " + name + "!")  
  
    return say_goodbye
```

Конечно, после запуска программы мы не увидим никакого сообщения, так как внутренняя функция нигде не вызывается. Исправим это. Сохраним ссылку на функцию `say_goodbye()` в переменной `f`:

```
f = say_name("Sergey")
```

А, затем, вызовем ее:

```
f()
```



Мы снова видим то же самое сообщение. **Вам не кажется здесь ничего странным?** Например, **откуда функция `say_goodbye()` берет значение**

Основы Python

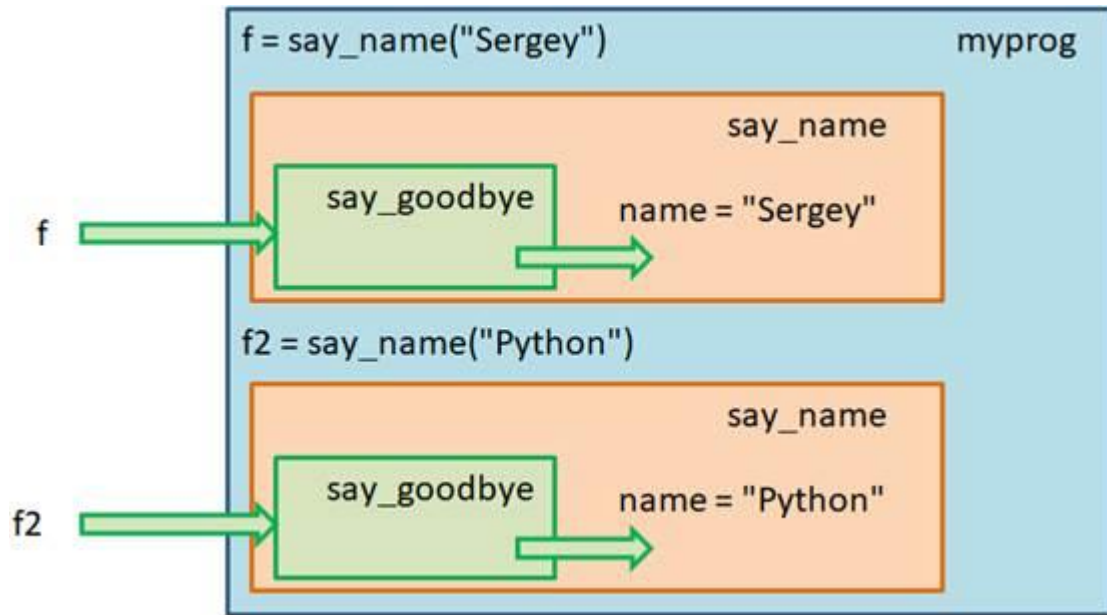
переменной name? Ведь внешняя функция `say_name()` выполнялась и завершилась, а значит, все ее локальные переменные вроде как тоже должны были бы исчезнуть? Но нет, мы обращаемся к переменной `name` и успешно получаем ее значение! **Почему?** Давайте разберемся.

Дело в том, что когда у нас имеется глобальная ссылка `f` на внутреннее, локальное окружение функции `say_goodbye()`, то это окружение продолжает существовать, оно не удаляется автоматически сборщиком мусора, именно из-за этой глобальной ссылки на него. А вместе с ним, продолжают существовать и все внешние локальные окружения, в данном случае – окружение функции `say_name()`, потому что также существует неявная, скрытая ссылка на него из внутреннего окружения. Такие ссылки формируются автоматически и позволяют, в частности, обращаться к переменным, объявленным в этих внешних окружениях. Именно поэтому функция `print()` в `say_goodbye()` имеет доступ к переменной `name` и эта переменная продолжает существовать, пока существует окружение `say_goodbye`, а значит и окружение `say_name`.

Вот такой эффект, когда мы «**держим**» внутреннее локальное окружение и имеем возможность продолжать использовать переменные из внешних окружений, в программировании называется замыканием. Замыкание в том смысле, что мы держим внутреннее окружение `say_goodbye` переменной `f` из глобального окружения. Получается цепочка ссылок, замыкающаяся на глобальном окружении. Мало того, при каждом новом вызове внешней функции, формируется свое новое, независимое локальное окружение, со своими локальными переменными и соответствующими значениями:

```
f = say_name("Sergey")
f2 = say_name("Python")
f()
f2()
```

Основы Python



Где может пригодиться такой функционал? Например, можно создать функцию-счетчик, которая бы увеличивала значение локальной переменной на единицу при каждом запуске:

```
def counter(start=0):  
    def step():  
        nonlocal start  
        start += 1  
        return start  
    return step
```

Обратите внимание, мы здесь используем ключевое слово **nonlocal**, чтобы переменная **start** изменялась во внешней локальной области, а не создавалась бы в текущей, локальной. Без этой строчки возникнет ошибка, из-за неопределенности: мы берем текущее значение **start** из вне, а потом создавали бы переменную с тем же именем внутри области **step**. Так делать нельзя. И строчка **nonlocal start** четко указывает брать переменную **start** из внешней локальной области, а не создавать в текущей.

Основы Python

Теперь можно сформировать несколько таких независимых счетчиков и выполнить их:

```
c1 = counter(10)
c2 = counter()
print(c1(), c2())
print(c1(), c2())
print(c1(), c2())
```

У нас, действительно, оба счетчика отработают независимо друг от друга.

И приведу еще один пример с замыканиями. Предположим, мы хотим сделать функцию, которая бы удаляла ненужные символы в начале и конце строки. Через замыкание это можно реализовать, следующим образом:

```
def strip_string(strip_chars=" "):
    def do_strip(string):
        return string.strip(strip_chars)

    return do_strip
```

Обратите внимание, вложенная функция тоже имеет параметр – строку, у которой будут удаляться ненужные символы. Далее, создадим два объекта с разным списком удаляемых символов:

```
strip1 = strip_string()
strip2 = strip_string(" !?,.;")
```

И вызовем вложенную функцию с одним аргументом:

```
print(strip1(" hello python!.. "))
print(strip2(" hello python!.. "))
```

Смотрите, первая функция **strip1** убрала только пробелы, а вторая еще и восклицательный знак с точками. Таким образом, мы можем многократно

Основы Python

использовать в программе функции `strip1` и `strip2`, передавая им разные строки.

Вот, что из себя представляют замыкания и вот так они работают. Для закрепления этого материала пройдите практические задания, а затем, на покорение нового материала – к новым урокам!

§45. Введение в декораторы функций

Собственно, последняя тема, которую я хочу затронуть – это **декораторы функций**. Чтобы понять их суть, вы должны хорошо знать про замыкания и вложенные функции.

Давайте, я сразу приведу пример, **из которого и сформулирую понятие декоратора?** Используя вложенную функцию `wrapper` и механизм замыканий, будем вызывать переданную функцию `func` внутри вложенной функции `wrapper`:

```
def func_decorator(func):  
    def wrapper():  
        print("----- что-то делаем перед вызовом функции -----")  
        func()  
        print("----- что-то делаем после вызова функции -----")  
  
    return wrapper
```

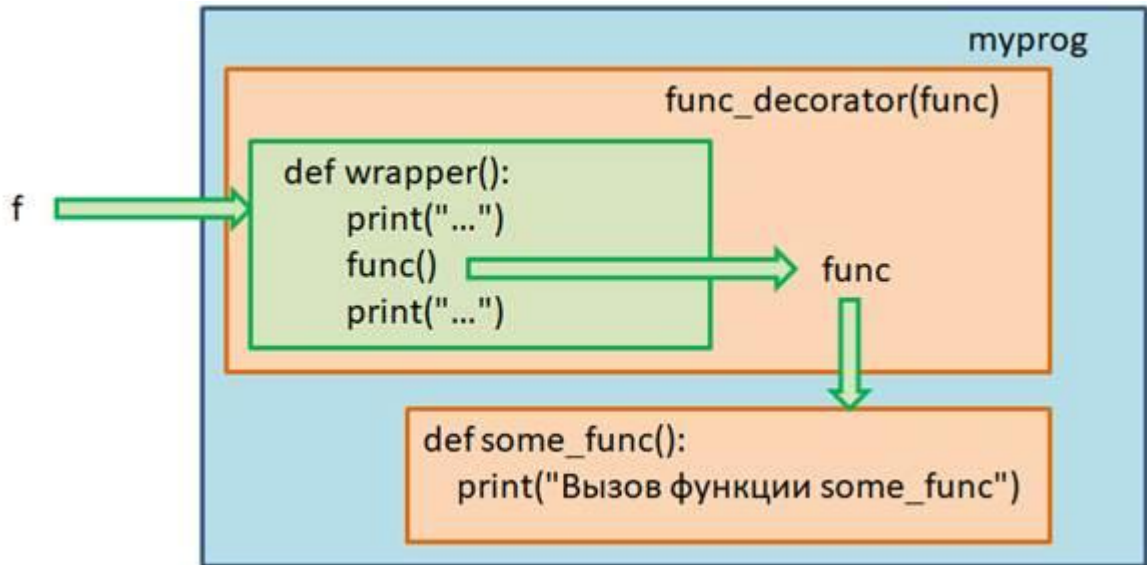
Далее, для примера, объявим некоторую функцию:

```
def some_func():  
    print("Вызов функции some_func")
```

И передадим ссылку на нее функции `func_decorator`:

```
f = func_decorator(some_func)
```

Основы Python



В результате, переменная **f** будет ссылаться на вложенную функцию **wrapper** и при ее вызове:

f()

соответственно запустится **wrapper()**, а она, в свою очередь, запустит переданную функцию **func()**, то есть, **some_func()**.

То есть, **декоратор** – это функция, которая принимает ссылку на другую функцию и расширяет ее функциональность за счет вложенной функции.

Часто, чтобы не создавать новые имена функций, вместо имени **f** используют то же самое имя функции:

```
some_func = func_decorator(some_func)
some_func()
```

И у нас получается, словно, мы модифицировали существующую функцию. В этом и заключается смысл декоратора – наполнить уже существующую функцию дополнительным функционалом.

Однако, если сейчас в нашу функцию **some_func()** добавить хотя бы один параметр:

Основы Python

```
def some_func(title):  
    print(f"title = {title}")
```

А, затем, вызвать ее с одним аргументом:

```
some_func = func_decorator(some_func)  
some_func("Python навсегда!")
```

то увидим ошибку, так как внутренняя функция **wrapper()** прописана без параметров. А именно на нее и ссылается сейчас **some_func**. В самом простом варианте, мы, конечно, может просто добавить этот параметр для **wrapper()**:

```
def func_decorator(func):  
    def wrapper(title):  
        print("----- что-то делаем перед вызовом функции -----")  
        func(title)  
        print("----- что-то делаем после вызова функции -----")  
  
    return wrapper
```

И теперь программа будет запускаться без ошибок. Но, очевидно, она перестанет работать, если число параметров функции **func()** изменится. **Как описать универсальную реализацию декоратора для функций с разным числом фактических и формальных параметров?** Конечно, для этого нужно определить вложенную функцию, принимающую произвольное число аргументов, следующим образом:

```
def func_decorator(func):  
    def wrapper(*args, **kwargs):  
        print("----- что-то делаем перед вызовом функции -----")  
        func(*args, **kwargs)  
        print("----- что-то делаем после вызова функции -----")  
  
    return wrapper
```

Основы Python

А, затем, переданные значения распаковываются и передаются функции `func()`. Получаем универсальную реализацию декоратора. И, действительно, если теперь добавить еще один параметр в функцию `some_func`:

```
def some_func(title, tag):  
    print(f"title = {title}, tag = {tag}")
```

То нам достаточно будет при вызове декоратора передать эти два аргумента:

```
some_func("Python навсегда!", 'h1')
```

Сам декоратор остается без изменений.

И, наконец, последний штрих. Сама по себе функция `some_func()` может возвращать какие-либо значения, например, помимо вывода сообщения в консоль, она еще будет возвращать сформированную строку:

```
def some_func(title, tag):  
    print(f"title = {title}, tag = {tag}")  
    return f"<{tag}>{title}</{tag}>"
```

Но, при попытке получить сейчас это значение:

```
some_func = func_decorator(some_func)  
res = some_func("Python навсегда!", 'h1')  
print(res)
```

Увидим **None**. И я, думаю, вы прекрасно понимаете **почему?** Да, это потому, что вложенная функция `wrapper()` ничего не возвращает. Исправим это и сделаем так, чтобы она возвращала значение функции `func()`:

```
def func_decorator(func):  
    def wrapper(*args, **kwargs):  
        print("----- что-то делаем перед вызовом функции -----")  
        res = func(*args, **kwargs)  
        print("----- что-то делаем после вызова функции -----")
```


Основы Python

```
return res
```

```
return wrapper
```

Вот теперь получили окончательную версию универсального декоратора функции и при выполнении программы видим не только его работу, но и возвращенное функцией значение.

Возможно, на протяжении всего этого урока вы задаете себе вопрос – **зачем это надо? Где применяется?** Приведу такой пример. Предположим, мы хотим протестировать различные функции на скорость их работы. Для этого возьмем функцию, которая реализует медленный **алгоритм Евклида** для поиска **НОД (наибольшего общего делителя)** двух натуральных чисел **a** и **b** (мы ее с вами делали на одном из прошлых занятий):

```
def get_nod(a, b):  
    while a != b:  
        if a > b:  
            a -= b  
        else:  
            b -= a  
    return a
```

И перед ней опишем декоратор-тестирующий:

```
def test_time(fn):  
    def wrapper(*args, **kwargs):  
        st = time.time()  
        res = fn(*args, **kwargs)  
        dt = time.time() - st  
        print(f"Время работы: {dt} сек")  
        return res  
    return wrapper
```

Основы Python

Мы здесь замеряем время работы функции и выводим его в консоль. Чтобы воспользоваться функцией `time`, импортируем этот модуль:

```
import time
```

и декорируем функцию `get_nod()`:

```
get_nod = test_time(get_nod)
res = get_nod(2, 1000000)
print(res)
```

После запуска программы видим время ее выполнения и возвращенный результат.

Мало того, если у нас будет еще какая-либо функция, например, тот же **алгоритм Евклида**, но с быстрой реализацией:

```
def get_fast_nod(a, b):
    if a < b:
        a, b = b, a
    while b:
        a, b = b, a % b

    return a
```

То мы легко можем применить тот же самый декоратор для тестирования скорости работы этой второй функции:

```
get_nod = test_time(get_nod)
get_fast_nod = test_time(get_fast_nod)
res = get_nod(2, 1000000)
res2 = get_fast_nod(2, 1000000)
print(res, res2)
```

Основы Python

Получается, что мы через декоратор `test_time` определили универсальный алгоритм для тестирования скорости работы любых функций. И это очень удобно!

И, наконец, последнее, о чем я хочу рассказать на этом занятии. Декораторы можно навешивать (применять) к функциям с помощью специального значка `@`. То есть, вместо строчек:

```
get_nod = test_time(get_nod)
get_fast_nod = test_time(get_fast_nod)
```

достаточно перед объявлением функций прописать `@test_time`. В результате, эти функции, как бы, «обертываются» нашим декоратором `test_time` и расширяются его функционалом. Если мы теперь запустим программу, то увидим тот же самый результат. А вот если у какой-либо функции уберем эту строчку, то будет обычный вызов без оценки скорости работы. Фактически, запись:

```
@test_time
def get_nod(a, b):
    ...
```

эквивалентна строчке:

```
get_nod = test_time(get_nod)
```

но со значком «собачка» программа выглядит, на мой взгляд, понятнее, да и записывать это декорирование проще. На практике, как правило, используют именно этот второй вариант.

На этом мы завершим наше первое знакомство с декораторами функций. Надеюсь, вы поняли, как они работают и для чего применяются? Отнеситесь серьезно к этому материалу, так как декораторы в **Python** используются довольно часто и важно правильно понимать принцип их работы.

Основы Python

§46. Декораторы с параметрами. Сохранение свойств декорируемых функций

На предыдущем занятии, мы познакомились с декораторами и рассмотрели некоторые примеры их работы. Здесь немного углубимся в эту тему и начнем с декораторов, которым можно дополнительно передавать аргументы.

Предположим, что мы создаем декоратор для вычисления производной функции. Используя уже имеющиеся знания, это можно сделать, следующим образом:

```
def func_decorator(func):  
    def wrapper(x, *args, **kwargs):  
        dx = 0.0001  
        res = (func(x + dx, *args, **kwargs) - func(x, *args, **kwargs)) / dx  
        return res  
    return wrapper
```

И к функции, например, `sin(x)` применить этот декоратор (не забываем `import math`):

```
@func_decorator  
def sin_df(x):  
    return math.sin(x)  
  
df = sin_df(math.pi/3)  
print(df)
```

После запуска программы видим искомый результат. Но, **что если, мы хотим управлять значением `dx`, передавая его как аргумент декоратору?**

Прописывать еще один параметр непосредственно у `func_decorator`, не лучший вариант:

```
def func_decorator(func, dx=0.0001):
```

Основы Python

В этом случае, у нас перестанет работать синтаксис с собачкой:

```
@func_decorator(dx=0.01)
def sin_df(x):
    return math.sin(x)
```

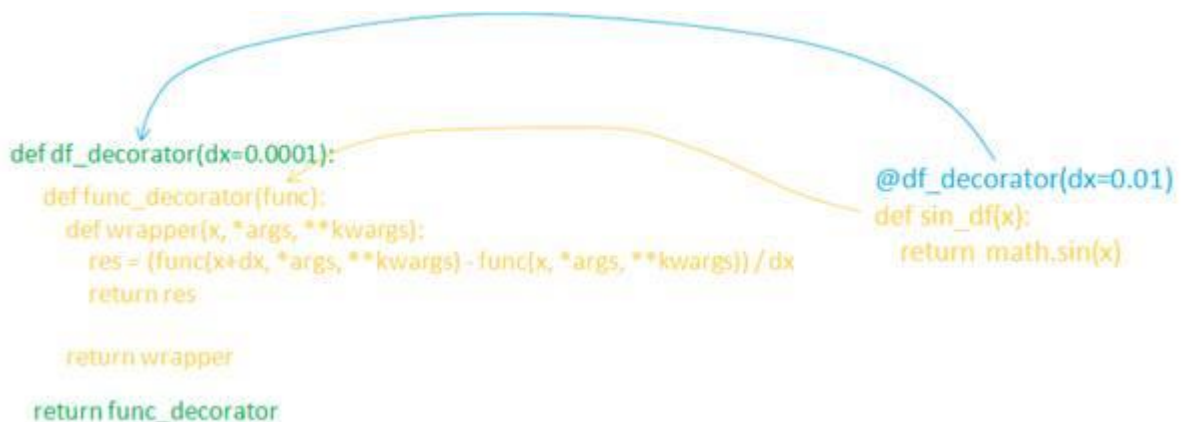
и попытка выполнить такую программу, приведет к ошибке. Для корректной реализации такой задачи следует обернуть имеющийся у нас декоратор в еще одну внешнюю функцию:

```
def df_decorator(dx=0.0001):
    def func_decorator(func):
        def wrapper(x, *args, **kwargs):
            res = (func(x+dx, *args, **kwargs) - func(x, *args, **kwargs)) / dx
            return res

        return wrapper
    return func_decorator
```

И применить уже ее к функции синуса:

```
@df_decorator(dx=0.01)
def sin_df(x):
    return math.sin(x)
```



Основы Python

В этом случае, аргумент будет передан в первую функцию, ссылка `sin_df` – во вторую вложенную функцию, а затем, через ссылку на третью функцию, мы можем вычислять значение производной:

```
df = sin_df(math.pi/3)
```

Все это эквивалентно следующим вызовам:

```
f = df_decorator(dx=0.01)
sin_df = f(sin_df)
```

Или, в более краткой форме:

```
sin_df = df_decorator(dx=0.01)(sin_df)
```

Но, запись через символ «собачки» куда нагляднее и проще, поэтому он, как правило, используется на практике.

Контроль имени и описания декорируемой функции

Во второй части занятия я хочу коснуться проблемы потери имени и описания декорируемой функции. **О чем здесь речь?** Вот смотрите, имя нашей декорированной функции можно узнать по специальному свойству `__name__`:

```
print(sin_df.__name__)
```

Сейчас оно принимает значение `wrapper`, что не удивительно, так как это и есть ссылка на эту функцию. Но изначально имя было другим (декоратор в комментариях и повторяем) – `sin_df`. Иногда важно сохранять исходные имена и при декорировании. **Как это сделать?** В самом простом варианте мы можем вручную внутри функции `func_decorator` изменить имя, оставив исходное:

```
def df_decorator(dx=0.0001):
    def func_decorator(func):
        def wrapper(x, *args, **kwargs):
            res = (func(x+dx, *args, **kwargs) - func(x, *args, **kwargs)) / dx
```

Основы Python

```
    return res

    wrapper.__name__ = func.__name__
    return wrapper

return func_decorator
```

И, как видим, это работает. И тот же самый фокус можем проделать с описанием функции:

```
@df_decorator(dx=0.01)
def sin_df(x):
    """Функция для вычисления производной синуса"""
    return math.sin(x)
```

Это описание доступно с помощью специального свойства `__doc__`:

```
print(sin_df.__doc__)
```

Для декорированной функции оно равно **None**, а для исходной – заданная строка. Также сохраняем описание, прописав в функции `func_decorator` строчку:

```
wrapper.__doc__ = func.__doc__
```

Все, теперь у нас сохраняются и имена и описания декорируемых функций.

Но есть более простой и продвинутый способ сохранять свойства `__name__` и `__doc__` - с помощью специального декоратора:

```
from functools import wraps
```

Если мы декорируем вложенную функцию `wrapper`:

```
@wraps(func)
```

то строчки можно убрать (я поставлю в комментарии):

Основы Python

```
# wrapper.__name__ = func.__name__  
# wrapper.__doc__ = func.__doc__
```

И, выполняя программу, видим, что имя функции и ее описание остаются прежними. То есть, декоратор **wraps** автоматически сохраняет эти свойства.

На этом мы завершим тему декораторов и функций языка **Python**. Этого материала, вам вполне хватит для программирования большинства задач и, кроме того, позволит понимать уже существующие программы. До встречи на следующем уроке!

§47. Импорт стандартных модулей. Команды `import` и `from`

Этим занятием мы открываем новую тему, связанную с модулями и пакетами. В некоторых программах этого курса я уже делал импорт стандартных модулей, например:

```
import math  
import time
```

чтобы использовать их функции. Но **что они из себя представляют?** В действительности, **модуль** – это обычный текстовый файл программы на языке **Python**. В этом легко убедиться, если в **PyCharm** нажать и удерживать клавишу **Ctrl** и щелкнуть по имени модуля, например, **math**. Откроется файл и мы можем ознакомиться с его содержимым.

Второй вопрос, а **что собственно делает конструкция `import`? Как он импортирует этот текст программы в нашу программу?** Давайте я с помощью функции **locals()** выведу все локальные переменные нашей программы:

```
print(locals())
```

В консоли у нас появился словарь и последними записями в нем идут имена **math** и **time**, которые ссылаются на соответствующие импортированные модули. Для более читаемого вида этой информации я импортирую еще один модуль:

Основы Python

```
import pprint
```

и из него вызову функцию с тем же именем:

```
pprint.pprint(locals())
```

Теперь мы видим все гораздо нагляднее. Кстати, дополнительно появилась еще одна строчка с именем **pprint** после его импортирования. Так вот, все эти имена: **math**, **pprint**, **time** – это переменные в нашей программе и каждая ссылается на свое пространство имен, со своими глобальными переменными. Например, чтобы посмотреть, что содержит модуль **math**, можно получить все ее переменные с помощью функции **dir**:

```
pprint.pprint(dir(math))
```

Мы видим полный список глобальных атрибутов этого модуля, включая имена функций и переменных. То есть, чтобы обратиться к определенной функции в своей программе, нам нужно написать имя импортируемого модуля и через точку указать имя функции или переменной, например:

```
a = math.ceil(1.8)
print(a)
print(math.pi)
```

То есть, в строчке **import math** автоматически создается пространство имен **math** в нашей программе, через которое мы можем обращаться к любому глобальному атрибуту этого модуля. И так с каждым импортируемым модулем.

Все эти модули, что я использовал, поставляются вместе с языком **Python** и составляют его стандартную библиотеку. **Что еще входит в нее?** Полный список модулей и подробное их описание приведено на странице официальной документации:

<https://docs.python.org/3/library/>

Основы Python

Я рекомендую вам в целом с ней ознакомиться, чтобы представлять ее возможности и использовать в своей программе готовый функционал, а не изобретать собственные велосипеды.

Далее, предположим, что мы в своей программе используем переменную с именем:

```
math = 'математика'
```

В итоге, в нашей программе, сначала при импорте **math** будет создана глобальная переменная **math** на импортируемый модуль, а затем, эта переменная станет ссылаться на строку. В результате мы уже не можем воспользоваться модулем **math** в своей программе и попытки обратиться к функции или переменной приведут к ошибкам.

Как разрешить эту ситуацию? Язык **Python** позволяет нам в момент импорта указывать псевдонимы импортируемых модулей, используя ключевое слово **as**, например:

```
import math as mt
```

И, далее, по программе использовать переменную **mt** в качестве имени модуля. На практике такая подстановка выполняется в двух целях. **Во-первых**, чтобы избежать возможного конфликта имен в программе и, **во-вторых**, для удобства, если имя модуля слишком длинное, то имеет смысл записать его более короткий псевдоним.

Когда импортируется модуль через **import**, то автоматически подключается все его пространство имен к нашей программе. Однако, часто из всего этого многообразия используется всего несколько функций или переменных. И, если мы не хотим подключать модуль целиком, а сделать выборочный импорт некоторых его атрибутов, то для этого следует использовать другую конструкцию:

```
from math import ceil, pi
```

Основы Python

Смотрите, теперь, при выводе:

```
pprint.pprint(locals())
```

мы не видим имя модуля **math**, но в пространстве имен нашей программы появляются имена **ceil** и **pi**. То есть, для их использования мы теперь должны обращаться к ним напрямую:

```
print(ceil(1.8))  
print(pi)
```

Но при таком импорте мы можем столкнуться с ситуацией, когда импортируемая функция (или переменная) переопределяется в нашей программе. Например, если прописать свою собственную функцию **ceil**:

```
def ceil(x):  
    print("своя функция ceil")  
    return x
```

то именно она и будет вызвана, потому что сначала происходит импорт, а затем, объявление функции. Если объявление функции сделать до импорта, то, наоборот, будет использована библиотечная функция, а наша затрется. То есть, берется то, что определяется последним.

Для разрешения такой ситуации (конфликта имен) можно у импортируемых элементов указывать псевдонимы через ключевое слово **as**:

```
from math import ceil as m_ceil, pi
```

И, далее, уже обращаться к функции **ceil** через имя **m_ceil**:

```
print(m_ceil(1.8))
```

Конструкция **from** позволяет выполнять импорт сразу всего пространства имен, если после **import** поставить *****:

```
from math import *
```

Основы Python

Тогда все функции, переменные и другие объекты будут напрямую импортированы в нашу программу, что также может вызвать конфликт имен, в частности, сейчас это происходит с функцией `ceil`. Поэтому делать такой импорт крайне не рекомендуется. Мы даже наперед точно не сможем сказать, какие имена будут добавлены в этом случае, особенно, если библиотеки меняются от версии к версии. Так что лучше такой импорт вообще не делать в своих программах.

И еще один момент. Формально в импорте мы можем через запятую перечислять имена импортируемых модулей, например:

```
import pprint, time, random
```

но стандарт **PEP8** так делать не рекомендует. Следует вместо одного такого импорта записывать три, каждый с новой строки:

```
import pprint
import time
import random
```

Это правило не касается конструкции **from ... import ...**. Здесь, конечно же, мы указываем конкретные имена через запятую.

Наконец, **последняя рекомендация** – располагать импорты в самом начале программы. Редко от этого правила отступают, но в большинстве случаев импортирование следует делать в первых строчках программы.

На этом мы с вами завершим первое знакомство с модулями языка **Python**. Надеюсь, теперь вы знаете, как выполнять импорт стандартных библиотек. На следующем занятии мы продолжим эту тему и поговорим о создании и импорте собственных модулей.

Основы Python

§48. Импорт собственных модулей

На этом занятии вы узнаете как создавать и импортировать свои собственные модули. Как мы говорили на предыдущем уроке, модули представляют собой обычный текстовый файл программы на **Python**. Поэтому, в текущем рабочем каталоге я создам еще один файл и назову его **mymodule.py**. (Щелкаем правой кнопкой по вкладке **mymodule** и выбираем пункт «Split and Move Right»). Запишем в этом модуле следующие строчки:

```
NAME = 'mymodule'
def floor(x):
    print("функция из mymodule")
    return int(x) if x >= 0 else int(x) - 1
```

В результате, в глобальном пространстве имен имеем две переменные: **NAME** и **floor**. Все локальные переменные, например, **x** сюда уже не попадают – только глобальные определения.

Импортируем этот модуль в нашу программу **ex.py**:

```
import mymodule
```

и если теперь ниже записать:

```
mymodule.
```

то, как раз, увидим наши определения – переменную **NAME** и имя функции **floor**. То есть, они были импортированы в нашу программу и доступны через пространство имен **mymodule**:

```
print(mymodule.floor(-5.4))
```

Или же, можем воспользоваться другой конструкцией:

```
from mymodule import floor
```

и напрямую вызвать эту функцию:

Основы Python

```
print(floor(-5.4))
```

Но что будет, если в файле **mymodule** тоже прописать импорт, скажем, библиотеки **math**, следующим образом:

```
import math
```

В этом случае, в модуле **mymodule** появляется глобальная переменная с именем **math**. Значит, при импорте уже модуля **mymodule** (в `ex.py`):

```
import mymodule  
import pprint
```

мы должны увидеть это имя:

```
pprint.pprint(dir(mymodule))
```

Действительно, в конце списка оно присутствует, а это значит, к модулю **math** можно обратиться через модуль **mymodule**:

```
a = mymodule.math.floor(-5.6)  
print(a)
```

То есть, мы здесь уже имеем иерархию модулей, сначала обращаемся к **mymodule**, затем к **math**, а потом к одной из его функций.

Конечно, чтобы не создавать таких цепочек, можно было бы импортировать библиотеку **math** через конструкцию:

```
from math import *
```

(Еще раз отмечу, что так делать не рекомендуется. Здесь, я лишь в учебных целях показываю, что произойдет). В итоге получаем неявное расширение модуля **mymodule** за счет модуля **math**. И все функции из **math** мы теперь напрямую можем вызывать из **mymodule**:

```
a = mymodule.floor(-5.6)
```

Основы Python

```
b = mymodule.ceil(-5.6)
print(a, b)
```

Причем, вот эта функция **floor** была переопределена, поэтому вызывается не библиотечный вариант, а наш. Так вот, такой импорт ни в коем случае делать не стоит, чтобы не смешивать свои и стандартные функции в одном месте. Лучше или перечислять, те элементы, что мы собираемся использовать, например:

```
from math import pi, ceil
```

или же воспользоваться простым импортом:

```
import math
```

Давайте теперь детальнее разберемся, как работает импорт модулей. **Первый вопрос, откуда Python «знает», где искать импортируемые модули?** Порядок поиска прописан в специальной коллекции **path** модуля **sys**:

```
import sys
pprint.pprint(sys.path)
```

Здесь первые строчки – это каталог с исполняемым модулем **ex.py** и рабочий каталог, а далее, идут дополнительные пути поиска, например, для файлов стандартной библиотеки. Поэтому, если мы переместим файл модуля **mymodule** из рабочего каталога, в какую-нибудь вложенную папку, например, с именем **folder** (создаем и перемещаем), то при импорте получим ошибку **ModuleNotFoundError**, так как к этой папке не прописан путь в коллекции **path**. Поэтому, нам дополнительно нужно указать каталог **folder**, в котором находится модуль **mymodule**, следующим образом:

```
import folder.mymodule
```

Конечно, в список **path** мы можем добавить путь для поиска модулей:

