Основы языка программирования Python

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Почему программисты используют Python?

Это самый типичный вопрос, который задают начинающие программисты, потому что на сегодняшний день существует масса других языков программирования. Учитывая, что число пользователей Python составляет порядка миллиона человек, достаточно сложно однозначно ответить на этот вопрос. Выбор средств разработки иногда зависит от уникальных особенностей и личных предпочтений.

Основные факторы, которые приводятся пользователями Python, примерно таковы:

- *Качество программного обеспечения*

Для многих основное преимущество языка Python заключается в удобочитаемости, ясности и более высоком качестве, отличающим его от других инструментов в мире языков сценариев. Программный код на языке Python читается легче, а значит, многократное его использование и обслуживание выполняется гораздо проще, чем использование программного кода на других языках сценариев. Единообразие оформления программного кода на языке Python облегчает его понимание даже для тех, кто не участвовал в его создании. Кроме того, Python поддерживает самые современные механизмы многократного использования программного кода, каким является объектно-ориентированное программирование (ООП).

- *Высокая скорость разработки*

По сравнению с компилирующими или строго типизированными языками, такими как C, C++ и Java, Python во много раз повышает производительность труда разработчика. Объем программного кода на языке Python обычно составляет треть или даже пятую часть эквивалентного программного кода на языке C++ или Java. Это означает меньший объем ввода с клавиатуры, меньшее количество времени на отладку и меньший объем трудозатрат на сопровождение. Кроме того, программы на языке Python запускаются сразу же, минуя длительные этапы компиляции и связывания, необходимые в некоторых других языках программирования, что еще больше увеличивает производительность труда программиста.

- *Переносимость программ*

Большая часть программ на языке Python выполняется без изменений на всех основных платформах. Перенос программного кода из операционной

системы Linux в Windows обычно заключается в простом копировании файлов программ с одной машины на другую. Более того, Python предоставляет массу возможностей по созданию переносимых графических интерфейсов, программ доступа к базам данных, веб-приложений и многих других типов программ. Даже интерфейсы операционных систем, включая способ запуска программ и обработку каталогов, в языке Python реализованы переносимым способом.

- *Библиотеки поддержки*

В составе Python поставляется большое число собранных и переносимых функциональных возможностей, известных как *стандартная библиотека*. Эта библиотека предоставляет массу возможностей, востребованных в прикладных программах, начиная от поиска текста по шаблону и заканчивая сетевыми функциями. Кроме того, Python допускает расширение как за счет ваших собственных библиотек, так и за счет библиотек, созданных сторонними разработчиками. Из числа сторонних разработок можно назвать инструменты создания веб-сайтов, программирование математических вычислений, доступ к последовательному порту, разработку игровых программ и многое другое. Например, расширение NumPy позиционируется как свободный и более мощный эквивалент системы программирования математических вычислений Mathlab.

- *Интеграция компонентов*

Сценарии Python легко могут взаимодействовать с другими частями приложения благодаря различным механизмам интеграции. Эта интеграция позволяет использовать Python для настройки и расширения функциональных возможностей программных продуктов. На сегодняшний день программный код на языке Python имеет возможность вызывать функции из библиотек на языке C/C++, сам вызываться из программ, написанных на языке C/C++, интегрироваться с программными компонентами на языке Java, взаимодействовать с такими платформами, как COM и .NET, и производить обмен данными через последовательный порт или по сети с помощью таких протоколов, как SOAP, XML-RPC и CORBA. Python – не обособленный инструмент.

- *Удовольствие*

Благодаря непринужденности языка Python и наличию встроенных инструментальных средств процесс программирования может приносить удовольствие. На первый взгляд это трудно назвать преимуществом, тем не менее,

удовольствие, получаемое от работы, напрямую влияет на производительность труда.

Из всех перечисленных факторов наиболее существенными для большинства пользователей являются первые два (качество и производительность).

Качество программного обеспечения

По своей природе Python имеет простой, удобочитаемый синтаксис и ясную модель программирования. Основное его преимущество состоит в том, что Python «каждому по плечу» – характеристики языка взаимодействуют ограниченным числом непротиворечивых способов и естественно вытекают из небольшого круга базовых концепций. Это делает язык простым в освоении, понимании и запоминании. На практике программистам, использующим язык Python, почти не приходится прибегать к справочным руководствам – это непротиворечивая система, на выходе которой, к удивлению многих, получается профессиональный программный код.

Философия Python по сути диктует использование минималистского подхода. Это означает, что даже при наличии нескольких вариантов решения задачи в этом языке обычно существует всего один очевидный путь, небольшое число менее очевидных альтернатив и несколько взаимосвязанных вариантов организации взаимодействий. Более того, Python не принимает решения за вас, когда порядок взаимодействий неочевиден – предпочтение отдается явному описанию, а не «волшебству». В терминах Python явное лучше неявного, а простое лучше сложного.

Помимо философии Python обладает такими возможностями, как модульное и объектно-ориентированное программирование, что естественно упрощает возможность многократного использования программного кода. Поскольку качество находится в центре внимания самого языка Python, оно также находится в центре внимания программистов.

Высокая скорость разработки

Во время бума развития Интернета во второй половине 1990-х годов было сложно найти достаточное число программистов для реализации программных проектов – от разработчиков требовалось писать программы со скоростью развития Интернета. Теперь, в эпоху экономического спада, картина изменилась.

Сегодня от программистов требуется умение решать те же задачи меньшим числом сотрудников.

В обоих этих случаях Python блистал как инструмент, позволяющий программистам получать большую отдачу при меньших усилиях. Он изначально оптимизирован для достижения *высокой скорости разработки* – простой синтаксис, динамическая типизация, отсутствие этапа компиляции и встроенные инструментальные средства позволяют программистам создавать программы за меньшее время, чем при использовании некоторых других инструментов. В результате Python увеличивает производительность труда разработчика во много раз по сравнению с традиционными языками программирования. Это значительное преимущество, которое с успехом может использоваться как во время бума, так и во время спада, а также во время любого промежуточного этапа развития индустрии программного обеспечения.

Является ли Python «языком сценариев»?

Python – это многоцелевой язык программирования, который зачастую используется для создания сценариев. Обычно он позиционируется как *объектно-ориентированный язык сценариев* – такое определение смешивает поддержку ООП с общей ориентацией на сценарии. Действительно, для обозначения файлов с программным кодом на языке Python программисты часто используют слово «сценарий» вместо слова «программа». Здесь термины «сценарий» и «программа» рассматриваются как взаимозаменяемые с некоторым предпочтением термина «сценарий» для обозначения простейших программ, помещающихся в единственный файл, и термина «программа» для обозначения более сложных приложений, программный код которых размещается в нескольких файлах.

Термин «язык сценариев» имеет множество различных толкований. Некоторые предпочитают вообще не применять его к языку Python. У большинства термин «язык сценариев» вызывает три разных ассоциации, из которых одни более применимы к языку Python, чем другие:

- *Командные оболочки*

Иногда, когда кто-то слышит, что Python – это язык сценариев, то представляет себе Python как инструмент для создания системных сценариев. Такие программы часто запускаются из командной строки с консоли и решают такие задачи, как обработка текстовых файлов и запуск других программ.

Программы на языке Python способны решать такие задачи, но это лишь одна из десятков прикладных областей, где может применяться Python. Это не только язык сценариев командной оболочки.

- *Управляющий язык*

Другие пользователи под названием «язык сценариев» понимают «связующий» слой, который используется для управления другими прикладными компонентами (то есть для описания сценария работы). Программы на языке Python действительно нередко используются в составе крупных приложений. Например, при проверке аппаратных устройств программы на языке Python могут вызывать компоненты, осуществляющие низкоуровневый доступ к устройствам. Точно так же программы могут запускать программный код на языке Python для поддержки настройки программного продукта у конечного пользователя, что ликвидирует необходимость поставлять и пересобирать полный объем исходных текстов.

Простота языка Python делает его весьма гибким инструментом управления. Тем не менее, технически – это лишь одна из многих ролей, которые может играть Python. Многие программисты пишут на языке Python автономные сценарии, которые не используют какие-либо интегрированные компоненты. Это не только язык управления.

- *Удобство в использовании*

Пожалуй, лучше всего представлять себе термин «язык сценариев» как обозначение простого языка, используемого для быстрого решения задач. Это особенно верно, когда термин применяется к языку Python, который позволяет вести разработку гораздо быстрее, чем компилирующие языки программирования, такие как C++. Ускоренный цикл разработки способствует применению зондирующего, поэтапного стиля программирования, который следует попробовать, чтобы оценить по достоинству.

Python предназначен не только для решения простых задач. Скорее, он упрощает решение задач благодаря своей простоте и гибкости. Язык Python имеет небольшой набор возможностей, но он позволяет создавать программы неограниченной сложности. По этой причине Python обычно используется как для быстрого решения тактических, так и для решения долговременных, стратегических задач.

Итак, является ли Python языком сценариев? Ответ зависит от того, к кому обращен вопрос. Вообще термин «создание сценариев», вероятно, лучше использовать для описания быстрого и гибкого способа разработки, который поддерживается языком Python, а не для описания прикладной области программирования.

Знакомство со средой

Что нам потребуется для выполнения программ на языке Python? Прежде, чем ответить на этот вопрос, рассмотрим, как запускаются программы на компьютере. Выполнение программ осуществляется операционной системой (Windows, Linux и пр.). В задачи операционной системы входит распределение ресурсов (оперативной памяти и пр.) для программы, запрет или разрешение на доступ к устройствам ввода/вывода и. т.д.

Для запуска программ на языке Python необходима программа-интерпретатор (виртуальная машина) Python. Данная программа скрывает от Python-программиста все особенности операционной системы, поэтому, написав программу на Python в системе Windows, ее можно запустить, например, в GNU/Linux и получить такой же результат.

Скачать и установить интерпретатор Python можно совершенно бесплатно с официального сайта: http://python.org. Для работы нам понадобится интерпретатор **Python версии 3** или выше 3.

Установив Python 3, вы получите в комплекте очень простую, но полезную IDE (Integrated Development Environment) под названием IDLE. Хотя есть много различных способов запустить код на Python, IDLE – это всё, что нужно для начала.

Запустите интерактивную графическую среду IDLE и дождитесь появления приглашения для ввода команд:

```
Type "copyright", "credits" or "license()" for more information.
>>>
```

В самом начале обучения Python можно представить как обычный интерактивный калькулятор. В интерактивном режиме IDLE найдем значения следующих математических выражений. После завершения набора выражения нажмите клавишу **`Enter`** для завершения ввода и вывода результата на экран.

```
>>> 3.0 + 6
9.0
>>> 4 + 9
13
>>> 1 - 5
-4
>>> + 6
6
>>>
```

Если по какой-либо причине совершить ошибку при вводе команды, то Python сообщит об этом:

```
>>> a
Traceback (most recent call last):
   File "<pyshell#0>", line 1, in <module>
     a
NameError: name 'a' is not defined
>>>
```

Не бойтесь совершать ошибки! Python поправит и подскажет, на что следует обратить внимание.

Можно заметить, что в этом интерактивном режиме нельзя внести изменения в выражение, которое уже ранее было выполнено. Приходится повторно набирать выражение и его запускать. В случае больших программ удобно использовать отдельные файлы с расширением .py. То есть, задача состоит в том, чтобы с помощью редактора создать файл с текстом программы. Для этого используем следующий способ.

В меню IDLE выберете File => New File. Появится окно текстового редактора, в котором можно набирать команды на языке Python.

Наберем следующий код:

```
a = 5
print(a)
print(a + 5)
```

Каждая новая команда пишется в новой строке.

В меню редактора выберем Save As и сохраним файл в произвольную директорию, указав имя primer1.py.

Чтобы выполнить программу в меню редактора выберем Run => Run Module (или нажмем <F5>). Результат работы программы отобразится в интерактивном режиме, например:

```
>>>
========== RESTART: C:/Python3/primer1.py ==========
5
10
>>>
```

Здесь нам следует познакомиться с функцией **print**, которая отображает содержимое переменных, переданных ей в качестве аргументов. Вспомните, что

в интерактивном режиме мы просто набирали имя переменной, что приводило к выводу на экран ее содержимого. Дело в том, что Python в интерактивном режиме самостоятельно подставляет вызов функции **print**, а в файле нам придется делать это вручную.

Команда `print()`

Она используется для вывода на экран. Внутри скобок через запятую мы указываем то, что необходимо вывести на экран. Если это какой-то текст, указываем его внутри кавычек. Кавычки могут быть как одинарными (английский апостроф), так и обычные двойные. Но обязательно, чтобы текст начинался и заканчивался кавычками одного типа. Команда **print** записывается только строчными буквами, другое написание недопустимо, так как в Python строчные и заглавные буквы считаются разными символами.

Вариант Python IDLE слишком прост, и для более серьёзной работы с Python есть другие среды разработки. Одна из них, довольно удобная для начинающих, – это **Wing-101**. Её можно скачать и установить с сайта https://wingware.com Там в пункте downloads надо выбрать установку под свою операционную систему и указать вариант для начинающих (for beginners).

В начале работы с указанной средой следует проверить настройки кодировок файлов. Сейчас широко используется в различных системах кодировка текста UTF-8, поэтому и здесь мы настроим редактор на эту кодировку.

Зайдите в меню **Edit** => **Preferences**, перейдите к категории **Files**. Для опции **Default Encoding** выберите значение **Unicode (UTF-8)**.

Команда `input()`

Она служит для ввода данных с клавиатуры. Так же, как и print(), всегда пишется с круглыми скобками. Команда работает так: когда программа доходит до места, где есть input(), она останавливается и ждёт, пока пользователь введёт строку с клавиатуры (ввод завершается нажатием клавиши Enter). Введённая строка подставляется на место input(). ‘ Поясним это примером:

```
print('Как тебя зовут?')
name = input()
print('Здравствуй,', name)
```

Запустите эту программу. Она выводит на экран строку «Как тебя зовут?» и дальше ждёт от пользователя ввода ответа на вопрос – ввода имени. Если вы ввели «Настя», то последует вывод на экран строки «Здравствуйте, Настя».

Команда присваивания

Знак «=» обозначает команду под названием «оператор присваивания». Оператор присваивает значение, которое имеет выражение справа от знака равно, переменной, которая находится слева от знака. То есть, вид команды следующий:

<имя переменной> = <выражение>

```
a = 5
name = input()
```

В языке Python разрешено *многократное присваивание* – в одном выражении используется несколько операторов присваивания. Примером такой ситуации может быть выражение

$x = y = 10$,

здесь переменным x и y присваивается значение 10.

А также можно применять *множественное присваивание* – это когда в левой части от оператора присваивания указано сразу несколько переменных. Примером такой ситуации может быть команда

$a, b = 1, 2$

В данном случае слева от оператора присваивания через запятую указаны переменные a и b, а справа (тоже через запятую) – значения 1 и 2. В результате переменная a получает значение 1, а переменная b получает значение 2. Принцип обработки такого рода выражений следующий: сначала вычисляются значения в правой части от оператора присваивания, а затем эти значения присваиваются переменным, указанным слева от оператора присваивания (количество переменных слева и количество значений справа должны совпадать). Поэтому, например, чтобы переменные a и b "обменялись" значениями, можно воспользоваться командой

$a, b = b, a$

Комментарии

Для удобства можно использовать комментарии, которые позволяют программисту делать для себя пометки или делать часть кода не выполнимой, не видимой для интерпретатора.

Если вы начнёте строку со знака решетки (#), то интерпретатор Python будет игнорировать всю эту строку. Программа будет выполняться так, как будто сроки нет. Такая строка называется *комментарием*.

Комментарии нужны в случаях:

1. Когда нужно добавить в программу какую-то пометку для человека, который будет читать эту программу (в том числе и для себя).
2. Когда нужно на время (в случае отладки, например) убрать какую-то строку из работы. Это называется «закомментировать» строку.

«Встрочные» комментарии находятся на той же строке, что и инструкция. Они должны отделяться по крайней мере двумя пробелами от инструкции. Если комментарии объясняют очевидное, то такие комментарии только отвлекают, и они не нужны. Пример неправильного комментария:

```
k = k + 1  # увеличение на 1
```

В Wing-101, если нужно закомментировать сразу несколько строчек подряд, то можно не делать это вручную, а воспользоваться соответствующей функцией в пункте меню **Source**. Выделите нужный фрагмент и выберите пункт **Toggle block comment**. Тем же способом можно и снять эти комментарии.

Переменные в Python

Мы уже имели дело с переменными. Но это было скорее "шапочное" знакомство. Здесь мы уделим переменным немного больше внимания. Надо сказать, они того заслуживают. Тем более что в Python переменные достаточно специфичны.

В первую очередь, что касается названия переменных: в принципе, это может быть практически любое имя (комбинация букв, цифр и символов подчеркивания), которое не совпадает ни с одним из ключевых слов Python.

Список ключевых слов Python представлен в таблице 1.

Таблица 1. Ключевые слова Python

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
deff	or	lambda	return	

Ключевые слова не могут быть модифицированы, и попытка использовать переменную с соответствующим именем приведет к ошибке.

Замечание

В данном случае мы не обсуждаем назначение ключевых слов. Список ключевых слов приведен исключительно для того, чтобы вы знали, как не следует называть переменные.

Помимо ключевых слов, в Python существует множество встроенных идентификаторов – таких, например, как названия встроенных функций (список не будем здесь приводить – он большой).

Формально мы можем задействовать переменную с именем, совпадающим с тем или иным идентификатором. Однако это может неожиданным образом повлиять на выполнении программы. Подобные ситуации считаются дурным тоном в программировании, и их следует избегать.

Имя переменной не может начинаться с цифры. Также крайне осторожно следует использовать подчеркивание в начале и конце имени переменной (переменные с начальным подчеркиванием и с двойным подчеркиванием в начале и конце имени обрабатываются по специальным правилам).

Для переменных не нужно явно объявлять тип. Однако это вовсе не означает, что типов данных в Python нет. Другое дело, что такое понятие как тип отождествляется именно с данными, а не с переменной.

В Python переменные ссылаются на данные, а не содержат их, как в некоторых других языках программирования. Каждая переменная "помнит", в каком месте в памяти находится некоторое значение. Это значение мы отождествляем со значением переменной (хотя на самом деле значением переменной является адрес ячейки памяти с соответствующими данными). Тем не менее, когда мы обращаемся к переменной, то выполняется автоматический переход по ссылке на данные, так что внешне иллюзия такая, как если бы переменная реально содержала определенное значение. Какие следствия из всего сказанного?

Во-первых, описанный выше механизм нередко является краеугольным камнем в понимании того, что происходит при выполнении программного кода и, в частности, при присваивании значений переменным.

Во-вторых, совершенно очевидно, что одна и та же переменная на разных этапах выполнения программы может ссылаться не просто на разные значения, но на значения разных типов (например, сначала ссылаться на текст, а затем на число).

В-третьих, хотя непосредственно у переменных типа нет, мы можем получить доступ через переменную к значению, на которое она ссылается, и узнать его тип. Осталось лишь выяснить, какие типы данных вообще возможны в Python.

Наш опыт работы с переменными пока что ограничивается текстовыми и числовыми значениями. Так вот, текстовые значения относятся к типу, который называется **`str`**. Числовые значения относятся к типу **`int`** (если это целые числа) или к типу **`float`** (если это действительные числа в формате значения с плавающей точкой).

Значения логического типа могут принимать два значения (истина или ложъ). В Python логический тип обозначается как **`bool`**. С логическими выражениями мы познакомимся, когда приступим к изучению условных инструкций и инструкций цикла.