```
sys.path.append(r"C:\PythonProjects\example\folder")
```

Основы Python

и тогда при импорте по-прежнему достаточно будет прописывать только **mymodule**. Но это делается крайне редко, обычно добавляют имена подкаталогов через точку.

Давайте теперь поменяем местами эти два файла (**ex.py** и **mymodule.py**). В этом случае для импорта достаточно будет также прописать:

```
import mymodule
```

так как пути поиска теперь будут включать и рабочий каталог, где находится **mymodule** и каталог с исполняемым файлом.

Вернем файл **ex.py** в рабочий каталог. И заметим, что в момент выполнения импорта модуль преобразуется интерпретатором языка **Python** сначала в байт-код, а затем, запускается на исполнение. Это важный момент. Смотрите, если в **mymodule** прописать строчку:

```
print(NAME)
```

и запустить файл **ex.py** на исполнение, то в консоли увидим результат срабатывания функции **print()**. Причем она сработала в момент импорта модуля до вызова функции **pprint()**. Это означает, что если в **mymodule** будет записана программа:

```
for i in range(5):  
    print(NAME)
```

то она будет выполнена. Так что с этим следует быть аккуратным.

Значит, получается, что подключаемые модули должны исключительно содержать определения переменных, функций и других объектов языка Python, но не их вызовы? **Не обязательно!** В каждом модуле есть специальная переменная **__name__**, которая принимает имя модуля, если он запускается при импорте:

```
print(__name__)
```


Основы Python

и значение `"__main__"`, если он запускается как самостоятельная программа (показываем). И это свойство часто используют для контроля исполнения модуля. Если в нем нужно прописать программу, выполняемую только при непосредственном запуске модуля, то ее следует поместить в следующее условие:

```
if __name__ == "__main__":  
    print("самостоятельный запуск")  
else:  
    print("запуски при импорте")
```

Такую проверку часто можно увидеть в питоновских программах и вы теперь знаете, для чего она нужна.

Давайте посмотрим, что произойдет, если в модульном файле сделать импорт проектного:

```
import ex
```

Чтобы результат был виден, в проектный файл добавим:

```
print("ex.py")
```

а в модульный:

```
print("mymodule")
```

После запуска видим, что сначала был выполнен проектный файл, затем, модульный, а потом, снова проектный. Но **почему это выполнение не пошло дальше по цепочке?** Дело в том, что модуль импортируется только один раз. Например, если прописать два импорта подряд:

```
import mymodule  
import mymodule
```

то увидим только одно сообщение. Это, как раз, и доказывает, что модуль был выполнен только один раз. Если же нам нужно сделать повторный импорт в

Основы Python

одной и той же программе, то для этого следует использовать специальную функцию:

```
importlib.reload(mymodule)
```

а перед этим импортируем специальный модуль

```
import importlib
```

который доступен, начиная с версии **Python 3.4**. В качестве аргумента этой функции передаем ранее импортированный модуль и он будет обновлен без перезапуска, поэтому второй раз функция **print()** в нем выполняться не будет.

Надеюсь, из этого занятия вам стало понятно, как создавать свои собственные модули, как их импортировать и как все это работает в деталях. Жду всех вас на следующем занятии!

§49. Установка сторонних модулей (pip install). Пакетная установка

На этом занятии мы познакомимся со способами установки сторонних (**внешних**) модулей для текущего интерпретатора **Питона**.

Я, думаю, вы прекрасно понимаете, что существующих стандартных модулей часто недостаточно для создания различных приложений. Например, если мы решаем какую-либо инженерную задачу, то часто применяются пакеты:

- **NumPy** – работа с многомерными массивами;
- **Matplotlib** – отображение графиков.

Если мы собираемся работать с графикой, то, обычно, используют модуль:

- **Pygame** – реализация простой 2D-графики.

Если разрабатываем бэкэнд сайта, то нам потребуется фреймворк, например:

- **Flask** – простой фреймворк (часто для информационных сайтов);
- **Django** – продвинутый фреймворк для самых сложных сайтов.

Основы Python

И так далее. Сторонние модули разрабатываются и обновляются постоянно. Для языка **Python** за долгие годы были разработаны сотни тысяч пакетов для самых разных нужд. В этом также одно из преимуществ данного языка – его удобно применять для решения самых разных задач.

Чтобы увидеть, какие пакеты уже установлены для текущего интерпретатора языка **Python**, нужно открыть командное окно или вкладку «**Terminal**» в **PyCharm**, набрать команду:

```
pip list
```

и отобразится список с номерами версий. Здесь слово **pip** означает обращение к специальному модулю, отвечающему за установку сторонних пакетов в интерпретатор. В самом простом варианте, чтобы установить новый модуль, нам нужно знать его название и, затем, выполнить команду:

```
pip install <название пакета>
```

Но **как нам узнать, какие пакеты существуют, как они называются и как их применять в программе?** Для этого существует специальный сайт (**репозиторий**) со списком сторонних пакетов и их кратким описанием:

<https://pypi.org/>

Конечно, отображать названия обычным списком просто нереально. Здесь, как в поисковой системе, нужно набрать примерное название и выполнить поиск. Например, найдем пакет **flask**:

В результате, получим множество пакетов, где это имя фигурирует. В самом начале, как правило, появляются наиболее релевантные результаты. И, действительно, здесь первой строчкой, как раз и стоит пакет **Flask** – микрофреймворк для разработки простых сайтов. Щелкнем по нему и увидим краткое описание, ссылку на подробную документацию и команду для установки этого пакета:

```
pip install Flask
```

Основы Python

Если прописать ее в таком виде, то будет найдена последняя версия и установлена на компьютер. При необходимости, можно явно указать версию для установки. Для этого после имени пакета ставится два равно и пишется номер версии:

```
pip install Flask==1.1.2
```

Другой способ установки пакетов – через **PyCharm**. Для этого нужно выбрать **Settings->Project Interpreter**. Увидим окно с текущим интерпретатором языка **Python** и набором внешних установленных пакетов, а также их версий. Чтобы добавить новый пакет, выбираем «+» и в открывшемся окне через поиск находим нужный пакет и нажимаем кнопку «**Install Package**».

Обратите внимание, я постоянно говорю об установке пакетов для текущего интерпретатора. Дело в том, что установка модулей осуществляется с привязкой к выбранной версии языка **Python**. У вас на компьютере может быть установлено несколько версий (**как у меня**) и у каждой версии свой набор установленных внешних модулей. Это сделано специально, так как некоторые пакеты привязаны к определенным версиям языка **Python**. Поэтому установка происходит для текущего интерпретатора.

Также вас может интересовать, а где располагаются эти установленные модули? Узнать это очень просто. Наведем курсор мыши на название любого пакета и увидим путь его расположения. Этот путь также присутствует в специальной коллекции:

```
sys.path
```

о которой мы с вами говорили на прошлом уроке. Поэтому, все установленные внешние пакеты могут быть импортированы в программу без дополнительных указаний каталогов и подкаталогов.

Наконец, последнее, о чем я хочу рассказать вам на этом занятии – это пакетная установка внешних модулей. Делается это с помощью команды:

```
pip install -r <текстовый файл>
```

Основы Python

А в текстовом файле нужно прописать имена устанавливаемых модулей, обычно, с указанием требуемых версий, но можно и без них (тогда будет установлена последняя версия пакета).

Этот файл можно сформировать вручную, но часто такая пакетная установка используется для переноса проекта с одного компьютера на другой. Поэтому, в файле нужно прописать все установленные внешние пакеты. Чтобы не делать этого вручную, можно выполнить команду:

```
pip freeze > requirements.txt
```

и в рабочем каталоге проекта появится текстовый файл **requirements.txt** с набором всех установленных модулей для текущего интерпретатора. Далее, мы можем указать этот файл в команде:

```
pip install -r requirements.txt
```

и эти пакеты будут установлены при их отсутствии.

Вот, в целом, порядок установки сторонних пакетов для выбранного интерпретатора языка **Python**. Для закрепления материала попробуйте выполнить установку какого-либо модуля и переходите к следующему уроку.

§50. Пакеты (package) в Python. Вложенные пакеты

На этом занятии речь пойдет о пакетах, узнаем, как их создавать и импортировать. Ранее мы с вами уже говорили о модулях – отдельных файлах с текстами программ, которые можно импортировать в другие программы. А пакет (**package**) – это специальным образом организованный подкаталог с набором модулей, как правило, решающих сходные задачи.

Давайте в качестве примера создадим пакет из модулей по обучающим курсам:

- HTML
- Java

Основы Python

- PHP
- Python

Это можно сделать, следующим образом. В **PyCharm** во вкладке «**Project**» щелкнуть правой кнопкой мыши и из выпадающего меню выбрать **New -> Python Package**. Здесь нам нужно придумать название пакета, пусть оно будет:

courses

Нажимаем **Enter** и пакет в рабочем каталоге создан. Посмотрим, что он из себя представляет. Переходим в рабочий каталог и видим в нем подкаталог с указанным именем **courses**. Внутри этого каталога находится только один пустой файл **__init__.py**. Чуть позже мы узнаем для чего он нужен.

Итак, пакет в **Python** – это обычный каталог, в котором обязательно должен располагаться специальный файл **__init__.py**.

Но пока наш пакет пустой, в нем нет ни одного модуля. Добавим их, то есть, добавим обычные **python**-файлы в этот каталог. Пусть они будут такими:

html.py:

```
def get_html():  
    print("курс по HTML")
```

java.py

```
def get_java():  
    print("Курс по Java")
```

php.py

```
def get_php():  
    pass
```

Основы Python

```
def get_mysql():  
    pass
```

python.py

```
def  
get_python():  
    print("курс по Python")
```

И, кроме того, в файл `__init__.py` добавим строчку:

```
NAME = "package courses"
```

И обратите внимание. Для корректной обработки модулей в пакете, все файлы следует создавать с кодировкой **UTF-8**.

Все, мы сформировали пакет и теперь можем его импортировать в нашу программу. Для этого записывается ключевое слово **import** и указывается имя пакета (**название подкаталога**):

```
import courses
```

Давайте посмотрим, что в итоге было импортировано:

```
print( dir(courses) )
```

Мы видим имя нашей переменной **NAME** и вспомогательные переменные, которые были созданы автоматически средой **Python**:

```
['NAME', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',  
'__name__', '__package__', '__path__', '__spec__']
```

Выведем значение переменной **NAME**:

```
print( courses.NAME )
```

Основы Python

И в консоли отображается строчка, прописанная в файле `__init__.py`. **О чем это говорит?** О том, что при импорте пакета файл `__init__.py` был выполнен. В действительности – это инициализатор пакета. В нем мы прописываем то, что следует импортировать при импорте пакета. Например, в нашем каталоге `courses` присутствуют четыре наших файла, но ни один из них не был использован в момент импорта. Это произошло, как раз, по той причине, что в файле `__init__.py` мы их никак не обрабатывали.

Чтобы получить доступ к функциям из модулей внутри пакета, их, в свою очередь, нужно импортировать в инициализаторе `__init__.py`, например, так:

```
import courses.python
```

Я, думаю, вы понимаете, **почему вначале указан подкаталог `courses`?** Так как модули находятся по нестандартному пути, то этот путь следует явно прописать. В данном случае, достаточно указать имя подкаталога, а затем, через точку имя файла.

Теперь, после запуска программы, мы видим, что в пространстве имен пакета `courses` появилось имя `python`. Поэтому, мы можем обратиться к этому модулю и вызвать его функцию:

```
courses.python.get_python()
```

Но, обычно, в инициализаторе пакетов импорт выполняют с помощью конструкции:

```
from courses.python import get_python
```

чтобы, затем, не указывать имя модуля в основной программе:

```
courses.get_python()
```

Но, все же, у такого способа импортирования модулей в инициализаторе пакета есть один существенный недостаток – мы, фактически, указываем полный путь к модулю (это называется **абсолютным импортом**). Представьте,

Основы Python

например, что в будущем, по каким-либо причинам, придется поменять название пакета. Тогда и все абсолютные импорты также придется переписывать. Поэтому здесь лучше воспользоваться **относительным** способом импортирования. Для этого, вместо имени основного подкаталога **courses**, ставится точка:

```
from .python import get_python
```

Эта точка означает «**использовать текущий каталог**». Так гораздо практичнее и мы теперь совершенно не привязаны к названию пакета.

Или, можно записать так:

```
from . import html, java, php, python
```

тогда импортируем все модули в текущем пакете. Но, мы вернем прежний импорт и запишем его со звездочкой для получения всех данных из файла:

```
from .python import *
```

Ранее, я вам говорил, что в программах так лучше не делать. Однако, внутри пакетов делают – это исключение из общего правила. **Почему такой импорт целесообразен?** Дело в том, что модули внутри пакетов постоянно совершенствуются. Появляются новые **функции, переменные, классы**. И, чтобы каждый раз не менять их список в импорте, прописывают звездочку, так удобнее. При этом конфликта имен можно избежать, контролируя импортируемые данные. Для этого внутри модуля прописывается специальная переменная **__all__** со списком импортируемых данных, если используется оператор *****. Например, если в модуле **php.py**, прописать:

```
__all__ = ['get_php', 'get_mysql']
```

то в пакет будут импортированы обе функции и мы их можем вызвать:

```
courses.get_php()  
courses.get_mysql()
```

Основы Python

ошибок никаких не будет. Но, если оставить только какую-нибудь одну:

```
__all__ = ['get_php']
```

то прежняя программа выполнится с ошибками, так как вторая функция не была импортирована.

Эту же коллекцию `__all__` можно записывать и в инициализаторе, например, так:

```
from .python import *
from .php import *

NAME = "package courses"

__all__ = ['python', 'php']
```

Тогда модули `python` и `php` импортируются, а переменная `NAME` нет.

И последнее, о чем я, думаю, следует сказать – это возможность создавать вложенные пакеты. Делается это очевидным образом, создаем еще один вложенный каталог, например, с именем:

`doc`

через **PyCharm** (также). И, далее, разместим в этом вложенном пакете два файла:

`java_doc.py`:

```
doc = """Документация по языку Java"""
```

`python_doc.py`:

```
doc = """Документация по языку Python"""
```

Затем, в файле инициализатора этого пакета запишем импорт этих модулей:

Основы Python

```
from . import python_doc, java_doc
```

А в инициализаторе основного пакета – импорт вложенного пакета:

```
from .doc import *
```

и скорректируем коллекцию:

```
__all__ = ['python', 'php', "doc"]
```

Все, теперь в основной программе можно обращаться к переменным вложенного пакета, следующим образом:

```
print(courses.python_doc.doc)
```

Мало того, в самих модулях вложенного пакета можно обращаться к модулям внешнего пакета. Для этого следует при импорте указать две точки (переход во внешний каталог) и далее имя модуля, например (в файле `python_doc.py`):

```
from ..python import get_python
```

Затем, можно вызвать эту функцию:

```
doc = """Документация по языку Python: """ + get_python()
```

соответственно, ее переписав (в файле `python.py`):

```
def get_python():  
    return "курс по Python"
```

Вот так относительно легко и просто можно создавать свои собственные пакеты, а также вкладывать один в другой. Принцип описания фрагментов программ на уровне пакета – довольно распространенная практика и вы теперь знаете, как все это работает.

Основы Python

§51. Функция `open`. Чтение данных из файла

На этом занятии мы с вами научимся читать данные из файлов. Думаю, вы все прекрасно понимаете, что такое файлы и что они хранятся, как правило, на внешних носителях. (Часто – это жесткий диск устройства или флеш-память или SSD-диск. Бывают и другие носители). Главная особенность файлов – сохранение информации после отключения устройства от питания.

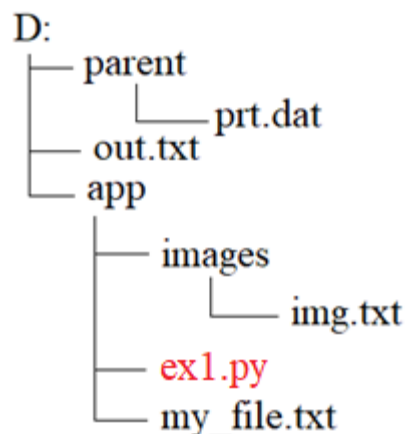
Начнем со знакомства с функцией:

```
open(file [, mode='r', encoding=None, ...])
```

которая открывает указанный файл на чтение или запись данных (по умолчанию – на чтение). Основные ее параметры, следующие:

- **file** – путь к файлу (вместе с его именем);
- **mode** – режим доступа к файлу (**чтение/запись**);
- **encoding** – кодировка файла.

Чтобы воспользоваться этой функцией, нужно правильно записать первый аргумент – **путь к файлу**. Давайте, посмотрим различные варианты его определения. Представим, что наш файл **ex1.py** находится в рабочем каталоге **app**:



Тогда, чтобы обратиться к файлу **my_file.txt** путь можно записать просто, как:

Основы Python

```
"my_file.txt"
```

или

```
"d:\\app\\my_file.txt"
```

или так:

```
"d:/app/my_file.txt"
```

Последние два варианта представляют собой абсолютный путь к файлу, то есть, полный путь, начиная с указания диска. Причем, обычно используют обратный слеш в качестве разделителя: так короче писать и такой путь будет корректно восприниматься как под ОС **Windows**, так и **Linux**. Первый же вариант – это относительный путь (относительно каталога с исполняемым файлом).

Теперь, предположим, что мы хотим обратиться к файлу **img.txt**. Это можно сделать так:

```
"images/img.txt"
```

или так:

```
"d:/app/images/img.txt"
```

Для доступа к **out.txt** пути будут записаны так:

```
"../out.txt"
```

```
"d:/out.txt"
```

Обратите внимание, здесь две точки означают переход к родительскому каталогу, то есть, выход из каталога **app** на один уровень вверх.

И, **наконец**, для доступа к файлу **prt.dat** пути запишутся так:

Основы Python

```
"../parent/prt.dat"  
"d:/ parent/prt.dat"
```

Вот принцип, по которому прописываются пути к файлам. В нашем случае мы имеем текстовый файл «**my_file.txt**», который находится в том же каталоге, что и программа **ex1.py**, поэтому путь можно записать, просто указав имя файла:

```
file = open("my_file.txt")
```

В результате переменная **file** будет ссылаться на файловый объект, через который и происходит работа с файлами. Если указать неверный путь, например, так:

```
file = open("my_file2.txt")
```

то возникнет ошибка **FileNotFoundError**. Поэтому пути нужно прописывать аккуратно. Позже мы увидим, как можно избежать такой ошибки, чтобы программа не прерывалась при неверно указанном файле.

Итак, функция **open()** по умолчанию открывает файл на чтение. Это значит, что в этом режиме мы можем только считывать информацию из файла, но не записывать. **Как прочитать данные из файла?** Для этого существует несколько методов у файлового объекта. Например, если выполнить метод:

```
print( file.read() )
```

то будет прочитан весь файл целиком и результат представлен в виде строки. Но сейчас мы видим с вами в консоли отображаются какие-то кракозябры. **Почему?** Все дело в несовпадении кодировок. В моем случае метод **read()** читает данные в кодировке **windows-1251**, а кодировка файла с текстом – **UTF-8**. Чтобы это поправить, в функции **open()** нужно явно указать кодировку файла через именованный аргумент **encoding**:

```
file = open("myfile.txt", encoding="utf-8" )
```

Основы Python

Теперь все работает корректно. Но, **что если нам нужно прочитать из файла всего несколько символов?** Для этого достаточно вызвать метод `read()` и первым аргументом указать максимальное количество читаемых символов, например:

```
print( file.read(4) )
```

Из файла будут считаны первые четыре символа. Правда, в консоли мы видим только три символа. Дело в том, что при кодировке **UTF-8** в файл автоматически добавляется первый (невидимый) символ с шестнадцатиричным кодом:

`file.read(4)`



#FEFF – Скажи-ка, дядя, ведь не даром
Я Python выучил с каналом
Балакирев что раздавал?
Ведь были ж заданья боевые,
Да, говорят, еще какие!
Недаром помнит вся Россия
Как мы рубили их тогда! EOF (End Of File)

Если мы снова вызовем метод:

```
print( file.read(4) )
```

то будут прочитаны следующие четыре символа. Вот эта стрелка (**на рисунке**) называется файловой позицией (**file position**), которая указывает, с какого места производить последующее считывание информации. Благодаря ей мы можем последовательно читать данные, просто вызывая метод `read()`. Когда доходим до конца файла, то здесь находится специальный символ **EOF**, означающий конец файла.

При необходимости, мы можем управлять положением файловой позиции. Для этого существует специальный метод:

Основы Python

`file.seek(offset[, from_what])`

Например, вот такая запись:

```
file.seek(0)
```

будет означать, что мы устанавливаем позицию в начало и тогда такие строчки:

```
print( file.read(4) )  
file.seek(0)  
print( file.read(4) )
```

будут считывать одни и те же первые символы. Если же мы хотим узнать текущую позицию в файле, то следует вызвать метод **tell**:

```
pos = file.tell()  
print( pos )
```

Следующий полезный метод **readline()** позволяет построчно считывать информацию из текстового файла:

```
s = file.readline()  
print( s )
```

Здесь концом строки считается символ переноса `'\n'`, либо конец файла. Причем, этот спецсимвол переноса будет также присутствовать в строке. Мы в этом можем убедиться, вызвав дважды функцию:

```
print( file.readline() )  
print( file.readline() )
```

Здесь в консоли строки будут разделены пустой строкой. Это как раз из-за того, что один перенос идет из прочитанной строки, а второй добавляется самой функцией **print()**. Поэтому, если их записать вот так:

```
print( file.readline(), end="" )
```


Основы Python

```
print( file.readline(), end="" )
```

то вывод будет построчным с одним переносом.

Если нам нужно последовательно прочитать все строки из файла, то для этого обычно используют цикл **for** следующим образом:

```
for line in file:  
    print( line, end="" )
```

Этот пример показывает, что объект файл является итерируемым и на каждой итерации возвращает очередную строку.

Или же, все строки можно прочитать методом:

```
s = file.readlines()
```

и тогда переменная **s** будет ссылаться на список с этими строками:

```
print( s )
```

Однако этот метод следует использовать с осторожностью, т.к. для больших файлов может возникнуть ошибка нехватки памяти для хранения полученного списка.

По сути это все методы для считывания информации из файла. Осталось только добавить, что как только мы завершаем работу с файлом, его следует закрыть. Для этого используется метод **close**:

```
file.close()
```

Конечно, прописывая эту строчку, мы не увидим никакой разницы в работе программы. Но, **во-первых**, закрывая файл, мы освобождаем память, связанную с этим файлом и, **во-вторых**, у нас не будет проблем в потере данных при их записи в файл. А, вообще, лучше **просто запомнить**: после завершения работы с файлом, его нужно закрыть.

Основы Python

На этом мы завершим наше первое знакомство с файлами. Из этого занятия вы должны запомнить, как открыть файл на чтение, как прочитать все данные или часть данных, как менять файловую позицию и не забывать закрывать открытые файлы после работы с ними. Жду всех вас на следующем уроке!

§52. Исключение `FileNotFoundError` и менеджер контекста (`with`) для файлов

На предыдущем занятии я отмечал, что если указать неверный путь к файлу, то возникнет ошибка:

`FileNotFoundError`

и программа досрочно прерывается. Это неприятный момент, тем более, что программист наперед не может знать, в каких условиях будет работать программа и вполне возможна ситуация, когда ранее доступный файл становится недоступным, например, из-за случайного его удаления, или изменения имени и т.п. То есть, при работе с файлами нужно уметь обрабатывать исключение `FileNotFoundError`, чтобы программа продолжала работать, даже если файл не будет найден.

Для обработки подобных ошибок (или, как говорят, исключений) существует специальная группа операторов:

`try / except / finally`

о которых мы подробно будем говорить в части объектно-ориентированного программирования на **Python**. Но, несмотря на то, что это выходит за рамки базового курса, я решил показать, как использовать эти операторы при работе с файлами. Иначе, ваши программы будут заведомо содержать серьезную уязвимость при обращении к файлам.

Формально, операторы `try / except / finally` имеют, следующий синтаксис (определение):

`try:`

блок операторов

Основы Python

критического кода
except [исключение]:
 блок операторов
 обработки исключения
finally:
 блок операторов
 всегда исполняемых
 вне зависимости, от
 возникновения исключения

И в нашем случае их можно записать, так:

```
try:
    file = open("my_file.txt", encoding='utf-8')
    s = file.readlines()
    print(s)
    file.close()
except FileNotFoundError:
    print("Невозможно открыть файл")
```

Смотрите, здесь функция **open()** находится внутри блока **try**, поэтому, если возникнет исключение **FileNotFoundError**, то выполнение программы перейдет в блок **except** и отобразится строка «Невозможно открыть файл». Иначе, мы прочитаем все строки из файла, отобразим их в консоли и закроем файл.

Однако, и в такой реализации не все гладко. В момент считывания информации из файла (в методе **readlines()**) также может возникнуть исключение из-за невозможности считывания информации из файла. Поэтому, совсем надежно можно было бы реализовать эту программу, следующим образом:

```
try:
    file = open("my_file.txt", encoding='utf-8')
```

Основы Python

```
try:
    s = file.readline()
    print(s)
finally:
    file.close()

except FileNotFoundError:
    print("Невозможно открыть файл")
except:
    print("Ошибка при работе с файлом")
```

Мы здесь прописываем еще один вложенный блок **try**, который будет учитывать все возможные исключения и при их возникновении мы обязательно перейдем в блок **finally** для закрытия файла. Это важная процедура – любой ранее открытый файл (функцией **open()**) следует закрывать, даже при возникновении исключений. И вот такая конструкция **try / finally** нам гарантирует его закрытие, что бы ни произошло в момент работы с ним. Но блок **try / finally** не отлавливает исключения, поэтому они передаются внешнему блоку **try** и здесь мы должны их обработать. Я сделал это через второй блок **except**, в котором не указал никакого типа исключений. В результате, он будет реагировать на любые не обработанные ошибки, то есть, в нашем случае – на любые ошибки, кроме **FileNotFoundError**.

Менеджер контекста для файлов

Более я не буду углубляться в работу блоков **try / except / finally**. Приведенного материала пока вполне достаточно, а в заключение этого занятия расскажу о замене блока **try / finally** так называемым файловым менеджером контекста, как это принято делать в программах на **Python**.

Так вот, в языке **Python** существует специальный оператор **with**, который, можно воспринимать как аналог конструкции **try / finally** и в случае работы с файлами записывается, следующим образом:

```
try:
```

Основы Python

```
with open("my_file.txt", encoding='utf-8') as file:  
    s = file.readlines()  
    print( s )
```

```
except FileNotFoundError:  
    print("Невозможно открыть файл")  
except:  
    print("Ошибка при работе с файлом")
```

Смотрите, мы здесь сразу вызываем функцию **open()**, создается объект **file**, через который, затем, вызываем методы для работы с файлом. Причем, все операторы должны быть записаны внутри менеджера контекста, так как после его завершения файл закрывается автоматически. Именно поэтому здесь нет необходимости делать это вручную.

При работе с менеджером контекста следует иметь в виду, что он не обрабатывает никаких ошибок (исключений) и все, что происходит, передается вышестоящему блоку **try**, где они и обрабатываются. Но в любом случае: произошли ошибки или нет, файл будет автоматически закрыт. В этом легко убедиться, если добавить блок **finally** для оператора **try**, в котором отобразить флаг закрытия файла:

```
try:  
    with open("my_file.txt", encoding='utf-8') as file:  
        s = file.readlines()  
        print( s )  
  
except FileNotFoundError:  
    print("Невозможно открыть файл")  
except:  
    print("Ошибка при работе с файлом")  
finally:  
    print(file.closed)
```

Основы Python

После запуска программы видим значение **True**, то есть, файл был закрыт. Даже если произойдет критическая ошибка, например, вызовем функцию `int()` для строки `s`:

```
int(s)
```

то, снова видим значение **True** – файл был закрыт. Вот в этом удобство использования менеджера контекста при работе с файлами.

На этом мы завершим наше ознакомительное занятие по обработке файловых ошибок. Пока будет достаточно запомнить использование операторов **try** / **except** / **finally** при работе с файлами, а также знать, как открывать файлы через менеджер контекста. На следующем уроке мы продолжим тему файлов и будем говорить о способах записи данных.

§53. Запись данных в файл в текстовом и бинарном режимах

На этом занятии вы узнаете, как можно сохранять данные в файл. Первое, что нам нужно сделать – это открыть файл на запись. Для этого вторым аргументом функции `open()` следует указать строку **"w"**:

```
file = open("out.txt", "w")
```

(Если не указывать ничего, как это мы делали на прошлых уроках, то файл будет открыт на чтение, или же, можно явно прописать **"r"** – значение по умолчанию).

Итак, мы открыли файл на запись и далее, чтобы поместить в него какую-либо текстовую информацию следует вызвать метод **write**, следующим образом:

```
file.write("Hello World!")  
file.close()
```

В результате будет создан файл **out.txt** (если его ранее не было) со строкой **«Hello World!»**. Причем, этот файл будет располагаться в том же каталоге, что

Основы Python

и исполняемый файл. Также после записи всей нужной информации в файл, его следует обязательно закрывать! Иначе, часть информации может не сохраниться. Как я уже говорил, метод `close()` нужно вызывать всегда после завершения работы с файлом.

Отлично, это у нас получилось! Давайте теперь запишем метод `write` со строкой «Hello»:

```
file.write("Hello")
```

и снова выполним эту программу. Смотрите, в нашем файле `out.txt` прежнее содержимое исчезло и появилось новое – строка «Hello». То есть, когда мы открываем файл на запись, то прежнее содержимое удаляется. Вот этот очень важный момент всегда следует иметь в виду – открывая файл на запись, его прежнее содержимое автоматически удаляется!

Те из вас, кто смотрел предыдущее занятие, сейчас могут заметить, что при открытии файла на запись могут возникать различные файловые ошибки, поэтому критический блок (кода открытия файла и записи информации) лучше поместить в блок `try`, а все операции с файлом выполнять через менеджер контекста:

```
try:
    with open("out.txt", "w") as file:
        file.write("Hello World!")
except:
    print("Ошибка при работе с файлом")
```

Об этом всегда следует помнить, при работе с файлами! Чтобы программа работала надежно даже в нештатных ситуациях, необходима конструкция `try / except` и менеджер контекста `with`.

Давайте теперь посмотрим, что будет, если вызвать метод `write` несколько раз подряд для одного и того же открытого файла:

```
file.write("Hello1")
```

Основы Python

```
file.write("Hello2")  
file.write("Hello3")
```

Смотрите, в файле появились все эти строчки друг за другом. То есть, здесь как и со считыванием, объект **file** записывает информацию, используя файловую позицию, и автоматически перемещает ее при выполнении записи новых данных. Поэтому все эти строки идут в файле друг за другом.

Если мы хотим, чтобы эти строки шли друг под другом, то в конце каждой из них достаточно указать символ переноса строки:

```
file.write("Hello1\n")  
file.write("Hello2\n")  
file.write("Hello3\n")
```

Однако, если нам все же нужно, мы можем выполнять дозапись информации в файл. Для этого его следует открыть в режиме **'a'** (сокращение от англ. слова **append** – добавление):

```
with open("out.txt", "a") as file:  
    ...
```

Выполняя эту программу, мы видим в файле уже шесть строчек.

Но будьте аккуратны. Если мы открываем файл на запись или дозапись, то читать информацию из этого файла уже не получится. Например, если попытаться воспользоваться методом **read()**:

```
file.read()
```

то возникнет ошибка. Чтобы ее увидеть я поставлю комментарии у блока **try** и снова запущу программу. Видим исключение **UnsupportedOperation**.

Если же мы хотим и записывать и считывать информацию, то можно воспользоваться режимом **a+**:

```
file = open("out.txt", "a+")
```


Основы Python

Так как здесь файловый указатель стоит на последней позиции, то для считывания информации, поставим его в самое начало:

```
file.seek(0)
print( file.read() )
```

А вот запись данных всегда осуществляется в конец файла, так как для записи используется другая файловая позиция и она указывает на конец файла.

Далее, если нам нужно записать несколько строк, представленных в какой-либо коллекции, то удобно воспользоваться методом **writelines()**:

```
file.writelines(["Hello1\n", "Hello2\n"])
```

Это часто используется, если в процессе обработки текста у нас получается список из строк и его требуется целиком поместить в файл. Здесь метод **writelines()** незаменим.

Чтение и запись в бинарном режиме доступа

До сих пор мы с вами говорили о чтении и записи информации в текстовом режиме доступа. Есть еще другой – бинарный режим работы с файлами. **Что такое бинарный режим доступа?** Это когда данные из файла считываются один в один без какой-либо обработки. Обычно он используется для сохранения и считывания объектов целиком. Давайте предположим, что нам нужно сохранить в файл вот такой список:

```
books = [
    ("Евгений Онегин", "Пушкин А.С.", 200),
    ("Муму", "Тургенев И.С.", 250),
    ("Мастер и Маргарита", "Булгаков М.А.", 500),
    ("Мертвые души", "Гоголь Н.В.", 190)
]
```

Для этого откроем файл на запись в бинарном режиме:

Основы Python

```
file = open("out.bin", "wb")
```

Затем, для работы с бинарными данными подключим специальный встроенный модуль **pickle**:

```
import pickle
```

И вызовем его метод **dump**:

```
pickle.dump(books, file)
```

Все, мы сохранили этот объект в файл. Теперь прочитаем эти данные. Откроем файл на чтение в бинарном режиме:

```
file = open("out.bin", "rb")
```

и вызовем метод **load** модуля **pickle**:

```
bs = pickle.load(file)
```

Все, теперь переменная **bs** ссылается на эквивалентный список:

```
print( bs )
```

Аналогичным образом можно записывать и считывать сразу несколько объектов. Например, так:

```
import pickle
```

```
book1 = ["Евгений Онегин", "Пушкин А.С.", 200]
```

```
book2 = ["Муму", "Тургенев И.С.", 250]
```

```
book3 = ["Мастер и Маргарита", "Булгаков М.А.", 500]
```

```
book4 = ["Мертвые души", "Гоголь Н.В.", 190]
```

```
try:
```

```
    with open("out.bin", "wb") as file:
```

```
        pickle.dump(book1, file)
```

Основы Python

```
pickle.dump(book2, file)
pickle.dump(book3, file)
pickle.dump(book4, file)
except:
    print("Ошибка при работе с файлом")
```

А, затем, считывание в том же порядке:

```
import pickle

try:
    with open("out.bin", "rb") as file:
        b1 = pickle.load(file)
        b2 = pickle.load(file)
        b3 = pickle.load(file)
        b4 = pickle.load(file)
except:
    print("Ошибка при работе с файлом")

print( b1, b2, b3, b4, sep="\n" )
```

Вот так в **Python** выполняется запись и считывание данных в бинарном режиме.

Надеюсь, из этого занятия вам стало понятно, как производится запись данных в файл, как делать дозапись информации, что такое бинарный и текстовый режимы доступа и как записывать и считывать информацию в бинарном режиме с помощью методов модуля **pickle**. До встречи на следующем занятии!

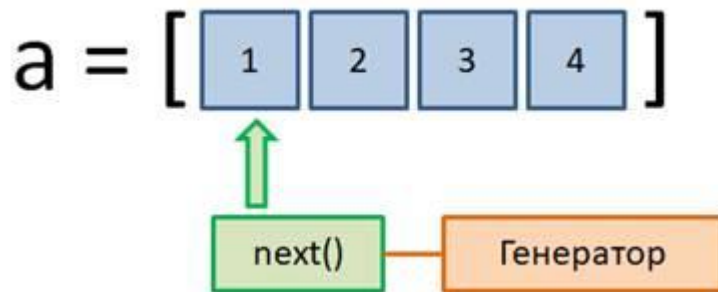
Основы Python

§54. Выражения генераторы

На этом занятии мы с вами поговорим о выражениях-генераторах. **Что это такое?** Смотрите, когда мы рассматривали генераторы списков, то в квадратных скобках описывали некий алгоритм формирования значений списка:

```
a = [x for x in range(1, 5)]
```

Так вот, все эти значения были сгенерированы внутренним генератором, с помощью которого и заполнялся весь этот список.



То же самое и для генерации множеств и словарей. Здесь также внутренний генератор автоматически заполнял значениями эти коллекции. Так вот, в **Python** можно определить такой генератор без привязки к какой-либо коллекции. Для этого используется синтаксис такой же, как и для генераторов коллекций, только все записывается в круглых скобках:

(**<формирование значения> for <переменная> in <итерируемый объект>**)

Обратите внимание, круглые скобки здесь не означают кортеж. Генераторов кортежей не существует. Когда мы пишем круглые скобки, то получаем на выходе «чистый» генератор. Например, строка:

```
a = (x ** 2 for x in range(6))
```

задает генератор для формирования квадратов целых чисел от 0 до 5. Если выведем значение переменной **a**, то увидим, что это объект-генератор:

Основы Python

<generator object <genexpr> at 0x032DEAE0>

Но **как получить нам конкретные значения из этого генератора?** Очень просто! Генератор сам по себе является также и итератором, то есть, его значения можно перебирать с помощью функции **next()**:

```
next(a)
...
next(a)
```

пока не получим исключение **StopIteration**. Соответственно, пройтись по значениям генератора можно только один раз. Повторно их перебрать уже не получится.

Помимо функции **next()** генераторы можно перебирать и в цикле **for**, так как они являются итерируемыми объектами:

```
gen = (x ** 2 for x in range(6))

for x in gen:
    print(x)
```

Но, опять же, перебрать его можно только один раз, поэтому повторный цикл **for** не выведет никаких значений:

```
for x in gen:
    print(x)
```

Некоторые функции, такие как:

list, set, sum, max, min и другие

позволяют работать непосредственно с итераторами. То есть, в качестве аргумента им можно передавать генератор, например:

```
a = (x ** 2 for x in range(6))
list(a)
```

Основы Python

будет сформирован список из значений на выходе генератора. Или так:

```
a = (x ** 2 for x in range(6))  
set(a)
```

получим множество из значений генератора. То же самое и для других функций:

```
sum((x ** 2 for x in range(6)))  
max((x ** 2 for x in range(6)))
```

И еще раз обратите внимание, использовать эти и другие подобные им функции для одного и того же генератора можно только один раз. Например, вызывая функцию `sum()` дважды:

```
a = (x ** 2 for x in range(6))  
sum(a)  
sum(a)
```

Во **втором** случае увидим **0**, так как элементы генератора нельзя обойти второй раз.

Но здесь остается один важный вопрос: **зачем вообще нужны эти выражения-генераторы?** Дело в том, что эти объекты по сравнению, например, с обычными списками не хранят в памяти все значения сразу, а генерируют их по мере необходимости, то есть, при переходе к следующему значению. Например, если возникает необходимость оперировать очень большим списком:

```
lst = list(range(1000000000000))
```

то у компьютера попросту не хватит памяти для его хранения и возникнет ошибка. А вот с генераторами таких проблем не возникнет, так как они не хранят все числа сразу, а формируют их по мере необходимости «**на лету**»:

```
lst = (x for x in range(10000000000))
```

Основы Python

```
for i in lst:
    print(i, end=" ")
    if i > 50:
        break
```

И, кроме того, работает такая конструкция достаточно быстро. Правда, для генераторов нельзя определить число элементов через функцию `len()`:

```
a = (x for x in range(10, 20))
len(a)
```

или получить доступ к его отдельному элементу по индексу:

```
a[2]
```

так как этих элементов попросту нет, они генерируются последовательно при вызове функции `next()`.

Если все же требуется работать со значениями генератора, как с элементами списка, то его сначала нужно преобразовать в этот список, а затем со списком и работать:

```
a = (x for x in range(10, 20))
b = list(a)
```

И, опять же, если попробовать преобразовать этот же генератор к списку еще раз:

```
b2 = list(a)
```

то получим пустой список, так как элементы генератора уже были перебраны в первой функции `list()` и сделать это повторно не получится.

Также не получится преобразовать генератор к списку, используя квадратные скобки:

Основы Python

```
[(x ** 2 for x in range(6))]
```

В этом случае первым элементом списка будет ссылка на объект-генератор, не более того. Также это означает, что у генераторов списков нельзя внутри прописывать круглые скобки, как это сделано сейчас. Теперь мы знаем, что это определение выражения-генератора внутри обычного списка.

Итак, из этого занятия вы должны были узнать, что из себя представляют выражения-генераторы, как перебираются их элементы и для чего они могут понадобиться. Помнить об ограничениях, накладываемые на генераторы, связанные с генерацией значений «на лету», то есть, нельзя использовать функцию `len()`, обращаться по индексу и перебирать его элементы только один раз. Для закрепления этого материала пройдите практические задания и переходите к следующему уроку.

§55. Функция-генератор. Оператор `yield`

На этом занятии мы с вами поговорим о функциях-генераторах. Но вначале вернемся к выражениям-генераторам, которые рассматривали на предыдущем занятии. Давайте предположим, что нам нужны средние арифметические значения для разных последовательностей целых чисел:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2, 3, 4, 5, 6, 7, 8, 9, 10
3, 4, 5, 6, 7, 8, 9, 10
4, 5, 6, 7, 8, 9, 10
5, 6, 7, 8, 9, 10
6, 7, 8, 9, 10
7, 8, 9, 10
8, 9, 10
9, 10

Для этого мы могли бы записать следующее выражение-генератор:

```
N = 10  
a = (sum(range(i, N+1))/len(range(i, N+1)) for i in range(N))
```


Основы Python

Но оно не очень удобно для восприятия и редактирования и, кроме того, здесь дважды записана функция `range()` при вычислении среднего значения. Поправить это можно двумя способами. В первом, объявить собственную функцию для вычисления среднего арифметического и вызвать ее в генераторе:

```
def avg(start, stop, step=1):  
    a = range(start, stop, step)  
    return sum(a) / len(a)
```

```
N = 10
```

```
a = (avg(i, N + 1) for i in range(1, N))  
print(list(a))
```

А во втором способе – создать собственную **функцию-генератор**, которая бы на выходе выдавала нужные значения. Давайте для начала запишем простую функцию-генератор, а потом вернемся к нашей исходной задаче. Функция будет просто возвращать значения списка:

```
def get_list():  
    for x in [1, 2, 3, 4]:  
        yield x
```

Смотрите, здесь в цикле записан новый для нас оператор **yield**, который возвращает текущее значение **x** и «замораживает» состояние функции до следующего обращения к ней (в том числе и все локальные переменные). Именно так определяются функции-генераторы. Если мы сейчас ее вызовем:

```
d = get_list()  
print(d)
```

Основы Python

то, смотрите, переменная **d** ссылается на объект-генератор, то есть, мы здесь имеем дело с генератором, значения которого можно перебирать с помощью функции **next()**:

```
print(next(d))
print(next(d))
```

Или, через цикл:

```
d = get_list()
for i in d:
    print(i, end=" ")
```

В этом и заключается роль оператора **yield**. Он превращает обычную функцию в генератор и при каждом вызове функции **next()** активизируется функция-генератор, возвращает очередное значение и «замораживает» свое состояние вместе с локальными переменными до следующего вызова функции **next()**. (Показываем это в режиме отладки).

Надеюсь, вы теперь хорошо представляете, как работает оператор **yield** и простая функция-генератор. Давайте вернемся к нашей исходной задаче и перепишем функцию **avg()** с использованием оператора **yield** (создаем еще одну):

```
def avg_gen(N, step=1):
    for i in range(1, N):
        a = range(i, N+1, step)
        yield sum(a) / len(a)
```

Принцип здесь тот же самый, на каждой итерации цикла будет возвращаться новое вычисленное среднее арифметическое значение. Далее, мы можем воспользоваться этой функцией-генератором и отобразить все ее значения, используя функцию **list()**:

```
b = avg_gen(N)
print(list(b))
```

Основы Python

Как видите, результат полностью совпадает с первоначальным. То есть, мы заменили выражение-генератор на функцию-генератор. Ну и, как всегда, возникает вопрос, **зачем это все было надо? Почему бы не пользоваться обычными генераторами? Что нам мешает это делать?** В целом, ничего. Преимущество здесь, главным образом, в удобстве использования. В выражении генератора мы можем записать лишь один оператор для формирования значения, а в функции-генераторе – произвольный фрагмент программы, реализующий нужную нам логику формирования очередного значения. В этом ключевое отличие функции-генератора от обычного генератора.

В заключение этого занятия приведу еще один пример использования функции-генератора. Предположим, что мы хотим найти все начальные индексы слова «генератор» в текстовом файле `lesson_54.txt`. Для этого вначале откроем этот файл на чтение:

```
try:
    with open("lesson_54.txt", encoding="utf-8") as file:
        a = find_word(file, "генератор")
        print(list(a))
except FileNotFoundError:
    print("Файл не найден!")
except:
    print("Ошибка обработки файла!")
```

И внутри блока `try` вызовем функцию-генератор, которая и будет последовательно возвращать индексы найденного слова «генератор». Саму функцию определим выше (по программе), следующим образом:

```
def find_word(f, word):
    g_indx = 0
    for line in f:
        indx = 0
        while(indx != -1):
            indx = line.find(word, indx)
```

Основы Python

```
if indx > -1:  
    yield g_indx + indx  
    indx += 1  
  
g_indx += len(line)
```

Мы здесь в **первом** цикле читаем файл по строкам. Во **втором** вложенном цикле **while** ищем указанное слово в строке, используя метод **find()**. И, если этот метод находит заданный фрагмент, то есть, возвращает значение больше -1, то функция генерирует на выходе значение индекса найденного слова как **g_indx + indx**. Здесь **g_indx** – это смещение по тексту для текущей строки, то есть, в ней мы суммируем длины предыдущих строк, чтобы сформировать индекс слова в тексте, а не в строке.

После запуска этой программы видим все найденные индексы данного слова по тексту. Как видите, мы в функции описали нетривиальную логику алгоритма поиска слова в текстовом файле. Сделать это с помощью обычного генератора было бы сложнее. В этом и преимущество функций-генераторов – они позволяют сразу, в одном месте кода, описывать нужный нам функционал.

На этом мы завершим с вами очередное занятие. Для закрепления материала, как всегда, пройдите практические задания и переходите к следующему уроку.

§56. Функция **map**. Примеры ее использования

На этом занятии мы с вами поговорим о функции **map()**. Мы ее ранее уже использовали и теперь пришло время в деталях понять, как она работает.

Вначале я приведу простой пример использования функции **map()** для преобразования строк чисел в обычные числа:

```
b = map(int, ['1', '2', '3', '5', '7'])
```

Основы Python

Первым аргументом указана ссылка на функцию, которая будет последовательно применяться к каждому элементу списка, а вторым аргументом – список из строк с числами. Вообще, вторым аргументом можно записывать любой итерируемый объект. На выходе функция **map()** возвращает итератор. То есть, переменная **b** – это итератор, который можно перебрать для преобразования строк в числа.

Вернемся к нашей программе и ниже дважды вызовем функцию **next()** для итератора **b**:

```
print(next(b))
print(next(b))
```

В консоли видим первые два преобразованных значения. Давайте переберем все их с помощью цикла **for**:

```
for x in b:
    print(x, end=" ")
```

Как видите, получили числа соответствующих строк. То есть, здесь действительно функция **map()** последовательно применила функцию **int()** к каждому элементу списка и на выходе мы видим уже обычные целые числа.

Мы также легко можем сохранить результат преобразования в новом списке, используя функцию **list()**:

```
a = list(b)
print(a)
```

Здесь функция **list()** автоматически перебрала итератор, неявно вызывая функцию **next()**, и сформировала соответствующие значения списка.

Этот же результат можно получить, используя генератор списка, следующим образом:

```
a = [int(x) for x in ['1', '2', '3', '5', '7']]
```

Основы Python

```
print(a)
```

Но здесь, в отличие от функции `map()` мы все значения уже храним в памяти, тогда как функция `map()` возвращает итератор и значения формируются по одному в процессе вызова функции `next()`.

А вот эквивалентный генератор:

```
a = (int(x) for x in ['1', '2', '3', '5', '7'])
```

нам бы, фактически, дал то же самое, что и функция `map()`. То есть, `map()` возвращает генератор, в котором некая функция применяется последовательно к элементам итерируемой последовательности. Используется она исключительно для удобства, чтобы не прописывать генератор в классическом виде.

Далее, кроме функции `list()` мы также с итератором можем применять некоторые другие функции, которые в качестве аргумента принимают итерированный объект, например:

```
print(sum(b))
```

Но, если следом попытаться перебрать итератор еще раз:

```
print(sum(b))
```

то увидим значение `0`, так как мы помним, что итератор можно перебирать только один раз. Вот это следует помнить, используя функцию `map()` – ее значения можно извлечь только один раз.

Вместо функции `sum()` можно также использовать функции `max()` и `min()`

```
print(max(b))  
print(min(b))
```

И другие, которые работают с итерируемыми последовательностями.

Основы Python

Конечно, вместо функции `int()` мы можем использовать любую другую, которая принимает один аргумент и возвращает некоторое значение. Например, можем взять список городов:

```
cities = ["Москва", "Астрахань", "Самара", "Уфа", "Смоленск", "Тверь"]
```

и применить к его элементам функцию `len()`, следующим образом:

```
b = map(len, cities)
print(list(b))
```

Видите, как легко и просто записан генератор для преобразования списка `cities`? В этом и заключается удобство использования функции `map()`.

Также мы можем применять к этим строкам их методы, например, переведем все в верхний регистр:

```
b = map(str.upper, cities)
```

Мы здесь указали объект `str` и его метод `upper()`, который возвращает новую строку со всеми заглавными буквами.

Также мы можем определять и свои функции, используемые в `map()`. Как я уже отмечал, функция обязательно должна принимать один аргумент и возвращать некоторое значение, например, так:

```
def symbols(s):
    return list(s.lower())
```

Мы здесь сначала преобразовываем строку в нижний регистр, а затем, формируем список из отдельных символов, который и возвращаем. Далее, в функции `map()` указываем эту функцию:

```
b = map(symbols, cities)
```

Основы Python

(Обратите внимание, без круглых скобок, то есть, передаем ссылку на нее, а не вызываем). И после запуска программы увидим наборы вложенных списков из отдельных символов исходных строк.

Конечно, если функция выполняет какую-либо простую операцию, то часто прописывают лямбда-функции непосредственно в `map()`. В нашем случае это можно сделать так:

```
b = map(lambda s: list(s.lower()), cities)
```

Результат будет прежним, а программа стала более простой и читабельной.

Или, можно преобразовать строки, записав их символы в обратном порядке:

```
b = map(lambda s: s[::-1], cities)
```

Как видите, используя механизм срезов и функцию `map()`, сделать это очень просто. Поэтому весь материал, что мы с вами проходим, нужно очень хорошо запоминать, чтобы уметь использовать в своих программах.

Ну и теперь, возвращаясь к уже знакомой нам конструкции:

```
s = map(int, input().split())  
print(list(s))
```

(Я ее записал здесь в две строки, чтобы было понятнее.) Вы понимаете, как она работает. Здесь `input().split()` возвращает список из строк введенных чисел, к каждой строке применяется функция `int()` и с помощью функции `list()` генератор `s` превращается в список из чисел.

Вот, что из себя представляет функция `map()`, которая довольно часто применяется на практике. Закрепите этот материал практическими заданиями и жду всех вас на следующем уроке.

Основы Python

§57. Функция `filter` для отбора значений итерируемых объектов

На прошлом занятии мы с вами детально познакомились с функцией `map()`. Здесь же будем говорить о похожей функции `filter()`:

`filter(func, *iterables)`

которая служит для фильтрации (отбора) элементов указанного итерированного объекта.

Работает она очень просто. Если функция `func` возвращает для текущего значения элемента **True**, то он будет возвращен, а при **False** – отброшен. Например, у нас имеется список из целых чисел:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

и мы хотим выбрать из него только четные значения. Для этого запишем функцию `filter()`, следующим образом:

```
b = filter(lambda x: x % 2 == 0, a)
print(b)
```

После запуска программы, видим, что переменная `b` ссылается на специальный объект `filter`. В действительности, это итератор, который можно перебрать с помощью функции `next()`:

```
print(next(b))
print(next(b))
```

В результате, увидим в консоли первые два четных значения из списка. Переберем их все с помощью цикла `for`:

```
for x in b:
    print(x, end=" ")
```

Или, можно сформировать новый список с помощью функции `list()`:

Основы Python

```
c = list(b)
print(c)
```

Или кортеж:

```
c = tuple(b)
```

И так далее. Можно перебрать итератор любыми известными нам функциями и операторами.

Иногда алгоритм обработки текущего значения может быть нетривиальным и записать его в лямбда-функцию довольно сложно. Ну, например, проверить, является ли число простым или нет. Напомню, что простым называется любое натуральное число, которое делится только на себя и на единицу. Такую проверку лучше вынести в отдельную функцию:

```
def is_prost(x):
    d = x-1
    if d < 0:
        return False

    while d > 1:
        if x % d == 0:
            return False
        d -= 1

    return True
```

А, затем, указать ее в **filter()**:

```
b = filter(is_prost, a)
```

На выходе получим только простые числа из списка. Конечно, это не самый лучший способ нахождения простых чисел, но как пример функции для **filter()** вполне подходит.

Основы Python

Функцию **filter** можно применять с любыми типами данных, например, строками. Пусть у нас имеется вот такой кортеж:

```
lst = ("Москва", "Рязань1", "Смоленск", "Тверь2", "Томск")
b = filter(str.isalpha, lst)

for x in b:
    print(x)
```

и мы вызываем метод строк **isalpha()**, который возвращает **True**, если в строке только буквенные символы. В результате в консоли увидим:

Москва Смоленск Тверь Томск

Вложенные вызовы функции filter()

Так как функция **filter()** возвращает итератор и в качестве второго аргумента также можно указывать любой итерируемый объект, то мы можем одну функцию **filter()** вложить в другую:

```
a = filter(func, *iterables)
      ↓
b = filter(func, a)
```

Например, первая (вложенная) функция **filter()** будет формировать простые числа, а вторая (внешняя) выбирать из простых чисел только нечетные. В нашей программе для этого достаточно прописать еще одну строчку:

```
b2 = filter(lambda x: x % 2 != 0, b)
```

и вывести полученный список:

```
c = tuple(b2)
print(c)
```

Или, то же самое можно реализовать и так:

Основы Python

```
b2 = filter(lambda x: x % 2 != 0, filter(is_prost, a))
```

Мы здесь вторым аргументом передаем результат работы вложенной функции **filter()**. И таких вложений можно делать сколько угодно. Хотя на практике, как правило, все эти вложения можно легко свести к одному простому вызову функции **filter()**, сформировав более сложное условие. Обычно, так и поступают. То есть, опять же, в нашем случае вместо этой вложенности можно немного модифицировать саму функцию **is_prost()**, следующим образом:

```
def is_prost(x):
    d = x-1
    if d < 0 or x % 2 == 0:
        return False

    while d > 1:
        if x % d == 0:
            return False
        d -= 1

    return True
```

Мы здесь вначале прописали составное условие **if d < 0 or x % 2 == 0**, которое сразу возвратит **False** для четного значения **x**. И при этом, программа будет работать быстрее.

Вот так работает и используется функция **filter()** для отбора определенных значений из итерируемых объектов. Для закрепления этого материала пройдите практические задания и переходите к следующему уроку.

§58. Функция **zip**. Примеры использования

На прошлых занятиях мы с вами познакомились с двумя функциями **map()** и **filter()**. На этом продолжим эту тему и поговорим о третьей часто применяемой функции:

Основы Python

`zip(iter1 [, iter2 [, iter3] ...])`

Она для указанных итерируемых объектов перебирает соответствующие элементы и продолжает работу до тех пор, пока не дойдет до конца самой короткой коллекции. Таким образом, она гарантирует, что на выходе будут формироваться наборы значений из всех переданных объектов.

Давайте в качестве простого примера возьмем два списка со значениями:

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8, 9, 10]
```

и передадим их в качестве аргументов функции `zip()`:

```
z = zip(a, b)
print(z)
```

После запуска программы увидим, что переменная `z` ссылается на объект `zip`. Как вы уже догадались, это итератор, элементы которого перебираются функцией `next()`:

```
print(next(z))
print(next(z))
```

Давайте переберем все пары через цикл `for`:

```
for x in z:
    print(x)
```

Здесь `x` является кортежем из соответствующих значений списков `a` и `b`. Причем последняя пара заканчивается числами (4, 8), когда функция `zip()` дошла до конца самого короткого списка.

Также всегда следует помнить, что функция `zip()` возвращает итератор. Это значит, что перебрать элементы можно только один раз. Например, повторный вызов оператора `for` для этой же функции:

Основы Python

```
for x in z:  
    print(x)
```

ничего не возвратит, так как мы уже один раз прошлись по его элементам в предыдущем цикле.

Если нам все же нужно несколько раз обходить выделенные пары элементов, то их следует вначале преобразовать, например, в кортеж:

```
z = tuple(zip(a, b))
```

А, затем, обходить несчетное число раз. Но в этом случае увеличивается расход памяти, так как мы сохраняем все элементы в кортеже, а не генерируем «**на лету**» с помощью итератора. Так что, без особой необходимости делать преобразование к кортежу или списку не стоит.

В качестве перебираемых коллекций могут быть любые итерируемые объекты, например, строки. Если мы добавим строку:

```
c = "python"
```

и вызовем функцию **zip()** и с ней:

```
z = zip(a, b, c)
```

то в консоли увидим кортежи уже из трех значений, причем последнее будет соответствующим символом строки. Также, напомним, что оператор **for** можно записать с тремя переменными, если перебираемые значения являются кортежами:

```
for v1, v2, v3 in z:  
    print(v1, v2, v3)
```

Иногда удобно кортеж сразу распаковать в отдельные переменные и использовать внутри цикла **for**.

Основы Python

Также все кортежи из функции `zip()` мы можем получить с помощью распаковки, используя запись:

```
z1, z2, z3, z4 = zip(a, b, c)
print(z1, z2, z3, z4, sep="\n")
```

Здесь неявно вызывается функция `next()` для извлечения всех элементов. Или, можно записать так:

```
z1, *z2 = zip(a, b, c)
print(z1, z2, sep="\n")
```

Тогда первое значение будет помещено в переменную `z1`, а все остальные – в список `z2`.

И в заключение этого занятия я покажу вам еще одно интересное преобразование, которое можно выполнить с помощью функции `zip()`. Если преобразовать итератор в список:

```
z = zip(a, b, c)
lz = list(z)
```

а, затем, распаковать его элементы с помощью оператора `*`:

```
print(*lz)
```

то мы получим уже не список, а четыре кортежа, следующих друг за другом. Если передать их в таком виде на вход функции `zip()`:

```
t1, t2, t3 = zip(*lz)
print(t1, t2, t3, sep="\n")
```

то на выходе будут сформированы три кортежа из соответствующих элементов этих четырех. Вообще, это эквивалентно записи:

```
t1, t2, t3 = zip((1, 5, 'p'), (2, 6, 'y'), (3, 7, 't'), (4, 8, 'h'))
```

Основы Python

Но те же самые кортежи мы можем получить гораздо проще, если распаковать непосредственно итератор `z`:

```
t1, t2, t3 = zip(*z)
```

Как видите, к итератору вполне применим оператор распаковки элементов и на практике иногда этим пользуются.

Надеюсь, из этого занятия вы узнали, как работает функция `zip()` и ее назначение. Для закрепления материала, как всегда, пройдите практические задания и переходите к следующему уроку.

§59. Особенности сортировки через `sort()` и `sorted()`

Здесь мы с вами затронем вопрос сортировки итерируемых объектов с помощью метода `sort()` и функции `sorted()`. Давайте вначале посмотрим на отличие в их вызовах. Если у нас имеется какой-либо список:

```
a=[1,-45,3,2,100,-4]
```

то этот объект имеет встроенный метод `sort`, который меняет его состояние и расставляет элементы по возрастанию:

```
a.sort()
```

Получим измененный список:

```
[-45, -4, 1, 2, 3, 100]
```

А вот коллекции кортежи или строки:

```
b=("ab", "bc", "wd", "gf")  
c = "hello world"
```

не имеют такого встроенного метода и попытка их отсортировать, записав:

```
b.sort()
```


Основы Python

```
c.sort()
```

приведет к ошибке. Для их сортировки как раз и можно воспользоваться второй функцией **sorted**:

```
sorted(b)
```

на выходе получим упорядоченный список

```
['ab', 'bc', 'gf', 'wd']
```

Обратите внимание, чтобы мы не передавали в качестве аргумента функции **sorted**, на выходе будем получать именно список отсортированных данных. В данном случае передаем кортеж, а получаем – список.

Или же, со строкой:

```
sorted(c)
```

результатом будет упорядоченная коллекция из символов:

```
[' ', 'd', 'e', 'h', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
```

Причем, эта функция не меняет исходные коллекции **b** и **c**, она возвращает новый список с отсортированными данными. В то время как метод **sort** для списка меняет этот список. Вот на это следует также обращать внимание. То есть, если нам нужно сохранить результат сортировки в переменной, это делается так:

```
res = sorted(b)
```

и **res** будет ссылаться на список:

```
['ab', 'bc', 'gf', 'wd']
```

Основы Python

Также следует иметь в виду, что сортировка данных возможна для однотипных элементов: или **чисел**, или **строк**, или **кортежей**, но не их комбинаций. Например, вот такой список:

```
a=[1,-45,3,2,100,-4,"b"]
```

отсортировать не получится:

```
a.sort()
```

возникнет ошибка, что строку нельзя сравнивать с числом. И то же самое с функцией **sorted**:

```
sorted(a)
```

Если уберем последний элемент:

```
a=[1,-45,3,2,100,-4]
```

то все будет работать:

```
sorted(a)
```

И этот пример также показывает, что список можно сортировать и с помощью метода **sort** и с помощью функции **sorted**. Разница только в том, что метод **sort** не создает новой коллекции, а меняет уже существующую. Функция же **sorted** не меняет исходную коллекцию, а создает новую с отсортированными элементами. Поэтому, для изменения коллекции а здесь следует записывать такую конструкцию:

```
a = sorted(a)
```

Оба этих подхода к сортировке поддерживают необязательный параметр

reverse = True/False

Основы Python

который определяет порядок сортировки: по возрастанию (**False**) или по убыванию (**True**). По умолчанию стоит значение **reverse=False**. Если мы запишем его вот так:

```
a = sorted(a, reverse=True)
```

то получим сортировку по убыванию:

```
[100, 3, 2, 1, -4, -45]
```

И то же самое с методом **sort**:

```
a.sort(reverse=True)
```

Давайте еще посмотрим, как будет работать сортировка для словаря:

```
d = {'river': "река", 'house': "дом", 'tree': "дерево", 'road': "дорога"}
```

Очевидно, для него мы можем использовать только функцию **sorted()**:

```
sorted(d)
```

которая на выходе даст список с ключами, отсортированными по возрастанию. И это логично, так как словарь по умолчанию возвращает именно ключи при его итерировании.

Также можно использовать конструкции вида:

```
sorted(d.values())
```

и

```
sorted(d.items())
```

Работают они очевидным образом.

Надеюсь, из этого занятия вы лучше стали понимать, как можно выполнять сортировку различных коллекций в языке **Python**. Для закрепления этого

Основы Python

материала вас ждут практические задания, а я буду всех ждать на следующем уроке, где мы продолжим эту тему и поговорим о тонкой настройке алгоритма сортировки данных.

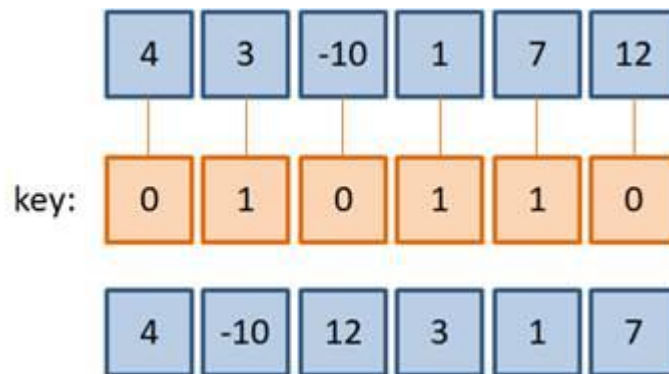
§60. Аргумент `key` для сортировки коллекций по ключу

На этом занятии мы увидим, как можно управлять алгоритмом сортировки с помощью специального параметра **key**, который имеется у метода `sort()` и функции `sorted()`.

По умолчанию сортировка коллекции выполняется по значениям ее элементов. Например:

```
a = [4, 3, -10, 1, 7, 12]
b = sorted(a)
print(b)
```

Но мы можем вместо этих значений указать другие, которые будут использованы для сортировки элементов. Например, вычислить показатель четности значений. Тогда последовательность будет выстроена по возрастанию этих ключей. В результате увидим сначала четные значения, а потом – нечетные.



Чтобы выполнить такую манипуляцию в функции `sorted()` прописывается аргумент **key** и ему присваивается ссылка на функцию, которая будет формировать альтернативное значение элемента, то есть, ключ:

Основы Python

```
b = sorted(a, key=is_odd)
```

А саму функцию можно определить, следующим образом:

```
def is_odd(x):  
    return x % 2
```

Здесь аргумент **x** – это текущее значение элемента коллекции, а то, что она возвращает, становится значением соответствующего ключа. То есть, для четных значений будем иметь **0**, а для нечетных – **1**. После запуска программы видим искомый результат сортировки.

Конечно, для простых функций, обычно, в аргументе **key** записывают лямбда-функцию. В нашем примере она будет выглядеть, следующим образом:

```
b = sorted(a, key=lambda x: x % 2)
```

Как видите, все достаточно просто.

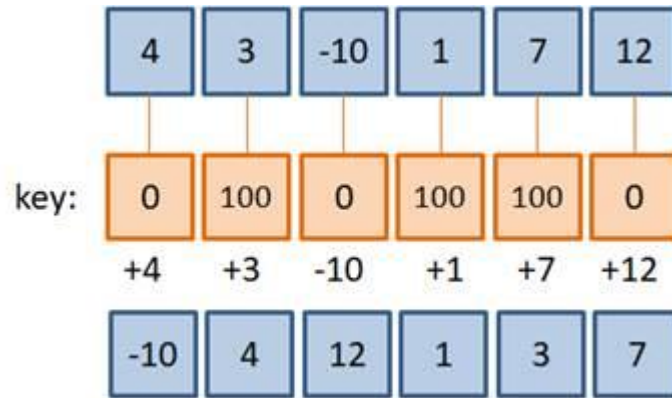
Тот же самый аргумент **key** можно указывать и в методе **sort()** для списка:

```
a.sort(key=lambda x: x % 2)
```

Он здесь работает абсолютно также, как и в функции **sorted()**.

Давайте теперь немного усложним нашу задачу и сделаем не просто разделение на четные и нечетные, а еще и отсортируем каждую группу по возрастанию. Для этого значения ключей я буду увеличивать на значения элементов и, кроме того, нечетным величинам изначально присваивать число **100**, чтобы гарантированно разделить четные и нечетные значения именно в нашей коллекции. Это исключительно учебный пример, показывающий возможности сортировки коллекций по ключу.

Основы Python



Итак, для формирования таких ключей, определим следующую функцию:

```
def key_sort(x):  
    return x if x % 2 == 0 else 100 + x
```

И укажем ее в функции `sorted()`:

```
b = sorted(a, key=key_sort)  
print(b)
```

Теперь у нас выполняется не только разделение на четные и нечетные значения, но и их сортировка внутри групп.

Я, думаю, что вы в целом поняли, как использовать аргумент **key** для управления сортировкой. И здесь, как всегда, извечный вопрос – **зачем это надо?** Давайте я приведу несколько более практичных примеров. Предположим, что у нас имеется список городов:

```
lst = ["Москва", "Тверь", "Смоленск", "Псков", "Рязань"]
```

И требуется их выстроить по длине. Для этого воспользуемся функцией `sorted()` и в аргументе **key** укажем стандартную функцию **len**:

```
print( sorted(lst, key=len) )
```

получим следующий результат:

```
['Тверь', 'Псков', 'Москва', 'Рязань', 'Смоленск']
```

Основы Python

Или можно сделать сортировку по последнему символу слова:

```
print( sorted(lst, key=lambda x: x[-1]) )
```

```
['Москва', 'Псков', 'Смоленск', 'Тверь', 'Рязань']
```

Или только по первому:

```
print( sorted(lst, key=lambda x: x[0]) )
```

```
['Москва', 'Псков', 'Рязань', 'Смоленск', 'Тверь']
```

И так далее.

Аргумент **key** часто используют для сортировки сложных структур данных. Допустим, у нас имеется вот такой кортеж, содержащий вложенные кортежи с информацией по книгам:

```
books = (  
    ("Евгений Онегин", "Пушкин А.С.", 200),  
    ("Муму", "Тургенев И.С.", 250),  
    ("Мастер и Маргарита", "Булгаков М.А.", 500),  
    ("Мертвые души", "Гоголь Н.В.", 190)  
)
```

И нам нужно его отсортировать по цене (последнее значение). Очевидно, это можно сделать так:

```
print( sorted(books, key=lambda x: x[2]) )
```

На выходе получим отсортированный список:

```
[('Мертвые души', 'Гоголь Н.В.', 190), ('Евгений Онегин', 'Пушкин А.С.', 200),  
('Муму', 'Тургенев И.С.', 250), ('Мастер и Маргарита', 'Булгаков М.А.', 500)]
```

Основы Python

Вот так используется аргумент **key** для управления сортировкой элементов произвольных коллекций данных. И теперь вы знаете, как его применять в своих программах.

§61. Функции `isinstance` и `type` для проверки типов данных

На этом занятии мы поговорим о функции `isinstance()`, с помощью которой можно выполнять проверку на принадлежность объекта определенным типам данных.

Например, объявим переменную:

```
a = 5
```

и для нее вызовем функцию `isinstance()`, следующим образом:

```
isinstance(a, int)
```

Увидим значение **True**, так как переменная **a** действительно ссылается на целочисленный объект. А вот если указать:

```
isinstance(a, float)
```

то уже будет возвращено значение **False**. Вот так в самом простом варианте работает эта функция. Но у нее есть один нюанс, связанный с булевым типом данных. Если определить переменную:

```
b = True
```

то, очевидно, функция:

```
isinstance(b, bool)
```

вернет значение **True**. Однако, если вместо **bool** записать **int**:

```
isinstance(b, int)
```


Основы Python

то тоже увидим **True**. Это связано с особенностью реализацией типа **bool**. Не буду углубляться в эти детали, здесь просто достаточно запомнить этот момент. Но тогда спрашивается, а **как отличать тип **bool** от типа **int****? Если нужна строгая проверка на типы, то лучше использовать знакомую нам функцию **type()** с проверкой на равенство:

```
type(b) == bool
```

или, используя оператор **is**:

```
type(b) is bool
```

Эта функция различает булевы типы от целочисленных:

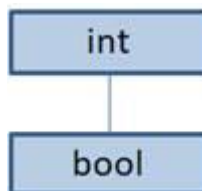
```
type(b) is int
```

Или, если мы хотим произвести множественную проверку, то воспользоваться оператором **in**:

```
type(b) in (bool, float, str)
```

Здесь мы сразу проверяем переменную **b** на три типа данных.

Сейчас у вас может возникнуть вопрос, **зачем нужна функция **isinstance()**, если проверку типов можно делать функцией **type()****? В целом, они действительно очень похожи, но **isinstance()** в отличие от **type()** делает проверку с учетом иерархии наследования объектов и была разработана для проверки принадлежности объекта тому или иному классу:



Например, тип **bool** наследуется от **int**, поэтому **isinstance()** выдает **True** для обоих типов, когда **b** – булева переменная. А функция **type()** даст **True** только для типа **bool**. То есть, здесь проверка происходит без учета иерархии. (Я

Основы Python

сейчас не буду углубляться и объяснять, что такое наследование – это уже раздел ООП и предмет отдельного курса).

Теперь, когда мы в целом познакомились с работой функции `isinstance()`, давайте рассмотрим следующий пример. Предположим, что у нас есть кортеж с произвольными данными:

```
data = (4.5, 8.7, True, "книга", 8, 10, -11, [True, False])
```

И наша задача вычислить сумму всех вещественных чисел этой коллекции. Для этого, очевидно, вначале каждый элемент кортежа нужно проверить на вещественный тип данных, а затем, выполнять суммирование. Я вначале сделаю это через обычный цикл `for`:

```
s = 0
for x in data:
    if isinstance(x, float):
        s += x

print(s)
```

Как видите, у нас получилась нужная сумма. Но эту же задачу можно реализовать лучше, с использованием ранее рассмотренной функции `filter()`:

```
s = sum(filter(lambda x: isinstance(x, float), data))
```

Видите, все записано в одну строчку и, кроме того, работает быстрее, так как используются встроенные функции вместо цикла `for`.

А вот если мы попробуем вычислить сумму целочисленных значений, просто изменив тип данных:

```
s = sum(filter(lambda x: isinstance(x, int), data))
```

Основы Python

то увидим неверное значение **8**, так как в коллекции **data** присутствует булево значение **True**, которое интерпретируется как целое число **1**. Здесь лучше применять строгую проверку с использованием функции **type()**:

```
s = sum(filter(lambda x: type(x) is int, data))
```

Теперь видим верное значение **7**.

С помощью функции **isinstance()** можно делать и множественные проверки. Например, мы хотим определить, **относится ли число к целому или вещественному типу данных?** Для этого достаточно записать кортеж из этих типов:

```
a = 5.5  
isinstance(a, (int, float))
```

Это эквивалентно записи:

```
isinstance(a, int) or isinstance(a, float)
```

Но первый вариант короче и потому чаще используется на практике.

Вот так в **Python** можно выполнять проверку типов данных для произвольных объектов. А также знаете отличия между работой функций **isinstance()** и **type()**. До встречи на следующем уроке.

§62. Функции **all** и **any**. Примеры их использования

На этом занятии мы поговорим о функциях **all()** и **any()**. Первым делом посмотрим, **что они делают и зачем нужны?** Начнем с функции **all()** и очень простого примера. Предположим, что есть список булевых значений:

```
a = [True, True, True, True]
```

И мы хотим узнать, **принимают ли все они значение **True**?** Для этого, как раз и используется функция **all()**, которая на вход принимает итерируемый объект и все его значения приводит к булевым величинам:

Основы Python

```
all(a)
```

Если все значения равны **True**, то на выходе получаем **True**. Если же, хотя бы одно значение принимает **False**, то на выходе будет **False**:

```
all([True, True, False, True])
```

Но, как вы понимаете, значения в списках могут быть любыми, например, такими:

```
a = [0, 1, 2.5, "", "python", [], [1, 2], {}]
```

И к нему тоже можно применить функцию **all()**:

```
all(a)
```

Как она работает? Помните, когда мы говорили о типе **bool**, то говорили, **что любое значение можно привести к этому типу?** В частности:

```
bool(0)
```

даст **False**, так как **0** воспринимается, как пустое значение. По тем же причинам **False** получается и для:

```
bool("")
```

```
bool([])
```

```
bool({})
```

А вот все, что не пустое, превращается в **True**:

```
bool(1)
```

```
bool("python")
```

Именно так вначале происходит преобразование списка, а затем, применяется функция **all()**. И, так как список содержит значения **False**, то и результат равен **False**.

Основы Python

Работу этой функции можно повторить с помощью обычных булевых операций. Если вначале объявить некую переменную со значением **True**:

```
all_res = True
for x in a:
    all_res = all_res and bool(x)

print(all_res)
```

А, затем, в цикле для каждого элемента выполнять оператор **and**, то эта переменная сохранит начальное значение **True** только в том случае, если не встретится ни один **False**. Я вам показал этот способ, чтобы вы знали как действовать в других языках программирования, где нет уже готовой функции **all()**. В **Python**, конечно, следует использовать эту функцию – это и удобнее и быстрее.

Вторая функция **any()** работает похожим образом, но возвращает **True**, если встретилось хотя бы одно такое значение. Например, для списка:

```
any(a)
```

мы увидим значение **True**. А вот если передать список со всеми **False**:

```
any([False, False, False, False])
```

то только в этом случае она вернет **False**.

Повторим также работу этой функции через булевы операции. Начальное значение переменной здесь нужно присвоить **False**:

```
any_res = False
for x in a:
    any_res = any_res or bool(x)

print(any_res)
```

Основы Python

А, затем, в цикле с помощью оператора **or** (или) корректировать это значение текущей булевой величиной. Если встретится хотя бы один **True**, то оператор **or** вернет **True** и оно сохранится в переменной **any_res**, что бы в дальнейшем не появлялось.

Вот принцип работы этих двух функций. Ну и, наверное, самый главный вопрос, **где это имеет смысл применять?** Давайте представим, что мы делаем игру «Крестики-нолики» и хотели бы на каждом шаге определять, **есть ли выигрышная позиция, например, у крестиков?** Для простоты сделаем это следующим образом. Все поле из девяти клеток я представлю одномерным списком (сейчас вы узнаете, зачем):

```
P = ['x', 'x', 'o', 'o', 'x', 'o', 'x', 'x', 'x']
```

x	x	o
o	x	o
x	x	x

Тогда, чтобы проверить выигрышные ситуации по строкам, можно воспользоваться функцией **all()**, следующим образом:

```
row_1 = all(map(lambda x: x == 'x', P[:3]))
row_2 = all(map(lambda x: x == 'x', P[3:6]))
row_3 = all(map(lambda x: x == 'x', P[6:]))

print(row_1, row_2, row_3)
```

Смотрите, мы здесь используем вложенный вызов функции **map()**, чтобы правильно преобразовать крестики в **True**, а нолики – в **False**, иначе бы все преобразовалось в **True**. Далее, срез для каждой строки на выходе функции **map()** обрабатывается функцией **all()** и получаем результат: **True** – есть выигрышная комбинация; **False** – нет выигрышной комбинации.

Основы Python

Кстати, внимательный и грамотный зритель здесь сразу должен возмутиться из-за дублирования кода – три раза записана одна и та же анонимная функция. Действительно, это лучше поправить и использовать свою собственную:

```
def true_x(a):  
    return a == 'x'
```

И, затем, указать ее в функции `map()`:

```
row_1 = all(map(true_x, P[:3]))  
row_2 = all(map(true_x, P[3:6]))  
row_3 = all(map(true_x, P[6:]))
```

Теперь известный принцип **DRY** – сохраняется. По аналогии можно сделать проверку выигрыша по столбцам:

```
col_1 = all(map(true_x, P[:,3]))  
col_2 = all(map(true_x, P[1::3]))  
col_3 = all(map(true_x, P[2::3]))  
  
print(col_1, col_2, col_3)
```

А по диагоналям сделайте самостоятельно – это несложно.

Для второй функции я приведу такой короткий пример. Предположим, мы делаем игру «Сапер» и игровое поле также представлено в виде одномерного списка длиной $N \times N$ элементов:

```
N = 10  
P = [0] * (N*N)
```

Далее, если в этом списке появляется хотя бы одна мина (отметим ее звездочкой):

```
P[4] = '*'
```

Основы Python

то игрок проигрывает. Чтобы узнать, наступил ли игрок на мину, удобно воспользоваться функцией `any()`:

```
loss = any(map(lambda x: x == '*', P))
print(loss)
```

После запуска программы увидим значение **True**, то есть, игрок проиграл. А если мину поставить в комментарий и снова запустить, то увидим значение **False**.

Вот два простых примера, где удобно применять эти две функции `any()` и `all()`.

§63. Расширенное представление чисел. Системы счисления

На этом занятии мы поговорим о различных форматах представления чисел в **Python**. Мы с вами неоднократно записывали в программе целые и вещественные значения в виде:

```
a = 500
b = 0.01
```

Это классическая запись. Еще существует экспоненциальная, когда числа записываются через степень десятки. Например, число:

$$500 = 5e2$$

Здесь **e2** – это **10** в квадрате, то есть, **100** и оно умножается на **5**, получаем **500**. Причем, обратите внимание, число **500** представляется как вещественное, а не целое.

Основы Python

Аналогичным образом можно задать и второе значение **0,01** как:

$$0.01 = 1e-2$$

Здесь **e-2** – это **10** в минус второй степени, то есть, **0,01** и все умножается на **1**. Причем, единичка вначале строго обязательна – это формат записи чисел: сначала идет число, которое умножается на степень десятки.

Зачем это может понадобиться? Часто в инженерных задачах оперируют очень большими или очень малыми числами. Например, мы хотим указать шаг сходимости градиентного алгоритма как **10⁻⁸**. Записывать это число в классическом виде:

$$0,00000001$$

не очень удобно, да и легко ошибиться в количестве нулей. А вот в экспоненциальной форме все очень удобно и наглядно:

$$1e-8$$

И то же самое с очень большими числами, например, число Авогадро (примерно):

$$6,02 * 10^{23} = 6.02e23$$

Прописывать такое число в классическом виде было бы совсем уж неудобно. Причем, вначале указано вещественное число **6,02**, а затем, степень десятки. Так тоже можно делать.

Думаю, вы теперь знаете и сможете применять в своих программах экспоненциальную запись для чисел. Далее, мы с вами посмотрим, как можно представлять числа не только в десятичном формате, как это делали до сих пор, но и в других системах счисления: **двоичной**, **шестнадцатеричной**, **восьмеричной**. На самом деле их бесконечно много, так как можно придумать любую свою систему, скажем **132**-ричную или еще какую, но на практике в **99%** случаях ограничиваются именно такими.

Основы Python

Вкратце я напомним, что в десятичной системе счисления все цифры умножаются на степени десятков, в зависимости от позиции их написания. Например:

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

При этом, нам достаточно иметь десять различных цифр для представления любого числа.

Если же переходим к двоичной системе счисления, что все числа кодируются двумя символами: **0** и **1** – это все цифры двоичной системы (они называются битами). В результате, получаем, например, такие записи чисел в этой системе счисления:

$$001 = 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1$$

$$1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$$

$$10001101 = 1 \cdot 2^7 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 = 141$$

Я здесь сразу привел их перевод в десятичную систему счисления, учитывая, что все позиции цифр – это соответствующие степени двойки. Так вот, в **Python** мы можем записывать числа непосредственно в двоичной системе, следующим образом:

```
a = 0b001
b = 0b1101
c = 0b10001101
```

Причем, все числа получаются целыми. Если нужно определить отрицательное значение в двоичной записи, то проще всего перед его определением поставить унарный минус:

```
d = -0b1111
```

Конечно, первая реакция человека, впервые столкнувшегося с двоичным представлением чисел, **зачем это нужно?** Куда привычнее использовать десятичную запись. Да, и часто именно ее применяют. Но бывают ситуации,

Основы Python

например, работа с числами на уровне **бит**, когда нам требуется включить или выключить отдельный бит числа, или проверить является ли текущий бит равным **1** (то есть включенным) или **0** (выключенным). Здесь двоичное представление нам помогает визуализировать такой процесс.

Следующая часто используемая система счисления – шестнадцатеричная. Здесь уже используется **16** различных обозначений. Для этого берут десять цифр из десятичной системы и еще первые шесть букв латинского алфавита:

0-9, A, B, C, D, E, F

Числа принимают уже такой вид:

1A, FB, C2DE

и т.п. Для перевода их в привычную нам десятичную систему, используется тот же принцип. Распишем их в виде:

$$1A = 1 \cdot 16^1 + A \cdot 16^0 = 26$$

$$FB = F \cdot 16^1 + B \cdot 16^0 = 240 + 11 = 251$$

$$C2DE = C \cdot 16^3 + 2 \cdot 16^2 + D \cdot 16^1 + E \cdot 16^0 = 49886$$

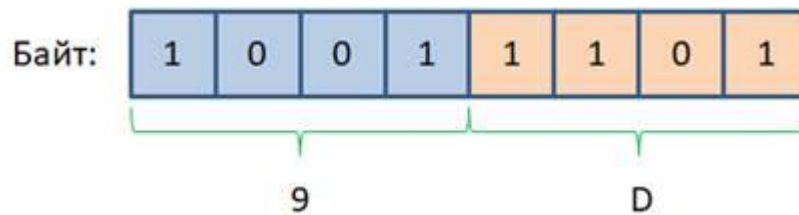
Здесь буква **A** соответствует значению **10** в десятичной системе, поэтому получаем значение **26**. Буква **F** = **15**, **B** = **11**, поэтому имеем значение **251**. А последнее число расшифруйте самостоятельно, должно получиться значение **49886**.

В чем ценность шестнадцатеричной системы счисления? Она получила широкое распространение благодаря удобному представлению байтовых данных. Один **байт** – это восемь **бит**. Если разделить байт на две части, то получим по четыре бита. В эти четыре бита можно записать одно из шестнадцати значений, то есть, одно из значений шестнадцатеричной системы счисления:

0 – F

Основы Python

Получается, что два символа шестнадцатеричной системы, как раз описывают один байт. Причем, делают это независимо друг от друга.



При определенной тренировке, глядя на шестнадцатеричную запись, мы легко можем ее перевести и в десятичную и в двоичную системы.

В **Python** можно записывать числа сразу в шестнадцатеричной системе, например:

```
a = 0x1A
```

Причем, можно использовать не только заглавные, но и малые латинские буквы:

```
b = 0xde
```

```
c = 0xaa3f
```

Везде получаем целые положительные значения. Для определения отрицательных чисел можно вначале указать унарный минус:

```
d = -0x342
```

Последняя восьмеричная система, которую мы рассмотрим, работает по аналогии с предыдущими, только используется основание для восьмерки. Соответственно, для определения чисел достаточно восемь различных обозначений – используют первые восемь цифр десятичной системы. Сами числа записываются в виде:

$$27 = 2 \cdot 8^1 + 7 \cdot 8^0 = 23$$

$$54 = 5 \cdot 8^1 + 4 \cdot 8^0 = 44$$

$$775 = 7 \cdot 8^2 + 7 \cdot 8^1 + 5 \cdot 8^0 = 509$$

Основы Python

Здесь я сразу перевел их в десятичный вид. Если нам нужно определить число в восьмеричной системе, то в **Python** используется следующая запись:

```
a = 0o27
b = 0o54
c = -0o775
```

Вот такие базовые системы счисления можно использовать в **Python** для определения чисел, а также применять экспоненциальную форму их представления.

§64. Битовые операции И, ИЛИ, НЕ, XOR. Сдвиговые операторы

На этом занятии речь пойдет о битовых операциях над целыми числами. Начнем с самой простой битовой операции **НЕ**, которая выполняет инверсию бит в соответствии с такой табличкой:

x	НЕ
0	1
1	0

Например, возьмем целое число:

```
a = 121
```

Чтобы посмотреть его битовое (**двоичное**) представление, можно воспользоваться функцией **bin()**:

```
bin(a)
```

На выходе будет строка с битами этого десятичного числа. А теперь выполним инверсию его бит. Для этого в **Python** используется оператор **~** (**тильда**), которая записывается перед числом или переменной:

```
b = ~a
```

Основы Python

В результате переменная **b** стала отрицательной и принимает значение -122. **Почему значение стало отрицательным и уменьшилось на -1?** Смотрите, любое число кодируется набором бит. В Python – это довольно длинная последовательность:

00000000 00000000 ... 00000000

Самый старший бит отвечает за знак числа: 0 – положительное; 1 – отрицательное. Так вот, если взять десятичное число 0:

```
d = 0
```

и инвертировать все его биты:

```
~d
```

то мы получим число, состоящее из всех битовых единиц, а это есть не что иное, как десятичное значение -1. Мы здесь наглядно видим, как операция отрицания превращает число 0 в отрицательное и уменьшает его на -1. Так происходит со всеми целыми положительными числами:

```
d = 10
```

```
~d
```

И с отрицательными:

```
d = -10
```

```
~d
```

За счет инвертирования всех бит числа.

Битовая операция И

Следующая битовая операция **И** применяется уже к двум операндам (**переменным** или **числам**) со следующей таблицей истинности:

x1	x2	И
----	----	---

Основы Python

0	0	0
0	1	0
1	0	0
1	1	1

и реализуется через оператор **&** (амперсанд). Например, представим, что у нас есть два числа (две переменные):

```
flags = 5  
mask = 4
```

Выполним для них битовую операцию **И**:

```
res = flags & mask
```

Переменная **res** ссылается на значение **4**. Давайте посмотрим, почему получился такой результат. В соответствии с таблицей истинности результирующие биты будут следующими (перечислить). В итоге, переменная **res** получается равной **4**.

flags	0	0	0	0	0	1	0	1
mask	0	0	0	0	0	1	0	0
res	0	0	0	0	0	1	0	0

Ну хорошо, разобрались, но **зачем все это надо?** Смотрите, если нам нужно проверить включен ли какой-либо бит числа (то есть установлен в **1**), то мы можем это сделать с помощью битовой операции **И**, следующим образом:

```
if flags & mask == mask:  
    print("Включен 2-й бит числа")  
else:  
    print("2-й бит выключен")
```

То есть, из-за использования операции **И** мы в переменной **flags** обнуляем все биты, кроме тех, что совпадают с включенными битами в переменной **mask**.

Основы Python

Так как **mask** содержит только один включенный бит – второй, то именно проверку на включенность этого бита мы и делаем в данном случае.

Чтобы убедиться, что все работает, присвоим переменной **flags** значение 1:

```
flags = 1
```

и запустим программу. Теперь, видим сообщение, что **2-й бит выключен**, что, в общем то, верно. Вот так, с помощью битовой операции **И** можно проверять включены ли определенные биты в переменных.

Также битовую операцию **И** используют для выключения определенных битов числа. Делается это, следующим образом:

```
flags = 13  
mask = 5  
flags = flags & ~mask
```

Что происходит здесь? Сначала выполняется инверсия бит маски, так как операция **НЕ** имеет более высокий приоритет, чем операция **И**. Получаем, что маска состоит из вот таких бит. Затем, идет операция поразрядное **И**, и там где в маске стоят 1 биты переменной **flags** не меняются, остаются прежними, а там где стоят в маске 0 – эти биты в переменной **flags** тоже становятся равными 0. За счет этого происходит выключение 2-го и 0-го битов переменной **flags**.

Кстати, последнюю строчку можно переписать и короче:

```
flags &= ~mask
```

Это будет одно и то же.

Битовая операция ИЛИ

Следующая битовая операция **ИЛИ** определяется оператором **|** и ее таблица истинности выглядит следующим образом:

Основы Python

x1	x2	ИЛИ
0	0	0
0	1	1
1	0	1
1	1	1

Обычно ее применяют, когда нужно включить отдельные биты переменной. Рассмотрим такую программу:

```
flags = 8
mask = 5
flags = flags | mask
```

Операция поразрядное **ИЛИ**, как бы собирает все единички из обеих переменных и получается такое своеобразное сложение (**объединение**).

flags	0	0	0	0	1	0	0	0
mask	0	0	0	0	0	1	0	0
flags	0	0	0	0	1	1	0	1

Кстати, в этом случае действительно получилось **8+5=13**. Но это будет не всегда так, например, если:

```
flags = 9
flags |= mask
```

то результат тоже будет **13**, так как операция **ИЛИ** включает бит вне зависимости был ли он уже включен или нет, все равно на выходе будет единица.

Битовая операция исключающее ИЛИ (XOR)

И последняя базовая операция работы с битами – исключающее ИЛИ (ее еще называют XOR). Она задается оператором **^** и имеет такую таблицу истинности:

Основы Python

x1	x2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Из нее видно, что данная операция позволяет переключать биты числа, то есть, если они были равны **0**, то станут **1** и, наоборот, если были **1** – станут **0**. Продемонстрируем это на таком примере:

```
flags = 9
mask = 1
flags = flags ^ mask
```

Здесь сначала получим значение **8**, так как **0**-й бит будет выключен.

flags	0	0	0	0	1	0	0	1
mask	0	0	0	0	0	0	0	1
flags	0	0	0	0	1	0	0	0

Но если повторить последнюю строчку:

```
flags ^= mask
```

то нулевой бит снова включится и переменная **flags** примет прежнее значение **9**. Вот так, с помощью операции **XOR** можно переключать отдельные биты переменных.

Интересной особенностью операции **XOR** является отсутствие потерь данных при ее работе. **Что это значит?** Смотрите, какую бы маску мы не взяли, дважды примененная маска дает исходное число:

```
flags = 9
```

Основы Python

```
mask = 111
flags ^= mask
flags ^= mask
```

Этот эффект часто используют для шифрования информации. Например, так работает пароль архиватора **zip**. В нем пароль – это маска, которая накладывается на заархивированные данные. Чтобы восстановить их, необходимо ввести пароль и повторить наложение этой маски.

Вот мы с вами рассмотрели основные битовые операции, и обратите внимание, что все их можно записывать в таком виде:

Полная форма	Краткая форма	Приоритет
$a = a \& b$	$a \&= b$	2 (И)
$a = a b$	$a = b$	1 (ИЛИ)
$a = a \wedge b$	$a \wedge= b$	1 (XOR)
$\sim a$		3 (НЕ)

Операторы сдвига бит

И в заключение занятия рассмотрим еще два распространенных битовых оператора:

>> сдвиг бит вправо
<< сдвиг бит влево

Предположим, у нас имеется переменная:

```
x = 160
```

Ее битовое представление:

```
bin(x) # 10100000
```

Давайте сдвинем все эти биты вправо на один бит. Сделать это можно, следующим образом:

Основы Python

```
x = x >> 1
```

Теперь битовое представление числа:

```
bin(x) # 1010000
```

А значение равно **80**. То есть, сдвигая биты вправо на один бит, мы разделили число на **2**. Если сдвинуть на два бита вправо:

```
x = x >> 2
```

то это уже эквивалентно делению на **4**. Давайте еще раз сдвинем на бит вправо:

```
x = x >> 1
```

будет пять, но **5** не делится на **2** нацело. **Что же получится при смещении бит вправо?** Смотрим:

```
x = x >> 1
```

Получаем значение **1**. То есть, здесь, как бы дробная часть была отброшена. Действительно, один из битов был просто потерян, это и объясняет данный эффект.

По аналогии можно делать смещения влево:

```
x = x << 1
```

Это уже будет эквивалентно умножению на два. Или, так:

```
x = x << 3
```

Это эквивалентно умножению на **8** (два в кубе). И так далее. Операции смещения бит влево и вправо выполняют целочисленное умножение и деление кратное двум. Причем эти операции умножения и деления работают значительно быстрее, чем традиционные арифметические операции

Основы Python

умножения и деления. Поэтому, разработчики различных алгоритмов для маломощных компьютеров стараются составить вычисления так, чтобы они базировались на сдвиговых операциях, исключая прямое умножение и деление.

Надеюсь, из этого занятия вы поняли, как работают битовые операции **И**, **ИЛИ**, **НЕ**, **XOR**, а также **битовые операции сдвигов**.

§65. Модуль **random** стандартной библиотеки

На этом занятии речь пойдет о модуле **random** стандартной библиотеки. Еще раз отмечу, что в **Python** имеется множество предустановленных модулей, которые поставляются вместе с интерпретатором языка. Подробно обо всех их можно почитать на странице официальной документации:

<https://docs.python.org/3/library/index.html>

Здесь много разных модулей и, программируя на **Python**, вы так или иначе будете с ними знакомиться. Среди них есть и модуль **random**. Откроем страницу с его описанием и видим здесь множество различных функций для генерации случайных значений (а точнее, псевдослучайных, то есть, их, все же можно просчитать). Я расскажу о некоторых из функций этого модуля, которые наиболее часто используются на практике.