Несколько позже мы познакомимся со списками, которые играют роль массивов в Python. Списки относятся к типу **`list`**. Еще в Python есть множества (о них мы тоже поговорим позже). Множества относятся к типу **`set`**. Существуют и другие типы данных, которые мы будем изучать постепенно, по мере нашего изучения языка Python.

От типа данных зависит то, какие операции могут с этими данными выполняться. Манипулировать с данными можем или с помощью функций, или операторов. Какие операторы, когда и как используются в Python, мы обсудим в следующем разделе.

Основные операторы

Обычно выделяют четыре группы операторов:

- арифметические;
- побитовые;
- операторы сравнения;
- логические операторы.

Арифметические операторы предназначены в первую очередь для выполнения арифметических вычислений. В таблице 2 перечислены и кратко описаны основные арифметические операторы языка Python.

Сразу следует отметить, что действие арифметических операторов достаточно точно соответствует "математической природе" этих операторов. При этом мы предполагаем, что речь идет о числовых расчетах.

Таблица 2. Арифметические операторы

Оператор	Описание
+	Оператор сложения. Вычисляется сумма двух чисел
-	Оператор вычитания. Вычисляется разность двух чисел
*	Оператор умножения. Вычисляется произведение двух чисел
/	Оператор деления. Вычисляется отношение двух чисел
//	Оператор целочисленного деления. Вычисляется целая часть от деления одного числа на другое
%	Оператор вычисления остатка от целочисленного деления. Вычисляется остаток от деления одного числа на другое
**	Оператор возведения в степень. Результатом является число, получающееся возведением первого операнда в степень, определяемую вторым операндом

Пример программного кода для выполнения несложных арифметических вычислений:

```
a = (5 + 2) ** 2 - 3 * 2  # Результат 43
b = 6-5/2  # Результат 3.5
c = 10 // 4 + 10 % 3  # Результат 3
# Результаты вычислений выводим на экран
print ("Результаты вычислений:")
```

```
print (a, b, c)
```

Результат выполнения этого программного кода представлен ниже:

```
Результаты вычислений:
43 3.5 3
```

Замечание

Возможно, некоторых пояснений потребует процедура вычисления значения выражения `10 // 4 + 10 % 3`. Так, значением выражения `10 // 4` является целая часть от деления 10 на 4 – это число 2. Значение выражения `10 % 3` – это число 1 (остаток от деления 10 на 3). В результате получаем сумму 2 и 1 – то есть число 3.

Язык Python позволяет создавать очень элегантные программные коды. Здесь мы воспользуемся возможностью, чтобы проиллюстрировать данное утверждение. Рассмотренный выше программный код мы перепишем немного иначе. В частности, воспользуемся функцией **eval** ( ), которая позволяет вычислять выражения, заданные в текстовом формате. Например, если некоторое алгебраическое выражение, записанное в соответствии с правилами синтаксиса языка Python, заключить в двойные кавычки, то получится текст. Если этот текст теперь указать аргументом функции **print**( ) , то на экране появится соответствующее алгебраическое выражение. Если же текст (с алгебраическим выражением) передать аргументом функции **eval**( ) , то соответствующее алгебраическое выражение будет вычислено.

Пример

```
a = "(5+2)**2 - 3*2" # Текстовое значение
b = "6 - 5/2"        # Текстовое значение
c = "10//4 + 10%3"   # Текстовое значение
# Результаты вычислений выводим на экран.
# Для "вычисления" текстовых выражений
# используем функцию eval()
print("Результаты вычислений:" )
print(a + " =", eval(a))
print(b + " =", eval(b))
print(c + " =", eval(c))
```

В результате выполнения этого программного кода получаем следующее:

```
Результаты вычислений:
(5+2)**2 - 3*2 = 43
6 - 5/2 = 3.5
10//4 + 10%3 = 3
```

Побитовые (или двоичные) **операторы** очень близки к арифметическим в том отношении, что тоже предназначены для работы с числовыми значениями. Только в случае побитовых операторов вычисления выполняются на уровне

двоичного кода числа. Для эффективного использования побитовых операторов необходимо неплохо разбираться в двоичном представлении чисел.

Замечание

В двоичном коде число представляется в виде последовательности нулей и единиц. Старший бит используется для определения знака числа: ноль соответствует положительному числу, а единица соответствует отрицательному числу.

Побитовые операторы перечислены и описаны в таблице 3.

Таблица 3. Побитовые операторы

Оператор	Описание
~	Побитовая инверсия (унарный оператор - у него один операнд). Результатом является число, получающееся заменой нулей на единицы и единиц на нули в побитовом представлении операнда (сам операнд при этом не меняется)
&	Побитовое И. При вычислении результата сравниваются побитовые представления операндов. Если на одной и той же позиции в операндах стоят единицы, то в числе-результате на этой же позиции будет единица. В противном случае (то есть если хотя бы в одной из двух позиций нуль) в числе-результате на соответствующей позиции будет нуль
|	Побитовое ИЛИ. Сравниваются побитовые представления операндов. Если на одной и той же позиции в операндах стоят нули, то в числе-результате на этой же позиции будет нуль. В противном случае (то есть если хотя бы в одной из двух позиций единица) в числе-результате на соответствующей позиции будет единица
^	Побитовое ИСКЮЧАЮЩЕЕ ИЛИ. Результат вычисляется сравнением побитовых представлений операндов. Если на одной и той же позиции в операндах стоят разные значения (у одного числа нуль, а у другого единица), то в числе-результате на этой же позиции будет единица. В противном случае (то есть если на соответствующих позициях в операндах стоят одинаковые числа) в числе-результате на этой позиции будет нуль
<<	Сдвиг влево. Результат вычисляется так: в побитовом представлении первого операнда выполняется сдвиг влево. Количество разрядов, на которые выполняется сдвиг, определяется вторым операндом. Младшие недостающие биты заполняются нулями
>>	Сдвиг вправо. Для вычисления результата в побитовом представлении первого операнда выполняется сдвиг вправо. Количество разрядов, на которые выполняется сдвиг, определяется вторым операндом. Биты слева заполняются значением самого старшего бита (для положительных чисел это нуль, а для отрицательных единица)

Чтобы проиллюстрировать методы использования побитовых операторов, рассмотрим небольшой программный код:

```
a = 70 >> 3
b = ~a
c = a << 1
print(a, b, c)
print(7|3, 7&3 , 7^3)
```

Результат выполнения этого программного кода такой:

```
8  -9  16
7  3  4
```

Замечание

На всякий случай приведем пояснения относительно результатов выполнения программного кода. Результат выражения 70 >> 3 – это число, получаемое сдвигом битового представления числа 70 на 3 позиции вправо (с потерей младших битов). Двоичный код для числа 70 имеет вид 0001000**110** (три бита, которые "пропадают" после сдвига вправо, выделены жирным шрифтом). После сдвига на 3 позиции вправо получаем 0001000. Это код числа 8. Такой же результат можно было получить проще, если вспомнить, что сдвиг вправо на одну позицию эквивалентен целочисленному делению (делению без остатка) на число 2. Если 70 трижды поделить без остатка на 2, получим 8 (35 после первого деления, 17 после второго деления, и 8 после третьего деления).

Далее, если применить побитовое инвертирование к числу 00…0001000 (значение переменной **a**), получим бинарный код 11…1110111, который соответствует отрицательному числу –9. Желающие могут выполнить проверку самостоятельно, однако если вспомнить, что результатом операции **~a+1** является код для значения **–a** (в данном случае это -8), то несложно догадаться, что **~a** соответствует значению **–a–1** (то есть в данном случае это -9).

Значение выражения **a<<1** получаем сдвигом бинарного кода для значения переменной **a** на одну позицию вправо с заполнением младшего бита нулём (соответствует умножению значения переменной **a** на 2). Поскольку значение переменной **a** равно 8, то значение выражения **a<<1** равняется 16.

В следующих выражениях используются числа 7 (бинарный код 00000111) и 3 (бинарный код 00000011). Для побитовых операций **|** (или), & (и), **^** (исключающее или) получаем следующие результаты:

```
|   00 ... 000111
    00 ... 000011
7   00 ... 000111

&   00 ... 000111
    00 ... 000011
3   00 ... 000011

^   00 ... 000111
    00 ... 000011
4   00 ... 000100
```

В левом нижнем углу жирным шрифтом выделен результат соответствующей операции в десятичной системе счисления. С логическими значениями мы столкнемся при проверке условий в условной инструкции (операторе). Значений у логического типа всего два: True (истина) и False (ложь). Для работы со значениями логического типа предназначены специальные операторы, которые принято называть логическими.

Используемые в Python логические операторы описаны в таблице 4.

Таблица 4. Логические операторы

Оператор	Описание
or	Бинарный оператор (у оператора два операнда). Логическое ***ИЛИ***. В общем случае результатом выражения **x or y** является `True`, если значение хотя бы одного из операндов **x** или **y** равно `True`. Если значения обеих операндов **x** и **y** равны `False`, результатом выражения **x or y** будет `False`. В Python выражения на основе оператора **or** вычисляются по упрощенной схеме: если первый операнд **x** интерпретируется как `True`, то в качестве результата возвращается **x**. Если первый операнд **x** интерпретируется как `False`, то в качестве результата возвращается второй операнд **y**
and	Бинарный оператор. Логическое ***И***. В общем случае результатом выражения **x and y** является значение `True`, если значения обеих операндов **x** и **y** равны `True`. Если значение хотя бы одного из операндов **x** или **y** равно `False`, результатом выражения **x and y** будет `False`. В Python выражения на основе оператора **and** вычисляются по упрощенной схеме: если первый операнд **x** интерпретируется как `False`, то в качестве результата возвращается **x**. Если первый операнд **x** интерпретируется как `True`, то в качестве результата возвращается второй операнд **y**
not	Логическое отрицание. Унарный оператор (у оператора один операнд). Результатом выражения **not x** будет значение `True`, если у операнда **x** значение `False`. Результатом выражения **not x** будет значение `False`, если у операнда **x** значение `True`

Замечание

Нередко на практике используется такая операция, как логическое исключающее или. Это бинарная операция. Ее результатом является истина, если операнды имеют разные значения. Если операнды имеют одинаковые значения, результатом операции исключающего

или является значение ложь. Другими словами, исключающее или – это проверка на предмет того, различные ли значения у операндов или нет.

Помимо непосредственно логических значений могут использоваться и числовые значения (да и не только). Если числовое значение встречается в том месте, где по идее должно быть значение логического типа, начинается интерпретация нелогического выражения как логического. Интерпретация выполняется так: нулевые числовые значения интерпретируются как False, а ненулевые значения интерпретируются как True.

Замечание

Ситуация даже более интересная, чем может показаться на первый взгляд. Логические значения True и False можно использовать, соответственно, как числа 1 и 0. Например, в арифметических расчетах вместо 1 можем использовать True, а вместо 0 можем использовать False . Однако к текстовому формату (например, будучи переданными аргументами функции print ( )) значения True и False приводятся не как 1 и 0, а как "True" и "False". Об этой особенности следует помнить.

Кроме того, при проверке условий или вычислении логических выражений могут использоваться не только непосредственно логические значения или числа, но и иные объекты. Как именно выполняется "логическая" интерпретация таких объектов мы узнаем, когда поближе познакомимся с методами ООП.

Программный код, приведенный ниже, даёт представление о том, как с применением логических операторов вычисляются логические выражения.

```
a = True
b = not a
print(a, b)
c = a and b
d = a or b
print(c, d)
```

Результат выполнения программного кода такой:

```
True  False
False  True
```

Хотя по своей сути логические операторы должны возвращать логические значение, в Python это далеко не всегда так. Результатом выражений на основе операндов **`or`** и **`and`** является один из операндов соответствующего выражения. А операнды не обязательно должны быть логического типа – достаточно, чтобы они могли интерпретироваться как логические значения.

Ситуацию иллюстрирует следующий пример

```
x = 10        # Числовая переменная
y = 20        # Числовая переменная
z = x and y   # Логическое и
print(z)      # Результат логического и
z = x or y    # Логическое или
print(z)      # Результат логического или
print(not x) # Логическое отрицание
```

В данном случае логические операторы используются с числовыми операндами. При выполнении программного кода получаем такой результат:

```
20
10
False
```

Переменные x и y ссылаются на целочисленные значения. Поскольку значения ненулевые, то при выполнении логических операций обе переменные интерпретируются как такие, что имеют значение True . Но интерпретироваться как значение True и ссылаться на значение True – это далеко не одно и то же.

При вычислении выражения x and y сначала проверяется значение первого операнда. Поскольку первый операнд x (значение 10) интерпретируется как True, в качестве результата выражения возвращается второй операнд – то есть y (значение 20). Аналогично, при вычислении выражения x or y, поскольку первый операнд x интерпретируется как True , он же возвращается в качестве результата. Но результатом является реальное числовое значение, на которое ссылается переменная x (то есть значение 10).

Иначе обстоят дела с оператором логического отрицания not . Результатом выражения not x является значение False. То есть в данном случае логический оператор дает вполне "ожидаемый" логический результат.

Операторы сравнения позволяют сравнивать на предмет равенства / неравенства различные значения. Обычно (хотя и не всегда) речь идет о числовых значениях. Результатом операций сравнения являются логические значения: True, если соответствующее соотношение верно (если отношение имеет место), и False, если неверно (отношение не имеет место). Здесь и далее мы будем говорить об операциях сравнения только в основном для числовых значений. Операторы сравнения языка Python представлены в таблице 5.

Таблица 5. Операторы сравнения

Оператор	Описание
<	Строго меньше. Результатом является `True`, если значение операнда слева от оператора *меньше* значения операнда справа от оператора. Иначе возвращается значение `False`
>	Строго больше. Результатом является `True`, если значение операнда слева от оператора *больше* значения операнда справа от оператора. Иначе возвращается значение `False`
<=	Меньше или равно. Результатом является `True`, если значение операнда слева от оператора *не больше* значения операнда справа от оператора. Иначе возвращается значение `False`
>=	Больше или равно. Результатом является `True`, если значение операнда слева от оператора *не меньше* значения операнда справа от оператора. Иначе возвращается значение `False`

Оператор	Описание
==	Равно. Результатом является `True`, если значение операнда слева от оператора *равно* значению операнда справа от оператора. Иначе возвращается значение `False`
!=	Не равно. Результатом является `True`, если значение операнда слева от оператора *не равно* значению операнда справа от оператора. Иначе возвращается значение `False`
is	Оператор проверки идентичности объектов. В качестве результата возвращается значение `True`, если оба операнда ссылаются на один и тот же объект. В противном случае (то есть если операнды ссылаются на разные объекты) возвращается значение `False`
is not	Оператор проверки неидентичности объектов. В качестве результата возвращается значение `True`, если операнды ссылаются на разные объекты. В противном случае (то есть если операнды ссылаются на один и тот же объект) возвращается значение `False`

В заключение раздела сделаем несколько важных замечаний. Первое касается так называемых сокращенных форм *оператора присваивания*.

Как мы уже знаем, оператором присваивания в Python служит знак равенства =. Также существуют так называемые сокращенные формы оператора присваивания. Речь идет вот о чем. Если необходимо выполнить команду вида x = x Θ y, где через x и y обозначены некоторые переменные, а через Θ мы формально обозначили один из арифметических или побитовых операторов, то эту команду можно записать в виде x Θ= y. Например, вместо команды x = x + y можно использовать команду x += y.

Что касается самого оператора присваивания (это второе замечание), то в Python разрешено многократное и множественное присваивание. Об этом сообщалось ранее при рассмотрении понятия оператора присваивания.

Вместе с тем, многократное и множественное присваивание следует использовать с крайней осторожностью, поскольку при работе с такими данными, как, например, списки, результаты подобных операций могут быть вполне неожиданными. Вся "сложность" ситуации связана в основном с тем, что в Python (как подчеркивалось и еще будет подчеркиваться) переменные не содержат значения, а ссылаются на них. Для некоторых типов данных этот момент является принципиальным. Все подобные "премудрости" мы будем изучать по мере знакомства с различными типами данных.

Третье замечание касается приоритета различных операторов. В сложных выражениях одновременно могут присутствовать самые разные операторы. Такие выражения вычисляются в соответствии с приоритетом операторов. Сначала вычисляются выражения и подвыражения с операторами, которые имеют более высокий приоритет, а уже затем вычисляются выражения и подвыражения с более низким приоритетом. Если несколько операторов имеют одинаковые приоритеты, выражение вычисляется слева направо. Вкратце приоритет операторов в Python следующий (в порядке уменьшения приоритета):

- побитовая инверсия, возведение в степень, знак числа (унарный минус или унарный плюс);
- умножение, деление, целочисленное деление, остаток от деления;
- сложение, вычитание;
- побитовые сдвиги;
- побитовое И;
- побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ;
- побитовое ИЛИ;
- оператор присваивания (включая и сокращенные формы).

Для изменения порядка вычисления выражения можно использовать круглые скобки. Более того, наличие в правильном месте круглых скобок (даже если острой необходимости в них нет), не только делает код более "надежным", но и значительно повышает его читабельность.

Операции над строками

Во всех примерах, которые мы рассматривали, переменные хранили строки. Мы вводили, выводили и хранили строки. Кроме уже описанных действий, строки еще можно складывать.

Пример

```
x = '10'
y = '20'
z = x + y
print(z)
```

Результатом выполнения программы будет строка 1020.

Конкатенация

Операция сложения для строк выполняет конкатенацию двух строк, то есть склеивает их содержимое вместе.

Дублирование

Для строк также можно выполнять умножение. Умножать можно строку на число или число на строку. Операция называется дублирование. В результате начальная строка будет повторена заданное количество раз.

Пример

```
x = '10'
y = '20'
print(x * 2 + y * 3)
```

Команда `in`

Команда **in** позволяет проверить, что одна строка находится внутри другой. Например: строка «золото» находится внутри строки «Сказка о золотом петушке». В таком случае обычно говорят, что одна строка является подстрокой для другой.

Пример

```
text = input()
if 'хорош' in text and 'плох' not in text:
    print('Текст имеет положительную эмоциональную окраску.')
elif 'плох' in text and 'хорош' not in text:
    print('Текст имеет отрицательную эмоциональную окраску.')
else:
    print('Текст имеет нейтральную или смешанную ' +
    'эмоциональную окраску.')
```

Первое условие окажется истинным, например, для строк «всё хорошо» и «какой хороший день», но не для «ВСЁ ХоРоШо» и не для «что-то хорошо, а что-то и плохо». Аналогично второе условие окажется истинным для строк «всё плохо», «плохое настроение» и т. д.

Длина строки

Длина строки – это количество символов в строке. Для определения длины строки в Python используется стандартная функция `len()`.

Пример

```
s = input()
print('Вы ввели слово', s, 'длиной', len(s), 'букв')
```

Здесь используется вызов функции. Он устроен так: пишем имя функции, а в скобках – те данные, которые мы передаём этой функции. Такие данные называются ***аргументами*** функции. На место вызова функции подставляется результат её работы – ***возвращаемое значение функции***.

Таким образом, функция `len()` возвращает длину своего аргумента.

`input()` – тоже функция, она может работать без аргумента. Если указан аргумент, то он будет напечатан в качестве сообщение перед вводом данных с клавиатуры. Но всегда считывает строку с клавиатуры и возвращает её.

`print()` – тоже функция. Она не возвращает никакого значения, зато выводит свои аргументы на экран. Эта функция может принимать сколько угодно аргументов, они разделяются запятыми.

Типы данных

У каждого элемента данных, который встречается в программе, есть свой тип. Например, «привет» – это строка, а вот 15.3 – это число (дробное). даже если данные не записаны прямо в программе, а получаются откуда-то еще, у них есть совершенно определенный тип. Например, на место input() всегда подставляется строка, а 2+2 даст именно число 4, а не строку ”4”.