Вначале, для использования этого модуля, его нужно импортировать. Как это делать мы с вами уже подробно говорили. В самом простом случае, можно записать:

```
import random
```

И, далее, из пространства имен **random** можно выбирать самые разные функции этого модуля. Вначале я воспользуюсь функцией **random()**:

```
a = random.random()
```

Основы Python

```
print(a)
```

Она при каждом вызове выдает случайное значение в диапазоне $[0.0; 1.0)$. Обратите внимание, что числа подчиняются равномерному закону распределения, то есть, в диапазоне $[0.0; 1.0)$ они могут принимать любое значение с равной вероятностью.



Другая похожая функция **uniform(a, b)** также генерирует случайные значения по равномерному закону, но в диапазоне от **a** до **b**:

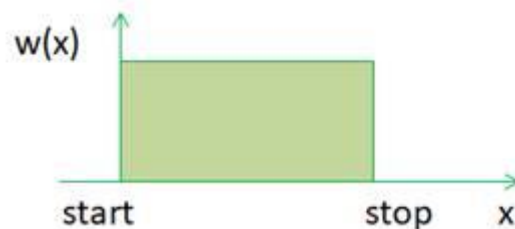
```
a = random.uniform(1, 5)
```

Если нам нужно моделировать целочисленные случайные значения с тем же равномерным распределением, то можно использовать или функцию:

```
a = random.randint(-3, 7)  # [-3; 7]
```

или функцию **randrange**:

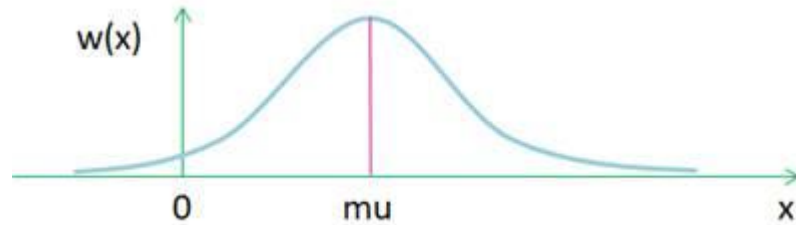
```
a = random.randrange(5)  
b = random.randrange(-3, 10, 2)
```



Однако часто на практике требуются случайные величины с другим законом распределения – **гауссовским** (или, еще говорят, **нормальным**). Закон

Основы Python

распределения имеет следующий вид и определяется двумя параметрами: математическим ожиданием (**средним значением**) и среднеквадратическим отклонением (**мерой разброса относительно МО**).



Особенность этих величин в том, что они чаще появляются в области **МО** и реже на более дальних расстояниях. Ценность этого распределения в том, что ему часто подчиняются многие реальные события: **колебания цен на нефть**, и **различных товаров**, **шумы при радиопередачах**, **погрешности измерений различных значений** и т.п. Это распределение окружает нас повсюду! И, конечно же, нужно уметь его моделировать. Для этого в **Python** существует функция **gauss(mu, sigma)**:

```
a = random.gauss(0, 3.5)
```

Во второй части этого занятия я расскажу о функциях модуля **random** при работе с последовательностями. Предположим, у нас имеется список:

```
lst = [4, 5, 0, -1, 10, 76, 3]
```

и мы хотим выбрать из него один элемент случайным образом. Для этого можно воспользоваться функцией:

```
a = random.choice(lst)
print(a)
```

После запуска видим случайно выбранное значение из списка **lst**.

Следующая функция **shuffle()** перемешивает элементы списка случайным образом:

Основы Python

```
random.shuffle(lst)
print(lst)
```

Причем, меняет сам список, поэтому работает только с изменяемыми коллекциями.

Третья функция **sample()** возвращает новый список с указанным числом неповторяющихся элементов, выбранных случайным образом из списка:

```
a = random.sample(lst, 3)
print(a)
```

Разумеется, максимальное число элементов не может превышать число элементов в списке **lst**.

И последнее, что я хочу отметить в работе с псевдослучайными величинами, это возможность формировать одинаковые случайные последовательности чисел при каждом новом запуске программы. **О чем здесь речь?** Смотрите, если мы, например, генерируем несколько целочисленных случайных величин:

```
a = [random.randint(0, 10) for i in range(20)]
print(a)
```

То, запуская программу снова и снова, будем видеть разные случайные значения в списке **a**. Часто именно так и должен работать генератор случайных величин. Но иногда требуется, чтобы каждый запуск программы приводил к одной и той же последовательности случайных чисел. Например, для повторяемости результатов экспериментов на разных компьютерах. Для этого нужно зафиксировать, так называемое, зерно генератора случайных чисел с помощью функции **seed()**, следующим образом:

```
random.seed(123)
```


Основы Python

Здесь число **123** – это, как раз и есть значение зерна. Оно задает начальное состояние датчика чисел, поэтому, запуская программу, каждый раз будем видеть одни и те же случайные числа в списке **a**.

Если выбрать другое значение зерна, то изменится и последовательность **СВ**. Это бывает очень удобно, не сохраняя последовательность случайных чисел, указывать только значение зерна для их генератора.

Вот такие возможности существуют в библиотеке **random**, которые следует знать и, по мере необходимости, применять на практике.

§66. Аннотация базовыми типами

На нескольких последующих занятиях мы поговорим об **аннотации типов**. **Что это такое и зачем это нужно?** Я напомним, что в **Python** тип переменной (**ссылки**) определяется типом объекта, на который она ссылается. То есть, динамически. Отсюда и пошло название **динамическая типизация**.

Например:

```
cnt = 0          # тип int
msg = "hello"    # тип str
lst = [1, 2, 3]  # тип list
```

Мало того, мы можем запросто присваивать другие типы данных этим же переменным:

```
cnt = -5.3       # тип float
msg = 0          # тип int
```

и никаких ошибок не будет. Все благодаря тому, что переменные – это ссылки на другие объекты, сами по себе они не хранят присваиваемые данные. Но **что если в программе мы хотим явно указать, что, например, переменная cnt работает исключительно с целыми значениями?** С типом **int**. **Как это сделать?** Очень просто. Если мы после переменной **cnt** поставим двоеточие, а затем укажем ожидаемый тип данных, на который она должна ссылаться по программе, например, **int**:

Основы Python

```
cnt: int  
cnt = 0
```

то в строчке программы, где происходит присвоение не целого значения:

```
cnt = -5.3      # тип float
```

интегрированная среда подсветит число **-5.3**. Это и есть пример **аннотации типов** и работы **статического анализатора**, который проверяет на соответствие указанным типам. При наведении курсора на подсветку увидим сообщение:

Expected type 'int', got 'float' instead

Но обратите внимание, программа при этом выполнится без ошибок. То есть, аннотация типов носит информационный характер, на ход выполнения программы она никак не влияет.

Так все же, для чего была придумана аннотация типов и с какой версии языка Python ее можно использовать? Сначала отвечу на вторую часть вопроса.

Активно аннотации стали внедряться, начиная с версии **Python 3.5**, и продолжают совершенствоваться до сих пор (на момент записи лекции – это версия **Python 3.10**). А теперь **каким целям она служит?**

- Для удобства восприятия стороннего кода.
- Для удобства редактирования кода, когда IDE «подсказывает» атрибуты указанного типа переменных.
- Для отслеживания некоторых явных ошибок на уровне несоответствия типов.

Вообще, тема аннотаций гораздо объемнее, чем может показаться на первый взгляд. Поэтому мы затронем лишь базовые моменты, которые используются примерно в **90%** случаях. Ну а за деталями, как всегда, можно обратиться к официальной документации:

<https://docs.python.org/3/library/typing.html>

Основы Python

Вернемся к строчкам аннотирования переменной `cnt`. Здесь можно сразу выполнять инициализацию после указания желаемого типа:

```
cnt: int = 0
```

Если присвоить другой тип данных:

```
cnt: int = 0.5
```

то значение **0.5** будет подсвечено, но программа, по-прежнему, выполнится без ошибок. И так мы можем делать с любыми переменными и любыми типами данных, включая и свои собственные (например, **классы**).

Аннотация типов в функциях

Конечно, для отдельных переменных программы аннотация применяется не часто. Обычно ее можно встретить при объявлении функций. Давайте посмотрим, как это делается.

Предположим, мы определяем функцию для умножения некоторого числа на два, следующим образом:

```
def mul2(x):  
    return x * 2
```

И, затем, разумеется, можем вызвать эту функцию:

```
res = mul2(5)  
print(res)
```

Но здесь в качестве аргумента можно передать не только число, а, например, список:

```
res = mul2([5])
```

или строку:

Основы Python

```
res = mul2("5")
```

При этом программа никаких ошибок выдавать не будет, но вместо умножения числа на два, увидим размножение элементов списка или символов строк. Если предполагается в качестве аргументов этой функции передавать только числа, то как раз это можно описать с помощью аннотации параметров функции, например, так:

```
def mul2(x: int):  
    return x * 2
```

Смотрите, теперь в вызове:

```
res = mul2("5")
```

интегрированная среда нам подсвечивает аргумент "5". Конечно, программа по-прежнему выполнится без ошибок, но теперь программист видит, что с аргументом функции что то не так. Если же записать целое число:

```
res = mul2(5)
```

то подсветка пропадет, т.к. типы совпадают.

Еще один плюс такого аннотирования – это подсказки интегрированной среды в виде набора атрибутов, присущих тому или иному типу. Например, если оставить только объявление функции без аннотирования:

```
def mul2(x):  
    return x * 2
```

То, записывая внутри функции «x.», мы увидим лишь общие атрибуты. Если же добавить аннотацию:

```
def mul2(x: float):  
    return x * 2
```

Основы Python

то появятся атрибуты для типа **float**. Это дополнительное удобство, который дает анализатор IDE.

Кстати, если нужно посмотреть, какие параметры функции как аннотированы, то можно обратиться к магической коллекции `__annotations__`, которая теперь есть у каждой функции:

```
print(mul2.__annotations__)
```

Увидим в консоли следующий словарь:

```
{'x': <class 'float'>}
```

Если в функцию **mul2** добавить еще один параметр, например, так:

```
def mul2(x: float, y):  
    return x * y
```

то словарь `__annotations__` не изменится, так как второй параметр не аннотирован. Давайте добавим аннотацию и для него:

```
def mul2(x: float, y: int):  
    return x * y
```

Теперь видим два ключа и два значения:

```
{'x': <class 'float'>, 'y': <class 'int'>}
```

Причем аннотации не мешают нам определять параметры со значениями по умолчанию. Мы по-прежнему можем использовать запись вида:

```
def mul2(x: float, y: int = 2):  
    return x * y
```

Тогда при передаче только одного первого аргумента, значение `y` будет принимать число **2**, прописанное по умолчанию.

Основы Python

Последнее, что осталось – это определить аннотацию для возвращаемого значения. Для этого после круглой скобки ставится стрелочка `->` и указывается возвращаемый тип:

```
def mul2(x: float, y: int) -> float:  
    return x * y
```

В результате в коллекции `__annotations__` появился ключ `'return'` со значением `float`:

```
{'x': <class 'float'>, 'y': <class 'int'>, 'return': <class 'float'>}
```

Если же функция ничего не возвращает (на самом деле она будет возвращать `None`), то указывается значение `None`:

```
def show_x(x: float) -> None:  
    print(f"x = {x}")
```

Вот так в самом простом варианте с использованием базовых типов выполняется аннотирование либо для отдельных переменных, либо для параметров функции и возвращаемого значения.

Модуль `typing` и типы `Union`, `Optional`, `Any`, `Final`

Если внимательно посмотреть на нашу функцию `mul2()`, то по идее вполне допустимо, чтобы параметры `x` и `y` принимали как тип `int`, так и тип `float`. **Как сделать такую составную, более сложную аннотацию типов?** Для этого можно воспользоваться специальными типами из модуля `typing`:

```
from typing import Union, Optional, Any, Final
```

Давайте рассмотрим их по порядку. Первый тип `Union` позволяет комбинировать несколько разных типов в один, образуя **составной тип**. Например, это можно сделать так:

```
def mul2(x: Union[int, float], y: Union[int, float]) -> Union[int, float]:  
    return x * y
```

Основы Python

То есть мы в квадратных скобках прописываем через запятую те типы, которые ожидаем увидеть у параметров **x**, **y**, а также у возвращаемого значения функции.

Сразу отмечу, что с версии **Python 3.10** такое объединение типов в аннотациях можно записывать с новым синтаксисом:

```
def mul2(x: int | float, y: int | float) -> Union[int, float]:  
    return x * y
```

Но вернемся к прежней записи и здесь:

```
Union[int, float]
```

фактически, определяет новый объект – новый тип данных. А раз это так, то мы можем сформировать переменную на этот объект следующим образом:

```
Digit = Union[int, float]
```

и, затем использовать при типизации:

```
def mul2(x: Digit, y: Digit) -> Digit:  
    return x * y
```

В ряде случаев, такой текст программы становится более читабельным. Хотя, усердствовать в этом тоже не стоит, т.к. программисту придется искать определение **Digit** и понимать, что оно означает.

Следующий тип **Optional** позволяет указать один какой-либо тип данных и еще автоматически добавляется тип **None**. Например:

```
Str = Optional[str]
```

эквивалентно

```
StrType = Union[str, None]
```

Основы Python

Этот случай выделили отдельно в тип **Optional**, т.к. такая комбинация с **None** довольно часто используется на практике. Например, объявим функцию **show_x**, у которой второй параметр **descr** по умолчанию будет принимать значение **None**:

```
def show_x(x: float, descr: Optional[str] = None) -> None:
    if descr:
        print(f'{descr} {x}')
    else:
        print(f'x = {x}')
```

Если теперь при вызове функции **show_x()** указать один аргумент:

```
show_x(55.6768)
```

то отработает блок **else**, а если указать два аргумента:

```
show_x(55.6768, 'x:')
```

то отработает первый блок по условию. Если же во втором аргументе указать не строку, а число:

```
show_x(55.6768, 10)
```

то интегрированная среда подсветит его из-за несоответствия ожидаемого типа данных. То есть, **Optional** введен для удобства, чтобы в **Union** не писать два типа, один из которых **None**.

Следующий тип **Any** означает буквально любой тип данных. Он используется, если параметр или переменная или возвращаемое значение функции может принимать любой тип. Или же, мы попросту не можем указать какой-либо конкретный тип данных, т.к. он нам неизвестен. Записывается такая нотация следующим образом:

```
def show_x(x: Any, descr: Optional[str] = None) -> None:
    if descr:
```


Основы Python

```
print(f'{descr} {x}')  
else:  
    print(f'x = {x}')
```

И, по сути, она эквивалентна поведению по умолчанию, когда мы не прописываем совсем никаких типов.

Наконец, последний тип **Final** появился в версии **Python 3.10** и служит для отметки констант в программе. Например, если прописать:

```
MAX_VALUE: Final = 1000
```

то при попытке позже поменять это значение:

```
MAX_VALUE = 2000
```

интегрированная среда нам подсветит эту переменную с указанием, что идет попытка изменить константную переменную. При этом программа отработает без ошибок.

На этом мы завершим первое занятие по аннотации типов. На следующем продолжим эту тему и посмотрим, как описывать типы для различных коллекций языка **Python**.

§67. Аннотации типов коллекций

На предыдущем занятии мы с вами увидели, как можно аннотировать отдельные переменные и функции простыми базовыми типами:

int, float, str, bool и т. п.

Давайте теперь посмотрим, как делать аннотацию для коллекций языка **Python**:

list, tuple, dict, set

Основы Python

Аннотация списков

Казалось бы, нет ничего проще. Достаточно также указать нужный нам тип, например, у переменной:

```
lst: list = [1, 2, 3]
```

и аннотация готова. И в целом да. В самом простом варианте мы так вполне можем сделать. Но обратите внимание, в этой аннотации нет никаких указаний на тип элементов списка. Поэтому можно прописать и числа, и строки:

```
lst_str: list = ['1', '2', '3']
```

и вообще любые типы данных. Интегрированная среда нам не выдаст никаких предупреждений. А вот если вместо списка записать, например, кортеж:

```
lst: list = (1, 2, 3)
```

то тогда уже увидим подсветку с предупреждением. То есть, тип `list` в аннотации отвечает только за определение списка. Типы элементов, при этом, могут быть любыми.

Но что если нам нужно дополнительно в аннотации указать и тип элементов списка? Начиная с версии **Python 3.9**, это можно сделать с помощью все того же базового типа `list` следующим образом:

```
lst: list[int] = [1, 2, 3]
```

Здесь в квадратных скобках указывается тип элементов списка.

Подразумевается, что все элементы списка имеют единый тип данных. Так чаще всего предполагается использование списков в языке **Python**.

Если указать другой тип, например, `str`:

```
lst: list[str] = [1, 2, 3]
```

Основы Python

Увидим подсветку кода. Но, как только в списке оказывается хотя бы один строковый элемент:

```
lst: list[str] = ['1', 2, 3]
```

подсветка пропадает. Вот это особенность работы данной аннотации в **PyCharm**. Но, забегаю вперед, скажу, что для языка **Python** есть довольно полезный модуль **mypy**, с помощью которого можно строже оценивать корректность программы для приведенной аннотации типов. И он находит все несоответствия. О модуле **mypy** речь пойдет в следующем занятии.

Вернемся к нашей нотации списка **list[str]**. Записывать ее в таком виде можно только, начиная с версии **Python 3.9** и выше. Если версия вашего интерпретатора ниже, то для достижения того же результата придется импортировать из модуля **typing** следующие типы:

List, Tuple, Dict, Set

```
from typing import List, Tuple, Dict, Set
```

а, затем, вместо **list** прописать **List**:

```
lst: List[str] = [1, 2, 3]
```

Эта конструкция работает абсолютно так же, как и типизация через **list**. Но **List** и другие подобные типы считаются устаревшими и в новых версиях лучше избегать такого способа обозначения.

Аннотация кортежей

С кортежами все в целом так же, как и со списками. Единственное ключевое отличие – это независимое указание типов для всех элементов кортежа. Например:

```
address: tuple[int, str] = (1, 'proporprogs.ru')
```

Основы Python

Возможно, здесь у вас возникает вопрос: **почему у списка мы указываем один тип для всех элементов, а у кортежа для каждого?** Как вы помните, кортеж – это неизменяемый тип данных. В частности, мы не можем добавлять или удалять элементы. Поэтому их число полагается известным и фиксированным. Часто кортежи в **Python** используются для группировки каких-либо данных в единый объект. Например, информация по книге может быть представлена в виде кортежа:

```
book: tuple[str, str, int]
book = ('Балакирев С.М.', 'Аннотация типов', 2022)
```

И мы наперед знаем какие данные и в каком порядке предполагается хранить. Поэтому прописать аннотацию для всех его элементов не представляет особого труда. И в дальнейшем она будет служить хорошей подсказкой для программиста.

Конечно, если вам понадобится аннотировать кортеж с произвольным числом элементов, то это можно сделать так:

```
elems: tuple[int, ...]
```

И тогда все следующие строчки:

```
elems = (1,)
elems = (1, 2)
elems = (1, 2, 3)
```

будут корректны для этой аннотации.

Я думаю, вы уже догадались, что для версий языка **Python** ниже **3.9** следует вместо **tuple** использовать тип **Tuple**, если мы хотим дополнительно указывать тип данных:

```
book: Tuple[str, str, int]
book = ('Балакирев С.М.', 'Аннотация типов', 2022)
```

Основы Python

Но с версии 3.9 лучше использовать стандартный тип **tuple** с квадратными скобками.

Аннотация словарей и множеств

Аналогичным образом выполняется аннотация словарей. Так как словари – изменяемый тип данных и число их элементов наперед не определено, то при аннотировании указываются лишь тип ключа и тип значения. Например:

```
words: dict[str, int] = {'one': 1, 'two': 2}
```

Или, для версий Python ниже 3.9:

```
words: Dict[str, int] = {'one': 1, 'two': 2}
```

То есть, эта аннотация ведет себя так же, как и при аннотировании списков.

И то же самое для множеств:

```
persons: set[str] = {'Сергей', 'Михаил', 'Наталья'}
```

или для Python ниже 3.9:

```
persons: Set[str] = {'Сергей', 'Михаил', 'Наталья'}
```

Здесь также указывается единый тип для всех элементов множества. Хотя подсветка будет появляться только тогда, когда все элементы множества или словаря не соответствуют указанному типу. Но модуль **mypy** находит все эти несоответствия.

Комбинирование типов

Разумеется, все приведенные способы типизации можно использовать не только для отдельных переменных, но и при объявлении функций. Например, так:

```
def get_positive(digits: list[int]) -> list[int]:
```

Основы Python

```
return list(filter(lambda x: x > 0, digits))
```

```
print(get_positive([1, -2, 3, 4, -5]))
```

И, смотрите, в аннотации указан один тип данных **int**, который ожидается увидеть у элементов списка. А что если мы собираемся хранить в списке и целые и вещественные числа. **Как тогда определить аннотацию?** Все очень просто. Для этого нам понадобится определить составной тип, используя **Union** из модуля **typing**:

```
def get_positive(digits: list[Union[int, float]]) -> list[Union[int, float]]:  
    return list(filter(lambda x: x > 0, digits))
```

То есть, внутри квадратных скобок можно прописывать любые типы, образуя вложенные конструкции. Я напомним, что начиная с версии **Python 3.10**, эту же нотацию можно определить и так:

```
list[int | float]
```

Такой вариант рекомендуемый, если позволяет версия.

Давайте еще немного усложним аннотацию. Предположим, что по умолчанию параметр **digits** должен принимать значение **None**. Мы уже знаем, что для этого следует воспользоваться типом **Optional** модуля **typing**.

Получим:

```
def get_positive(digits: Optional[list[Union[int, float]]] = None) -> list[Union[int, float]]:  
    return list(filter(lambda x: x > 0, digits))
```

И такие комбинации можно делать сколь угодно глубокие. Только здесь возникает проблема с визуальным восприятием таких конструкций. Чтобы было нагляднее можно воспользоваться алиасами (переменными на те или иные типы). Например, вначале объявить алиас:

Основы Python

```
Digit = Union[int, float]
```

А, затем, прописать его при аннотировании:

```
def get_positive(digits: Optional[list[Digit]] = None) -> list[Digit]:  
    return list(filter(lambda x: x > 0, digits))
```

Так воспринимается гораздо лучше. Разумеется, алиасы можно было бы и дальше определять, например:

```
def get_positive(digits: Optional[ListDigits] = None) -> ListDigits:  
    return list(filter(lambda x: x > 0, digits))
```

Но увлекаться этим не стоит. Иначе, программисту постоянно придется искать определения алиасов и смотреть, что за типы в них прописаны. Во всем должна быть мера.

Аннотации вызываемых объектов (Callable)

В заключение этого занятия отмечу еще один тип **Callable** из модуля **typing**, который позволяет аннотировать вызываемые объекты. Часто это обычные функции, которые передаются как параметры. Давайте я приведу следующий пример. Пусть у нас имеется функция, которая из ряда целых чисел указанного диапазона выбирает числа по заданному критерию и формирует список:

```
def get_digits(flt: Callable[[int], bool], lst: Optional[list[int]] = None) -> list[int]:  
    if lst is None:  
        return []  
  
    return list(filter(flt, lst))
```

Здесь первый параметр отмечен как вызываемый тип (**Callable**), у которого ожидается целочисленный аргумент и булево возвращаемое значение. Затем, мы можем воспользоваться этой функцией следующим образом:

Основы Python

```
def even(x):  
    return bool(x % 2 == 0)  
  
print(get_digits(even, [1, 2, 3, 4, 5, 6, 7]))
```

Увидим отображение списка только из четных значений. Но если поменять функцию **even** так, чтобы она возвращала не булево значение, а строку:

```
def even(x):  
    return str(x % 2 == 0)
```

То появится подсветка кода из-за несоответствия возвращаемого типа и того, что указан в аннотации.

В общем случае тип **Callable** описывается по синтаксису:

Callable[[TypeArg1, TypeArg2, ...], ReturnType]

Например, если нужно аннотировать функцию, у которой нет параметров и она ничего не возвращает:

```
def hello_callable():  
    print("Hello Callable")
```

То это будет выглядеть так:

```
Callable[[], None]
```

Надеюсь, из этого занятия вы поняли, как выполнять аннотирование различных коллекций языка **Python**.

Основы Python

§68. Аннотации типов на уровне классов

На предыдущем занятии мы с вами в целом рассмотрели различные способы аннотаций отдельных переменных и функций. Здесь же посмотрим на аннотацию на уровне классов.

Итак, давайте рассмотрим простую аннотацию, например, типом **dict**:

```
tr: dict = {'car': 'машина'}
```

В действительности, здесь **dict** – это класс, который определяет работу со словарем. А имя класса воспринимается, как название типа. Значит, при аннотации мы можем указывать любые классы в том числе и **object** – класс, от которого неявно наследуются все классы в **Python 3**. Давайте это сделаем и посмотрим к чему приведет:

```
x: object = None  
x = "123"  
x = 123
```

Смотрите, мы совершенно спокойно можем прописывать любые типы данных, унаследованные от **object**, и интегрированная среда нам не выдаст никаких предупреждений. **Почему так произошло?** Дело в том, что к базовому типу **object** можно привести любой другой тип, который от него наследуется. А это практически все типы данных в **Python**. Поэтому строки, числа и любые другие стандартные объекты допускается присваивать переменной **x** без нарушения данной аннотации.

Или такой пример. Объявим два класса:

```
class Geom: pass  
class Line(Geom): pass
```

и аннотируем переменную типом **Geom**:

```
g: Geom
```

Основы Python

Далее, присвоим этой переменной объект класса **Line**:

```
g = Line()
```

Как видите интегрированная среда не подсвечивает код, значит, данные соответствуют типу, указанному при аннотации – базовому классу **Geom**. А если убрать наследование у класса **Line**:

```
class Line: pass
```

то сразу появляется подсветка, т.к. теперь типы **Line** и **Geom** независимы.

Все эти примеры показывают, как именно аннотация типов учитывает иерархию наследования классов при проверке корректности присваиваемых данных.

Отличие между object и typing.Any

Учитывая все сказанное, внимательный слушатель может спросить: **а в чем отличия между типом object и типом Any из модуля typing?** Если все типы данных в **Python** наследуются от **object**, значит это эквивалентно использованию типа **Any**? Почти. Но есть один нюанс. Его можно сформулировать так:

Тип Any совместим с любым другим типом, а тип object – ни с каким другим.

Давайте я поясню смысл этой фразы на конкретном примере. Допустим, у нас имеется некая переменная аннотированная типом **Any**:

```
a: Any = None
```

А, затем, мы определим еще одну переменную с аннотацией типа **str**:

```
s: str
```

После этого можно совершенно спокойно присвоить первую переменную второй:

Основы Python

```
s = a
```

Подсветки кода, т.е. каких-либо предупреждений не будет. Но, если вместо **Any** прописать **object**:

```
a: object = None
```

то появляется подсветка, т.к. тип **object** не совместим с типом **str**. То есть, с точки зрения приведенной аннотации будет неверным присваивать переменной типа **str** переменную типа **object**. Потому что **str** наследуется от **object**, а не наоборот. А вот если бы типы были записаны в другом порядке:

```
a: str = '123'  
s: object  
s = a
```

то никаких предупреждений не возникало бы. Когда же мы указываем **Any**, то совершенно не имеет значения какой тип будет присвоен этой переменной или куда она далее будет присваиваться:

```
a: Any = '123'  
s: object  
s = a
```

Any – это эквивалент отключения проверки типов, то есть, поведение статического анализатора по умолчанию, словно никакая типизация не была указана. Однако, через **Any** мы явно показываем, что переменная может ссылаться на произвольный тип данных. Вот в этом отличие между **Any** и **object**.

Модуль `myru`

До сих пор мы непосредственно в интегрированной среде смотрели на подсветку кода и понимали, что где то нарушили аннотацию типов. Но доверять целиком и полностью такой анализ интегрированной среде неправильно. Вполне могут возникать случаи, когда она не подсвечивает

Основы Python

нарушения типов и я вам это уже демонстрировал. Чтобы строже оценивать корректность программы с точки зрения аннотации типов, часто на практике используют специальный модуль:

myru

Мало того, этим модулем можно воспользоваться, если под рукой нет интегрированной среды, а неточности как-то выявить нужно. Подробнее о нем можно почитать на странице официальной документации:

<https://mypy.readthedocs.io/en/stable/>

Чтобы воспользоваться этим модулем его нужно вначале установить. Делается это уже известной вам командой:

pip install mypy

После установки в терминале достаточно записать команду:

mypy <имя файла>

В моем случае программа находится в файле **main.py**, а сам файл хранится в текущем рабочем каталоге. Поэтому команда примет вид:

mypy main.py

Если нужно анализировать сразу несколько файлов, то они прописываются через пробел.

После запуска команды здесь же в терминале видим строчки, в которых нарушена типизация. Все достаточно просто и удобно.

Аннотация с помощью Type и TypeVar

Но давайте вернемся непосредственно к теме «**аннотация типов**» и предположим, что у нас объявлены два класса:

Основы Python

```
class Geom: pass
class Point2D(Geom): pass
```

и некая функция, которая должна создавать экземпляры переданных ей классов, унаследованных от **Geom**:

```
def factory_point(cls_geom):
    return cls_geom()
```

Обратите внимание, здесь предполагается, что параметр **cls_geom** будет ссылаться на сам класс, а не объект класса. Почему это важно, вы сейчас увидите. Используя текущие знания по аннотации типов, первое, что приходит в голову – это прописать в функции следующие определения:

```
def factory_point(cls_geom: Geom) -> Geom:
    return cls_geom()
```

Но нам здесь интегрированная среда сразу подсвечивает фрагмент **cls_geom()**. **Почему это произошло?** Как раз по той причине, что аннотация : **Geom** подразумевает, что параметр **cls_geom** будет ссылаться на объекты класса **Geom**, а не на сам класс **Geom**. Вот это очень важно понимать, когда вы прописываете аннотации типов. Везде подразумеваются объекты тех типов, которые указываются. Но **как тогда поправить эту ситуацию?** Очень просто. Для этого существует специальный тип **Type** из модуля **typing**. Если мы перепишем аннотацию в виде:

```
def factory_point(cls_geom: Type[Geom]) -> Geom:
    return cls_geom()
```

то никаких нарушений уже не будет. Тем самым мы указали, что параметр **cls_geom** будет ссылаться непосредственно на класс **Geom**, а не его объекты. А далее, используя переменную **cls_geom**, создается объект этого класса и возвращается функцией.

Давайте теперь воспользуемся этой функцией. Если ее вызвать так:

Основы Python

```
geom = factory_point(Geom)
point = factory_point(Point2D)
```

то с аннотациями никаких конфликтов не будет. Но, если мы дополнительно аннотируем и переменные **geom** и **point** соответствующими типами:

```
geom: Geom = factory_point(Geom)
point: Point2D = factory_point(Point2D)
```

то во второй строчке появится подсветка кода. Очевидно это из-за того, что мы явно указываем ожидаемый тип **Point2D**, а в определении функции прописан тип **Geom**. И, так как **Geom** – базовый класс для **Point2D**, то возникает конфликт аннотаций.

Для исправления таких ситуаций в **Python** можно описывать некие общие типы с помощью класса **TypeVar**. Например:

```
T = TypeVar("T", bound=Geom)
```

Мы здесь объявили универсальный тип с именем **T** и сказали, что он должен быть или классом **Geom** или любым его дочерним классом. Далее, в самой функции, достаточно прописать этот тип:

```
def factory_point(cls_geom: Type[T]) -> T:
    return cls_geom()
```

и он будет автоматически вычисляться при вызове функции. Когда передается класс **Geom**, то **T** будет соответствовать этому типу, а когда передается **Point2D** – то тип **T** будет **Point2D**. И так далее. Вот смысл универсальных типов при формировании аннотаций.

Для полноты картины сразу отмечу здесь, что класс **TypeVar** можно использовать и в таких вариациях:

```
T = TypeVar("T") # T – произвольный тип без ограничений
T = TypeVar("T", int, float) # T – тип связанный только с типами int и float
```

Основы Python

Подробнее обо всем этом можно почитать в официальной документации. Но я не вижу большого смысла глубоко погружаться в эту тему, т.к. различные вариации данного класса используются на практике не часто.