Пользователь может ввести с клавиатуры какие-то цифры, но в результате input( ) вернет строку, состоящую из этих цифр. Если мы попытаемся, например, прибавить к этой строке 1, получим ошибку.

А если нам надо работать с числами? Мы пока будем рассматривать целые и вещественные числа.

Когда речь идет о числовых данных, они записываются без кавычек. А для вещественных чисел, чтобы отделить дробную часть от целой, используют точку.

На прошлом занятии мы складывали две строки:

```
print('10' + '20')
```

и получали результат – строку ”1020“.

Давайте попробуем в этом примере убрать кавычки. В таком случае речь пойдет уже не о строках, а о двух целых числах.

И результатом функции **`print(10 + 20)`** будет целое число 30.

А если мы попробуем сложить два вещественных числа

```
print(10.0 + 20.0)
```

результатом будет вещественное число 30.0.

Числовые данные

Обычно числовые литералы набираются в десятичной системе – то есть в той системе, которой мы пользуемся в повседневной жизни. Это десять цифр от 0 до 9 включительно, с помощью которых мы и вводим в программном коде нужное нам целое или действительное число. Целые числа набираются в виде последовательности цифр и, если нужно, знака числа. Действительные числа набираются практически так же, но только у них есть целая и дробная части, а в качестве разделителя целой и дробной частей используем точку (об этом говорилось выше).

Замечание

Литерал – это фиксированное значение, которое не может быть изменено в программе. Обычно под литералами подразумевают числа и текст, которые используются явно в программном коде. Например, в команде *x* = 1.5 через *x* обозначена переменная, а 1.5 – это литерал (числовой). Другой пример: в команде *name* = "Иван Иванович" текст "Иван Иванович" является литералом, а *name* – это переменная.

Но в принципе в программном коде для целых чисел значения можно указывать не только в десятичной системе счисления, но и в двоичной, восьмеричной и шестнадцатеричной.

Двоичную систему счисления мы уже обсуждали. В восьмеричной системе числа записываются с помощью восьми цифр: от 0 до 7 включительно. Аналогично обстоят дела с шестнадцатеричной системой счисления. Только теперь используется, как несложно догадаться, 16 "символов": десять цифр от 0 до 9 и еще шесть букв от A до F включительно. Эти буквы обозначают числа от 10 до 15 (то есть буква A соответствует числу 10, буква B соответствует числу 11, и так далее, вплоть до буквы F, которая соответствует числу 15).

Если литералы задаются не в десятичной системе счисления, то начинаться они должны со специального префикса, который "идентифицирует", к какой системе счисления относится литерал. Для бинарных чисел (литералы в двоичной системе) префикс состоит из нуля и буквы *b* (большой или маленькой). Например, литерал 0b101 означает число 5. Вполне законной будет, скажем, команда *x*=0b101, которой переменной *x* присваивается числовое значение 5.

Замечание

Если бы нам понадобилось присвоить переменной x значение –5 (то есть отрицательное значение), причем набрать литерал в двоичном коде, то соответствующая команда выглядела бы как x = –0b101 .То есть мы используем знак "минус" в литерале. Все то, о чем мы говорили выше о бинарном кодировании отрицательных чисел в компьютере, о принципе "дополнения до нуля" – в данном случае нам не нужно. Почему? Потому что там речь шла о том, как отрицательные числа представлены в памяти компьютера. Здесь речь идет о формальной записи числа с использованием двоичного кода. То, что мы пишем в программном коде, предназначено для того, кто будет этот код читать – то есть для программиста, пользователя, читателя. Поэтому здесь никто не запрещает нам использовать в числовом литерале знак "минус", даже если этот литерал представляет число в двоичной системе счисления. Компьютер, когда придет его черед взяться за выполнение программного кода, автоматически переведет указанный литерал в "правильное" с точки зрения компьютера представление. Это же замечание (относительно знака числа) касается и прочих систем счисления: восьмеричной и шестнадцатеричной.

Признаком числа, записанного в восьмеричной системе счисления, является префикс *0o* (ноль и большая или маленькая буква *o*), а числа, записанные в шестнадцатеричной системе, начинаются с префикса *Ox* (ноль и большая или маленькая буква *x*). Например, число 123 в восьмеричной системе счисления запишется как 0o173. Это же число в шестнадцатеричной системе представляется литералом Ox7B.

Комплексные числа вводятся в "естественном" формате, то есть в виде суммы действительной и мнимой части. Признаком мнимости числа является суффикс *j* (большая или маленькая буква), который размещается сразу после числового значения. Другими словами, если после числового литерала указать (без пробелов или иных разделителей) букву j, то получится мнимое число. Например, комплексное число 3+2i запишется в программном коде как 3+2j. Мнимая единица *i* в виде программного кода реализуется как 1j, и так далее. Также для создания комплексного числа можем воспользоваться функцией *complex*( ). Первым аргументом функции передается действительная часть комплексного числа, а второй аргумент – мнимая часть комплексного числа. Так, число 3+2i можем получить с помощью команды *complex*(3, 2) , а мнимой единице i соответствует инструкция *complex*(0, 1).

При вводе очень больших или, напротив, очень маленьких (по модулю) чисел удобно воспользоваться представлением числа в виде мантиссы и показателя степени. В качестве разделителя мантиссы и показателя степени используют букву *e* (большую или маленькую). Например, числовой литерал 1.2*e*3 обозначает число $1.2 \cdot 10^3$, а числовой литерал 1.2*e*-5 соответствует числу $1.2 \cdot 10^{-5}$.

Выше мы уже рассматривали арифметические и побитовые операторы. С целочисленными значениями (данные типа *int*) могут использоваться и те, и другие. С действительными значениями (данные типа *float*) используются арифметические операторы. Также для выполнения математических вычислений предназначен целый ряд встроенных функций. Некоторые полезные в работе математические функции представлены в таблице 6.

Таблица 6. Некоторые математические функции

Функция	Описание
`abs()`	Вычисление модуля числа. Число, для которого вычисляется модуль, указывается аргументом функции. Может использоваться с комплексными числами. Например, результатом каждого из выражений abs(5.0) , abs(-5.0) и abs(3 + 4 j) является значение 5.0
`bin()`	Функция предназначена для преобразования числа из десятичной системы счисления в двоичную. Исходное число указывается аргументом функции, а результатом является текстовое представление для двоичного кода числа. Например, результатом выражения bin(9) будет текст "0b1001"
`complex()`	Функция используется для создания комплексных чисел на основе (переданных аргументами функции) действительной и мнимой частей числа, или на основе текстового представления комплексного числа. Например, результатом выражений complex(3, 4) и complex ("3 + 4 j") будет комплексное число 3 + 4 j
`float()`	Функция используется для преобразования числовых значений и текстовых представлений для действительных чисел в числовые значения типа *float*. Например, результатом каждого из выражений *float*(5), *float*("5"), *float*("5.") и *float*("5.0") является значение 5.0
`hex()`	Функция предназначена для преобразования числа из десятичной системы счисления в шестнадцатеричную. Аргумент функции – число в десятичной системе счисления. Результат функции – текстовое представление этого числа в шестнадцатеричной системе. Например, результатом выражения *hex*(123) будет текст "0x7B"
`int()`	Функция для преобразования объекта (например, текста) в целое число. Если аргументом функции передано действительное число (тип float), то результат вычисляется отбрасыванием дробной части в действительном числе. Если аргументом указать текстовое представление целого числа, результатом будет само это число. Причем число (в текстовом представлении) может быть не только в десятичной системе, но и в двоичной, восьмеричной или шестнадцатеричной. В этом случае вторым аргументом указывается целое число, определяющее исходную систему счисления (соответственно, 2, 8 или 16). Например, результатом каждого из выражений *int*(123.4), *int*("123"), *int*("0b1111011", 2), *int*("0ol73", 8) и *int*("0x7B", 16) является значение 123

`max()`	Функция для вычисления максимального значения из набора чисел. Например, результатом выражения *max* (-2, 4, 9, -1) является значение·9
`min()`	Функция для вычисления минимального значения из набора чисел. Например, результатом выражения *min* (-2, 4, 9, -1) является значение·-2
`oct()`	Функция предназначена для преобразования числа из десятичной системы счисления в восьмеричную. Аргументом указывается число в десятичной системе счисления, а результатом является текстовое представление этого числа в восьмеричной системе. Например, результатом выражения *oct*(9) будет текст "0o11"
`pow()`	Функция для возведения в степень числа. Если у функции два аргумента, то результатом является первое число в степени, определяемой вторым числом. Другими словами, результатом выражения *pow(x, y)* является значение *x**y*. Если у функции три аргумента, то после возведения в степень вычисляется остаток от деления на третий аргумент: то есть результатом выражения *pow(x, y, z)* является значение *(x**y) % z*. Например, результатом выражения *pow*(2, 3) будет значение 8, а результатом выражения *pow*(2, 3, 5) будет значение 3
`round()`	Функция предназначена для округления действительных значений до целочисленных. Аргументом указывается округляемое значение. Округление выполняется по таким правилам: • если дробная часть округляемого значения меньше 0.5, округление выполняется до ближайшего меньшего целого числа; • если дробная часть округляемого значения больше 0.5, округление выполняется до ближайшего большего целого числа; • если дробная часть округляемого значения равняется 0.5, округление выполняется до ближайшего четного целого числа. Если функции передать второй аргумент, то он будет определять количество знаков в дробной части, до которых будет выполняться округление (то есть в этом случае округление выполняется не до целого числа, а до действительного с количеством разрядов после запятой, определяемым вторым аргументом функции). Например, результатом выражения *round*(4.6) (дробная часть 0.6) есть значение 5 , у выражения *round*(-4.6) (дробная часть 0.4) значение -5 , у выражения *round*(4.4) (дробная часть 0.4) значение 4 , у выражения *round*(-4.4) (дробная часть 0.6) значение -4 , у выражения *round*(4.5) (дробная

часть 0.5) значение 4, а у выражения *round*(-4.5) (дробная часть 0.5) – соответственно, -4. Для сравнения: результат выражения *round* (1.23456, 3) – число 1.235

Большое количество математических функций становится доступным после подключения модуля math. Модули обсуждаются далее, но сразу отметим, что ничего сложного в подключении модуля нет: для этого в программу достаточно добавить инструкцию import math. После этого, например, можем в программном коде использовать тригонометрические функции, такие, как sin( ) (синус), cos( ) (косинус), tan( ) (тангенс) и многие другие. Правда, для указанного способа импортирования модуля, при вызове функций из этого модуля придется перед именем функции указывать название модуля (разделитель имени модуля и имени функции – точка): например, math.sin( ) , math.cos( ) или math.tan( ). Также в этом модуле определены иррациональные постоянные: $\pi \approx 3.14159265$ (инструкция math.pi) и $e \approx 2.7182818$ (инструкция math.e).

Подключение модулей

Обычно под модулем подразумевается некоторый файл с программой или блоком программного кода. При написании больших программ не всегда удобно весь программный код хранить в одном файле. Поэтому его разбивают на отдельные части или на модули. Возможен и другой вариант: при написании собственной программы мы хотим использовать часть программного кода, который был создан ранее (и находится в каком-то определенном модуле). Чтобы использовать такой код, необходимо импортировать соответствующий модуль. Для импортирования модулей используется инструкция ***import***, после которой указывается название подключаемого (импортируемого) модуля. Мы будем Импортировать встроенные модули Python – то есть те модули, которые являются составной частью среды разработки на языке Python. Например, если мы хотим импортировать математический модуль ***math*** (модуль содержит различные математические функции и утилиты), то используем инструкцию ***import math***. Если импортируемых модулей несколько, то после инструкции ***import*** имена импортируемых модулей перечисляются через запятую.

После того, как модуль подключен, описанные в нем функции, переменные и другие полезные конструкции можно использовать в программном коде. Но при этом каждый раз необходимо явно указывать имя модуля, в котором описана переменная или функция. Используется так называемый *точечный синтаксис*: сначала указываем имя модуля, и через точку имя переменной или название функции (со всеми полагающимися аргументами). Например, если мы хотим использовать переменную, которая описана в модуле, то соответствующая инструкция будет выглядеть как `модуль.переменная`. Причем предварительно командой `import` модуль необходимо импортировать модуль. Более конкретно, в модуле `math` есть переменная `pi` со значением постоянной $\pi \approx 3.14159265$. Если мы хотим увидеть значение этой переменной,

то сначала командой `import math` подключаем модуль `math`, а затем командой `print(math.pi)` отображаем значение переменной `pi` из модуля `math`.

Вместо того чтобы использовать имя модуля при обращении к его "содержимому", можем для модуля создать "псевдоним". Модуль подключаем командой в формате **`import`** `модуль` **`as`** `имя`. Другими словами, в инструкции подключения модуля после имени модуля через ключевое слово **`as`** можно указать идентификатор, который будет использоваться вместо имени модуля. То есть, модуль все равно подключается, но когда мы обращаемся к переменным и функциям этого модуля, то указываем не имя модуля, а идентификатор (тот, который после ключевого слова **`as`**). Например, если мы подключаем модуль командой `import модуль as имя`, то обращаться к переменной из модуля нужно в формате `имя.переменная`. Если вспомнить о модуле `math` и подключить его командой `import math as m`, то распечатать значение переменной `pi` можем командой `print (m.pi)`.

Замечание

Если мы подключили модуль с "псевдонимом", то обращаться к содержимому модуля придется через "псевдоним" – попытка использовать имя модуля приведет к ошибке.

Очевидным образом бросается в глаза неудобство, связанное с необходимостью при обращении к переменным и функциям модуля в явном виде указывать имя модуля (или его "псевдоним"). Но здесь же кроется и главное преимущество: непосредственно в программном коде мы можем определить переменную (функцию или что-то еще) с таким же именем, как и в подключаемом модуле. В этом случае наличие или отсутствие имени модуля при обращении к переменной (или функции) позволяет однозначно идентифицировать, о какой программной конструкции идет речь.

Вместе с тем, можно подключать не весь модуль, а только некоторые его утилиты (переменные или функции). Скажем, если из модуля нас интересует только одна переменная, то можем воспользоваться командой `from модуль import переменная`. После этого переменную можно использовать без ссылки на имя модуля. Так, чтобы использовать в программном коде переменную `pi` без указания перед ее именем названия модуля `math`, используем инструкцию импорта `from math import pi`.

Если мы хотим подключить все утилиты из некоторого модуля, полезной будет инструкция `from модуль import *` (то есть после инструкции импорт указываем звездочку).

Тернарный оператор

Часто возникает необходимость выполнять в программном коде различные команды в зависимости от того, выполняется или нет некоторое условие. Вообще такая задача решается с помощью условного оператора, с которым мы

познакомимся в следующей главе. Вместе с тем, есть "упрощенная" форма условного оператора, которую, по аналогии с языками C++ и Java, можно было бы назвать тернарным оператором (хотя, конечно, рассматриваемая далее конструкция резко контрастирует с обычными представлениями об операторе).

Замечание

Обычно операторы бывают унарные и бинарные. У унарного оператора один операнд, у бинарного оператора два операнда. Тернарный оператор – оператор, у которого три операнда. Один из операндов – это проверяемое условие. Еще два операнда – значения, одно из которых возвращается в качестве результата, в зависимости от проверяемого условия.

Тернарный оператор возвращает значение. Причем это значение зависит от истинности или ложности некоторого условия. Шаблон тернарного оператора такой:

```
значение_1 if условие else значение_2
```

Тернарный оператор – достаточно сложная конструкция. Сначала указывается выражение (значение_1), которое определяет значение тернарного оператора при истинности условия. Между этим выражением' и условием указывается ключевое слово if. После условия указывается ключевое слово else, а затем выражение (значение_2), определяющее результат тернарного оператора, если условие не выполнено. Небольшой пример использования тернарного оператора приведен ниже.

```
# Считывается первое число
a = float(input("Введите первое число: "))
# Считывается второе число
b = float(input("Введите второе число: "))
# Первое значение
value_1 = "Первое число больше второго."
# Второе значение
value_2 = "Второе число не меньше первого."
# Вызывается тернарный оператор
res = value_1 if a>b else value_2
# Отображается результат
print(res)
```

Результат выполнения этого программного кода может быть таким (жирным шрифтом выделен ввод пользователя):

```
Введите первое число: 12
Введите второе число: 30
Второе число не меньше первого.
```

Или таким:

```
Введите первое число: 30
Введите второе число: 12
Первое число больше второго.
```

Управляющие инструкции

К управляющим инструкциям в данном случае мы относим условный оператор (со всеми его возможными модификациями), а также операторы цикла (в языке Python их два). По большому счету, именно управляющие инструкции делают программу действительно программой – то есть целостной конструкцией, а не банальным набором последовательно выполняемых команд. Но обо всем по порядку. В первую очередь рассмотрим условный оператор, причем начнем с самых простых его версий

Условный оператор `if`

Условный оператор (или, как его еще иногда называют, условная инструкция) позволяет в зависимости от истинности или ложности определенного условия выполнять различные блоки программного кода. Общая схема, в соответствии с которой функционирует условный оператор, выглядит так:

- Проверяется некоторое условие: обычно это выражение, значение которого может интерпретироваться как истинное (значение True) или ложное (значение False).
- Если выражение-условие интерпретируется как истинное, выполняется выделенная специальным образом последовательность команд.
- Если условие интерпретируется как ложное, выполняется другая (но тоже выделенная специальным образом) последовательность команд.
- После того, как одна или другая последовательность команд условного оператора выполнена, управление передается той команде, которая находится после команды вызова условного оператора.

Фактически, речь идет о том, что имеется два набора команд, а решение о том, какой из наборов команд выполнять, принимается по результатам проверки выражения-условия. Таким образом, в программе создается "точка ветвления".

В языке Python условный оператор реализуется в виде программного блока со следующим шаблоном (жирным шрифтом выделены ключевые элементы):

```
if условие:
   команды_1
else:
   команды_2
```

После ключевого слова **if** указывается условие – выражение логического типа (или допускающее интерпретацию в терминах True и False). После условия ставится двоеточие (то есть **:**). Далее следует блок команд, которые выполняются, если условие истинно (в шаблоне это *команды_l*). Блок команд выделяется с помощью отступов (это стандартный способ выделения блоков

команд в языке Python). В принципе, количество отступов может быть произвольным – главное, чтобы для каждой команды блока делалось одно и то же количество отступов. Но общепринятым является правило делать 4 отступа (4 пробела).

После окончания блока команд (выполняемых при истинном условии) следует ключевое слово **else** (и двоеточие **:**). Ключевое слово **else** должно быть на том же уровне (так те же количество отступов или пробелов), что и ключевое слово **if**. Далее – еще один блок команд (в шаблоне обозначены как *команды_2*), которые выполняются, если условие ложно.

Пример

```
print("Введите число A: ")
a = int(input())
print("Введите число B: ")
b = int(input())
if b == 0:
    print("На ноль делить нельзя!")
else:
    print("A/B =", a/b)
```

Описанный выше условный оператор обычно называют **if**-оператором или **if-else**-оператором. Вместе с тем, этот оператор может использоваться и в несколько ином виде. Так, допускается использование упрощенной формы оператора без **else**-блока. Шаблон использования условного оператора в этом случае такой:

```
if условие:
    команды
```

Одна очень полезная модификация условного оператора позволяет последовательно проверять несколько условий. Если смотреть в корень, то это на самом деле система вложенных условных операторов. Используется следующий шаблон (жирным шрифтом выделены ключевые элементы):

```
if условие_1:
    команды_1
elif условие_2:
    команды_2
elif условие_3:
    команды_3
elif условие_N:
    команды_N
else:
    команды
```

В данном случае начало традиционное: после ключевого слова **if** указывается условие для проверки (обозначено *условие_1*). Если условие истинно, выполняются команды **if**-блока (обозначено как *команды_1*). Если выражение *условие_1* ложно, проверяется условие, указанное после ключевого слова **elif** (в шаблоне условие обозначено как *условие_2*, после условия ставится двоеточие **:**).