Аннотация типов в классах

В заключение этой темы добавлю пару слов об аннотации типов внутри классов. В целом все делается практически также как и в случае с переменными и функциями. Давайте распишем класс **Point2D** следующим образом:

```
class Point2D:
    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

Обратите внимание, что параметр **self** не принято аннотировать. Также метод **__init__** всегда возвращает значение **None**.

Воспользоваться этим классом можно так:

```
p = Point2D(10.5, 20)
```

В этом случае первый аргумент будет подсвечен, т.к. не соответствует целому типу. Но программа отработает без ошибок. Главный вопрос здесь:

существует ли аннотация типов у локальных атрибутов x, у объекта класса Point2D? Давайте проверим. Запишем команду:

```
p.x = '10'
```

и интегрированная среда нам не выдает никакой подсветки. Но если перейти в терминал и выполнить команду:

муру main.py

то статический анализатор модуля **муру** отметит две строчки. То есть, с точки зрения **муру** в локальных атрибутах **x, y** ожидаются целые значения.

Основы Python

Однако если нужно аннотировать атрибуты класса или его объектов, то лучше это явно прописать непосредственно в самом классе следующим образом:

```
class Point2D:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

В заключение этого занятия покажу еще одну особенность аннотации типов в классах. Объявим метод с именем `copy()`:

```
class Point2D:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

    def copy(self) -> Point2D:
        return Point2D(self.x, self.y)
```

Предполагается, что он должен возвращать копию объекта класса **Point2D**. Однако просто так записать имя класса внутри самого класса не получится. Здесь есть два способа обойти этот момент. Первый (**устаревший**), заключить имя класса в кавычки, прописать его как строку:

```
def copy(self) -> 'Point2D':
    return Point2D(self.x, self.y)
```


Основы Python

Но можно сделать лучше. Если импортировать из модуля `__future__` объект `annotations`:

```
from __future__ import annotations
```

то после этого можно убрать кавычки у имени класса:

```
def copy(self) -> Point2D:  
    return Point2D(self.x, self.y)
```

Вы спросите почему это явно не внедрили в новых версиях языка Python? Зачем требуется что то дополнительно импортировать? Ответ прост. Это сделано специально для обратной совместимости с более ранними версиями. Разработчики языка решили, что это важно. По крайней мере пока. Поэтому просто запомните это обстоятельство.

Следует ли использовать аннотацию типов

На данный момент мы с вами рассмотрели основные возможности типизации данных в языке **Python**. Конечно, это далеко не все, что существует по этой теме, но в большинстве случаев данного материала вполне достаточно, чтобы описывать типы в ваших программах. И завершить я бы хотел краткими указаниями, когда и для чего следует вообще использовать инструмент аннотации типов.

Если вы пишете небольшие короткие программы, которые достаточно просто охватить взглядом, то как таковая аннотация мало чего добавляет в текст программы. Программист и так легко сможет понять какие данные куда передаются. Поэтому излишняя аннотация может даже ухудшить восприятие кода. На мой взгляд для небольших программ достаточно аннотировать ключевые функции или даже совсем ее не делать, как это было в ранних версиях языка и при этом программы воспринимались вполне нормально.

Если же пишется более-менее крупный проект, состоящий из нескольких модулей, то здесь грамотное аннотирование заметно упрощает понимание кода сторонними программистами. Да и сам автор программы спустя

Основы Python

продолжительное время сможет быстро восстановить в памяти все нюансы ее работы. Здесь аннотация типов действительно играет положительную роль и ее стоит использовать. Однако во всем должна быть мера. Можно аннотациями так замусорить текст программы, что получим обратный эффект. Всегда следует помнить, что цель аннотирования – это упрощение восприятия текста программы. И, как только, аннотации начинают мешать – это верный признак усмирить свой энтузиазм и вернуться непосредственно к написанию кода. Увлекаться аннотацией типов не стоит.

Я думаю, что приведенный материал позволит вам теперь грамотно применять аннотацию типов в своих программах и будет действительно помогать лучше понимать программный код.

§69. Конструкция `match/case`. Первое знакомство

В Python версии 3.10 появилась новая конструкция под названием `match/case`, которая позволяет достаточно гибко анализировать переменные на соответствие шаблонов и для найденного совпадения выполнять некоторые заданные операции. Данный оператор имеет следующий синтаксис:

```
match <переменная>:
    case <шаблон_1>:
        операторы
    ...
    case < шаблон_n>:
        операторы
    case _:
        иначе (default)
```

На первый взгляд, вроде понятно, только **что означает здесь слово «шаблон»?** Это и некоторые другие моменты мы, как раз, и будем постепенно раскрывать на занятиях по оператору `match/case`.

Давайте вначале запишем эту конструкцию в очень простом варианте:

```
cmd = "top"
```

Основы Python

```
match cmd:
    case "top":
        print("вверх")
    case "left":
        print("влево")
    case "right":
        print("вправо")
    case _: # wildcard
        print("другое")

print("проверки завершены")
```

Смотрите, здесь есть некая переменная **cmd** на строку «**top**», затем, она указывается после оператора **match**, внутри которого записаны блоки **case**. После каждого **case** указано значение в виде строки. Как только значение переменной оказывается равным значению после **case**, то выполняются операторы внутри этого блока **case** и далее выполнение программы переходит к следующей строчке после **match** – функции **print("проверки завершены")**.

То есть, как только выполняется один из блоков **case**, оператор **match** завершает свою работу. Соответственно, внутри **match** обязательно должен быть записан хотя бы один оператор **case**. А общее число блоков **case** может быть произвольным. Конечно, всегда следует помнить о читабельности программы и удобстве ее последующего редактирования.

Внутри каждого блока **case** должен быть хотя бы один оператор. Например, запись вида:

```
match cmd:
    case "top":
    case "left":
    case "right":
        print("вверх, влево или вправо")
    case _: # wildcard
```

Основы Python

```
print("другое")
```

приведет к синтаксической ошибке. Если нужно в одном блоке **case** учесть сразу несколько констант, это можно сделать с помощью оператора '|' следующим образом:

```
match cmd:
    case "top" | "left" | "right":
        print("вверх, влево или вправо")
    case _: # wildcard
        print("другое")
```

Обратите внимание на последний блок **case** с символом подчеркивания. Это, так называемый, **wildcard** символ. В **Python** мы его обычно используем, если по синтаксису нужно прописать переменную, но в программе она нам не нужна. В частности здесь блок **case** с переменной `_` будет отработывать всегда, если не отработали все предыдущие блоки. То есть, это некий аналог условного оператора **else** в **if**. Например, если переменная:

```
cmd = "top2"
```

то мы увидим сообщения:

```
другое
проверки завершены
```

Как раз здесь отработал последний блок **case**.

Те из вас, кто знаком с другими языками программирования, например, **C++** или **Java**, возможно сейчас смотрят на эту программу и думают, ага, так это же аналог конструкции **switch/case** для проверки переменных на равенство заданных констант. Но не спешите с выводами. В действительности оператор **match/case** гораздо более гибкий, чем **switch/case** и гибкость определяется тем, что после **case** прописывается не просто константа, а шаблон проверки. Сейчас используется простейший шаблон, когда мы сравниваем переменную

Основы Python

cmd на равенство указанным после **case** значениям. Но вариаций здесь гораздо больше и об это мы сейчас пойдет речь.

Шаблоны сравнений оператора **case**

Вместо конкретных значений после оператора **case** можно записывать переменные. Например, так:

```
cmd = "top"

match cmd:
    case command:
        print(f'команда: {command}')
```

В результате при выполнении блока **case** будет создана переменная **command** (если ранее ее не было в программе) и ссылаться на то же значение, что и переменная **cmd**, то есть, на строку «**top**». Поэтому при выполнении программы увидим на экране сообщение:

команда: top

То есть, фактически, в блоке **case** выполняется присваивание:

```
command = cmd
```

и выводится значение переменной **command** в консоль. Причем, этот блок **case** будет выполняться всегда при любых значениях переменной **cmd**. Потому что такой шаблон не делает никаких проверок, он просто создает переменную **command** и присваивает значение переменной, указанной после оператора **match**. Причем, если после этого **case** прописать еще какой-либо, например:

```
match cmd:
    case command:
        print(f'команда: {command}')
    case "top":
```

Основы Python

```
print("top")
```

то возникнет синтаксическая ошибка, связанная с тем, что первым указан блок **case**, перехватывающий все возможные варианты, и все другие **case** просто игнорируются. Поэтому такой общий шаблон можно указывать только последним:

```
match cmd:
    case "top":
        print("top")
    case command:
        print(f"команда: {command}")
```

И если здесь вместо переменной **command** прописать нижнее подчеркивание, то получим уже знакомый нам блок **default** (**отбойник**), который перехватывает все, что не вошло в предыдущие блоки:

```
match cmd:
    case "top":
        print("top")
    case _:
        print(f"другая команда")
```

Шаблоны проверки типов

Давайте теперь усложним наш шаблон и в первом блоке **case** будем перехватывать все строковые команды, то есть, те, которые записаны в виде строки. Сделать это можно следующим образом:

```
cmd = "top"

match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case _: # wildcard
        print(f"другая команда")
```

Основы Python

Смотрите, после **case** записано **str()**. Мы привыкли такую запись воспринимать как создание или пустой строки или преобразование объекта в строку. Здесь же это не что иное, как проверка переменной **cmd** строковому типу. То есть, если переменная **cmd** является строкой, то только в этом случае выполнится этот блок **case** и будет создана переменная **command**, которая, как и ранее, ссылается на значение переменной **cmd**.

Обратите еще раз внимание на конструкцию «**str() as command**». Вначале идет проверка строкового типа переменной **cmd**, и если это так, то после ключевого слова **as** можно указать переменную, которая будет ссылаться на эту строку. Кстати, если нам нужна только проверка на строковый тип данных, то фрагмент «**as command**» можно не писать, например:

```
match cmd:
    case str():
        print(f"строковая команда")
    case _: # wildcard
        print(f"другая команда")
```

Или, вместо «**as command**» можно написать:

```
match cmd:
    case str(command):
        print(f"строковая команда: {command}")
    case _: # wildcard
        print(f"другая команда")
```

Но выглядит это несколько путанно, так как в **Python** мы привыкли воспринимать такую конструкцию «**str(command)**», как преобразование объекта в строку. Поэтому предпочтительнее, на мой взгляд, использовать ключевое слово **as**.

По аналогии можно проверять и другие типы данных, например:

```
cmd = 10
```

Основы Python

```
match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case int() as command:
        print(f"целочисленная команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case _: # wildcard
        print(f"другая команда")
```

При выполнении этой программы будет выполнен второй блок **case**, так как переменная **cmd** целочисленная. Однако, если мы ей присвоим булево значение, например:

```
cmd = True
```

то снова отработает второй блок **case**, а не третий, как этого можно было бы ожидать. **Почему так произошло?** Я думаю, что многие из вас уже догадались, что проверка на тип данных в операторах **case** выполняется по принципу функции **isinstance()**, то есть, с учетом цепочки наследования типов. В частности, булевый тип наследуется от целочисленного, поэтому **isinstance()** и для целых чисел и для булевых значений возвращает **True**.

Разрешить эту проблему очень просто. Так как шаблоны в операторах **case** проверяются по порядку (**сверху-вниз**), то сначала следует поставить проверку на булевый тип, а потом на целочисленный:

```
match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case int() as command:
        print(f"целочисленная команда: {command}")
    case _: # wildcard
```


Основы Python

```
print(f"другая команда")
```

Тогда все проверки на тип данных будут проходить корректно. По аналогии мы можем делать проверку на любые типы данных, которые имеются в языке Python.

Guard (защитник)

Давайте сделаем еще один шаг усложнения наших шаблонов и предположим, что для целочисленных команд дополнительно нужно проверять, чтобы они были в диапазоне от 0 до 9 включительно. Для этого после определения переменной **command** можно прописать ключевое слово **if** и указать дополнительное условие. В нашем случае оно будет выглядеть так:

```
match cmd:
    case str() as command:
        print(f"строковая команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case int() as command if 0 <= command <= 9:
        print(f"целочисленная команда: {command}")
    case _: # wildcard
        print(f"другая команда")
```

Теперь блок «**case int() as command if 0 <= command <= 9**» отработает только в том случае, если переменная **cmd** является целочисленной и значение переменной **command** (которая ссылается на переменную **cmd**) лежит в диапазоне [0; 9]. Причем, обратите внимание, оператор **if** будет отработывать после команды **int()**, то есть, после проверки на целый тип данных и после формирования переменной **command**. Поэтому если проверка дошла до **if**, то мы точно знаем, что определена переменная **command** и она имеет целочисленный тип. Проверка шаблона в **case** всегда выполняется строго слева-направо.

Основы Python

В конструкции **match/case** оператор **if** получил название **guard** (то есть, защитник). Условия в этом операторе можно прописывать по тем же правилам, что и в условных операторах **if** языка **Python**.

По аналогии можно прописать **guard** и для строковых команд, например, так:

```
cmd = "c_top"

match cmd:
    case str() as command if len(command) < 10 and command[0] == 'c':
        print(f"строковая команда: {command}")
    case bool() as command:
        print(f"булева команда: {command}")
    case int() as command if 0 <= command <= 9:
        print(f"целочисленная команда: {command}")
    case _: # wildcard
        print(f"другая команда")
```

Если же какая-либо переменная может принимать несколько разных типов, например, **int** и **float**, то их можно записать через оператор **|** следующим образом:

```
case int() | float() as command if 0 <= command <= 9:
```

И так далее, то есть здесь мы можем определять любые нужные нам условия в шаблонах оператора **case**.

Возможно, некоторым из вас на протяжении всего занятия не давал покоя вопрос, **зачем все это нужно. У нас же в Python есть условные операторы if/elif/else и через них можно делать все те же самые проверки?** Все верно. Конструкция **match/case** не дает нам ничего принципиально нового. Она лишь может упростить написание программного кода (при ее грамотном использовании), а также повысить читаемость текста программы. Это, наверное, главные причины, по которым данные операторы были добавлены в язык **Python**. Как мы дальше увидим, шаблоны в операторах **case** позволяют

Основы Python

очень гибко, просто и удобно обрабатывать коллекции с данными, но об этом речь пойдет уже на следующем занятии.

§70. Конструкция match/case с кортежами и списками

На предыдущем занятии был рассмотрен шаблон, содержащий константу, либо одну переменную. Часто этого бывает достаточно, если мы анализируем отдельные объекты, а не коллекции данных. Но давайте представим, что хотели бы обрабатывать информацию, представленную в виде кортежа или списка. **Как в этом случае можно строить шаблоны в операторах case?**

Для простоты определим кортеж из трех элементов, содержащий информацию о книге, например, такой:

```
cmd = ("Балакирев С.М.", "Python", 2000.78)
```

Здесь первые две строки – это автор и название книги, а последнее число – цена книги. Если сейчас прописать в операторе **case** одну переменную с проверкой на тип данных, например, так:

```
match cmd:
    case tuple() as book:
        print(f"кортеж: {book}")
    case _: # wildcard
        print("непонятный формат данных")
```

то у нас отобразится в консоли строка:

```
кортеж: ('Балакирев С.М.', 'Python', 2000.78)
```

То есть, пока все работает ровно так, как на предыдущем занятии: формируется новая переменная **book**, которая ссылается на кортеж с данными о книге. Но, так как кортеж относится к упорядоченным типам данных (**Sequence Types**), то его можно распаковывать непосредственно в шаблонах следующим образом:

Основы Python

```
match cmd:
    case author, title, price:
        print(f"Книга: {author}, {title}, {price}")
    case _: # wildcard
        print("непонятный формат данных")
```

Фактически, здесь используется уже знакомая нам конструкция распаковки последовательностей вида:

```
author, title, price = cmd
```

Причем, этот шаблон сработает только в том случае, если кортеж содержит ровно три элемента. Давайте ради интереса добавим четвертое значение – год издания:

```
cmd = ("Балакирев С.М.", "Python", 2000.78, 2022)
```

После запуска программы увидим строчку:

непонятный формат данных

то есть, шаблон не сработал. В нем мы явно прописали три элемента, а в переданном оказалось четыре. Поэтому перешли в блок **default** (отбойник).

Но **как можно было бы сделать так, чтобы вне зависимости от числа элементов кортежа брать только первые три элемента?** Очень просто! Снова вспоминаем правила распаковки последовательностей, где мы с вами использовали оператор ***** для чтения всех остальных элементов, если они есть:

```
match cmd:
    case author, title, price, *_:
        print(f"Книга: {author}, {title}, {price}")
    case _: # wildcard
        print("непонятный формат данных")
```

Основы Python

То есть в `*` будут попадать все элементы, начиная с четвертого. Если же в кортеже всего три элемента, то шаблон также сработает, только в `*` не будет ни одного элемента.

Сразу отмечу, что все то же самое будет работать и со списками. Если вместо кортежа указать список:

```
cmd = ["Балакирев С.М.", "Python", 2000.78, 2022]
```

то на работе рассматриваемых шаблонов это никак не отразится, т.к. это все та же упорядоченная коллекция (**Sequence Types**).

Далее, если, например, дополнительно мы хотим ограничить размер кортежа или списка определенным числом элементов, то для этого можно записать **guard**, так же, как мы это делали для отдельных переменных:

```
match cmd:
    case author, title, price, *_ if len(cmd) < 6:
        print(f"Книга: {author}, {title}, {price}")
    case _: # wildcard
        print("непонятный формат данных")
```

Но чтобы шаблон выглядел в более читаемом виде, допустимо использовать круглые или квадратные скобки:

```
case (author, title, price, *) if len(cmd) < 6:
```

или

```
case [author, title, price, *_] if len(cmd) < 6:
```

Обратите внимание, здесь эти скобки не создают ни кортежей, ни списков. Это группирующие скобки, чтобы часть шаблона воспринималось как единое целое. Позже мы увидим, для чего они нужны.

Основы Python

Давайте теперь предположим, что мы хотели бы еще делать проверку на тип данных элементов списка `cmd`. Нет ничего проще. Для каждой отдельной переменной можно прописать:

```
case [str() as author, str() as title, float() as price, *_] if len(cmd) < 6:
```

Или сделать это только у некоторых переменных, например, у цены:

```
case [author, title, float() as price, *_] if len(cmd) < 6:
```

Мало того, если переменная может принимать несколько разных типов, то их можно перечислить через оператор `|` следующим образом:

```
case [author, title, float() | int() as price, *_] if len(cmd) < 6:
```

Если нам нужно указать какие-либо ограничения на длину строк автора и заголовка, то это делается в **guard**, например, так:

```
case [str(author), str(title), price, *_] if len(cmd) < 6 and len(author) < 50 and len(title) < 100:
```

И так далее, то есть, здесь с отдельными переменными можно делать все то, что мы рассматривали на предыдущем занятии.

Далее, предположим, что информация о книге может быть представлена в нескольких форматах, например, двух разных:

```
cmd = ["Балакирев С.М.", "Python", 2000.78]
```

или

```
cmd = [1, "Балакирев С.М.", "Python", 2000.78, 2022]
```

И нам нужно в обоих случаях выделять **автора**, **заголовок** и **цену**. Все остальное не имеет значения. **Как сформировать обработку таких данных?** Конечно, здесь мы могли бы прописать два отдельных шаблона, но это привело бы к некоторому дублированию кода, что не хорошо. Лучше

Основы Python

определить один шаблон, объединяющий эти два варианта. Сделать это можно следующим образом:

```
match cmd:
    case (author, title, price) | (_, author, title, price, _):
        print(f"Книга: {author}, {title}, {price}")
    case _: # wildcard
        print("непонятный формат данных")
```

Смотрите, мы используем группирующие скобки и указываем, как могут быть организованы данные внутри коллекции. На местах тех элементов, что нам не нужны, стоит символ подчеркивания. А оператор '|' означает «или». То есть, может отработать или первый вариант формата, или второй. Причем, обратите внимание, число переменных в группах должно быть одинаковым и их имена совпадать. Это строго обязательно. То есть, мы не можем, например, во второй группе дополнительно выделить год издания:

```
case (author, title, price) | (_, author, title, price, year):
```

Будет синтаксическая ошибка. Только одни и те же переменные во всех группах одного и того же шаблона. Если нам нужно во втором варианте дополнительно выделять год издания и с ним что то делать, то тогда вариант с объединением не подойдет и нужно определять два разных оператора **case**:

```
match cmd:
    case (author, title, price):
        print(f"Книга: {author}, {title}, {price}")
    case (_, author, title, price, year):
        print(f"Книга: {author}, {title}, {price}, {year}")
    case _: # wildcard
        print("непонятный формат данных")
```

И для практики. Если мы бы здесь не хотели обрабатывать данные в виде кортежей, то в первом операторе **case** следовало бы сделать проверку на этот тип данных, например, так:

Основы Python

```
match cmd:
    case tuple():
        print("формат кортежа недопустим")
    case (author, title, price):
        print(f"Книга: {author}, {title}, {price}")
    case (_, author, title, price, year):
        print(f"Книга: {author}, {title}, {price}, {year}")
    case _: # wildcard
        print("непонятный формат данных")
```

Причем, обратите внимание, проверка на тип **tuple** должна идти вначале перед распаковкой коллекции. Я, думаю, вы понимаете **почему?** Тогда если переменная **cmd** принимает тип **tuple**, то сразу отработает первый оператор **case** и все другие будут проигнорированы.

Вот, в целом, такие вариации построения шаблонов проверок возможны при обработке упорядоченных коллекций в конструкции **match/case**. В заключение лишь отмечу, что обычная строка языка **Python** не относится к типу **Sequence Types**, поэтому делать ее распаковку внутри шаблонов не получится. С ней можно только как с единым целым – со строкой целиком.

На следующем занятии мы продолжим эту тему и увидим, как можно строить шаблоны для проверки формата других коллекций, таких как словарей и множеств.

§71. Конструкция **match/case** со словарями и множествами

На предыдущих занятиях мы увидели, как можно строить шаблоны проверок для отдельных переменных и упорядоченных коллекций типа **list** и **tuple**. Теперь пришло время обработки других коллекций типа **dict** (словарь) и **set** (множество). Начнем со **словарей**.

Давайте предположим, что на вход блока обработки поступает запрос в виде следующего словаря:

```
request = {'url': 'https://proporprogs.ru/', 'method': 'GET', 'timeout': 1000}
```


Основы Python

Мы хотим выбрать из него данные по ключам «**url**» и «**method**». Шаблон проверки в этом случае будет выглядеть так:

```
match request:
    case {'url': url, 'method': method}:
        print(f"Запрос: url: {url}, method: {method}")
    case _: # wildcard
        print("неверный запрос")
```

Обратите внимание, мы здесь использовали фигурные скобки. Как только мы их прописываем, шаблон сразу ожидает данные в формате «**ключ-значение**». Далее, мы говорим, что в словаре **request** должны быть два ключа «**url**» и «**method**» с какими то произвольными значениями. Если эти ключи действительно присутствуют, то создаются переменные **url** и **method**, которые ссылаются на соответствующие значения ключей. Затем, с помощью **print()**, мы выводим в консоль строку с этими значениями переменных.

То есть, данный шаблон будет срабатывать всякий раз, как только на вход поступает словарь с ключами «**url**» и «**method**». При этом наличие или отсутствие других ключей не играет никакой роли. Например, в словаре **request** есть еще один ключ «**timeout**», но, тем не менее, шаблон срабатывает. Это отличает проверку словарей от проверки кортежей и списков, когда мы должны были указывать ровно то число элементов, которые в них ожидаются. Для словарей же достаточно лишь наличие указанных ключей и не более того.

Далее, если мы дополнительно хотим сделать проверку на тип данных в значениях словаря, то это можно сделать следующим образом:

```
case {'url': str(url), 'method': int() as method}:
```

Я здесь специально проверку типа записал двумя разными способами, чтобы напомнить о них. В **первом** ключе мы требуем, чтобы была строка, а во **втором** – чтобы было целое число. Конечно, сейчас шаблон не сработает, т.к. **method** является строкой, но если мы его изменим на число:

Основы Python

```
request = {'url': 'https://proproprogs.ru/', 'method': 1, 'timeout': 1000}
```

то он среагирует на этот словарь.

Записывая после ключей имена переменных, мы, тем самым, определяем шаблон для произвольных значений заданного типа (или любого типа, если этих проверок нет). Но можно записать ключ с конкретным значением, например, так:

```
case {'url': url, 'method': method, 'timeout': 1000}:
```

Тогда оператор **case** отработает только если ключ **'timeout'** принимает значение **1000** и будет игнорировать все другие значения. Например, если:

```
request = {'url': 'https://proproprogs.ru/', 'method': 'GET', 'timeout': 2000}
```

то перейдем в последний блок **case** (**отбойник**). Но, если мы скомбинируем эти два варианта в одном шаблоне:

```
case {'url': url, 'method': method, 'timeout': 1000} | {'url': url, 'method': method, 'timeout': 2000}:
```

то сможем отлавливать и значение **1000** и значение **2000** для ключа **'timeout'**. То есть, здесь оператор **|** работает так же, как и с упорядоченными коллекциями.

Также мы, при необходимости, можем прописывать **guard** (**защитника**) при обработке словарей. Например, если в запросе **request** ожидается не более трех ключей, то такой шаблон можно записать в виде:

```
case {'url': url, 'method': method} if len(request) <= 3:
```

И так далее, здесь допустимо использовать все те же самые конструкции, которые мы рассматривали на предыдущих занятиях.

Давайте теперь предположим, что нам нужно построить шаблон проверки словаря, в котором есть ключи **«url»** и **«method»** и дополнительно может быть

Основы Python

еще не более двух ключей. **Как это сделать?** Для этого следует воспользоваться оператором упаковки ****** всех остальных данных словаря, кроме указанных ключей, например, так:

```
match request:
    case {'url': url, 'method': method, **kwargs} if len(kwargs) <= 2:
        print(f"Запрос: url: {url}, method: {method}")
    case _: # wildcard
        print("неверный запрос")
```

В результате в переменной **kwargs** будут находиться все остальные ключи с их значениями, кроме ключей «**url**» и «**method**». А далее, мы просто проверяем, чтобы словарь **kwargs** содержал не более двух дополнительных ключей.

Теперь, например, для запроса (словаря):

```
request = {'id': 2, 'url': 'https://proporprogs.ru/', 'method': 'GET', 'timeout': 2000}
```

шаблон сработает, а для словаря:

```
request = {'id': 2, 'url': 'https://proporprogs.ru/', 'method': 'GET', 'timeout': 2000,
'date': 100}
```

не сработает, т.к. дополнительно имеем уже три ключа. Причем расположение ключей в словаре не имеет никакого значения при работе шаблона. Главное, чтобы существовали ключи «**url**» и «**method**» и кроме них было не более двух дополнительных ключей.

По аналогии можно организовать проверку на наличие строго двух ключей в словаре:

```
case {'url': url, 'method': method, **kwargs} if not kwargs:
```

Мы здесь в защитнике проверяем, чтобы словарь **kwargs** был пустым.

Основы Python

Давайте теперь несколько усложним структуру словаря и предположим, что на вход поступают следующие данные:

```
json_data = {'id': 2, 'type': 'list', 'data': [1, 2, 3], 'access': True, 'date': '01.01.2023'}
```

Здесь множество ключей, причем, один из них (**data**) может содержать список, если ключ **«type»** принимает значение **«list»**. Шаблон обработки можно записать в виде:

```
match json_data:
    case {'type': 'list', 'data': lst}:
        print(f"JSON-данные: type-list: {lst}")
    case _: # wildcard
        print("неверный запрос")
```

Первый блок **case** сработает только в том случае, если ключ **«type»** имеет значение **«list»** и имеется еще один ключ **«data»**. Ожидается, что переменная **lst** в этом случае будет ссылаться на список. Конечно, для надежности, мы можем сделать дополнительную проверку типа данных:

```
case {'type': 'list', 'data': list() as lst}:
```

Тогда гарантированно, при срабатывании блока **case**, переменная **lst** будет принимать тип **list**. Видите, как легко, просто и элегантно мы сделали обработку нужного нам формата данных с помощью операторов **match/case**. Гораздо проще, чем если бы использовали условные операторы **if/elif/else**. Именно в этом преимущество конструкции **match/case** перед условными операторами.

Или вот, еще несколько более сложный пример шаблона обработки проверки и обработки входных данных. Пусть имеется словарь вида:

```
json_data = {'id': 2, 'access': True, 'info': ['01.01.2023', {'login': '123', 'email': 'email@m.ru'}], True, 1000]}
```

Основы Python

Нам нужно проверить наличие ключей 'access' и 'info' и выделить значение ключа 'access' и **email** из ключа 'info'. Сделать это можно следующим образом:

```
match json_data:
    case {'access': access, 'info': [_, {'email': email}, _, _]}:
        print(f"JSON: access: {access}, email: {email}")
    case _: # wildcard
        print("неверный запрос")
```

Смотрите, как красиво все выглядит! Мы буквально берем значение ключа 'access' и нужное значение для **email** из списка ключа 'info', попутно проверяя наличие этих ключей и списка в словаре **json_data**. Причем список обязательно должен содержать четыре элемента и во втором храниться словарь с ключом **email**. Попробуйте сделать то же самое через операторы **if/elif/else** и вы увидите, насколько сложнее будет выглядеть эта же программа! Тогда как конструкция **match/case** с легкостью решает поставленную задачу.

Обработка множеств

Вот в целом так строятся проверки для словарей. Помимо них есть еще одна похожая коллекция – множество (**set**). Предположим, мы бы хотели выбирать все множества, состоящие из трех элементов, например:

```
primary_keys = {1, 2, 3}
```

Как это можно сделать? Если прописать по аналогии со словарем:

```
match primary_keys:
    case {a, b, c}:
        print(f"Primary Keys: {a}, {b}, {c}")
    case _: # wildcard
        print("неверный запрос")
```

то получим синтаксическую ошибку. Для множеств так записывать шаблоны нельзя. Если попробовать убрать фигурные скобки, то будут ожидать на

Основы Python

входе кортежи или списки. **Как же быть с множествами?** Вариант, фактически, один. Прописать явно тип **set** и присвоить некоторой переменной ссылку на множество:

```
match primary_keys:
    case set() as keys:
        print(f"Primary Keys: {keys}")
    case _: # wildcard
        print("неверный запрос")
```

А проверку, чтобы множество содержало ровно три элемента, можно сделать с помощью **guard**:

```
case set() as keys if len(keys) == 3:
```

По сути, только так мы можем отбирать множества и через **guard** записывать некоторые проверки.

На этом мы завершим наше очередное занятие по конструкции **match/case**. На следующем заключительном занятии по этой теме рассмотрим некоторые ограничения и особенности работы этих операторов, а также посмотрим на некоторые типовые примеры их использования.

§72. Конструкция match/case. Примеры и особенности использования

На предыдущих занятиях мы в целом увидели, как можно прописывать различные шаблоны для проверок отдельных констант и переменных, а также упорядоченных коллекций **list**, **tuple** и более сложных – **dict**, **set**. На этом занятии посмотрим некоторые реальные приложения конструкции **match/case**, а также увидим ограничения при использовании этих операторов.

Давайте предположим, что нам нужно организовать подключение к БД. Для этого формируется словарь с необходимыми данными, например, такой:

```
request = {'server': '127.0.0.1', 'login': 'root', 'password': '1234', 'port': 24}
```

Основы Python

Затем, с помощью нашей функции `connect_db()` выполним проверку данных в запросе `request` и вернем результат подключения:

```
result = connect_db(request)
print(result)
```

Саму же функцию запишем в следующем виде:

```
def connect_db(connect: dict) -> str:
    match connect:
        case {'server': host, 'login': login, 'password': psw, 'port': port}:
            return f"connection: {host}@{login}.{psw}:{port}"
        case {'server': host, 'login': login, 'password': psw}:
            port = 22
            return f"connection: {host}@{login}.{psw}:{port}"
        case _: # wildcard
            return "error connection"
```

Здесь сразу бросается в глаза дублирование кода в первых двух блоках `case`, а именно, строчка с возвращением `f`-строки. Давайте это поправим. Вынесем ее из блока `match` и запишем в самом конце функции:

```
def connect_db(connect: dict) -> str:
    match connect:
        case {'server': host, 'login': login, 'password': psw, 'port': port}:
            pass
        case {'server': host, 'login': login, 'password': psw}:
            port = 22
        case _: # wildcard
            return "error connection"

    return f"connection: {host}@{login}.{psw}:{port}"
```

Смотрите, в **первом** блоке `case` записан оператор `pass`, который ничего не делает. Но он нужен, т.к. **Python** не разрешает объявление пустых блоков

Основы Python

case, в них обязательно должен быть хотя бы один оператор. Именно поэтому я и прописал **pass**. Во **втором** блоке также стоит один оператор, который создает переменную **port** со значением **22**. Если первые два блока **case** не сработают, то третий отработает в любом случае и функция **connect_db()** вернет строку **"error connection"**. Значит, если мы доходим до последней строки функции, то сработал или первый **case**, или второй. В любом случае у нас будут переменные **host**, **login**, **psw** и **port**, так как все созданные переменные внутри блока **match** сохраняются и за его пределами. Поэтому формирование **f**-строки пройдет без ошибок и будет возвращен требуемый результат. Так мы избавляемся от дублирования кода при одинаковом наборе переменных.

Давайте рассмотрим еще один пример. Предположим, у нас может информация о книгах быть представлена в разных форматах, например:

```
book_1 = ('Балакирев', 'Python', 2022)
book_2 = ['Балакирев', 'Python ООП', 2022, 3432.27]
book_3 = {'author': 'Балакирев', 'title': 'Нейросети', 'year': 2020}
book_4 = {'author': 'Балакирев', 'title': 'Keras + Tensorflow', 'price': 5430, 'year': 2020}
```

Наша задача написать функцию, которая бы обрабатывала все эти варианты представления книг и выдавала кортеж в формате:

(автор, название, год, цена)

Пусть эта функция называется **book_to_tuple()** и вызывается в программе следующим образом:

```
result = book_to_tuple(book_1)
print(result)
```

Саму функцию удобно реализовать с помощью операторов **match/case**, например, так:

```
def book_to_tuple(data: dict | tuple | list) -> tuple | None:
```


Основы Python

```
match data:
    case author, title, year:
        price = None
    case author, title, year, price, _:
        pass
    case {'author': author, 'title': title, 'year': year, 'price': price}:
        pass
    case {'author': author, 'title': title, 'year': year}:
        price = None
    case _: # wildcard
        return None

return author, title, year, price
```

Смотрите, параметр **data** может быть разных типов: **словарь**, **кортеж** или **список**. На выходе функция должна выдавать или кортеж, или значение **None**, если входные данные не соответствуют ожидаемым форматам. Далее, первый блок **case** распаковывает кортеж или список, состоящий из трех элементов в формате: **автор**, **название**, **год издания**. Если шаблон срабатывает, то формируется четвертая переменная **price** со значением **None**. Следующий шаблон ожидает также список или кортеж, состоящий, как минимум из четырех элементов в порядке: **автор**, **название**, **год издания**, **цена**. Далее могут идти какие-либо другие элементы, мы их просто игнорируем. Следующие два блока проверяют словари. Словарь может содержать или три ключа, тогда формируется четвертая переменная **price** со значением **None**, или четыре ключа с полным набором необходимых переменных. Если ни один из шаблонов не срабатывает, то попадаем в последний блок **case**, который возвращает значение **None**, то есть, несоответствие форматов. Если же отработывает один из предыдущих блоков **case**, то переходим к последней команде функции, где возвращается кортеж со значениями переменных **author**, **title**, **year**, **price**.

Основы Python

Как видите, все достаточно просто и очевидно. Но здесь опять же есть небольшое дублирование кода: дважды записана команда `price = None`. По логике ее можно вынести за пределы оператора `match` и записать перед ним:

```
def book_to_tuple(data: dict | tuple | list) -> tuple | None:
    price = None
    match data:
        case author, title, year:
            pass
        case author, title, year, price, _:
            pass
        case {'author': author, 'title': title, 'year': year, 'price': price}:
            pass
        case {'author': author, 'title': title, 'year': year}:
            pass
        case _: # wildcard
            return None

    return author, title, year, price
```

Получим абсолютно тот же самый результат. (Операторы `pass` можно не считать дублированием кода).

Давайте немного усложним задачу и сделаем дополнительную проверку на целочисленный тип переменной `year` и ее принадлежность диапазону значений (`min_year`; `max_year`). Если сделать это «в лоб», то получим что то вроде:

```
def book_to_tuple(data: dict | tuple | list, min_year=1800, max_year=3000) -> tuple | None:
    price = None
    match data:
        case author, title, int(year) if min_year < year < max_year:
            pass
        case author, title, int(year), price, *_ if min_year < year < max_year:
```

Основы Python

```
    pass
    case {'author': author, 'title': title, 'year': int(year), 'price': price} if min_year <
year < max_year:
    pass
    case {'author': author, 'title': title, 'year': int(year)} if min_year < year <
max_year:
    pass
    case _: # wildcard
    return None

return author, title, year, price
```

Видите, здесь идет явное дублирование кода в шаблонах. Оставить так или поправить, конечно, решать разработчику, но, на мой взгляд, здесь целесообразно скомбинировать оператор **match** с оператором **if** следующим образом:

```
def book_to_tuple(data: dict | tuple | list, min_year=1800, max_year=3000) -> tuple
| None:
    price = None
    match data:
        case author, title, int(year):
            pass
        case author, title, int(year), price, *_:
            pass
        case {'author': author, 'title': title, 'year': int(year), 'price': price}:
            pass
        case {'author': author, 'title': title, 'year': int(year)}:
            pass
        case _: # wildcard
            return None

    if not (min_year < year < max_year):
        return None
```

Основы Python

```
return author, title, year, price
```

То есть, мы буквально всю проверку вынесли в одну строчку оператора **if**. Этот пример показывает, что нас никто не ограничивает в различных комбинациях операторов **match** и **if** для написания более удобного и читабельного текста программы.

В заключение этого занятия покажу еще одну особенность конструкции **match/case**. Для простоты предположим, что у нас имеется некая целочисленная переменная:

```
cmd = 10
```

и в операторе **match** хотели бы выделить две ситуации со значениями **3** и **5**. Очевидно, сделать это можно следующим образом:

```
match cmd:
    case 3:
        print("3")
    case 5:
        print("5")
```

Но опытный разработчик знает, что явно прописывать константы в программе не лучший ход. Правильнее было бы сначала задать константы с числами **3** и **5**, а уже потом их использовать в операторах **case**. Давайте так и сделаем:

```
CMD_3 = 3
CMD_5 = 5
```

```
cmd = 3
```

```
match cmd:
    case CMD_3:
        print("3")
```

Основы Python

```
case CMD_5:  
    print("5")
```

Однако, при запуске обнаруживается синтаксическая ошибка. **Почему?** Дело в том, что в **Python** нет констант как таковых. Здесь объявлены две переменные **CMD_3** и **CMD_5**, которые указываются в операторах **case**. В действительности, это аналогично тому, что мы в **case** пропишем произвольную переменную, например, такую:

```
match cmd:  
    case _:  
        print("3")  
    case CMD_5:  
        print("5")
```

Да, принципиально мы ничего не изменили, просто написали другое имя переменной. Но вы помните, что она означает – любые данные, т.е. срабатывает всегда, поэтому второй блок **case** здесь приводит к ошибке. Если мы его уберем, то программа всегда будет выводить число **3**:

```
match cmd:  
    case _:  
        print("3")
```

Но это не то, что нам нужно. Все же, **как можно использовать переменные как константы в операторах case?** Здесь есть, по крайней мере, два варианта. Если сами константы объявлены в текущем модуле и менять мы этого не хотим, то можно пойти на следующую хитрость:

```
match cmd:  
    case int(cmd) as x if x == CMD_3:  
        print("3")  
    case int(cmd) as x if x == CMD_5:  
        print("5")
```

Основы Python

Однако, выглядит это слишком громоздко. Тут проще было бы воспользоваться операторами `if/elif/else`. Во втором случае разработчики разрешают нам использовать переменные как константы, если переменная указана через точку. **Но что это значит?** Например, мы можем переменные `CMD_3` и `CMD_5` вынести в отдельный файл (**модуль**), а потом подключить его через `import` в текущем модуле:

```
import consts

cmd = 3

match cmd:
    case consts.CMD_3:
        print("3")
    case consts.CMD_5:
        print("5")
```

Тогда никаких проблем не возникнет и программа будет работать, как задумано.

Или, можно пойти на еще одну хитрость и определить константы внутри какого-либо класса, например, так:

```
class Consts:
    CMD_3 = 3
    CMD_5 = 5

cmd = 3

match cmd:
    case Consts.CMD_3:
        print("3")
    case Consts.CMD_5:
        print("5")
```

Основы Python

В этом случае тоже все отработает как надо без ошибок.

Вот такие приемы и особенности есть у конструкции **match/case**, которые мне удалось выяснить. Я думаю, что материал этих занятий по операторам **match/case** позволит вам достаточно уверенно и легко использовать их в своих программах, там, где они действительно позволят упростить программный код и сделают его более читаемым и удобным для редактирования.

Практические упражнения

Упражнения -1: Базовые конструкции языка Python

Эта программа на **Python** представляет собой пример, который демонстрирует различные основные концепции программирования на **Python**.

Вот как она работает:

1. Импорт модуля **math**:

1. Программа начинается с импорта модуля **math**, который предоставляет доступ к различным математическим функциям.

2. Присваивание переменных и вывод на экран:

1. Затем определяются несколько переменных (**name**, **age**, **height**, **is_student**) и присваиваются им значения. Эти переменные хранят различные типы данных, включая строки, целые числа, числа с плавающей запятой и логические значения.
2. С помощью инструкции **print** выводятся значения этих переменных на экран.

3. Числовые типы и арифметические операции:

1. Определяются две числовые переменные (**num1** и **num2**) и выполняются арифметические операции над ними (сложение, вычитание, умножение и деление).
2. Результаты этих операций сохраняются в переменных и выводятся на экран.

4. Математические функции:

Основы Python

1. Используется модуль **math** для выполнения математических операций, таких как **вычисление квадратного корня** и **возведение числа в степень**.
 2. Результаты сохраняются в переменных (**sqrt_num1** и **power_num1**) и выводятся на экран.
5. **Ввод и преобразование:**
1. С помощью функции **input** программа принимает ввод от пользователя и сохраняет его в переменной **user_input**.
 2. Затем программа пытается преобразовать введенное пользователем значение в целое число с помощью функции **int()**. Если преобразование успешно, программа выводит преобразованное целое число. Если преобразование не удалось (например, если пользователь ввел нечисловое значение), программа обрабатывает исключение с помощью конструкции **try-except**.
6. **Логические типы и операторы сравнения:**
1. Выполняются операции сравнения над двумя числовыми переменными (**x** и **y**), и результаты сохраняются в логических переменных (**is_greater**, **is_equal** и **is_not_equal**).
 2. Результаты сравнений выводятся на экран.
7. **Логические операторы:**
1. Программа демонстрирует использование логических операторов (**and**, **or**, **not**) с логическими переменными (**is_true** и **is_false**).
 2. Результаты этих логических операций выводятся на экран.

При запуске этой программы она выполнит каждый раздел последовательно и отобразит результаты на экране. Пользователи могут ввести число, и программа попытается преобразовать его в целое число, демонстрируя обработку ошибок с помощью конструкции **try-except**. Кроме того, программа покажет результаты различных математических и логических операций.

```
import math
```


Основы Python

1) Переменные, оператор присваивания, функции типа и идентификатора

```
name = "Алиса"
```

```
age = 30
```

```
height = 5.8
```

```
is_student = False
```

```
print("Имя:", name)
```

```
print("Возраст:", age)
```

```
print("Высота:", height)
```

```
print("Студент:", is_student)
```

2) Числовые типы, арифметические операции

```
num1 = 10
```

```
num2 = 5
```

```
addition = num1 + num2
```

```
subtraction = num1 - num2
```

```
multiplication = num1 * num2
```

```
division = num1 / num2
```

```
print("Добавление:", addition)
```

```
print("Вычитание:", subtraction)
```

```
print("Умножение:", multiplication)
```

```
print("Разделение:", division)
```

3) Математические функции и работа с математическим модулем

```
sqrt_num1 = math.sqrt(num1)
```

```
power_num1 = math.pow(num1, 2)
```

```
rounded_num2 = round(num2)
```

```
print("Квадратный корень из числа 1:", sqrt_num1)
```

```
print("число 1 возведено в степень 2:", power_num1)
```

```
print("Округленное значение числа 2:", rounded_num2)
```

Основы Python

4) Функции Print() и input()

```
user_input = input("Введите номер: ")  
print("Вы вошли:", user_input)
```

5) Преобразование строк в числа int() и float()

try:

```
num_input = int(user_input)  
print("Преобразовано в целое число:", num_input)
```

except ValueError:

```
print("Ввод не является допустимым целым числом.")
```

6) Логический тип bool, операторы сравнения и операторы and или not.

```
x = 10
```

```
y = 5
```

```
is_greater = x > y
```

```
is_equal = x == y
```

```
is_not_equal = x != y
```

```
print("X больше, чем y?", is_greater)
```

```
print("Является ли x равным y?", is_equal)
```

```
print("Разве x не равен y?", is_not_equal)
```

Использование логических операторов

```
is_true = True
```

```
is_false = False
```

```
result_and = is_true and is_false
```

```
result_or = is_true or is_false
```

```
result_not = not is_true
```

```
print("верно и неверно:", result_and)
```

```
print("верно или неверно:", result_or)
```

```
print("не правда:", result_not)
```

Основы Python

Имя: Алиса
Возраст: 30
Высота: 5.8
Студент: False
Добавление: 15
Вычитание: 5
Умножение: 50
Разделение: 2.0
Квадратный корень из числа 1: 3.1622776601683795
число 1 возведено в степень 2: 100.0
Округленное значение числа 2: 5
Введите номер: 30
Вы вошли: 30
Преобразовано в целое число: 30
X больше, чем y? True
Является ли x равным y? False
Разве x не равен y? True
верно и неверно: False
верно или неверно: True
не правда: False

Имя: Алиса
Возраст: 30
Высота: 5.8
Студент: False
Добавление: 15
Вычитание: 5
Умножение: 50
Разделение: 2.0
Квадратный корень из числа 1: 3.1622776601683795
число 1 возведено в степень 2: 100.0
Округленное значение числа 2: 5
Введите номер: Имеда

Основы Python

Вы вошли: Имеда

Ввод не является допустимым целым числом.

X больше, чем y? True

Является ли x равным y? False

Разве x не равен y? True

верно и неверно: False

верно или неверно: True

не правда: False

Упражнения -2: Строки и списки

Вот пример **Python**, охватывающий упомянутые вами темы, с подробным описанием каждой команды:

Теперь опишем каждую команду подробно:

1. Знакомство со строками. Основные операции со строками: Мы определяем две строки: `string1` и `string2`.
2. Введение в индексы и срезы строк:

`char_at_index` сохраняет первый символ строки `1`, используя индексацию. Подстрока сохраняет первые `5` символов строки `2` с использованием нарезки.

3. Основные строковые методы:

`Uppercase_string` преобразует строку `string1` в верхний регистр. `concatenated_string` объединяет строку `1` и строку `2`.

4. Специальные символы, экранирование символов, необработанные строки:

`escaped_string` включает в строку двойные кавычки с использованием escape-символа `.` `raw_string` — это необработанная строка, которая не экранирует специальные символы, такие как `.`

5. Форматирование строк: метод форматирования и F-строки:

Основы Python

`formatted_string` использует метод `format` для вставки значений в строку шаблона. `f_string` — это f-строка, которая непосредственно встраивает значения в строку.

6. Списки — операторы и функции для работы с ними:

`my_list` — список, содержащий целые числа от 1 до 5.

7. Нарезка списка и сравнение списков:

`Sliced_list` хранит фрагмент `my_list` с индексом от 1 до 3. `list_comparison` проверяет, равен ли `my_list` [1, 2]

8. Основные методы списка:

`add` добавляет элемент (6) в конец `my_list`. `Remove` удаляет первое вхождение числа 3 из `my_list`. `len` вычисляет длину `my_list`.

9. Вложенные списки, многомерные списки:

`nested_list` — это список, содержащий списки в качестве своих элементов, образующий вложенный или многомерный список.

Наконец, операторы **печати** отображают на экране результаты каждой операции или переменной.

1) Знакомство со строками. Основные операции со строками

`string1 = "Привет, "`

`string2 = "Мир!"`

2) Введение в индексы и срезы строк

`char_at_index = string1[0]` # Доступ к символу с индексом 0 (первый символ)

`substring = string2[0:5]` # Нарезка для получения символов от индекса от 0 до 4 (5-й символ)

3) Основные строковые методы

Основы Python

```
uppercase_string = string1.upper()      # Преобразовать строку в верхний
регистр
concatenated_string = string1 + string2  # Объединить две строки

# 4) Специальные символы, экранирование символов, необработанные строки
escaped_string = "Это \"цитировать\""  # Использование escape-символа для
включения кавычек
raw_string = r"C:\Users\Username\Documents" # Создание необработанной
строки (экранирование не требуется)

# 5) Форматирование строк: метод форматирования и F-строки
formatted_string = "Меня зовут {} и я {} лет.".format("Алиса", 25)
f_string = f"Меня зовут {Имеда} и я {55} лет."

# 6) Списки — операторы и функции для работы с ними
my_list = [1, 2, 3, 4, 5]

# 7) Нарезка списка и сравнение списков
sliced_list = my_list[1:4]              # Нарезка для получения элементов с индекса от
1 до 3
list_comparison = my_list == [1, 2]     # Сравнение, если список равен [1, 2]

# 8) Основные методы списка
my_list.append(6)                       # Добавление элемента в конец списка
my_list.remove(3)                       # Удаление первого появления значения 3
list_length = len(my_list)              # Получение длины списка

# 9) Вложенные списки, многомерные списки
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Создание вложенного списка

# Печать результатов
print("1):", string1, string2)
print("2):", char_at_index, substring)
print("3):", uppercase_string, concatenated_string)
```

Основы Python

```
print("4:", escaped_string, raw_string)
print("5:", formatted_string, f_string)
print("6:", my_list)
print("7:", sliced_list, list_comparison)
print("8:", my_list, list_length)
print("9:", nested_list)
```

1): Привет, Мир!
2): П Мир!
3): ПРИВЕТ, Привет, Мир!
4): Это "цитировать" C:\Users\Username\Documents
5): Меня зовут Алиса и я 25 лет. Меня зовут Имеда и я 55 лет.
6): [1, 2, 4, 5, 6]
7): [2, 3, 4] False
8): [1, 2, 4, 5, 6] 5
9): [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Упражнения -3: Условные операторы, циклы, генераторы списков

Вот пример **Python**, охватывающий упомянутые вами темы, с подробным описанием каждой команды:

Теперь опишем каждую команду подробно:

1. **Условное утверждение если. конструкция if-else:**
Мы устанавливаем **x** равным **10** и используем оператор **if**, чтобы проверить, больше ли **x** **5**. Если **true**, он печатает сообщение; в противном случае он печатает другое сообщение.
2. **Вложенные условия и множественный выбор. конструкция if-elif-else:**
Программа присваивает значение оценки и использует **if-elif-else** для определения оценки на основе этого значения.
3. **Тернарный условный оператор. Вложенное троичное условие:**
Он присваивает значение активности на основе значения **is_sunny**, используя троичное условное выражение.

Основы Python

4. Оператор цикла **while**:

Он демонстрирует цикл **while**, который выполняется до тех пор, пока **count** меньше или равно 5.

5. Операторы цикла прерывают, продолжают и еще:

Он использует цикл **for** с операторами **Break** и **continue** для управления выполнением цикла. Он также показывает блок **else**, который выполняется, когда цикл завершается нормально (**без останова**).

6. Оператор цикла **for**. функция диапазона():

Цикл **for** используется с функцией **range()** для итерации от 2 до 5.

7. Примеры оператора цикла **for**. функция перечисления():

Цикл **for** перебирает список фруктов и использует **enumerate()** для доступа как к индексу, так и к элементу.

8. Итератор и итерируемые объекты. Функции **iter()** и **next()**:

Он демонстрирует создание итератора с помощью **iter()** и получение значений с помощью **next()**.

9. Вложенные циклы. Примеры проблем с вложенными циклами:

Он использует вложенные циклы **for** для печати комбинаций (i, j).

10. Треугольник Паскаля как пример работы вложенных циклов:

Он генерирует треугольник Паскаля, используя вложенные циклы, с помощью функции **generate_pascals_triangle()**.

11. Список понятий:

Он создает список квадратов чисел, используя понимание списка.

12. Генераторы вложенных списков:

Он генерирует матрицу, используя вложенные списки.

Каждая команда демонстрирует различную концепцию или использование в **Python**: от условий и циклов до списков и вложенных структур.

```
# 1) Условное утверждение если. конструкция if-else
```

```
x = 10
```

```
if x > 5:
```

```
    print("x больше, чем 5")
```


Основы Python

```
else:
    print("x не больше, чем 5")

# 2) Вложенные условия и множественный выбор. конструкция if-elif-else
grade = 85
if grade >= 90:
    print("A")
elif grade >= 80:
    print("B")
elif grade >= 70:
    print("C")
else:
    print("D")

# 3) Тернарный условный оператор. Вложенное троичное условие
is_sunny = True
activity = "Выйти на улицу" if is_sunny else "Остаться дома"

# 4) Оператор цикла while
count = 1
while count <= 5:
    print(f"Считать: {count}")
    count += 1

# 5) Операторы цикла прерывают, продолжают и еще
for i in range(10):
    if i == 3:
        break
    if i == 1:
        continue
    print(i)
else:
    print("Цикл завершен без встречи с оператором прерывания")
```

Основы Python

6) Оператор цикла for. функция диапазона()

```
for i in range(2, 6):  
    print(i)
```

7) Примеры оператора цикла for. функция перечисления()

```
fruits = ["яблоко", "банан", "вишня"]  
for index, fruit in enumerate(fruits):  
    print(f"Index {index}: {fruit}")
```

8) Итератор и итерируемые объекты. функции iter() и next()

```
my_list = [1, 2, 3]  
my_iter = iter(my_list)  
print(next(my_iter)) # Выход: 1  
print(next(my_iter)) # Выход: 2
```

9) Вложенные циклы. Примеры проблем с вложенными циклами

```
for i in range(3):  
    for j in range(2):  
        print(f"({i}, {j})")
```

10) Треугольник Паскаля как пример работы вложенных циклов

```
def generate_pascals_triangle(n):  
    triangle = []  
    for i in range(n):  
        row = [1]  
        if i > 0:  
            for j in range(1, i):  
                row.append(triangle[i - 1][j - 1] + triangle[i - 1][j])  
            row.append(1)  
        triangle.append(row)  
    return triangle
```

```
pascals_triangle = generate_pascals_triangle(5)  
for row in pascals_triangle:
```

Основы Python

```
print(row)
```

11) Список понятий

```
squared_numbers = [x ** 2 for x in range(1, 6)]
```

12) Генераторы вложенных списков

```
matrix = [[i * j for j in range(1, 4)] for i in range(1, 4)]
```

x больше, чем 5

В

Считать: 1

Считать: 2

Считать: 3

Считать: 4

Считать: 5

0

2

2

3

4

5

Index 0: яблоко

Index 1: банан

Index 2: вишня

1

2

(0, 0)

(0, 1)

(1, 0)

(1, 1)

(2, 0)

(2, 1)

[1]

Основы Python

```
[1, 1]  
[1, 2, 1]  
[1, 3, 3, 1]  
[1, 4, 6, 4, 1]
```

Упражнения -4: Словари, кортежи и множества

Вот пример **Python**, охватывающий упомянутые вами темы, с подробным описанием каждой команды:

Теперь опишем каждую команду подробно:

1. **Знакомство со словарями (dict). Основные операции со словарями:**
студент — словарь с информацией о студенте. Доступ к элементам с помощью **ключей** и **метода get()**. Изменение элементов словаря и добавление новой пары ключ-значение. Удаление элемента словаря с помощью **del**.
2. **Словарные методы, перебирающие элементы словаря:**
Демонстрирует такие методы, как **keys()**, **values()** и **items()**, для извлечения ключей, значений и пар ключ-значение из словаря. Перебор элементов словаря с использованием цикла **for**.
3. **Кортежи и их методы:**
my_tuple — это кортеж, содержащий целые числа. Доступ к элементам по индексу и нарезка кортежа.
4. **Множества и их методы:**
my_set — это набор, содержащий целые числа. Добавление и удаление элементов из набора с помощью **add()** и **Remove()**.
5. **Операции над множествами, сравнение множеств:**
Демонстрирует операции над множествами, такие как объединение, пересечение и разность. Проверяет, является ли один набор подмножеством другого, используя **issubset()**.
6. **Генераторы наборов и генераторы словарей:**
Использование понимания множеств для создания набора квадратов

Основы Python

чисел. Использование словарного понимания для создания словаря квадратов чисел с соответствующими ключами.

Каждая команда демонстрирует различные операции и методы, связанные со словарями, кортежами и наборами, включая создание, доступ, модификацию и итерацию.

1) Знакомство со словарями (dict). Основные операции со словарями

```
student = {  
    "имя": "Алиса",  
    "возраст": 25,  
    "главный": "Информатика"  
}
```

Доступ к элементам словаря

```
name = student["имя"]  
age = student.get("возраст")
```

Изменение элементов словаря

```
student["возраст"] = 26  
student["год"] = 3
```

Удаление элементов словаря

```
del student["главный"]
```

2) Словарные методы, циклически перебирающие элементы словаря

```
keys = student.keys()      # Получить ключи словаря  
values = student.values()  # Получить словарные значения  
items = student.items()    # Получить пары ключ-значение
```

```
for key in student:  
    print(key, student[key])
```

Основы Python

3) Кортежи и их методы

```
my_tuple = (1, 2, 3, 4, 5)
```

Доступ к элементам кортежа

```
element = my_tuple[2]
```

Нарезка кортежа

```
sliced_tuple = my_tuple[1:4]
```

4) Множества и их методы

```
my_set = {1, 2, 3, 4, 5}
```

Добавление и удаление элементов из набора

```
my_set.add(6)
```

```
my_set.remove(3)
```

5) Операции над множествами, сравнение множеств

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}
```

```
union_set = set1.union(set2) # Союз наборов
```

```
intersection_set = set1.intersection(set2) # Пересечение множеств
```

```
difference_set = set1.difference(set2) # Разница наборов
```

```
is_subset = set1.issubset(set2) # Проверьте, является ли set1 подмножеством set2
```

6) Генераторы множеств и генераторы словарей

```
squares_set = {x ** 2 for x in range(1, 6)} # Установить понимание
```

```
dict_squared = {x: x ** 2 for x in range(1, 6)} # Понимание словаря
```

имя Алиса

Основы Python

возраст 26

год 3

Упражнения -5: Функции

Вот пример **Python**, охватывающий упомянутые вами темы, с подробным описанием каждой команды:

1) Функции: первое введение, определение определения и их вызов.

```
def greet(name):  
    return f"Привет, {name}!"
```

```
greeting = greet("Алиса")  
print(greeting)
```

2) Оператор возврата в функциях. Функциональное программирование

```
def square(x):  
    return x ** 2
```

```
result = square(5)  
print(result)
```

3) Евклидов алгоритм поиска НОД

```
def gcd(a, b):  
    while b:  
        a, b = b, a % b  
    return a
```

```
result = gcd(48, 18)  
print(result)
```

4) Именованные аргументы. Фактические и формальные параметры

```
def multiply(a, b):  
    return a * b
```

Основы Python

```
result = multiply(b=3, a=4)
print(result)
```

5) Функции с произвольным количеством параметров *args и **kwargs

```
def add_numbers(*args):
    return sum(args)
```

```
sum_result = add_numbers(1, 2, 3, 4, 5)
print(sum_result)
```

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

```
print_info(имя="Алиса", возраст=25)
```

6) Операторы * и ** для упаковки и распаковки коллекций

```
numbers = [1, 2, 3, 4, 5]
print(*numbers) # Распаковка списка
```

```
def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result
```

```
nums = (2, 3, 4)
result = multiply(*nums) # Упаковка аргументов
```

7) Рекурсивные функции

```
def factorial(n):
    if n == 0:
        return 1
    else:
```


Основы Python

```
    return n * factorial(n - 1)

result = factorial(5)
print(result)

# 8) Анонимные (лямбда) функции
multiply = lambda x, y: x * y
result = multiply(3, 4)
print(result)

# 9) Область видимости переменных. Ключевые слова глобальные и
нелокальные
x = 10 # Глобальная переменная

def modify_global():
    global x
    x = 20

modify_global()
print(x) # x сейчас 20

# 10) Замыкания в Python
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
result = closure(5)
print(result)

# 11) Введение в декораторы функций
def my_decorator(func):
    def wrapper():
```

Основы Python

```
print("Что-то происходит до вызова функции.")
func()
print("Что-то происходит после вызова функции.")
return wrapper

@my_decorator
def say_hello():
    print("Привет!")

say_hello()

# 12) Декораторы с параметрами. Сохранение свойств декорированных функций
def repeat(num):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(num):
                func(*args, **kwargs)
            return wrapper
        return decorator

@repeat(3)
def greet(name):
    print(f"Привет, {name}!")

greet("Алиса")
```

Привет, Алиса!

25

6

12

15

Основы Python

имя: Алиса

возраст: 25

1 2 3 4 5

120

12

20

15

Что-то происходит до вызова функции.

Привет!

Что-то происходит после вызова функции.

Привет, Алиса!

Привет, Алиса!

Привет, Алиса!

Теперь опишем каждую команду подробно:

1. **Функции: первое введение, определение определения и их вызов:**
Определяет функцию **Greeting**, которая принимает один аргумент (**имя**) и возвращает приветственное сообщение. Вызывает функцию приветствия с аргументом «**Алиса**» и сохраняет результат в приветствии.
2. **Оператор возврата в функциях. Функциональное программирование:**
Определяет функцию **Square**, которая вычисляет квадрат числа. Вызывает функцию **Square** с аргументом **5** и сохраняет результат в **result**.
3. **Евклидов алгоритм нахождения НОД:**
Определяет функцию **НОД**, которая вычисляет наибольший общий делитель двух чисел с помощью алгоритма **Евклида**. Вызывает функцию **gcd** с аргументами **48** и **18** и сохраняет результат в **result**.
4. **Именованные аргументы. Фактические и формальные параметры:**
Определяет функцию умножения, которая принимает два именованных аргумента (**a** и **b**) и возвращает их произведение. Вызывает функцию умножения с именованными аргументами и сохраняет результат в **result**.
5. **Функции с произвольным количеством параметров args и kwargs:**
Определяет функцию **add_numbers**, которая принимает любое

Основы Python

количество аргументов и возвращает их сумму. Определяет функцию `print_info`, которая принимает аргументы ключевого слова и печатает их.

6. Операторы "и" для упаковки и распаковки коллекций:

Демонстрирует использование операторов `и` для упаковки и распаковки аргументов. Определяет функцию умножения, которая принимает произвольное количество аргументов и возвращает их произведение.

7. Рекурсивные функции:

Определяет рекурсивную функцию факториал для вычисления факториала числа. Вызывает функцию факториала с аргументом `5` и сохраняет результат в `result`

8. Анонимные (лямбда) функции:

Определяет анонимную функцию (лямбда) умножения для вычисления произведения двух чисел. Вызывает лямбда-функцию с аргументами `3` и `4` и сохраняет результат в `result`.

9. Область видимости переменных. Ключевые слова глобальные и нелокальные:

Демонстрирует использование глобальных и нелокальных ключевых слов для области видимости переменных.

10.Замыкания в Python:

Определяет замыкание с использованием вложенных функций и возвращает внутреннюю функцию. Вызывает замыкание с аргументом и сохраняет результат в `result`.

11.Введение в декораторы функций:

Определяет декоратор `my_decorator`, который оборачивает функцию и добавляет поведение до и после вызова функции.

Применяет `my_decorator` **к функции `say_hello`, используя символ `**@`.

12.Декораторы с параметрами. Сохранение свойств декорированных функций:

Определяет повторение декоратора, которое принимает параметр `num` для повторения функции несколько раз. Применяет декоратор повтора к функции приветствия, чтобы повторить ее три раза при вызове с аргументом «Алиса».

Основы Python

Упражнения -6: Простое программа интерактивное образовательное меню.