Если это, второе, условие, истинно, выполняются команды для данного **elif**-блока (команды этого блока обозначены как *команды_2*), и на этом работа условного оператора завершается. Если же и выражение *условие_2* ложно, проверяется условие в следующем **elif**-блоке, и в случае его истинности – команды этого блока.

Если при переборе **elif**-блоков ни одного истинного условия не обнаружено, выполняются команды **else**-блока, который размещается в шаблоне самым последним.

Пример

```
print("Вычисление стоимости покупки с учетом скидки")
print("Введите сумму покупки и нажмите <Enter> ")
fSumma = float(input())
if fSumma > 1000.0:
    print("Вам предоставляется скидка 5%")
    print("Сумма с учетом скидки:", fSumma * 0.95)
elif fSumma > 500.0:
    print("Вам предоставляется скидка 3%")
    print("Сумма с учетом скидки:", fSumma * 0.97)
else:
    print("Скидка не предоставляется")
```

Оператор цикла `while`

Операторы цикла позволяют многократно выполнять предопределенный набор или группу команд. В языке Python есть два оператора цикла. Сначала мы познакомимся с оператором цикла, в котором используется **`while`**-инструкция. Поэтому обычно такой оператор цикла называют оператором цикла **`while`**, или **`while`**-оператором.

Шаблон у этого оператора цикла простой (жирным шрифтом выделены ключевые элементы):

`while` *условие*:
    команды

Принцип работы оператора цикла очень простой: первым делом, как доходит очередь до выполнения этого оператора, проверяется условие, которое указано после ключевого слова **`while`**. Если условие истинно, выполняются

команды в теле оператора цикла. После выполнения команд снова проверяется условие. Если условие истинно, выполняются команды, а затем проверяется условие, и так далее – до тех пор, пока при проверке условия не окажется, что оно ложно. На этом выполнение оператора цикла заканчивается и выполняется команда, следующая после условного оператора.

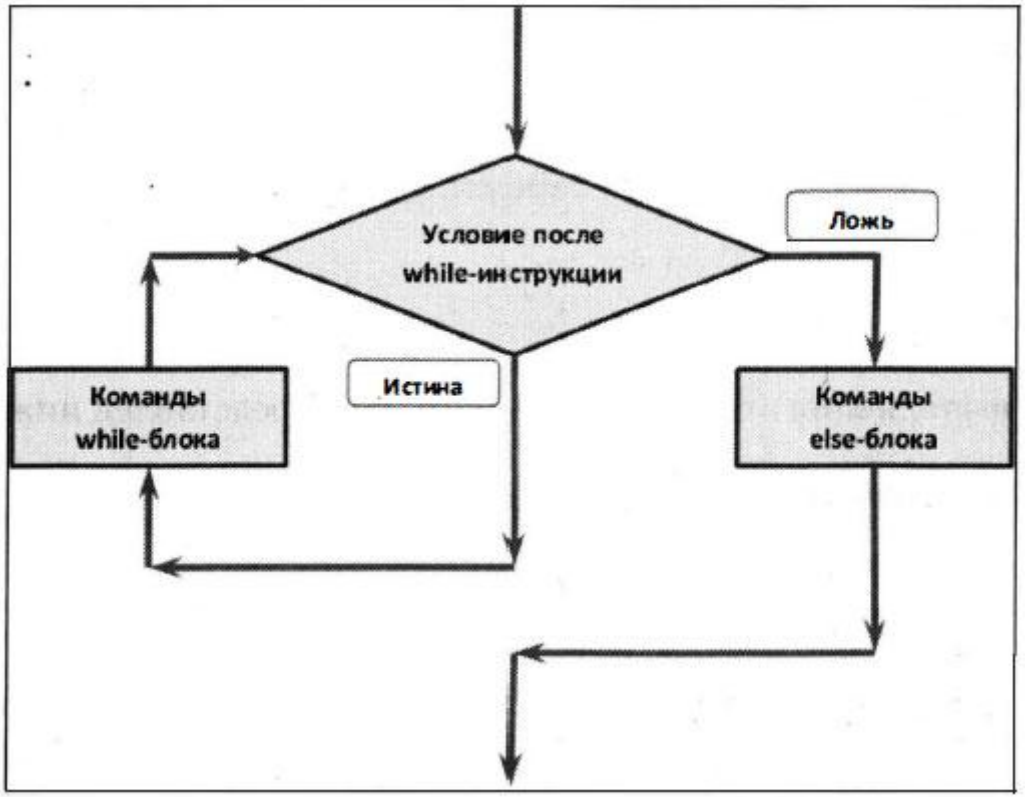
У оператора цикла имеется и "расширенная" версия с ключевым словом **`else`**. Шаблон для оператора цикла в этом случае таков (жирным шрифтом выделены ключевые элементы):

```
while условие:
    команды_1
else:
    команды_2
```

Кроме **`while`**-блока в этом случае есть еще и `else`-блок: после ключевого слова **`else`** (и двоеточия) указывается блок команд, которые выполняются только в том случае, если условие после инструкции **`while`** является ложным.

В частности, в самом начале выполнения оператора цикла проверяется условие после инструкции **`while`**. Если условие ложно, выполняются команды в **`else`**-блоке и на этом работа оператора цикла завершается. Если условие истинно, выполняются команды **`while`**-блока. После этого проверяется условие. Если условие ложно, выполняется **`else`**-блок и завершается работа оператора цикла. Если условие истинно, выполняются команды **`while`**-блока и снова проверяется условие, и так далее.

Схема выполнения такой "расширенной" версии оператора цикла иллюстрируется ниже



Если внимательно проанализировать эту схему, то несложно сообразить, что команды **`else`**-блока выполняются один и только один раз, причем на завершающей стадии выполнения оператора цикла. Такого же эффекта можно добиться, если эти команды разместить вне оператора цикла сразу после него. Поэтому в **`else`**-блоке было бы мало пользы, если бы не два ключевых слова (две инструкции): **`break`** и **`continue`**. Инструкция **`break`** завершает работу оператора цикла *без выполнения* **`else`**-блока. Инструкция **`continue`** завершает выполнение текущего цикла и позволяет перейти сразу к проверке условия в **`while`**-блоке.

Кроме оператора цикла **`while`**, в языке Python есть еще один оператор цикла, который, отталкиваясь от ключевого слова, входящего в выражение для этого оператора, называют оператором цикла **`for`**.

Оператор цикла `for`

По сравнению с оператором цикла **`while`**, оператор цикла **`for`** можно было бы назвать более "специализированным". С одной стороны, оператор очень гибкий и эффективный. С другой стороны, его успешное использование может вызвать некоторые трудности. Проще говоря, при работе с оператором **`for`** (в зависимости от режима его использования) возникает много всяких нюансов. Поэтому мы пока что рассмотрим самые простые и наиболее понятные схемы применения оператора цикла **`for`**.

Шаблон использования этого оператора такой (жирным шрифтом выделены ключевые элементы):

```
for элемент in последовательность:
    команды
```

После ключевого слова **for** указывается некий `элемент` (если проще, то название переменной), который последовательно принимает значения из `последовательности` (упорядоченный набор значений). Между `элементом` и *последовательностью* – ключевое слово **in**. Завершается данная синтаксическая конструкция двоеточием **:**. Затем идет блок команд оператора цикла. В операторе цикла команды выделяются отступами – традиционно, четырьмя.

Замечание

Нередко (но не всегда) в качестве последовательности в операторе цикла **for** используются списки. Со списками мы еще не знакомы. Основные сведения о списках (равно как и об иных типах данных, содержащих коллекции значений и подпадающих под определение последовательности) представлены в следующих главах. Здесь нас списки интересуют только в контексте использования их в операторе цикла **for**. Важно знать, что список – это набор упорядоченных элементов (оформленных соответствующим образом). Причем элементы могут быть разного типа. Чтобы создать список, достаточно в квадратных скобках через запятую перечислить элементы списка. Например, конструкция [1, 2, 3, 4, 5] представляет собой список из пяти натуральных чисел.

В качестве последовательности можно использовать и текстовые значения. В этом случае перебираются буквы в тексте.

Выполняется оператор цикла следующим образом: `элемент` (переменная, указанная перед ключевым словом **in**) принимает первое значение в `последовательности`, указанной после ключевого слова **in**. После этого выполняются команды оператора цикла. Затем `элемент` принимает второе значение из `последовательности`, и снова выполняются команды оператора цикла. И так далее, пока не будет завешен перебор всех значений в `последовательности`.

Замечание

В блоке команд оператора цикла можно использовать инструкции `break` и `continue`. Эффект такой же, как и при использовании этих инструкций в операторе цикла **while**: выполнение инструкции `break` приводит к завершению оператора цикла, а выполнение инструкции `continue` приводит к завершению текущего цикла и переходу к следующему.

В операторе цикла **for** может использоваться **else**-блок.

Шаблон оператора цикла **for** с **else**-блоком выглядит следующим образом (жирным шрифтом выделены ключевые элементы):

```
for элемент in список:
    команды_1
else:
    команды_2
```

Команды в **else**-блоке выполняются, фактически, по завершении оператора цикла, как мы его понимали бы без **else**-блока. Весь "выигрыш" от **else**-блока в том, что если работа оператора цикла завершается вследствие выполнения инструкции **break**, то команды в блоке **else** не выполняются.

Работу оператора цикла **for** проиллюстрируем на несложных примерах. В первую очередь рассмотрим пример, в котором вычисляется сумма натуральных чисел. Еще одно важное новшество – использование функции **range**( ), с помощью которой мы будем создавать виртуальную последовательность натуральных чисел. Чтобы создать такой объект, как виртуальная последовательность чисел, аргументами функции **range**( ) передаются первое число *виртуальной* последовательности и последнее число последовательности плюс один. Например, если нас интересует последовательность чисел от 1 до 5 включительно, то для создания такой последовательности можем воспользоваться инструкцией **range**(1, 6). Таким образом, последнее число в последовательности на единицу меньше второго аргумента функции **range**( ).

Замечание

Если функции **range**( ) передать только один аргумент, то сформированная последовательность будет начинаться с нуля. Если функции **range**( ) передать три аргумента, то третий аргумент будет определять шаг арифметической прогрессии: первый аргумент определяет начальное числовое значение в последовательности, а каждое следующее получается прибавлением шага прогрессии (третий аргумент) – но не больше, чем значение второго аргумента. Можно также сказать, что если третий аргумент не указан, то по умолчанию это значение равно единице, а если не указан первый аргумент, то его значение по умолчанию нулевое.

Еще одно замечание касается термина *виртуальный*. Почему мы говорим о числовой последовательности, сформированной функцией **range**( ), как о виртуальной? Все дело в том, что на самом деле функцией возвращается некоторый объект, который допускает с собой такое обращение, как если бы это была последовательность чисел.

Но если мы присвоим результат функции **range**( ) некоторой переменной и затем захотим проверить значение этой переменной, последовательности мы не увидим – будет формальное обозначение результата с ссылкой на название функции **range**( ).

Однако обращаем внимание, что для использования функции **range**( ) в операторе цикла **for** никаких дополнительных действий по "визуализации" виртуальной последовательности предпринимать не нужно.

Ниже представлен программный код, в котором сумма натуральных чисел вычисляется с помощью оператора цикла **for**.

```
print("Сумма натуральных чисел")
n = 100 # Количество слагаемых
```

```
# Формируем текст для отображения результата
text = "1 + 2 + . . . + " + str(n) + " ="
# Переменная для записи суммы
s = 0
# Оператор цикла для вычисления суммы
for i in range (1, n + 1):
    # Добавляем слагаемое к сумме
    s = s + i
# Отображаем результат
print(text, s)
```

При выполнении данного программного кода мы получаем следующий результат:

```
Сумма натуральных чисел
1 + 2 + . . . + 100 = 5050
```

Замечание

Сумму можно вычислить с помощью встроенной функции **sum** ( ). Аргументом функции передается список элементов, сумма которых вычисляется. Так, сумму натуральных чисел от 1 до n (если значение этой переменной задано) можем вычислить командой `sum(range(1, n + 1))`. Здесь аргументом функции `sum()` передается виртуальная последовательность `range(1, n + 1)` – так тоже можно делать.

Следующий пример иллюстрирует, как в операторе цикла **for** в качестве последовательности для перебора значений используется текст. Если более конкретно, то мы берем в качестве основы текст, и затем с помощью оператора цикла **for** распечатываем текст по буквам с небольшими текстовыми вставками. Теперь обратимся к программному коду:

```
# Текст для оператора цикла
txt = "Python"
i = 1  #  переменная для нумерации букв
# Оператор цикла
for s in txt:
    # Формируем в спомогательный текст
    t = str(i) + "-я буква:"
    # Выводим сообщение
    print(t, s)
    # Изменяем номер буквы
    i = i + 1
# Команда после завершения оператора цикла
print("Работа программы завершена!")
```

В переменную `txt` записывается базовый текст, который мы планируем "перебирать" по буквам в операторе цикла. Также мы объявляем с начальным единичным значением переменную `i` . Мы при выводе букв из текста `txt` собираемся указывать номер каждой буквы, и для этого нам понадобилась отдельная переменная. Но отдельно подчеркнем: хотя в операторе цикла эта

переменная и используется, перебор элементов последовательности (букв в тексте) выполняется с помощью другой переменной – мы ее назвали `s`. Инструкция `s in txt` в операторе цикла после ключевого слова `for` означает, что переменная `s` будет последовательно принимать буквенные значения из текста в переменной `txt`. Поэтому о переменной `s` мы можем думать как об очередной букве в тексте `txt`. Причем перебираются буквы строго в той последовательности, как они находятся в тексте.

В операторе цикла командой `t=str(i)+"-я буква:"` мы формируем и записываем в переменную `t` вспомогательный текст, который получается объединением текстового представления порядкового номера `i` буквы в тексте и текста `"-я буква:"`. Для перевода числового значения в текстовое использована функция `str( )`.

В результате выполнения команды `print(t, s)` в строке вывода вспомогательный текст с номером буквы и сама буква. После этого командой `i = i + 1` на единицу увеличивается номер для следующей буквы.

После завершения оператора цикла командой `print("Работа программы завершена!")` выводится финальное сообщение. Ниже приведен результат выполнения программы:

```
1-я буква: P
2-я буква: y
3-я буква: t
4-я буква: h
5-я буква: o
6-я буква: n
Работа программы завершена!
```

Также обращаем внимание и напоминаем, что текст в Python можно выделять как двойными, так и одинарными кавычками.

Хотя условные операторы и операторы цикла представляют собой достаточно мощное средство и позволяют решать многие нетривиальные задачи, в Python существует еще одна синтаксическая конструкция, которая в известном смысле может быть отнесена к управляющим инструкциям – во всяком случае, один из возможных вариантов ее использования сводится к организации точек ветвления в программе. Речь идет об обработке исключительных ситуаций.

Вложенные циклы

Циклы называются вложенными (т. е. один цикл находится внутри другого), если внутри одного цикла во время каждой итерации необходимо выполнить другой цикл. Так для каждого витка внешнего цикла выполняются все витки внутреннего цикла. Основное требование для таких циклов: чтобы все действия вложенного цикла располагались внутри внешнего.

При использовании вложенных циклов стоит помнить, что изменения, внесенные внутренним циклом в какие-либо данные, могут повлиять на внешний.

Рассмотрим следующую задачу: необходимо вывести в строку таблицу умножения заданного числа. Задача решается так:

```
k = int(input())
for i in range(1, 10):
    print(i, "*", k, "=",k*i, sep="", end="\t")
```

Результат:

```
5
1*5=5   2*5=10   3*5=15   4*5=20   5*5=25   6*5=30   7*5=35   8*5=40 9*5=45
```

А если нам нужно вывести таблицу умножения для всех чисел от 1 до k? Очевидно, что в этом случае предыдущую программу нужно повторить k раз, где вместо k будут использоваться числа от 1 до k включительно.

Эту задачу можно записать двумя циклами, где для каждого значения внешнего цикла будут выполняться все значения внутреннего цикла.

Программа будет выглядеть так:

```
k = int(input())
for j in range(1, k+1):
    for i in range(1, 10):
        print(i, "*", j, "=",j*i, sep="", end="\t")
    print()
```

Результат:

```
5
1*1=1   2*1=2   3*1=3   4*1=4   5*1=5   6*1=6   7*1=7   8*1=8   9*1=9
1*2=2   2*2=4   3*2=6   4*2=8   5*2=10  6*2=12  7*2=14  8*2=16  9*2=18
1*3=3   2*3=6   3*3=9   4*3=12  5*3=15  6*3=18  7*3=21  8*3=24  9*3=27
1*4=4   2*4=8   3*4=12  4*4=16  5*4=20  6*4=24  7*4=28  8*4=32  9*4=36
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25  6*5=30  7*5=35  8*5=40  9*5=45
```

Оператор *break* и *continue* во вложенных циклах

Рассмотрим другую задачу: представьте, что необходимо распечатать все строки таблицы умножения для чисел от 1 до 9 и столбцы, кроме столбца для числа k.

Тогда нам нужно будет пропустить выполнение шага внутреннего цикла, когда придет k-й столбец

```
k = int(input())
for j in range(1, 10):
    for i in range(1, 10):
```

```
        if i == k:
            continue
        print(i, "*", j, "=",j*i, sep="", end="\t")
    print()
```

Результат:

```
5
1*1=1   2*1=2   3*1=3   4*1=4   6*1=6   7*1=7   8*1=8   9*1=9
1*2=2   2*2=4   3*2=6   4*2=8   6*2=12  7*2=14  8*2=16  9*2=18
1*3=3   2*3=6   3*3=9   4*3=12  6*3=18  7*3=21  8*3=24  9*3=27
1*4=4   2*4=8   3*4=12  4*4=16  6*4=24  7*4=28  8*4=32  9*4=36
1*5=5   2*5=10  3*5=15  4*5=20  6*5=30  7*5=35  8*5=40  9*5=45
1*6=6   2*6=12  3*6=18  4*6=24  6*6=36  7*6=42  8*6=48  9*6=54
1*7=7   2*7=14  3*7=21  4*7=28  6*7=42  7*7=49  8*7=56  9*7=63
1*8=8   2*8=16  3*8=24  4*8=32  6*8=48  7*8=56  8*8=64  9*8=72
1*9=9   2*9=18  3*9=27  4*9=36  6*9=54  7*9=63  8*9=72  9*9=81
```

Обратите внимание: если оператор `break` или `continue` расположен внутри вложенного цикла, он действует именно на вложенный цикл, а не на внешний. Нельзя выскочить из вложенного цикла сразу на самый верхний уровень.

Обработка исключительных ситуаций

Какой бы витиевато-изощренный программный код ни создавался, "обойти" все "опасные места" и предусмотреть все возможные варианты развития ситуации далеко не всегда удается. Особенно эта проблема актуальна, когда в программе часть данных считывается из файла или вводится в программу пользователем уже в процессе выполнения программы. В этих случаях велика вероятность получения программой некорректных данных. В таких случаях при выполнении программы могут возникать ошибки. Самое главное и наиболее неприятное следствие возникновения ошибки – это преждевременное "аварийное" завершение программы. В Python существует механизм, который позволяет программе выполняться даже после того, как возникла ошибка. Все это называется общим и емким термином: *обработка исключительных ситуаций*.

Ошибки, или исключительные ситуации, бывают разные. Обычно выделяют три типа ошибок:

- Ошибки, связанные с неправильным синтаксисом команд. Другими словами, это ошибки, связанные с неправильно набранным кодом. Такие ошибки выявляются просто и без особых усилий: при попытке запуска на выполнение программы с неправильными командами, скорее всего, будет выведено сообщение о месте ошибки и причине ее возникновения.

- Ошибки, связанные с неправильным алгоритмом выполнения программы. То есть ошибки, которые связаны с тем, что программа как таковая составлена неправильно (хотя с формальной точки зрения все команды корректные). Это очень коварные ошибки, поскольку обычно никаких предупреждающих сообщений не появляется, программа работает, а результат – неправильный. Возможное решение проблемы: тестирование программы на модельных примерах или задачах, для которых известен правильный результат.
- Ошибки времени выполнения, возникающие в процессе выполнения программы и связанные с некорректностью переданных в программу данных, недоступностью ресурсов и пр.

Понятно, что далеко не для каждой ошибки можно выполнить обработку так, чтобы программа продолжала выполняться. Однако в некоторых случаях такая обработка может быть выполнена. Собственно о таких потенциально "отлавливаемых" ошибках, их перехвате и обработке речь будет идти далее.

Общая идея, заложенная в основу метода обработки исключительных ситуаций, такая: программный код, в котором теоретически может возникнуть ошибка, выделяется специальным образом – образно говоря, "берется на контроль". Если при выполнении этого программного кода ошибка не возникает, то ничего особенного не происходит. Если при выполнении "контролируемого" кода возникает ошибка, то выполнение кода останавливается и автоматически создается исключение – объект, содержащий описание возникшей ошибки. С практической точки зрения мы можем думать об исключении как о некотором сообщении, которое генерируется интерпретатором в силу возникшей ошибки при выполнении кода или из-за некоторых других обстоятельств, близких по своей природе к ошибке выполнения кода. Хотя ошибка и исключение – это не одно и то же (исключение является следствием ошибки), мы, если это не будет приводить к недоразумениям, обычно будем отождествлять эти понятия.

Для обработки исключительных ситуаций в языке Python используется конструкция **`try-except`**. Существуют разные вариации использования этой конструкции. Мы начнем с наиболее простых ситуаций.

Можем поступить следующим образом: после ключевого слова **`try`** и двоеточия размещается блок программного кода, который мы подозреваем на предмет возможного возникновения ошибки. Этот код будем называть основным или контролируемым. По завершении этого блока указывается ключевое слово **`except`** (с двоеточием), после которого идет еще один блок программного кода. Этот код будем называть вспомогательным или кодом обработки исключения. То есть шаблон такой (жирным шрифтом выделены ключевые элементы):

```
try:
    # основной код
except:
    # вспомогательный код
```

Если при выполнении основного кода в блоке **try** ошибка не возникла, то вспомогательный программный код в блоке **except** выполняться не будет. Если при выполнении основного кода в блоке **try** возникла ошибка, то выполнение кода **try**-блока прекращается, и выполняется вспомогательный код в блоке **except**. После этого управление передается следующей команде после конструкции **try–except**.

Пример использования конструкции **try–except** в описанном выше формате приведен в листинге ниже. В этом примере речь идет о решении линейного уравнения вида. Это уравнение имеет очевидное решение , но это при условии, что параметр отличен от нуля. Поскольку параметры и вводятся пользователем, то, как говорится, возможны варианты.

```
print("Решаем уравнение ax = b")
# Начало try-блока (основной код)
try:
    # Определяем первый параметр. Возможна ошибка
    # при преобразовании текста в число
    a = float(input("Введите a: "))
    # Определяем второй параметр. Возможна ошибка
    # при преобразовании текста в число
    b = float(input("Введите b: "))
    # Решение уравнения. Возможна ошибка при делении на ноль
    x = b/a
    # Отображается значение для корня уравнения.
    # Команда выполняется, если до этого не возникли ошибки
    print("Решение уравнения: x =", x)
# Начало except-блока (вспомогательный код)
except:
    # Команда выполняется только если ранее
    # при выполнении основного кода возникла ошибка
    print("Вы ввели некорректные данные!")
# Команда выполняется после блока try-except
print("Спасибо, работа программы завершена.")
```

Ниже показано, как может выглядеть результат выполнения программы в случае, если все данные пользователем вводятся корректно (здесь и далее ввод пользователя выделен жирным шрифтом):

```
Решаем уравнение ax = b
Введите a: 5
Введите b: 8
Решение уравнения: x = 1.6
Спасибо, работа программы завершена.
```

Если вместо числового значения ввести нечто другое, можем получить совсем иной результат:

```
Решаем уравнение ax = b
Введите a: text
```

```
Вы ввели некорректные данные!
Спасибо, работа программы завершена.
```

Практически такой же результат получаем, если для переменной *a* указать нулевое значение:

```
Решаем уравнение ax = b
Введите a: 0
Введите b: 3
Вы ввели некорректные данные!
Спасибо, работа программы завершена.
```

Исключительные ситуации бывают разные – типы ошибок разные. И более утонченная обработка исключительных ситуаций подразумевает и более индивидуальный, так сказать, подход. Речь идет о том, чтобы обработка ошибок базировалась на типе или характере ошибки. Разумеется, это возможно. Причем описанная выше схема обработки исключительных ситуаций претерпевает минимальные изменения. Если более конкретно, то ситуация выглядит так: в конструкции **try–except** после блока **try** указывается несколько **except**-блоков, причем для каждого блока явно указывается тип ошибки, который обрабатывается в этом блоке. Ключевое слово, определяющее тип ошибки, указывается после ключевого слова **except** соответствующего блока.

Замечание

При возникновении ошибки генерируется исключение и создается объект, описывающий ошибку. Поэтому тип ошибки – это на самом деле класс, на основе которого создается объект исключения (более точное название, принятое в Python – экземпляр исключения). Другими словами, "типология" ошибок базируется на иерархии классов. Каждый класс соответствует какой-то ошибке. Работа с классами и объектами (экземплярами класса) обсуждается немного позже. Да и особенности использования классов и объектов здесь нас мало интересуют. Во всяком случае, не будет большой проблемой, если мы станем интерпретировать название класса ошибки просто как некоторое ключевое слово, определяющее тип, к которому относится ошибка.

В этом случае шаблон программного кода, содержащий обработку ошибок разного типа, имеет такой вид: ·

```
try:
    # основной код
except Тип_ошибки_1:
    # вспомогательный код
except Тип_ошибки_2:
    # вспомогательный код
. . . .
except Тип_ошибки_N:
    # вспомогательный код
```

Этот код выполняется следующим образом. Выполняются команды **try**-блока. Если возникла ошибка, то выполнение команд **try**-блока прекращается и

начинается последовательный просмотр **except**-блоков на предмет совпадения типа ошибки, которая возникла, и типа ошибки, указанного после ключевого слова в **except**-блоке. Как только совпадение найдено, выполняются команды соответствующего **except**-блока, после чего управление переходит к команде после конструкции **try–except**.

Совпадение типов при поиске нужного **except**-блока для обработки ошибки не обязательно должно быть "буквальным". Дело вот в чем. Мы уже знаем, что тип ошибки – это на самом деле класс, который описывает эту ошибку. Но у классов могут быть подклассы (производные классы). Это примерно так, как если бы у типа были подтипы. Или, другим словами, для некоторых категорий ошибок существует более детальная классификация по сравнению с остальными ошибками. Например, класс `ZeroDivisionError`, соответствующий ошибке деления на ноль, является подклассом класса `ArithmeticError` (арифметическая ошибка). А еще у класса `ArithmeticError` есть подклассы `FloatingPointError` (ошибка при операциях с плавающей точкой) и `OverflowError` (ошибка переполнения). Так вот, при обработке исключительных ситуаций перехватываются не только ошибки определенного (указанного в **except**-блоке) класса, но и ошибки подклассов этого класса. Например, если мы в **except**-блоке укажем класс ошибки `ZeroDivisionError`, то будут перехватываться ошибки типа только `ZeroDivisionError` (деление на ноль). Но если в **except**-блоке указать класс `ArithmeticError`, то в этом блоке будут перехватываться и обрабатываться ошибки типа `ZeroDivisionError`, `FloatingPointError` и `OverflowError`.

Если при переборе **except**-блоков совпадение (по типу ошибки) не найдено, выполнение кода прекращается и появляется сообщение об ошибке. Правда, могут использоваться вложенные конструкции **try–except**. В этом случае необработанная ошибка может перехватываться внешним **except**-блоком (но такая возможность должна быть предусмотрена в программе).

Если при выполнении **try**-блока ошибок не было, коды в **except**-блоках не выполняются.

Что касается классов исключений, то кроме перечисленных выше, наибольший интерес с практической точки зрения могут представлять такие:

- Исключение `ValueError`: возникает при рассогласовании типов (например, функции нужен аргумент числового типа, а передается текст).
- Исключение `IndexError`: возникает, когда индекс (например, для элемента списка) указан неправильно (выходит за границы допустимого диапазона). Списки обсуждаются в одном из следующих занятий.
- Исключение `KeyError`: возникает при неверно указанном ключе словаря. Словари обсуждаются в одном из следующих занятий.

- Исключение `NameError`: возникает, если не удается найти локальное или глобальное имя (переменную) с некоторым названием.
- Исключение `SyntaxError`: связано с наличием синтаксических ошибок.
- Исключение `TypeError`: связано с несовместимостью типов, когда для обработки (например, в функции) требуется значение определенного типа, а передается значение другого типа.

Более подробно классы исключений и методы работы с ними будут обсуждаться после того, как мы познакомимся с классами и объектами. Иногда приходится несколько иную задачу: для нескольких типов ошибок создавать один **except**-блок. В этом случае после ключевого слова **except** в круглых скобках через запятую перечисляются те типы ошибок, для которых выполняется обработка в данном блоке.

В инструкции **try–except** помимо блоков **try** и **except** могут также использоваться блоки **else** и **finally.** Общий шаблон инструкции в этом случае такой:

```
try:
    # основной код
except Тип_ошибки_1:
    # вспомогательный код
except Тип_ошибки_2:
    # вспомогательный код
. . . .
except Тип_ошибки_N:
    # вспомогательный код
else:
    # код для случая, если ошибки не было
finally:
    # код, который выполняется всегда
```

Программный блок с ключевым словом **else** размещается после последнего **except**-блока и содержит программный код, который выполняется только в том случае, если при выполнении основного кода в **try**-блоке ошибок не было.

Программный код, размещенный в блоке **finally**, выполняется в любом случае, независимо от того, возникла ошибка при выполнении кода **try**-блока или нет.

Множества

Множество – составной тип данных, представляющий собой несколько значений (элементов множества) под одним именем. Этот тип называется **set**. Чтобы задать множество, нужно в фигурных скобках перечислить его элементы.

Здесь создается множество из четырех элементов (названий млекопитающих), которое затем выводится на экран:

```
mammals = {'cat', 'dog', 'fox', 'elephant'}
print(mammals)
```

Введите этот код в Python и запустите программу несколько раз. Скорее всего, вы увидите, что порядок перечисления млекопитающих разный, так происходит потому, что элементы в множестве не упорядочены. Это позволяет быстро выполнять операции над множествами, о которых мы скоро поговорим чуть позже.

Создание множества

Для создания пустых множеств обязательно вызывать функцию `set`.

```
empty = set()
```

Элементами множества могут быть строки или числа. А также множество может содержать и строки, и числа.

Пример

```
mammals_numbers = {'cat', 5, 'dog', 3, 'fox', 12, 'cow'}
print(mammals_numbers)
```

Как видим, Python опять выводит элементы множества в случайном порядке. Заметьте, если поставить в программе оператор вывода множества на экран несколько раз, не изменяя само множество, порядок вывода элементов не изменится. Правило упорядочивания элементов в множестве выбирается случайным образом при запуске интерпретатора Python.

Может ли элемент входить в множество несколько раз? Это было бы странно, так как совершенно непонятно, как отличить один элемент от другого. Нет смысла хранить несколько одинаковых объектов – удобно иметь контейнер, сохраняющий только уникальный объекты. Поэтому множество содержит каждый элемент только один раз. Следующий фрагмент кода это демонстрирует:

```
birds = {'raven', 'sparrow', 'dove', 'hawk',
'dove'}
print(birds)
```

Итак, у множеств есть три ключевые особенности:

- Порядок элементов в множестве не определен
- Элементы множеств – строки и/или числа
- Множество не может содержать одинаковых элементов

Выполнение этих трех свойств позволяет организовать элементы множества в структуру со сложными взаимосвязями, благодаря которым можно быстро проверять наличие элементов в множестве, объединять множества и т. д. Но пока давайте обсудим ограничения.

Операции над множеством

Простейшая операция – вычисление количества элементов множества. Для этого служит функция `len()`.

```
my_set = {12, 23, 34, 45}
print(len(my_set))
```

Результат:

```
4
```

Очень часто необходимо обойти все элементы множества в цикле. Для этого используется цикл **`for`** и оператор **`in`**, с помощью которых можно перебрать не только все элементы диапазона (как мы это делали раньше, используя `range`), но и элементы множества:

```
my_set = {12, 23, 34, 45}
for elem in my_set:
    print(elem)
```

Результат:

```
12
23
34
45
```

Код для работы с множествами нужно писать таким образом, чтобы он правильно работал при любом порядке обхода. Для этого надо знать два правила:

- Если мы не изменяли множество, порядок обхода элементов тоже не изменится
- После изменения множества порядок элементов может измениться произвольным образом

Чтобы проверить наличие элемента в множестве, можно воспользоваться уже знакомым оператором **`in`**:

```
if elem in my_set:
    print('Элемент есть в множестве')
else:
    print('Элемента нет в множестве')
```

Добавление элемента в множество делается при помощи `add`:

```
new_elem = 'e'
my_set.add(new_elem)
```

`add` – это метод множества. В данном случае – множества `my_set`. Правило записи методов: `Объект.метод`

Если элемент, равный `new_elem`, уже существует в множестве, оно не изменится, поскольку не может содержать одинаковых элементов. Ошибки при этом не произойдет.

С удалением элемента сложнее. Для этого есть сразу три метода: **`discard`** (удалить заданный элемент, если он есть в множестве, и ничего не делать, если его нет), **`remove`** (удалить заданный элемент, если он есть, и породить ошибку keyError, если нет) и **`pop`**. Метод **`pop`** удаляет некоторый элемент из множества и возвращает его как результат. Порядок удаления при этом неизвестен.

```
my_set = {12, 23, 34}
my_set.discard(12)  # удалён
my_set.discard(19)  # не удалён, ошибки нет
my_set.remove(23)  # удалён
print(my_set)  # в множестве остался один элемент 34
my_set.remove(28)  # не удалён, ошибка KeyError
```

Метод **`pop`** удаляет из множества случайный элемент и возвращает его значение

```
my_set = {12, 23, 34}
elem = my_set.pop(12)  # удалён
print('удалённый элемент', elem)
print('после удаления:', my_set)
```

Если попытаться применить **`pop`** к пустому множеству, произойдет ошибка `keyError`.

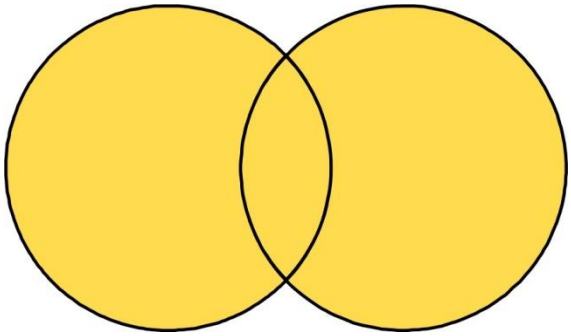
Очистить множество от всех элементов можно методом `clear`

```
my_set.clear()
```

Операции над множествами

Есть четыре операции, которые из двух множеств делают новое множество: объединение, пересечение, разность и симметричная разность.

Объединение



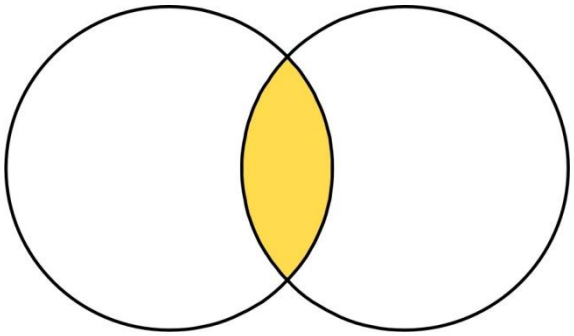
Объединение двух множеств включает в себя все элементы, которые есть хотя бы в одном из них. Для этой операции существует метод **`union`**:

```
union_set = set1.union(set2)
```

Или можно использовать оператор **|**

```
union_set = set1 | set2
```

Пересечение



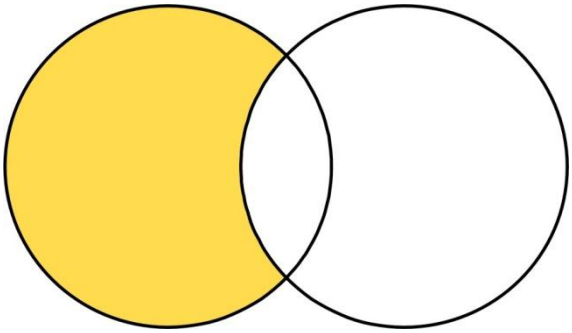
Пересечение двух множеств включает в себя все элементы, которые есть в обоих множествах:

```
intersection_set = set1.intersection(set2)
```

Или аналог:

```
intersection_set = set1 & set2
```

Разность



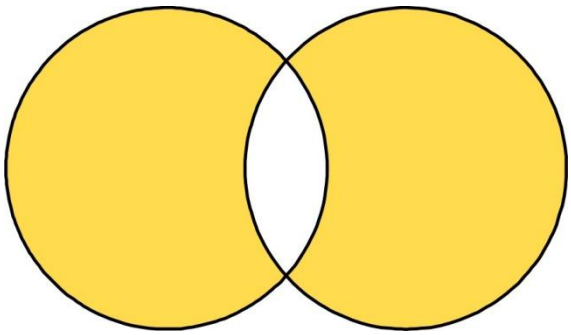
Разность двух множеств включает в себя все элементы, которые есть в первом множестве, но которых нет во втором:

```
diff_set = set1.difference(set2)
```

Или аналог:

```
diff_set = set1 - set2
```

Симметричная разность



Симметричная разность двух множеств включает в себя все элементы, которые есть только в одном из этих множеств:

```
symm_diff_set = set1.symmetric_difference(set2)
```

Или аналогичный вариант:

```
symm_diff_set = set1 & set2
```

Примеры

```
set1 = {12, 23, 34}
set2 = {12, 34, 45}
union_set = set1.union(set2)  # {12, 23, 34, 45}
```

```
intersection_set = set1.intersection(set2)  # {12, 34}
diff_set = set1 - set2          # {23}
sum_diff_set = set1 ^ set2      # {23, 45}
```

Сравнение множеств

Все операторы сравнения множеств, а именно: **==**, **<**, **>**, **<=**, **>=**, возвращают `True`, если сравнение истинно, и `False` – в противном случае.

Равенство и неравенство множеств

Множества считаются равными, если они содержат одинаковые наборы элементов. Равенство множеств, как в случае с числами и строками, обозначается оператором ==.

Неравенство множеств обозначается оператором !=. Он работает противоположно оператору ==.

```
if set1 == set2:
    print('Множества равны')
else:
    print('Множества не равны')
```

Обратите внимание на то, что у двух равных множеств могут быть разные порядки обхода, например, из-за того, что элементы в каждое из них добавлялись в разном порядке.