```
!pip install ipywidgets
!apt-get install -y python-opengl
from IPython.display import YouTubeVideo

# Определите идентификаторы видео YouTube в том же порядке, что и в
меню
video_ids = [
    "btuxcr7Sxw4",
    "fb5f0jX9tng",
    "qJwkaKgNxfE",
]

# Определите меню тем обучения
training_topics = [
    "Тема 1: Введение в Python — установка на компьютер Python для
начинающих",
    "Тема 2: Параметры выполнения команд — переход на PyCharm Python
для начинающих",
    "Тема 3: Переменные, оператор присваивания, функции type() и
id() Python для начинающих",
    "Выход"
]

while True:
    # Отобразить меню
    print("\nМеню программы обучения:")
    for idx, topic in enumerate(training_topics, start=1):
        print(f"{idx}. {topic}")

    # Получите выбор пользователя
    choice = input("Введите номер темы, которую хотите изучить (или «exit» для
выхода): ")
```

Основы Python

```
# Преобразуйте пользовательский ввод в нижний регистр для сравнения
без учета регистра
choice = choice.lower()

if choice == 'Выход':
    print("Выход из программы. До свидания!")
    break

# Преобразуйте выбор в целое число (при условии, что это допустимое
число)
try:
    choice = int(choice)
except ValueError:
    print("Неверный Ввод. Пожалуйста, введите действительный номер темы
или «выйдите»")
    continue

# Проверьте, находится ли выбор в допустимом диапазоне
if 1 <= choice <= len(training_topics):
    if choice == len(training_topics):
        print("Выход из программы. До свидания!")
        break

    selected_topic = training_topics[choice - 1]
    youtube_video_id = video_ids[choice - 1]

    print(f"Вступительное видео для {selected_topic}...")
    display(YouTubeVideo(youtube_video_id, width=640, height=360,
autoplay=True))
else:
    print("Неверный номер темы. Пожалуйста, выберите действительный
номер темы или «выйдите».")
```

Основы Python

Вот описание каждой части программы:

1. **!pip install ipywidgets** и **!apt-get install -y python-opengl**: эти команды используются для установки необходимых пакетов Python. **ipywidgets** используется для интерактивных виджетов (хотя он не используется в этой конкретной программе), а **python-opengl** обеспечивает поддержку OpenGL в Python.
2. **from IPython.display import YouTubeVideo**: эта строка импортирует класс **YouTubeVideo** из модуля **IPython.display**. Этот класс позволяет встраивать и отображать видео YouTube непосредственно в среде Jupyter Notebook или IPython.
3. **video_ids**: Этот список содержит идентификаторы видео YouTube. Каждый идентификатор соответствует определенному видео YouTube, которое представляет собой введение в тему.
4. **training_topics**: В этом списке содержится меню тем обучения. Каждый элемент представляет собой строку, представляющую тему, которую пользователи могут выбирать. Последний пункт «Выход» (Выход) предназначен для выхода из программы.
5. **The while True: цикл**: создается бесконечный цикл, который продолжается до тех пор, пока пользователь не решит выйти, набрав «ВЫХОД» или выбрав опцию **выхода** из меню.
6. **Displaying the menu**: внутри цикла программа отображает пользователю меню тем обучения, перебирая список **Training_topics** и печатая каждую тему с соответствующим номером.
7. **Getting user input**: Программа предлагает пользователю ввести номер темы, которую он хочет изучить, или ввести «ВЫХОД», чтобы выйти из программы. Затем он считывает введенные пользователем данные.
8. **Lowercasing the input**: Пользовательский ввод преобразуется в нижний регистр с помощью **choice = choice.lower()**, чтобы обеспечить сравнение без учета регистра.
9. **Checking for exit**: Если пользователь вводит «Выход» (Выход), программа печатает прощальное сообщение и выходит из цикла, фактически завершая программу.

Основы Python

10. **Converting input to an integer:** Программа пытается преобразовать ввод пользователя в целое число, используя блок **try-Exception**. Если входные данные не являются допустимым целым числом, отображается сообщение об ошибке и происходит переход к следующей итерации цикла.
11. **Checking for valid input:** Если входные данные являются допустимым целым числом, программа проверяет, находится ли выбранное число в допустимом диапазоне (от 1 до количества тем обучения). Если оно действительно, программа продолжает отображать соответствующее видео **YouTube**, используя класс **YouTubeVideo**. Если выбранное число соответствует варианту **выхода**, программа завершается.
12. **Displaying the video:** Программа отображает вступительное видео по выбранной теме, создавая объект **YouTubeVideo** с соответствующим идентификатором видео, а затем отображая его с заданной шириной, высотой и настройками автозапуска.

Эта программа предоставляет удобный способ доступа к образовательному контенту, позволяя пользователям выбирать темы и просматривать соответствующие видеоролики непосредственно в среде **Jupyter Notebook** или **IPython**.

Упражнения -7: Простая программа Python, которая использует различные основные команды Python

```
# Определить переменные
name = "Джон"
age = 30
height = 175.5
is_student = True

# Базовый ввод и вывод
print("Привет, " + name + "!")
print("Тебе", age, "лет.")
print("Ваш рост", height, "см.")
if is_student:
```


Основы Python

```
print("Ты студент.")
else:
    print("Вы не студент.")

# Основные расчеты
num1 = 10
num2 = 5
sum_result = num1 + num2
difference_result = num1 - num2
product_result = num1 * num2
division_result = num1 / num2

print("Сумма:", sum_result)
print("Разница:", difference_result)
print("Продукт:", product_result)
print("Разделение:", division_result)

# Условные операторы
if age < 18:
    print("Ты несовершеннолетний.")
elif age >= 18 and age < 65:
    print("Ты взрослый.")
else:
    print("Вы пенсионер.")

# Зацикливание
print("Считая от 1 к 5:")
for i in range(1, 6):
    print(i)

# Списки
fruits = ["яблоко", "банан", "вишня"]
fruits.append("апельсин")
print("Фрукты:", fruits)
```

Основы Python

```
# Функции
def square(x):
    return x * x

result = square(5)
print("Площадь 5:", result)

# Словари
student = {
    "имя": "Алиса",
    "возраст": 22,
    "главный": "Информатика"
}
print("Имя студента:", student["имя"])

# Манипулирование строками
message = "Это простая программа на Python."
print("Верхний регистр:", message.upper())
print("Строчные буквы:", message.lower())

try:
    index = message.index("простой")
    print("Слово «простой» находится в индексе:", index)
except ValueError:
    print("Слово «простой» не найдено в строке.")

# Обработка файлов
file_name = "sample.txt"

# Open the file in write mode (use "w" for write mode)
with open(file_name, "w") as file:
    file.write("Это пример файла.")
```

Основы Python

```
# Open the file in read mode (use "r" for read mode)
with open(file_name, "r") as file:
    contents = file.read()
    print("Содержимое файла:", contents)

# Обработка ошибок
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Ошибка деления на ноль.")
```

Привет, Джон!

Тебе 30 лет.

Ваш рост 175.5 см.

Ты студент.

Сумма: 15

Разница: 5

Продукт: 50

Разделение: 2.0

Ты взрослый.

Считая от 1 к 5:

1

2

3

4

5

Фрукты: ['яблоко', 'банан', 'вишня', 'апельсин']

Площадь 5: 25

Имя студента: Алиса

Верхний регистр: ЭТО ПРОСТАЯ ПРОГРАММА НА PYTHON.

Строчные буквы: это простая программа на python.

Слово «простой» не найдено в строке.

Содержимое файла: Это пример файла.

Основы Python

Ошибка деления на ноль.

Эта программа охватывает основные концепции **Python**, включая переменные, ввод/вывод, вычисления, условные операторы, циклы, списки, функции, словари, манипуляции со строками, обработку файлов и обработку ошибок. Вы можете запустить эту программу, чтобы увидеть, как эти основные команды **Python** работают вместе

Упражнения - 8: Программа на Python, которая использует API шуток для извлечения и отображения случайных шуток

Вот программа на **Python**, которая использует API шуток для извлечения и отображения случайных шуток:

Чтобы запустить этот код, вам необходимо установить библиотеку **requests**:

```
!pip install requests
```

```
import requests
```

```
def fetch_random_joke():
```

```
    url = "https://official-joke-api.appspot.com/random_joke"
```

```
    response = requests.get(url)
```

```
    if response.status_code == 200:
```

```
        joke_data = response.json()
```

```
        setup = joke_data["setup"]
```

```
        punchline = joke_data["punchline"]
```

```
        return setup, punchline
```

```
    else:
```

```
        return "Извините, на этот раз я не смог подобрать шутку.", ""
```

```
def main():
```

```
    print("Добро пожаловать в генератор шуток!")
```

Основы Python

```
while True:
    input("Нажмите Enter, чтобы получить случайную шутку...")
    setup, punchline = fetch_random_joke()

    print("\nШутка:")
    print(setup)
    input("Нажмите Enter, чтобы открыть прикол...")
    print("Прикол:")
    print(punchline)

    another_joke = input("\nХотите услышать еще одну шутку? (да/нет): ")
    if another_joke.lower() != "да":
        print("Благодарим вас за использование генератора шуток!")
        break

if __name__ == "__main__":
    main()

# Дополнительная шутка
print("\nДополнительная шутка:")
print("Почему букашки такие дружелюбные?")
print("Потому что они нашли в вашем ноутбуке 'Ctrl' и 'C'!")

# Что смешного
print("\nЧто говорит печка микроволновка другой печке?")
print("- Ну как, брат, готова к очередной горячей встрече?")
```

Добро пожаловать в генератор шуток!

Нажмите Enter, чтобы получить случайную шутку...

Шутка:

Why are pirates called pirates?

Нажмите Enter, чтобы открыть прикол...

Прикол:

Основы Python

Because they arrrr!

Хотите услышать еще одну шутку? (да/нет): нет
Благодарим вас за использование генератора шуток!

Дополнительная шутка:

Почему букашки такие дружелюбные?

Потому что они нашли в вашем ноутбуке 'Ctrl' и 'C'!

Что говорит печка микроволновка другой печке?

- Ну как, брат, готова к очередной горячей встрече?

Эта программа извлекает шутки из «**Официального API шуток**» и отображает их пользователю. Он интерактивный и позволяет пользователю получить новую шутку и раскрыть кульминацию при каждом нажатии клавиши **Enter**. Пользователь может продолжать получать шутки до тех пор, пока не решит выйти из программы. Это веселая и интерактивная программа, способная развлечь широкую аудиторию.

Упражнения -9: Программа на **Python** от начала до конца, которая служит математическим калькулятором и графопостроителем

Вот программа на **Python** от начала до конца, которая служит математическим калькулятором и графопостроителем. В коде используются библиотеки **Sympy** и **Matplotlib** для выполнения символьных математических вычислений и построения математических функций:

```
!pip install sympy matplotlib
```

```
import sympy as sp
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
def main():
```

```
    print("Математический калькулятор и графический редактор")
```

Основы Python

```
while True:
    expression = input("Введите математическое выражение (или «выход», чтобы выйти): ")

    if expression.lower() == "exit":
        break

    try:
        x = sp.symbols('x')
        equation = sp.sympify(expression)
        sp.pprint(equation)

        # Решите уравнение символически
        solution = sp.solve(equation, x)
        print("Символическое решение:")
        sp.pprint(solution)

        # Постройте график
        plot_graph(equation)
    except Exception as e:
        print(f"Ошибка: {e}")

def plot_graph(equation):
    x = np.linspace(-10, 10, 400) # Создайте диапазон значений x
    y = [equation.subs('x', xi).evalf() for xi in x] # Оцените уравнение для каждого x

    plt.figure(figsize=(8, 6))
    plt.plot(x, y, label=str(equation), color='b')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('График уравнения')
    plt.grid(True)
    plt.legend()
```

Основы Python

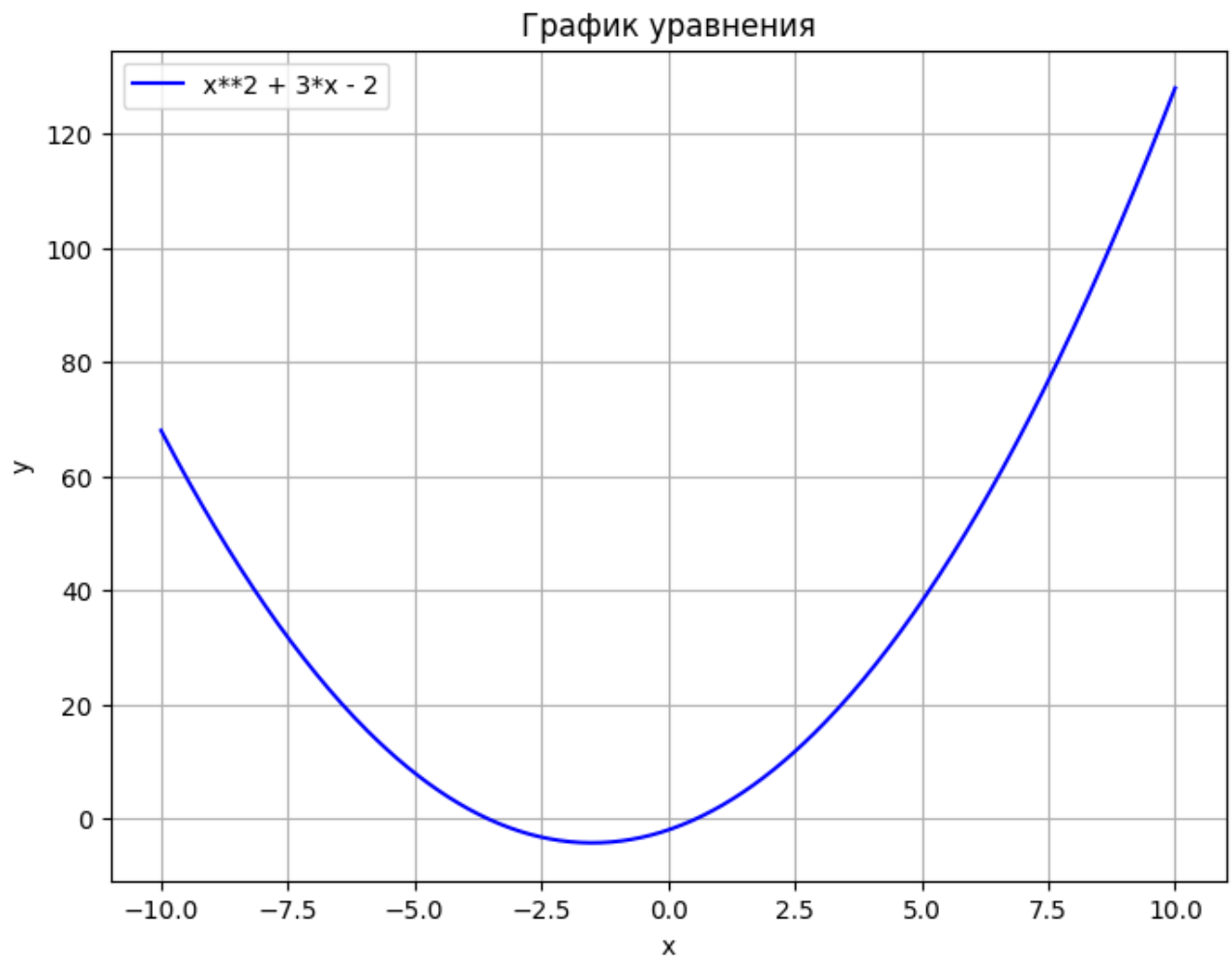
```
plt.show(block=True)
```

```
if __name__ == "__main__":  
    main()
```

$$x^2 + 3x - 2$$

Символическое решение:

$$\left[\frac{3 \pm \sqrt{17}}{2}, -\frac{\sqrt{17} \pm 3}{2} \right]$$

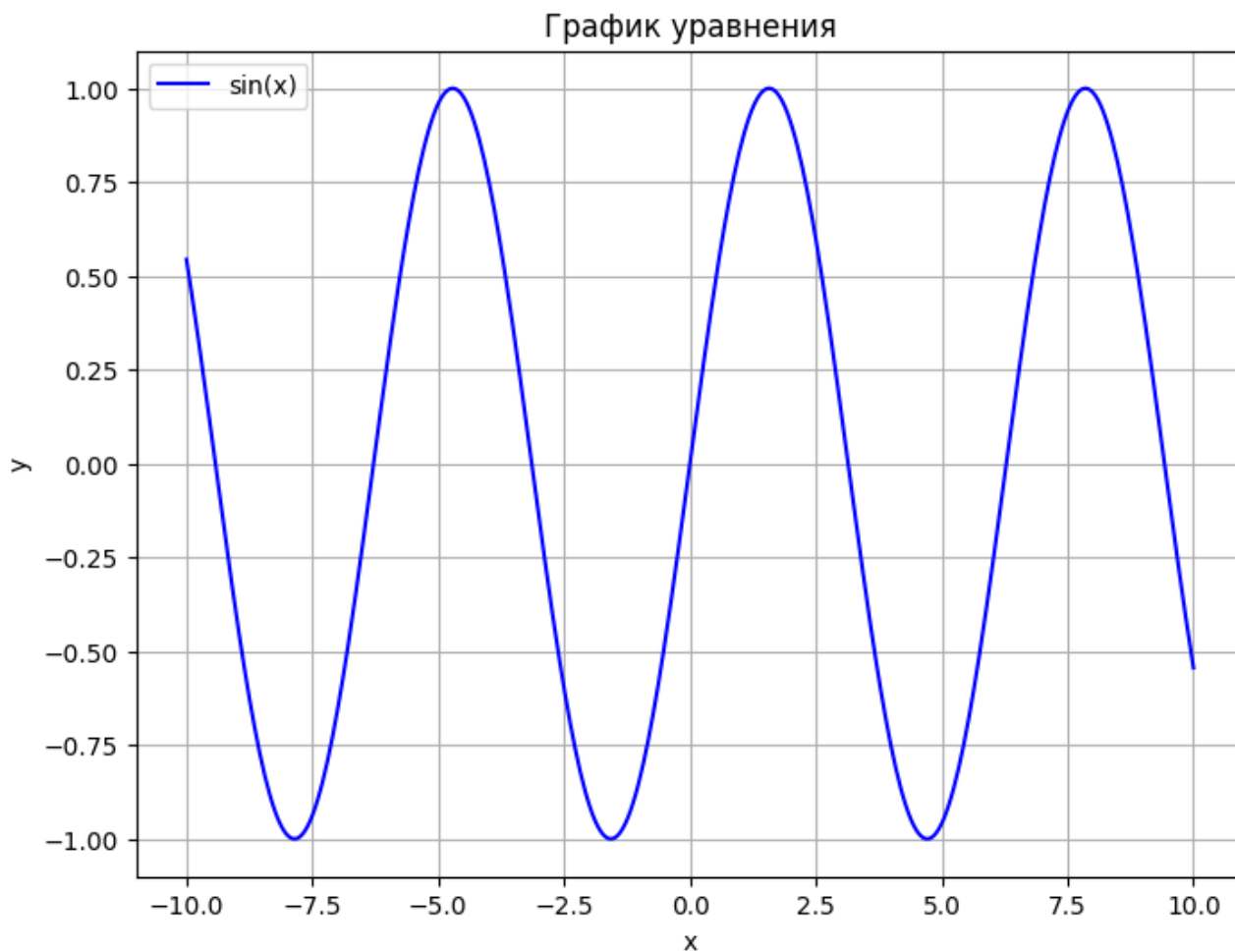


$\sin(x)$

Основы Python

Символическое решение:

$[0, \pi]$



Чтобы запустить эту программу, убедитесь, что у вас установлены библиотеки **Sympy** и **Matplotlib**. Вы можете установить их с помощью **pip**:

Вот как работает программа:

1. Запустите программу, и она предложит вам ввести математическое выражение.
2. Вы можете вводить математические выражения, такие как « $x^2 + 3x - 2$ » или « $\sin(x)$ », или любое другое допустимое математическое выражение.
3. Программа отобразит символическое представление выражения и его символическое решение (если оно разрешимо).
4. Он также построит график данного уравнения.

Основы Python

5. Чтобы выйти из программы, просто введите «exit».

Помните, что для просмотра графика важно запустить этот код в среде, поддерживающей графический вывод (например, **Jupyter Notebook**, IDE, такой как **Anaconda** или **PyCharm**, или скрипт **Python**, выполняемый в среде с возможностями графического пользовательского интерфейса).

Я подробно опишу каждую команду и раздел предоставленной программы **Python**:

1. **!pip install sympy matplotlib**: - Эта команда обычно запускается в командной строке или терминале, он использует менеджер пакетов **pip** для установки двух библиотек **Python**: **Sympy** (символическая математика) и **matplotlib** (библиотека построения графиков), **!** в начале команды используется для выполнения команд оболочки из **Jupyter Notebook** или среды **JupyterLab**. В стандартной среде **Python** эту команду можно запустить непосредственно из командной строки без знака **!**.
2. **import sympy as sp**: - Импортирует библиотеку **Sympy** и для удобства переименовывает ее в **sp**, **** Sympy**** — библиотека символьной математики и алгебраических манипуляций.
3. **import matplotlib.pyplot as plt**: - Импортирует модуль **matplotlib.pyplot** и переименовывает его в **plt**, **matplotlib** — широко используемая библиотека для создания графиков и визуализаций.
4. **import numpy as np**: - Импортирует библиотеку **numpy** и переименовывает ее в **np**, **numpy** обычно используется для числовых операций и создания массивов.
5. **def main()**:: - Определяет функцию с именем **main**, которая служит точкой входа в программу.
6. **print("Математический калькулятор и графический редактор")**: - Печатает сообщение на русском языке. Это название программы, которое переводится как «Математический калькулятор и редактор графиков».

Основы Python

7. **while True::** - Запускает бесконечный цикл для многократного запроса пользовательского ввода и обработки математических выражений, пока пользователь не решит выйти.
8. **expression = input("Введите математическое выражение (или «выход», чтобы выйти): ")**: - Предлагает пользователю ввести математическое выражение на русском языке или ввести «выход» (выход) для выхода из программы. Пользовательский ввод сохраняется в переменном выражении.
9. **if expression.lower() == "exit"::** - Проверяет, равен ли ввод пользователя, преобразованный в нижний регистр, «выходу». Если это так, программа выйдет из бесконечного цикла, позволяя пользователю выйти.
10. **x = sp.symbols('x')**: - Определяет символическую переменную «**x**» с помощью **Sympy**. Этот символ используется для обозначения неизвестных в символьной математике.
11. **equation = sp.simplify(expression)**: - Анализирует ввод пользователя (математическое выражение) и преобразует его в символьное уравнение с помощью **Sympy**.
12. **sp.pprint(equation)**: - **Pretty-выводит** символическое уравнение на консоль. Это отобразит уравнение в более удобном для чтения формате.
13. **solution = sp.solve(equation, x)**: - Символьно решает уравнение для «**x**», используя **Sympy**. Результат сохраняется в переменной **Solution**.
14. **print("Символическое решение:")**: - Печатает сообщение на русском языке, которое переводится как «**Символическое решение**».
15. **sp.pprint(solution)**: - **Pretty-выводит** символическое решение на консоль.
16. **plot_graph(equation)**: - Вызывает функцию **plot_graph** для создания и отображения графика для данного уравнения.
17. **except Exception as e::** - Начинает блок обработки исключений для обнаружения любых ошибок, которые могут возникнуть.
18. **print(f"Ошибка: {e}")**: - В случае возникновения ошибки вместе с сообщением об ошибке печатается сообщение об ошибке на русском языке, которое переводится как «**Ошибка**».
19. **def plot_graph(equation)::** - Определяет функцию с именем **plot_graph**, которая создает и отображает график для данного уравнения.

Основы Python

20. **Inside the plot_graph function:** - `x = np.linspace(-10, 10, 400)`: - Создает массив из 400 равномерно расположенных значений `x` от -10 до 10, используя `numpy`, `y = [equation.subs('x', xi).evalf() for xi in x]`: - Вычисляет символьное уравнение для каждого значения `x` в массиве, преобразуя результат в числовое значение с помощью `evalf`, `plt.figure(figsize=(8, 6))`: * - *Создает фигуру для графика заданного размера 8х6 дюймов*, `plt.plot(x, y, label=str(equation), color='b')`: - Строит график со значениями `x` и соответствующими значениями `y`, помечая график уравнением и устанавливая синий цвет, `plt.xlabel('x')` and `plt.ylabel('y')`: - Устанавливает метки осей `X` и `Y`, `plt.title('График уравнения')`: - Устанавливает заголовок графика на русском языке, что переводится как «График уравнения», `plt.grid(True)`: - Отображает линии сетки на графике, `plt.legend()`: - Добавляет легенду к графику, `plt.show(block=True)`: - Отображает график, оставляя окно открытым до тех пор, пока оно не будет закрыто вручную.
21. `if name == "main":`: - Проверяет, запускается ли скрипт как основная программа (а не импортируется как модуль).
22. `main()`: - Вызывает основную функцию, чтобы начать выполнение программы.

Эта программа позволяет пользователю вводить математические выражения, вычислять символьные решения и создавать графики для этих выражений, все на русском языке (Программу можно конвертировать на любой язык, это будет вашим домашним заданием).

Упражнения -10: Простая программа на **Python**, которая позволяет пользователю интерактивно решать арифметические задачи

Простая программа на **Python**, которая позволяет пользователю интерактивно решать арифметические задачи. Программа постоянно будет подсказывать пользователю проблемы и отображать решения. Основное внимание уделяется операциям сложения, вычитания, умножения и деления.

```
def main():  
    print("Интерактивное решение арифметических задач")
```

Основы Python

```
while True:
    print("\nВыберите операцию:")
    print("1. Добавление (+)")
    print("2. Вычитание (-)")
    print("3. Умножение (*)")
    print("4. Разделение (/)")
    print("5. Выход")

    choice = input("Введите свой выбор (1/2/3/4/5): ")

    if choice == "1":
        add()
    elif choice == "2":
        subtract()
    elif choice == "3":
        multiply()
    elif choice == "4":
        divide()
    elif choice == "5":
        print("До свидания!")
        break
    else:
        print("Неверный выбор. Пожалуйста, выберите действительный вариант.")

def add():
    num1 = float(input("Введите первое число: "))
    num2 = float(input("Введите второе число: "))
    result = num1 + num2
    print(f"Result: {num1} + {num2} = {result}")

def subtract():
    num1 = float(input("Введите первое число: "))
    num2 = float(input("Введите второе число: "))
```

Основы Python

```
result = num1 - num2
print(f"Result: {num1} - {num2} = {result}")

def multiply():
    num1 = float(input("Введите первое число: "))
    num2 = float(input("Введите второе число: "))
    result = num1 * num2
    print(f"Result: {num1} * {num2} = {result}")

def divide():
    num1 = float(input("Введите дивиденды: "))
    num2 = float(input("Введите делитель: "))
    if num2 == 0:
        print("Ошибка: деление на ноль не допускается..")
    else:
        result = num1 / num2
        print(f"Результат: {num1} / {num2} = {result}")

if __name__ == "__main__":
    main()
```

Интерактивное решение арифметических задач

Выберите операцию:

1. Добавление (+)
2. Вычитание (-)
3. Умножение (*)
4. Разделение (/)
5. Выход

Введите свой выбор (1/2/3/4/5): 1

Введите первое число: 10

Введите второе число: 25

Result: 10.0 + 25.0 = 35.0

Основы Python

Выберите операцию:

1. Добавление (+)
2. Вычитание (-)
3. Умножение (*)
4. Разделение (/)
5. Выход

Введите свой выбор (1/2/3/4/5): 2

Введите первое число: 35

Введите второе число: 20

Result: $35.0 - 20.0 = 15.0$

Выберите операцию:

1. Добавление (+)
2. Вычитание (-)
3. Умножение (*)
4. Разделение (/)
5. Выход

Введите свой выбор (1/2/3/4/5): 3

Введите первое число: 60

Введите второе число: 4

Result: $60.0 * 4.0 = 240.0$

Выберите операцию:

1. Добавление (+)
2. Вычитание (-)
3. Умножение (*)
4. Разделение (/)
5. Выход

Введите свой выбор (1/2/3/4/5): 4

Введите дивиденды: 80

Введите делитель: 4

Результатт: $80.0 / 4.0 = 20.0$

Выберите операцию:

Основы Python

1. Добавление (+)
2. Вычитание (-)
3. Умножение (*)
4. Разделение (/)
5. Выход

Введите свой выбор (1/2/3/4/5): 4

Введите дивиденды: 45

Введите делитель: 0

Ошибка: деление на ноль не допускается..

Выберите операцию:

1. Добавление (+)
2. Вычитание (-)
3. Умножение (*)
4. Разделение (/)
5. Выход

Введите свой выбор (1/2/3/4/5): 5

До свидания!

Эта программа предоставляет пользователям интерактивное меню для выбора арифметических операций и ввода чисел. Затем он отображает результат выбранной операции. Пользователи могут продолжить решение проблем или выйти из программы

Упражнения -11: Простая программа на **Python**, которая выполняет интеллектуальную задачу, моделируя базового чат-бота

Вот простая программа на **Python**, которая выполняет интеллектуальную задачу, моделируя базового **чат-бота**. Чат-бот может участвовать в разговоре, отвечать на вопросы и предоставлять заранее определенные ответы на основе ввода пользователя.

```
import random
```

```
def chatbot():
```


Основы Python

```
print("Привет! Я ваш дружелюбный чат-бот. Как я могу помочь вам  
сегодня?")

while True:
    user_input = input("Ты: ")

    if user_input.lower() in ["привет", "привет", "привет"]:
        response = "Чат-бот: Привет!"
    elif "Как вы" in user_input.lower():
        response = "Чат-бот: Я всего лишь программа, но спасибо, что  
спросили! Могу я чем-нибудь помочь?"
    elif "Ваше имя" in user_input.lower():
        response = "Чат-бот: Я всего лишь чат-бот, поэтому у меня нет имени,  
но вы можете называть меня Чат-бот!"
    elif "exit" in user_input.lower():
        response = "Чат-бот: До свидания! Если у вас возникнут  
дополнительные вопросы в будущем, не стесняйтесь вернуться."
        break
    else:
        responses = [
            "Чат-бот: Я не уверен, что понимаю. Не могли бы вы  
перефразировать это?",
            "Чат-бот: Я здесь, чтобы помочь. Что еще вы хотели бы знать?",
            "Чат-бот: Извините, я не могу ответить на этот вопрос.",
        ]
        response = random.choice(responses)

    print(response)

if __name__ == "__main__":
    chatbot()
```

Привет! Я ваш дружелюбный чат-бот. Как я могу помочь вам сегодня?
Ты: Я хочу изучать языка питон

Основы Python

Чат-бот: Я здесь, чтобы помочь. Что еще вы хотели бы знать?

Ты: я хочу изучать javascript

Чат-бот: Я здесь, чтобы помочь. Что еще вы хотели бы знать?

Ты: exit

Эта программа-чат-бот ведет базовый разговор с пользователем. Он предоставляет ответы на основе ввода пользователя. Вы можете расширить ответы и добавить более сложную логику, чтобы сделать чат-бот более умным и способным решать различные задачи и вопросы.

Посмотрите наши полные проекты здесь ==>

<https://github.com/ImedaSheriphadze>

Для получения более подробной информации свяжитесь с нами по

координатам: e-mail: isheriphadze@gmail.com

mob: +995(555) 45-92-70

Telegram: https://t.me/Imeda_Sheriphadze