Теперь перейдем к операторам **<=, >=**. Они означают «является подмножеством» и «является надмножеством».

Подмножество и надмножество

Подмножество – некоторая выборка элементов множества, которая может быть как меньше множества, так и совпадать с ним, на что указывают символы «**<**» и «**=**» в операторе **<=**. Наоборот, надмножество включает все элементы некоторого множества и, возможно, какие-то еще.

```
set1 = {12, 23, 34}
print(set1 <= set1)   #  True
set2 = {12, 23}
print(set2 <= set1)   #  True
set3 = {12}
print(set3 <= set1}   #  True
set4 = {12, 77}
print(set4 <= set1}   #  False
```

Операция `set1 < set2` означает, что `set1` является подмножеством `set2`, но целиком не совпадает с ним. Операция `set1 > set2` означает, что `set1` является надмножеством `set2`, но целиком не совпадает с ним.

Строки. Индексация

На прошлом занятии мы познакомились с коллекцией, которая называется множество. Вспомним, что основная особенность коллекций – возможность хранить несколько значений под одним именем. Можно сказать, что коллекция является контейнером для этих значений.

Но еще до изучения множеств мы уже знали тип данных, который ведет себя подобно коллекции. Этот тип данных – строка. Действительно, ведь строка фактически является последовательностью символов. В некоторых языках программирования есть специальный тип данных `char`, позволяющий хранить один символ. В Python такого типа данных нет, поэтому можно сказать, что строка – последовательность символов.

Что мы уже знаем о работе со строками в языке Python. Мы умеем создавать строки четырьмя способами: задавать напрямую, считывать с клавиатуры функцией `input()`, преобразовывать число в строку функцией `str` и склеивать из двух других строк операцией **+**. Кроме того, мы умеем узнавать длину строки, используя функцию `len`, и проверять, является ли одна строка частью другой, используя операцию `in`.

В отличие от множеств, в строках важен порядок элементов (символов). Действительно, если множества {1, 2, 3} и {3, 2, 1} – это одинаковые множества, то строки POB и BOP – две совершенно разные строки.

Наличие порядка дает нам возможность пронумеровать символы. Нумерация символов начинается с 0.

Индекс

По индексу можно получить соответствующий ему символ строки. Для этого нужно после самой строки написать в квадратных скобках индекс символа.

```
word = 'Hello!'
first_letter = word[0]
print(first_letter)    #  H
fifth_letter = word[4]
print(first_letter)    #  o
```

Если попытаться получить букву, номер которой слишком велик, в этом случае Python выдаст ошибку:

```
print(word[6])
#  builtins.IndexError: string index out of range
```

Конечно, номер в квадратных скобках – не всегда фиксированное число, которое прописано в программе. Его, например, можно считать с клавиатуры или получить в результате арифметического действия.

Отрицательные индексы

Кроме «прямой» индексации (начинающейся с 0), в Python разрешены отрицательные индексы: `word[-1]` означает последний символ строки `word`, `word[-2]` – предпоследний и т д.

Изменить отдельный символ строки невозможно, т. е. строка относится к ***неизменяемым*** типам данных в Python.

Перебор элементов строки

На предыдущем занятии мы узнали, что цикл **`for`** можно использовать для перебора элементов множества. Таким же образом можно использовать цикл **`for`**, чтобы перебрать все буквы в слове:

```
text = 'Здравствуйте, мои дорогие!'
count_o = 0
for letter in text:
    if letter == 'o':
        count_o += 1
print(count_o)
```

Но, так как символы в строке пронумерованы, у нас есть еще один способ перебрать все элементы в строке: перебрать все индексы, используя уже знакомую нам конструкцию `for i in range(…)`.

```
text = 'Здравствуйте, мои дорогие!'
count_o = 0
for i in range(len(text)):
    if text[i] == 'o':
        count_o += 1
print(count_o)
```

Хранение текстов в памяти компьютера. Кодирование

Рассмотрим, как строки хранятся в памяти компьютера.

Поскольку компьютер умеет хранить только двоичные числа, для записи нечисловой информации (текстов, изображений, видео, документов) прибегают к кодированию. Самый простой случай кодирования – сопоставление кодов текстовым символам. Один из самых распространенных форматов такого кодирования – таблица ASCll (American standard code for information interchange).

Изначально в этой таблице каждому символу был поставлен в соответствие 7-битный код, что позволяло идентифицировать 128 различных

символов. В таблице коды меньше 32 являются служебными и не предназначены для непосредственного вывода на экран (перевод строки, табуляция и т. д.).

Этого хватало на латинские буквы обоих регистров, знаки препинания и спецсимволы – например, перевод строки или разрыв страницы. Позже код расширили до 8 бит (1 байта), что позволяло хранить уже 256 различных значений: в таблицу помещались буквы второго алфавита (например, кириллица) и дополнительные графические элементы (символы псевдографики). Таблицы стали нумеровать – каждому алфавиту свой номер. Например, для кириллицы это был номер 866 (обозначали CP866 или DOS866) – сюда входили буквы русского, украинского и белорусского алфавитов.

В некоторых относительно низкоуровневых языках (например, в C) можно в любой момент от представления строки в памяти перейти к последовательности байтов, начинающейся по какому-либо адресу.

Сейчас однобайтные кодировки отошли на второй план, уступив место Юникоду.

Юникод

Юникод – таблица, которая содержит соответствия между числом и каким-либо знаком, причем число может быть 16-разрядным (2 байта) и более. Это позволяет одновременно использовать любые символы любых алфавитов и дополнительные графические элементы. Кроме того, в Юникоде каждый символ, помимо кода, имеет некоторые свойства: например, буква это или цифра. Это позволяет более гибко работать с текстами.

В Юникод все время добавляются новые элементы, а сам размер этой таблицы не ограничен и будет только расти, поэтому сейчас при хранении в памяти одного юникод-символа может потребоваться от 1 до 8 байт. Отсутствие ограничений привело к тому, что стали появляться символы на все случаи жизни.

Если использовать юникод-числа для отображения соответствующих символов, то перед числовым кодом (в 16-ричном виде) следует указать спец-символ **`\u`**. Например,

```
print('\u2603'). #  получим маленького снеговика: ☃
```

Все строки в Python хранятся именно как последовательность юникод-символов.

Функция `ord`

Для того чтобы узнать код некоторого символа, существует функция **`ord`** (от order – порядок).

```
ord('Б')  # =>  1041
```

Функция `chr`

Зная код, всегда можно получить соответствующий ему символ. Для этого существует функция **`chr`** (от character – символ):

```
chr(1041)  # =>  Б
```

Строки. Срезы

В самом простом варианте срез (slice) строки – ее кусок от одного индекса включительно и до другого не включительно (как для `range`). То есть это новая, более короткая строка.

Срез записывается с помощью квадратных скобок, в которых указывается *начальный* и *конечный* индекс, разделенные двоеточием.

```
text = 'Hello, world!'
print(text[0:5])   # Hello
print(text[7:12])  # world
```

Если не указан начальный индекс, срез берется от начала (от 0). Если не указан конечный индекс, срез берется до конца строки.

```
text = 'Hello, world!'
print(text[:5])   # Hello
print(text[7:])   # world!
```

Разрешены отрицательные индексы для отсчета с конца списка. В следующем примере из строки, содержащей фамилию и инициалы, будет извлекаться фамилия.

```
full_name = 'Петров И. А.'
print(full_name[:-6])
```

Как и для `range`, в параметры среза можно добавить третье число – шаг обхода. Этот параметр не является обязательным и записывается через второе двоеточие.

Интересное отличие среза от обращения по индексу к отдельному элементу состоит в том, что мы не получим ошибку при указании границ среза за пределами строки. В срез в таком случае попадут только те элементы, которые находятся по валидным индексам среза.

```
dog = 'Собака'
print(dog[2:200])  #  бака
```

Шаг может быть и отрицательным – для прохода по строке в обратном порядке. Если в этом случае не указать начальный и конечный индекс среза, ими станут последний и первый индексы строки соответственно (а не наоборот, как при положительном шаге).

```
text = 'Собака'
text_rev = text[::-1]
print(text_rev)  #  акабоС
```

Итак, с помощью квадратных скобок можно получить доступ как к одному символу строки, так и к некоторой последовательности символов, причем совсем необязательно идущих подряд!

Списки

Мы уже знаем тип данных, который называется **множество** и является **коллекцией (контейнером)**, то есть позволяет хранить несколько элементов данных, и тип, который тоже обладает свойствами коллекции и называется **строка**. Теперь мы познакомимся с еще одним типом-коллекцией, который называется **список** (list).

Списки являются очень гибкой структурой данных и широко используются в программах. Рассмотрим основные свойства списка в сравнении с теми коллекциями, которые мы уже знаем:

- Список хранит несколько элементов под одним именем (как и множество)
- Элементы списка могут повторяться (в отличие от множества)
- Элементы списка упорядочены и проиндексированы, доступна операция среза (как в строке)
- Элементы списка можно изменять (в отличие от строки)
- Элементами списка могут быть значения любого типа: целые и действительные числа, строки и даже другие списки

Создание списков

Для создания списка используется операция присваивания.

Чтобы задать готовый список, нужно справа от знака присваивания в квадратных скобках перечислить его элементы через запятую. Здесь создается список из первых пяти простых чисел, который помещается в переменную `primes` (простые числа):

```
primes = [2, 3, 5, 7, 11]
print(primes)  # выводим на экран список целиком
```

Для того чтобы создать пустой список, можно воспользоваться конструкцией [ ] или функцией `list`.

```
empty1 = [ ]  # это пустой список
empty2 = list()  # и это тоже пустой список
```

Ранее при изучении операций со строками был рассмотрен способ создания строки из заданного количества повторений фрагмента. Такую строку можно легко составить путем умножения на число:

```
print('Abc'*3)  # AbcAbcAbc
```

Аналогично можно поступать и со списком:

```
print([1, 2, 3]*3)   # [1, 2, 3, 1, 2, 3, 1, 2, 3]
my_list = [0]*3      # [0, 0, 0]
```

Индексация в списках

Чтобы получить отдельный элемент списка, нужно записать после него (или имени переменой, связанной с данным списком) в квадратных скобках номер (индекс) нужного элемента. Индекс отсчитывается с нуля, как в строках. Так же, как и в строках, для нумерации с конца разрешены отрицательные индексы.

```
print('Сумма первых 2 простых чисел:', primes[0] + primes[1])
print('Последнее из простых чисел в списке:', primes[-1])
```

Как и в строках, попытка обратиться к элементу с несуществующим индексом вызовет ошибку:

```
print(primes[5])    # ошибка: index out of range
```

Добавление элемента в список

Добавление элемента в конец списка делается при помощи метода `append` (этот метод аналогичен методу `add`, используемому для добавления элементов в множество):

```
primes.append(13)
primes.append(19)
```

Кроме того, расширить имеющийся список можно любым итерабельным (перечисляемым) объектом с помощью метода `extend`:

```
list1 = [10, 20, 30]
list2 = [40, 50, 60]
list1.extend(list2)
print(list1)  # [10, 20, 30, 40, 50, 60]
```

Расширение списка строкой

Строка является итерируемой, поэтому, если расширить список строкой, то добавится каждый символ – например:

```
list1 = [10, 20, 30]
list2 = 'Сумма'
```

```
list1.extend(list2)
print(list1)  # [10, 20, 30, 'C', 'y', 'м', 'м', 'a']
```

Расширение списка множеством

Множество также является итерируемым типом, поэтому если расширить список множеством, то элементы множества добавятся в конец списка, но в произвольном порядке:

```
list1 = [10, 20, 30]
list2 = {'A', 'B', 'C'}
list1.extend(list2)
print(list1)  # [10, 20, 30, 'C', 'A', 'B']
```

Изменение элемента списка

В отличие от отдельных символов в строках, элемент списка можно поместить слева от ”**=**” в операторе присваивания и тем самым изменить этот элемент:

```
primes[6] = 17   #  Исправляем ошибку
```

Тем не менее, многие вещи, которые можно делать со строками, можно делать и со списками:

```
print(len(primes))   # выводим длину списка
primes += [19,23,29]   # списки можно складывать, как и строки
print(primes)  # выведет [2, З, 5, 7, 11, 13, 17, 19, 23, 29]
#     можно проверять, содержится ли в списке элемент
if 1 in primes:
    print('Мы считаем единицу простым числом')
e1se:
    print('Мы не считаем 1 простым числом. И это правильно')
```

Перебор элементов списка

Во время выполнения программы текущее количество элементов списка всегда известно. Поэтому, если нужно что-то сделать с каждым элементом списка (например, напечатать его на экране), можно перебрать элементы с помощью цикла **`for`**. При этом, как и для строк, возможны два варианта перебора – перебор индексов и перебор самих элементов.

```
for i in range(len(primes)):
    # выведем по очереди все элементы списка
    print('Простое число номер', i+1, primes[i])
for p in primes:
    print('Квадрат числа', p, '=', p**2)  # и их квадраты
```

Заметим, что при использовании конструкции

```
for i in range(len(имя_списка))
```

индексы перебираются в цикле очень удобно: от 0 включительно до длины списка не включительно. Таким образом, можно перебрать все элементы списка.

Цикл `for` нередко используется и для формирования списка, если мы заранее знаем, сколько элементов в нем должно быть:

```
n = 10
a = []
print('Введите', n, 'значений:')
for i in range(n):
    a.append(input())
print('Получился список строк:', a)
```

Срезы списков

Как и для строк, для списков определена операция взятия среза:

```
months = ['январь', 'февраль', 'март', 'апрель', 'май', 'июнь', 'июль', 'август', 'сентябрь', 'октябрь', 'ноябрь', 'декабрь']
spring = months[2:5]   # spring == ['март', 'апрель', 'май']
for month in spring:
    print(month)
```

Использование срезов

Срезы можно использовать и для присваивания новых значений элементам списка. Например, если мы решим перевести на английский названия летних месяцев, это можно сделать с помощью среза:

```
months[5:8] = ['June', 'July', 'August']
```

Теперь список months будет выглядеть так:

```
['январь', 'февраль', 'март', 'апрель', 'май', 'June', 'July', 'August', 'сентябрь', 'октябрь', 'ноябрь', 'декабрь']
```

Удаление элементов

С помощью функции `del` можно удалять элементы списка.

```
m = ['01','02','03','04','05','06','07','08','09','10']
del m[3]
print(m)  # ['01','02','03','05','06','07','08','09','10']
```

Функция `del` работает и со срезами: например, так можно удалить все элементы с четными индексами исходного списка:

```
a = [1, 2, 3, 4, 5, 6, 7, 8]
del a[::2]
```

```
print(a)   # [2, 4, 6, 8]
```

Списки и массивы

Во многих языках программирования (да и в самом Python, в недрах стандартной библиотеки есть другой тип данных с похожими свойствами – массив. Поэтому списки иногда называют массивами, хоть это и не совсем правильно.

Элементы массива имеют одинаковый тип и располагаются в памяти одним куском, а элементы списка могут быть разбросаны по памяти как угодно и могут иметь разный тип. Все это замедляет работу списков по сравнению с массивами, но придает им гораздо большую гибкость. Из этого различия вытекает и «питонский путь» формирования списка: не создавать «пустой массив» и заполнять его значениями, а добавлять значения к изначально пустому списку.

Кортежи

Мы уже знаем такие коллекции, как списки, множества и строки. Теперь мы рассмотрим еще один тип данных, являющийся коллекцией, который называется **кортеж** – tuple.

Кортежи очень похожи на списки, они тоже являются индексированной коллекцией, только вместо квадратных в них используются круглые скобки (причем их часто можно пропускать):

```
#  кортеж из двух элементов; тип элементов может быть любой
card = ('7', 'пик')
#  пустой кортеж (из 0 элементов)
empty = ( )
# кортеж из 1 элемента - запятая, чтобы отличить от обычных скобок
t = (18,)
# длина, значение отдельного элемента, сложение - как у списков
print(len(card), card[0], card + t
```

Кортежи можно сравнивать между собой:

```
(1, 2) == (1, 3)  # Fa1se
(1, 2) < (1, 3)    # True
(1, 2) < (5,)     # True
('7', 'червей') < ('7', 'треф')     # True
```

Обратите внимание: операции == и != применимы к любым кортежам, независимо от типов элементов. А вот операции <, >, <=, >= применимы только в том случае, когда соответствующие элементы кортежей имеют один тип. Поэтому сравнивать `('7', 'червей')` и `('7', 'треф')` можно, а вот кортежи `(1, 2)` и `('7', 'пик')` нельзя – интерпретатор Python выдаст

ошибку. При этом сравнение происходит последовательно элемент за элементом, а если элементы равны – просматривается следующий элемент.

Неизменяемость

Важнейшее техническое отличие кортежей от списков – неизменяемость. Как и к строке, к кортежу нельзя добавить элемент методом `append`, а существующий элемент нельзя изменить, обратившись к нему по индексу. Это выглядит недостатком, но в дальнейшем мы поймем, что у кортежей есть и преимущества.

Есть и семантическое, то есть смысловое, отличие. Если списки предназначены скорее для объединения неопределенного количества однородных сущностей, то кортеж – быстрый способ объединить под одним именем несколько разнородных объектов, имеющих различный смысл.

Так, в примере выше кортеж `card` состоит из двух элементов, означающих достоинство карты и ее масть.

Еще одним приятным отличием кортежей от списков является то, что они могут быть элементами множества:

```
a = {('7', 'треф'), ('7', 'червей')}
print(a)      # {('7', 'треф'), ('7', 'червей')}
```

Присваивание кортежей

Кортежи можно присваивать друг другу. Именно благодаря этому работает красивая особенность Python – уже знакомая нам конструкция вида `a,b = b,a`

Как известно, по левую сторону от знака присваивания = должно стоять имя переменной либо имя списка с индексом или несколькими индексами. Они указывают, куда можно «положить» значение, записанное справа от знака присваивания. Однако слева от знака присваивания можно записать и кортеж из таких обозначений (грубо говоря, имен переменных), а справа – кортеж из значений, которые следует в них поместить. Значения справа указываются в том же порядке, что и переменные слева (здесь скобки вокруг кортежа необязательны):

```
n, s = 10, 'Hello'
# то же самое, что
n = 10
s = 'Hello'
```

В примере выше мы изготовили кортеж, стоящий справа от =, прямо на этой же строчке. Но можно заготовить его и заранее:

```
cards = [('7', 'пик'), ('Д', 'треф'), ('Т', 'пик')]
value, suit = cards[0]
print('Достоинство карты:', value )
print('Масть карты: ', suit)
```

Самое приятное: сначала вычисляются все значения справа, и лишь затем они кладутся в левую часть оператора присваивания. Поэтому можно, например, поменять местами значения переменных a и b ,написав: a,b = b,a

```
a, b = 1, 2   # здесь   a = 1, b = 2
a, b = b, a   # теперь  a = 2, b = 1
```

С использованием кортежей многие алгоритмы приобретают волшебную краткость. Например, вычисление чисел Фибоначчи:

```
n = int(input())
f1, f2 = 0, 1
for i in range(n):
    print(f2)
    f1, f2 = f2, f1 + f2
```

Сортировка пузырьком

Итак, у нас есть удобный способ поменять местами значения двух переменных. Теперь рассмотрим алгоритм, в котором эта операция играет важную роль.

Часто бывает нужно, чтобы данные не просто содержались в списке, а были отсортированы (например, по возрастанию), то есть, чтобы каждый следующий элемент списка был не меньше предыдущего.

В качестве данных могут выступать числа или строки. Скажем, отсортированный список [4, 1 ,9, 3, 1] примет вид [1, 1, 3, 4, 9].

Классический алгоритм сортировки – сортировка пузырьком (или сортировка обменом).

Она называется так потому, что элементы последовательно «всплывают» (отправляются в конец списка), как пузырьки воздуха в воде. Сначала всплывает самый большой элемент, за ним – следующий по старшинству и т. д. Для этого мы сравниваем по очереди все соседние пары и при необходимости меняем элементы местами, ставя больший элемент на более старшее место

А полный код программы, которая считывает, сортирует и выводит список, выглядит, например, так:

```
n = int(input( ))     # количество элементов
a = []
for i in range(n):     # считываем элементы списка
    a.append(int(input( )))
for i in range(n-1):
    for j in range(n - 1 - i):
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1], a[j]

print(a)
```

Преобразования между коллекциями

Итак, на данный момент мы знаем уже четыре вида коллекций: строки, списки, множества и кортежи.

Может возникнуть вопрос: можно ли из одной коллекции сделать другую? Например, преобразовать строку в список или во множество? Конечно, да, для этого можно использовать функции `list`, `set` и `tuple` Если в качестве аргумента передать этим функциям какую-либо коллекцию, новая коллекция будет создана на ее основе.

Зачем нужно преобразование коллекций?

Преобразование строки в список позволяет получить список символов. В некоторых задачах это может быть полезно: например, если мы хотим изменить один символ строки:

```
s = 'карова'    # написали с ошибкой
a = list(s)     # a = ['к', 'а', 'р', 'о', 'в', 'а']
a[1] = 'о'      # a = ['к', 'о', 'р', 'о', 'в', 'а']
```

С этой же целью может потребоваться преобразование кортежа в список:

```
# В кортеже (писатель, дата рождения) допущена ошибка
writer = ('Лев Толстой', 1827)
a = list(writer) # a = ['Лев Толстой', 1827]
a[1] = 1828      # a = ['Лев Толстой', 1828]
```

Преобразование списка или строки во множество позволяет получить очень интересные результаты. Как вы помните, все элементы множества должны быть уникальны, поэтому при преобразовании списка во множество каждый элемент останется только в одном экземпляре. Таким образом, можно очень легко убрать повторяющиеся элементы и узнать, сколько элементов встречается в списке хотя бы один раз:

```
a = [1, 2, 1, 1, 2, 2, 3, 3, 2]
print('Количество элементов в списке без повторений:',
   len(set(a)))
```

Таким же образом можно получить все буквы без повторений, которые встречаются в строке:

```
a = set('Преобразование строки')
print(a, len(a))
# {'к','н','и','а','з',' ','б','е','П','т','р','в','о','с'} 14
```

Преобразование множества в список тоже возможно, но при этом нужно учитывать, что элементы множества не упорядочены и при преобразовании множества в список порядок элементов в нем предсказать заранее не всегда возможно:

```
names = {'Иван', 'Петр', 'Сергей'}
```

```
print(list(names) )
# Возможные варианты вывода на экран: ['Сергей', 'Иван', 'Петр'],
# ['Сергей', 'Петр', 'Иван'], ['Петр', 'Иван', 'Сергей']
#  и так далее
```

Методы `split` и `join`

Изучая множества и списки, мы уже неоднократно встречались с методами-функциями, «приклеенными» к объекту (списку или множеству) и изменяющими его содержимое.

Методы есть не только у списков и множеств, но и у строк. Сегодня мы изучим два очень полезных метода строк – **`split`** и **`join`** Они противоположны по смыслу: **`split`** разбивает строку по произвольному разделителю на список «слов», а **`join`** собирает из списка слов единую строку через заданный разделитель.

Чтобы вызвать эти методы, необходимо использовать уже знакомый нам синтаксис. После имени переменной, содержащей объект-строку, или просто после строки через точку пишется имя метода, затем в круглых скобках указываются аргументы. **`split`** и **`join`**, в отличие, например, от метода списков **`append`** или метода множеств **`add`**, не изменяют объект, которому принадлежат, а создают новый (список или строку) и возвращают его, как это делают обычные функции типа **`len`**

Метод `split`

Метод `split` можно вызвать вообще без аргументов или с одним аргументом-строкой. В результате строка разбивается на части, разделенные любыми символами пустого пространства (набором пробелов, символом табуляции и т. д.). Во втором случае разделителем слов считается строка-аргумент. Из получившихся слов формируется список.

В этом примере все сравнения истинны, т. е. все вызовы функции `print` выведут True.

```
s = ' раз два три'
print(s.split())                        # ['раз', 'два', 'три']
print('   one two   three'.split()) #  ['one', 'two', 'three']
print('192.168.1.1'.split('.'))      # ['192', '168', '1', '1']
print('A##B##C'.split('##'))         # ['A','B','C]
```

Метод `join`

`join` же всегда принимает один аргумент – список слов, которые нужно склеить. Разделителем (точнее, «соединителем») служит та самая строка, чей

метод **join** вызывается. Это может быть и пустая строка, и пробел, и символ новой строки, и что угодно еще

```
s = ['Тот', 'Кого', 'Нельзя', 'Называть']
print(''.join(s))    # 'ТотКогоНельзяНазывать'
print(' '.join(s))   # 'Тот Кого Нельзя Называть'
print('-'.join(s))   # 'Тот-Кого-Нельзя-Называть'
print('|'.join(s))   # 'Тот|Кого|Нельзя|Называть'
```

Итак, **split** служит для преобразования строки в список, а **join** – для преобразования списка в строку:

Обратите внимание: **split** и **join** – методы строк. Попытка вызвать такой метод у объекта, не являющегося строкой, вызовет ошибку!

Списочные выражения

Для генерации списков из неповторяющихся элементов в Python имеется удобная синтаксическая конструкция – списочное выражение (*list comprehension*). Она позволяет создавать элементы списка в цикле `for`, не записывая цикл целиком.

Например, если нам необходимо создать список квадратов целых чисел от 0 до 9 включительно, мы можем записать следующий код:

```
squares = []
for i in range(10):
    squares.append(i ** 2)
print(squares)    # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

То же самое, но гораздо короче можно сделать с помощью списочного выражения:

```
squares = [i**2 for i in range(10)]
print(squares)
```

А если нам необходимы квадраты не всех чисел, а только четных? Тогда можно добавить условие:

```
even_squares
for i in range(10):
    if i % 2 == 0:
        even_squares.append(i**2)
print(even_squares)     # [0, 4, 16, 36, 64]
```

То же самое, но короче, с помощью списочного выражения:

```
even_squares = [i**2 for i in range(10) if i % 2 == 0]
print(even_squares)
```

В списочном выражении можно пройти по двум или более циклам:

```
print([i*j for i in range(3) for j in range(3)])
# 0, 0, 0, 0, 1, 2, 0, 2, 4
```

На самом деле квадратные скобки не являются неотъемлемой частью списочного выражения. Если их не поставить, выражение будет вычисляться по мере надобности – когда очередной элемент становится нужен. Заключая списочное выражение в квадратные скобки, мы тем самым даем инструкцию сразу создать все элементы и составить из них список. Пока что мы в основном будем пользоваться именно такими списочными выражениями – заключенными в квадратные скобки и превращенными таким образом в список.

Использование списочных выражений

Списочные выражения часто используются для инициализации списков. Дело в том, что в Python не принято создавать пустые списки, чтобы потом заполнять их значениями, если можно этого избежать.

Если все-таки необходимо создать пустой список (скажем, длиной 10) и заполнить его нулями (не может же он быть совсем пустой), это легко сделать, используя умножение списка на число: [0] * 10.

Списочные выражения часто используют в аргументах методов `split` и `join`. Например, комбинация метода `split` и списочного выражения позволяют нам удобно считать числа, записанные в одну строку.

```
a = [int(x) for x in '976 929 289 80 7677'.split()]
evil, good = [int(x) for x in '666 777'.split()]
print(evil, good)      #  666 777
```

Здесь строка (обычно она не задается прямо в выражении, а получается из `input()`) разделяется на отдельные слова с помощью `split`. Затем списочное выражение пропускает каждый элемент получившегося списка через функцию `int`, превращая строку '976' в число 976. Можно собрать все получившиеся значения в один список или разложить их по отдельным переменным с помощью множественного присваивания, как во второй строчке примера.

Рассмотрим и пример использования метода `join` вместе со списочным выражением. Выведем на одной строке список квадратов натуральных чисел от 1 до 9: 1^2=1, 2^2=4, 3^2=9,. . . Для этого сначала с помощью списочного выражения сформируем список строк вида ['1^2=1', '2^2=4', '3^2=9',. . .], а затем «склеим» его в одну строку методом `join`:

```
print(', '.join(str(i)+'^2='+ str(i**2) for i in range(1,10)))
```

Заметьте, что в аргументе функции `join` стоит списочное выражение, не заключенное в квадратные скобки (это можно сделать, но необязательно).

Будьте внимательны! Обычно возведение в степень обозначают «крышечкой» перед степенью, но в Python эта «крышечка» обозначает совсем

другое, а возведение в степень выполняется оператором **. В примере «крышечка» используется только как обозначение операции возведения в степень.

Методы списков и строк

В материалах занятия приводятся таблицы с почти полным перечнем методов списков и строк, которые можно использовать как справочный материал. Рассматривается неявное приведение объектов к булеву типу. Приводятся примеры цепочек вызова методов. Появление метода **pop** позволяет познакомиться с понятием стека.

Эта тема отличается от прочих: большую часть его материала не нужно запоминать, но можно использовать как справочный материал. С этой же целью вводятся функции `dir` и `help`.

Мы уже знакомы с некоторыми функциями для работы со списками, строками и множествами. Мы знаем также методы для работы с элементами множеств (`add`, `pemove`, `discard` и т. д.), метод `append`, который позволяет добавлять элементы в список, метод `split` для разбиения строки на список «слов» и метод `join` для «сшивки» списка строк в одну. Однако мы не знаем, как делать со списками многие другие вещи: добавлять элементы в произвольное место, удалять элементы и т. д. Такие задачи решаются с помощью других функций и методов.

Далее приводится перечень основных методов списков и строк, а также функций и специальных синтаксических конструкций для работы с ними, включая уже знакомые нам. Если при написании программы вы забудете, как называется тот или иной метод или какие у него аргументы, смело заглядывайте в материалы этого занятия, в документацию, пользуйтесь поиском в Интернет описанными здесь функциями *`dir`* и *`help`*.

Главное – усвоить основные возможности методов для стандартных типов строки и списка, а детали (например, порядок аргументов в методе `insert` или название именованного аргумента в методе `split`) всегда можно посмотреть здесь или в документации.

В таблице предполагается, что **`a`** и **`a2`** – списки, **`x`** – элемент списка, **`s`** и **`s2`** – строки, **`c`** – символ (строка длины 1), **`n`** – индекс элемента списка или строки, `start`, `stop`, `step` – границы среза (т. е. тоже индексы) и шаг среза, **`k`** – натуральное число.

Для методов и функций также даны примеры выражений, которые будут истинными `True` и демонстрируют действие этого метода. Например, выражения `5 in [2,3,5]` и `'abc'.isalpha()` равны `True`. Их можно подставить в оператор `if` (и соответствующий блок будет выполнен) или в функцию `print` (и тогда она выведет `True`).

Методы списков

Операция	Описание	Пример
`x in a`	Проверка, что x содержится в a	`5 in [2, 3, 5]`
`x not in a`	Проверка, что x не содержится в a То же, что и `not (x in a)`	`5 not in [2, 3, 6]`
`a + a2`	Конкатенация списков, то есть новый список, в котором сначала идут все элементы a, а затем все элементы a2	`[2, 4] + [5, 3] == [2, 4, 5, 3]`
`a * k`	Список a, повторенный k раз	`[2, 3] * 3 == [2, 3, 2, 3, 2, 3]`
`a[n]`	n-й элемент списка, отрицательные n — для отсчета с конца	`[2, 3, 7][0] == 2` `[2, 3, 7][-1] == 7`
`a[start:stop:step]`	Срез списка	`[2, 3, 7][:2] == [2, 3]`
`len(a)`	Длина списка	`len([2, 3, 7]) == 3`
`max(a)`	Максимальный элемент списка	`max([2, 3, 7]) == 7`
`min(a)`	Минимальный элемент списка	`min([2, 3, 7]) == 2`
`sum(a)`	Сумма элементов списка	`sum([2, 3, 7]) == 12`
`a.index(x)`	Индекс первого вхождения x в a (вызовет ошибку, если `x not in a`, то есть если x отсутствует в a)	`[2, 3, 7].index(7) == 2`
`a.count(x)`	Количество вхождений x в a	`[2, 7, 3, 7].count(7) == 2`

`a.append(x)`	Добавить x в конец a	`a = [2, 3, 7]`
		`a.append(8)`
		`a == [2, 3, 7, 8]`

`a.extend(a2)`	Добавить элементы коллекции a2 в конец a	`a = [2, 3, 7]`
		`a.extend([8, 4, 5])`
		`a == [2, 3, 7, 8, 4, 5]`

`del a[n]`	Удалить n-й элемент списка	`a = [2, 3, 7]`
		`del a[1]`
		`a == [2, 7]`

`del a[start:stop:step]`	Удалить из a все элементы, попавшие в срез	`a = [2, 3, 7]`
		`del a[:2]`
		`a == [7]`

Операция	Описание	Пример
`a.clear()`	Удалить из `a` все элементы (то же, что `del a[:]`)	`a.clear()`
`a.copy()`	Копия `a` (то же, что и полный срез `a[:]`)	`b = a.copy()`
`a += a2` `a *= k`	Заменить содержимое списка на `a + a2` и `a * k` соответственно	

`a.insert(n, x)`	Вставить `x` в `a` на позицию `n`, подвинув последующую часть дальше	`a = [2, 3, 7]`
		`a.insert(0, 8)`
		`a == [8, 2, 3, 7]`

`a.pop(n)`	Получить `n`-й элемент списка и одновременно удалить его из списка. Вызов метода без аргументов равносилен удалению последнего элемента:
	`a.pop() == a.pop(-1)`
		`a = [2, 3, 7]`
		`a.pop(1) == 3`
		`a == [2, 7]`

`a.remove(x)`	Удалить первое вхождение `x` в `a`, в случае `x not in a` — ошибка	`a = [2, 3, 7]`
		`a.remove(3)`
		`a == [2, 7]`

`a.reverse()`	Изменить порядок элементов в `a` на обратный (перевернуть список)	`a = [2, 3, 7]`
		`a.reverse()`
		`a == [7, 3, 2]`

`a.sort()`	Отсортировать список по возрастанию	`a = [3, 2, 7]`
		`a.sort()`
		`a == [2, 3, 7]`

`a.sort(reverse=True)`	Отсортировать список по убыванию	`a = [3, 2, 7]`
		`a.sort(reverse = True)`
		`a == [7, 3, 2]`

`bool(a)`	Один из способов проверить список на пустоту (возвращает `True`, если список непустой, и `False` в противном случае)	

Методы строк

Операция	Описание	Пример
`s2 in s`	Проверка, что подстрока `s2` содержится в `s`	`'m' in 'team'`
`s2 not in s`	Проверка, что подстрока `s2` не содержится в `s` то же, что `not (s2 in s)`	`'I' not in 'team'`
`s + s2`	Конкатенация (склейка) строк, то есть строка, в которой сначала идут все символы из `s`, а затем все символы из `s2`	`'tea' + 'm' == 'team'`
`s * k`	Строка `s`, повторенная `k` раз	`'ha' * 3 == 'hahaha'`
`s[n]`	`n`-й элемент строки, отрицательные `n` — для отсчета с конца	`'team'[2] == 'a'` `'team'[-1] == 'm'`
`s[start:stop:step]`	Срез строки	`'mama'[:2] == 'ma'`
`len(s)`	Длина строки	`len('abracadabra') == 11`

`s.find(s2)`
`s.rfind(s2)`
	Индекс начала первого или последнего вхождения подстроки `s2` в `s` (вернет `-1`, если `s2 not in s`)	`s = 'abracadabra'`
		`s.find('ab') == 0`
		`s.rfind('ab') == 7`
		`s.find('x') == -1`

`s.count(s2)`	Количество неперекрывающихся вхождений `s2` в `s`	`'abracadabra'.count('a') == 5`
`s.startswith(s2)` `s.endswith(s2)`	Проверка, что `s` начинается с `s2` или оканчивается на `s2`	`'abracadabra'.startswith('abra')`
`s += s2` `s *= k`	Заменить содержимое строки на `s + s2` и `s * k` соответственно	

`s.isdigit()`
`s.isalpha()`
`s.isalnum()`
	Проверка, что в строке `s` все символы — цифры, буквы (включая кириллические), цифры или буквы соответственно	`'100'.isdigit()`
		`'abc'.isalpha()`
		`'E315'.isalnum()`

`s.islower()` `s.isupper()`	Проверка, что в строке `s` не встречаются большие буквы, маленькие буквы. Обратите внимание, что для обеих этих функций знаки препинания и цифры дают True	`'hello!'.islower()` `'123PYTHON'.isupper()`

Операция	Описание	Пример
`s.lower()` `s.upper()`	Строка `s`, в которой все буквы (включая кириллические) приведены к верхнему или нижнему регистру, т. е. заменены на строчные (маленькие) или заглавные (большие)	`'Привет!'.lower() == 'привет!'` `'Привет!'.upper() == 'ПРИВЕТ!'`
`s.capitalize()`	Строка `s`, в которой первая буква — заглавная	`'привет'.capitalize() == 'Привет'`

`s.lstrip()`
`s.rstrip()`
`s.strip()`
	Строка `s`, у которой удалены символы пустого пространства (пробелы, табуляции) в начале,
	в конце или с обеих сторон
		`' Привет! '.strip() == 'Привет!'`

`s.ljust(k, c)` `s.rjust(k, c)`	Добавляет справа или слева нужное количество символов `c`, чтобы длина `s` достигла `k`	`'Привет'.ljust(8, '!') ==` `'Привет!!'`
`s.join(a)`	Склеивает строки из списка `a` через символ `s`	`'+'.join(['Вася', 'Маша']) ==` `'Вася+Маша'`
`s.split(s2)`	Список всех слов строки `s` (подстрок, разделенных строками `s2`)	`'Раз два три!'.split('а') ==` `['Р', 'з дв', ' три!']`

`s.replace(s2, s3)`	Строка `s`, в которой все неперекрывающиеся
	вхождения `s2` заменены на `s3`
	Есть необязательный третий параметр, с помощью которого можно указать, сколько раз производить замену
		`'Раз два три!'.replace('а', 'я')`
		`=='Ряз двя три!'`
		`'Раз два три!'.replace('а', 'я', 1)`
		`== 'Ряз два три!'`

`list(s)`	Список символов из строки строки `s`	`list('Привет') ==` `['П', 'р', 'и', 'в', 'е', 'т']`
`bool(s)`	Проверка, что строка не пустая (возвращает `True`, если не пустая, и `False` в противном случае)	
`int(s)` `float(s)`	Если в строке `s` записано целое (дробное) число, получить это число, иначе — ошибка	`int('25') == 25`
`str(x)`	Представить любой объект `x` в виде строки	`str(25) == '25'`

Обратите внимание: никакие методы строк, включая `s.replace( )`, не изменяют саму строку **`s`**. Все они лишь возвращают измененную строку, в отличие от большинства методов списков. `a.sort()`, например, ничего не возвращает, а изменяет сам список **`a`**.

Функции `dir` и `help`

Для получения информации о списке методов любого объекта (в том числе тех, о которых вы вряд ли хотели узнать) в Python существует специальная функция `dir`. Например, `dir([ ])` вернет все методы списков, и оператор `print(dir([ ]))` выведет длинный список, оканчивающийся так:

`'index', 'insert', 'pop' , 'remove', 'reverse' , 'sort'`.

Если же нам нужно узнать справочную информацию по конкретному методу или типу данных, для этого существует функция `help`. `help([ ])` выведет на экран много информации, большая часть которой пока лишняя. А вот `help([ ].insert)` выведет на экран краткую справку именно по методу `insert`, подсказывая, что первый аргумент этого метода – индекс, а второй – тот объект, который нужно вставить в список на этот индекс.

Заметьте: при вызове справки по методу `help([ ].insert)` после `insert` нет скобок (...) – ведь мы не хотим вызвать этот метод, чтобы вставить что-то в какой-то список. Функции `he1p` в качестве аргумента передается сам метод `insert`

Цепочки вызовов

Бывает удобно строить последовательные цепочки вызовов методов.

Например, `s.strip( ).lower().rep1ace('ё', 'e')` выдаст строку **`s`**, в которой убраны начальные и конечные пробелы, все буквы сделаны маленькими, после чего убраны все точки над Ё. В результате этого преобразования строка 'Зелёный клён' превратится в 'зеленый клен'.

Другой пример:

'мало Средн МНОГО'.lower().split().index('средне')

В данном примере строка сначала полностью приводится к нижнему регистру (все буквы при помощи метода lower(). Затем она превращается в список слов [' мало', 'средне', 'много'] при помощи split(). Далее метод index находит в этом списке слово «средне» на позиции номер 1.

Использование методов списков. Структура данных «Стек»

Приведем пример использования методов списков `append` и `pop` для моделирования структуры данных «Стек». Эта структура данных часто используется при решении различных задач (например, при вычислении выражений).

Представьте себе стопку сложенных футболок или любых других неодинаковых предметов, которые можно сложить в стопку. Из стопки удобно забрать самый верхний предмет – тот, который положили в нее последним. Если вы полностью разберете стопку, будете брать футболки в порядке, обратном тому, в котором их в нее клали.

Структура данных называется стек (stack, что и значит «стопка»). По-английски такую организацию данных называют LIFO – Last In First 0ut – Первым Пришел Последним Ушел!

Если пользоваться только методами `append` и `pop` без аргументов, список оказывается как раз такой стопкой.

Так, в примере ниже порядок вывода будет обратным порядку ввода:

```
stack = []
for i in range(5):
    print('Какую футболку вы кладёте сверху стопки?')
    stack.append(input())
whi1e stack: # пока стопка не закончилась
    print('Сегодня вы надеваете футболку', stack.pop())
```

Результат:

```
Какую футболку вы кладёте сверху стопки?
белую
Какую футболку вы кладёте сверху стопки?
красную
Какую футболку вы кладёте сверху стопки?
синюю
Какую футболку вы кладёте сверху стопки?
зеленую
Какую футболку вы кладёте сверху стопки?
веселую
Сегодня вы надеваете футболку веселую
Сегодня вы надеваете футболку зеленую
Сегодня вы надеваете футболку синюю
Сегодня вы надеваете футболку красную
Сегодня вы надеваете футболку белую
```

Здесь использовалась конструкция `while stack`: В условии оператора **`if`** или **`while`** любой объект интерпретируется как **`bool`**: либо как `True`, либо как `False`. В случае списков и строк в `False` превращаются только `[]` и `''` соответственно (а в случае чисел – только ноль).

Иными словами, можно было бы написать `while bool(stack):` или `while stack != []` и получить тот же самый результат, но самый короткий и общепринятый вариант – просто `while stack:`

Форматированный вывод. f-строки

Начиная с версии 3.6 в Python появился новый тип строк – f-строки, которые улучшают читаемость кода и работают быстрее других способов форматирования.

f-строки, которые буквально означают formatted string, задаются с помощью литерала f перед кавычками:

```
>>>  'обычная строка'
>>>  f'f-строка'
```

f-строки делают очень простую вещь: они берут значения переменных, которые есть в текущей области видимости, и подставляют их в строку. В самой строке вам лишь нужно указать имя этой переменной в фигурных скобках.

```
name ='Сергей'
age = 19
print(f'Меня зовут {name}. Мне {age} лет.')
```

Результат:

```
Меня зовут Сергей. Мне 19 лет.
```

В качестве подставляемого значения внутри фигурных скобок может быть: имя переменной (`f'Меня зовут {name}'`), элемент списка (`f'Третий месяц в году – {month[2]}'`) или словаря, методы объектов (`f'Имя: {name.upper()}'`). f-строки позволяют выполнять базовые арифметические операции

```
(f'({x} + {y}) / 2 = {(x+y)/2}')
```

и даже вызывать функции `(f'13/7 = {round(13/7)}')`

Кроме того, вы можете задавать форматирование для чисел.

– Указать необходимое количество знаков после запятой, спецификатор **f** отвечает за вывод чисел с плавающей точкой (тип `float`):

```
print(f'Число пи по Архимеду = {(22/7):.2f}')
```

– Представить результат в двоичной системе счисления, используя спецификатор **b**:

```
print(f'10 в двоичной системе счисления = {(10):b}')
```

Аналогично для шестнадцатеричной системы счисления используется спецификатор **x**, а для восьмеричной – **o**.

Вложенные списки. Двумерные вложенные списки (матрицы)

Язык Python не ограничивает нас в уровнях вложенности: элементами списка могут быть списки, их элементами могут быть другие списки, элементами которых в свою очередь могут быть другие списки и т. д. Но для

решения практических задач сначала важно научиться работать с двумерными списками.

С помощью таких списков очень удобно представить прямоугольную таблицу (матрицу) – каждый вложенный список при этом будет являться строкой. Именно такая структура данных используется, например, для представления игровых полей при программировании таких игр, как шахматы, крестики-нолики, морской бой.

Создание двумерного списка

Важно понять, что список списков принципиально ничем не отличается, например, от списка чисел. Чтобы задать список списков в программе, мы также перечисляем элементы через запятую в квадратных скобках:

```
table = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Если элементы списка вводятся через клавиатуру (каждая строка таблицы на отдельной строке, всего **n** строк, числа в строке разделяются пробелами), для ввода списка можно использовать следующий код:

```
table = []
for i in range(n):
    row = [int(el) for el in input().split()]
    table.append(row)
```

В этом примере мы используем метод `append`, передавая ему в качестве аргумента другой список. Так у нас получается список списков.

Списочные выражения

Для создания вложенных списков можно использовать списочные выражения. Например, список из предыдущего примера можно создать так:

```
table=[[int(el) for el in input().split()] for i in range(n)]
```

Попробуем теперь составить список размером 10x10 элементов, заполненный нулями (такая задача нередко возникает при написании различных программ). Может показаться, что сработает конструкция

`a = [[0] * 10] * 10`, но это не так.

Самый короткий способ выполнить такую задачу – при помощи списочного выражения:

```
[[0] for _ in range(10)]
```

Обратите внимание: в этом примере используется переменная. Это вполне законное имя переменной, как и, например, i. Однако по соглашению оно используется для переменной-счетчика только в том случае, когда принимаемые этой переменной значения не важны, а важно лишь количество итераций.

Перебор элементов двумерного списка. Вывод списка на экран

Для доступа к элементу списка мы должны указать индекс этого элемента в квадратных скобках. В случае двумерных вложенных списков мы должны указать два индекса (каждый в отдельных квадратных скобках), в случае трехмерного списка – три индекса и т. д. В двумерном случае сначала указывается номер строки, затем – номер столбца (сначала выбирается вложенный список, а затем – элемент из него).

```
table = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(table[0,0], table[0,1], table[0,2])
```

Для того чтобы перебрать все элементы матрицы (чтобы, например, вывести их на экран), обычно используются вложенные циклы. Например, список из предыдущего примера можно вывести на экран таким образом:

```
for i in range(3):
    for j in range(3):
        print(table[i][j], end='\t')
    print()
```

В этом примере мы перебирали индексы элементов. А что будет, если перебирать сами элементы? Например, если мы хотим подсчитать сумму всех элементов матрицы, можно написать такой цикл:

```
s = 0
for row in table:
    s += sum(row)
print(s)
```

Матрица

Матрица – прямоугольная табличка, заполненная какими-то значениями, обычно числами.

В математике встречается множество различных применений матриц, поскольку с их помощью многие задачи гораздо проще сформулировать и решить. Мы же сконцентрируемся на том, как хранить матрицу в памяти компьютера.

В первую очередь от матрицы нам нужно уметь получать элемент в i-й строке и j-м столбце. Чтобы этого добиться, обычно поступают так: заводят список строк матрицы, а каждая строка сама по себе тоже является списком элементов. То есть, мы получили список списков чисел. Теперь, чтобы получить элемент, нам достаточно из списка строк матрицы выбрать i-ю строку и из неё взять j-й элемент.

Давайте заведем простую матрицу M размера 2x3 (2 строки и 3 столбца) и получим элемент на позиции (1, 3). Обратите внимание: в математике нумерация

строк и столбцов идет с единицы, а не с нуля. И, по договоренности среди математиков, сначала всегда указывается строка, а лишь затем – столбец.

Элемент на i-й строке, j-м столбце матрицы M в математике обозначается $M_{i\,j}$. Итак:

```
matrix = [[1, 2, 3],  [4, 5, 6]]
print(matrix[0][2])
```

`matrix` – вся матрица, `matrix[0]` – список значений в первой строке, `matrix[0][2]` – элемент в третьем столбце в этой строке.

Чтобы перебрать элементы матрицы, приходится использовать двойные циклы. Например, выведем на экран все элементы матрицы, перебирая их по столбцам:

```
for col in range(3):
    for row in range(2):
        print(matrix[row][col])
```

Знакомство со словарями

Списки – удобный и самый популярный способ сохранить большое количество данных в одной переменной. Списки индексируют все хранящиеся в них элементы. Первый элемент, как мы помним, лежит по индексу 0, второй – по индексу 1 и т. д. Такой способ хранения позволяет быстро обращаться к элементу списка, зная его индекс.

```
actors    = ['Джонни Депп', 'Эмма Уотсон', 'Билли Пайпер']
print(actors[1])
```

Представим, что мы делаем свою онлайн-энциклопедию об актерах мирового кино (наподобие Википедии). Для каждого актера нужно сохранить текст статьи о нем. Ее название – строим из фамилии и имени актера. Как правильно хранить такие данные?

Можно создать список кортежей. Каждый кортеж будет состоять из двух строк – названия и текста статьи.

```
actors = [('Джонни Депп', 'Джон Кристофер Депп Второй родился'
           ' 9 июня 1963 года в Овенсборо, Кентукки.')
('Сильвестр Сталлоне',
 'Сильвестр Гарденцио Сталлоне родился в Нью-Йорке.'
 'Его отец, парикмахер Фрэнк Сталлоне – иммигрант из Сицилии.')]
```

Со временем количество статей значительно вырастет. Чтобы найти нужную статью по названию, нам придется написать цикл **`for`**, который пройдет по всем элементам списка `actors` и найдет в нем кортеж, первый элемент которого равен искомому названию.

Корень этой проблемы в том, что списки индексируются целыми числами. Мы же хотим находить информацию не по числу, а по строке – названию статьи. Было бы здорово, если бы индексами могли быть не числа, а строки. В списках это невозможно, однако возможно в словарях!

Словарь (в Python он называется `dict`) – тип данных, позволяющий, как и список, хранить много данных.

В отличие от списка, в словаре для каждого элемента можно самому определить «индекс», по которому он будет доступен. Этот индекс называется ***ключом***.

Создание словаря

Вот пример создания словаря для энциклопедии об актерах мирового кино:

```
actors = {'Джонни Депп': 'Джон Кристофер Депп Второй родился 9 июня 1963 года в Овенсборо, Кентукки',

' Сильвестр Сталлоне': 'Сильвестр Гарденцио Сталлоне родился в Нью-Йорке. Его отец, парикмахер Фрэнк Сталлоне– иммигрант из Сицилии.',

'Эмма Уотсон': 'Эмма Шарлотта Дуерр Уотсон родилась в семье английских адвокатов. В 5 лет переехала вместе с семьей из Парижа в Англию.'}
```

Элементы словаря перечисляются в фигурных скобках (как и элементы множества!) и разделяются запятой. До двоеточия указывается ключ, а после двоеточия – значение, доступное в словаре по этому ключу.

Пустой словарь можно создать двумя способами:

```
d = dict()
# или так
d = {}
```

Вспомните, что создать пустое множество можно, только используя функцию `set( )`. Теперь понятно, почему это так – пустые фигурные скобки зарезервированы для создания словаря.

Обращение к элементу словаря

После инициализации словаря мы можем быстро получать статью про конкретного актера:

```
print(actors['Эмма Уотсон'])
```

Обращение к элементу словаря выглядит как обращение к элементу списка, только вместо целочисленного индекса используется ключ. В качестве ключа можно указать выражение: Python вычислит его значение, прежде чем обратится к искомому элементу.

```
first_name =   'Сильвестр'
last_name = 'Сталлоне'
print(actors[first_name + ' ' +last_name])
```

Если ключа в словаре нет, возникнет ошибка:

```
print(actors['Несуществующий ключ'])
KeyError: 'Несуществующий ключ'
```

Добавление и удаление элементов

Важная особенность словаря – его динамичность. Мы можем добавлять новые элементы, изменять их или удалять. Изменяются элементы точно так же, как в списках, только вместо целочисленного индекса в квадратных скобках указывается ключ:

```
actors['Эмма Уотсон'] = 'Новый текст статьи об Эмме Уотсон '
```

Также в словари можно добавлять новые элементы и удалять существующие.

Добавление элемента

Добавление синтаксически выглядит так же, как и изменение:

```
actors['Брэд Питт']='Уильям Брэдли Питт, более известный как'\
    ' Брэд Питт – американский актёр и продюсер.' \
    ' Лауреат премии «Золотой глобус» за 1995 год.'
```

Удаление элемента

Для удаления можно использовать инструкцию **`del`** (как и в списках):

```
del actors['Джонни Депп']
# больше в словаре нет ни ключа 'Джонни Депп',
# ни соответствующего ему значения
print(actors['Джонни Депп'])
```

Результат:

```
KeyError: 'Джонни Депп'
```

Удалять элемент можно и по-другому:

```
actors.pop('Джонни Депп')
```

Единственное отличие этого способа от вызова de1 – он возвращает удаленное значение. Можно написать так:

```
deleted_value = actors.pop('Джонни Депп')
```

Переменной `deleted_value` присвоится значение, которое хранилось в словаре по ключу 'Джонни Депп'. В остальном этот способ идентичен оператору `del`. В частности, если ключа `'Джонни Депп'` в словаре нет, возникнет ошибка keyError.

Чтобы ошибка не появлялась, этому методу можно передать второй аргумент. Он будет возвращен, если указанного ключа в словаре нет. Это позволяет реализовать безопасное удаление элемента из словаря:

```
deleted_value = actors.pop('Джонни Депп', None)
```

Если ключа `'Джонни депп'` в словаре нет, в `deleted_value` попадет `None`.

Проверка наличия элемента в словаре

Оператор **`in`** позволяет проверить, есть ли ключ в словаре:

```
if 'Джонни Депп' in actors:
    print('У нас есть статья про Джонни Деппа')
```

Проверить, что ключа нет, можно с помощью аналогичного оператора **`not in`**:

```
if 'Сергей Безруков' not in actors:
    print('У нас нет статьи о Сергее Безрукове')
```

Нестроковые ключи

Решим следующую задачу. Пусть дан длинный список целых чисел `numbers`. Мы знаем, что некоторые числа встречаются в этом списке несколько раз. Нужно узнать, сколько именно раз встречается каждое из чисел.

```
numbers = [1, 10, 1, 6, 4, 16, 4, 2, 2, 1, 10, 1]
counts = {}
for number in numbers:
    if number not in counts:
        counts[number] = 1
    else:
        counts[number] += 1
```

Просто так сделать `counts[number] += 1` нельзя: если ключа `number` в словаре нет, возникнет ошибка keyError.

В результате работы этой программы все элементы из списка `numbers` окажутся ключами словаря `counts`. Значением `counts[x]` будет количество раз, которое число **`x`** встретилось в списке `numbers`. Как это работает?

Почему для этой задачи не стоит использовать список, хотя ключи – обычные целые числа? Потому что, используя словарь, мы можем решить эту задачу и для вещественных чисел, и для очень больших целых чисел, и вообще для любых объектов, которые можно сравнивать.

Методы словарей

Взять значение в словаре можно не только с помощью квадратных скобок, но и с помощью метода `get`:

```
article = actors.get('Джонни Депп')
```

Преимущество метода в том, что, кроме ключа, он может принимать и второй аргумент – значение, которое вернется, если заданного ключа нет:

```
article = actors.get('Джонни Депп', \
    'Статья о Джонни Деппа не найдена')
```

Воспользуемся этим приемом для улучшения нашей программы в задаче о повторяющихся числах:

```
numbers = [1, 10, 1, 6, 4, 12, 4, 2, 2, 1, 10, 1]
counts = {}
for number in numbers:
    counts[number] = counts.get(number, 0) + 1
```

Все ключи словаря можно перебрать циклом `for`:

```
for actor_name in actors:
    print(actor_name, actors[actor_name] )
```

Метод keys()

Другой способ сделать то же самое – вызвать метод `keys()`

```
for actor_name in actors.keys():
    print(actor_name, actors[actor_name])
```

С помощью метода `keys()` можно получить список всех ключей словаря:

```
actors_names = list(actors.keys( ))
```

Метод values()

Есть и парный метод `values()`, возвращающий все значения словаря:

```
all_articles = list(actors.values())
```

Он позволяет, например, проверить, есть ли какое-нибудь значение `value` среди значений словаря:

```
value in d.values()
```

Метод items()

Если вы хотите перебрать элементы словаря **`d`** так, чтобы в переменной `key` оказывался ключ, а в `value` – соответствующее ему значение, это можно сделать с помощью метода `items()` и цикла `for`.

```
for key, value in d.items():
```

Например:

```
for actor_name, article in actors.items():
    print(actor_name, article)
```

Допустимые типы ключей в словаре

Мы уже выяснили, что ключами в словарях могут быть строки и целые числа. Кроме этого, ключами могут быть вещественные числа и кортежи.

Ключами в словаре не могут быть другие словари. В принципе в одном словаре могут быть ключи разных типов, однако обычно принято использовать однотипные ключи.

Вообще, есть строгий способ определить, может ли объект быть ключом в словаре. Для этого объект должен быть неизменяемым. Неизменяемые объекты не могут поменять значение в себе во время выполнения программы. Неизменяемыми в Python являются числа, строки и кортежи. Именно их обычно и используют в качестве ключей словарей.

Вот как может выглядеть словарь с ключами-кортежами. В качестве ключа используются координаты, а в качестве значения – название города.

```
cities={
   (55.75, 37.5): 'Москва',
   (59.8, 30.3): 'Санкт-Петербург',
   (54.32, 48.39): 'Ульяновск'
}
print(cities[(55.75, 37.5)])
cities[(53.2, 50.15)] = 'Самара'
```

Возможно, нам захочется развернуть этот словарь, то есть построить такой, в котором ключами будут города, а значениями – их координаты.

```
coordinates = {}
for coordinate, city in cities.items():
    coopdinates[city] = coordinate
```

Если в исходном словаре были повторяющиеся значения, некоторые из них потеряются при разворачивании словаря. Это объясняется тем, что значения в словаре могут повторяться, а вот ключи обязаны быть уникальными.

Значениями в словаре, в отличие от ключей, могут быть объекты любого типа – числа, строки, кортежи, списки и даже другие словари. Вот, например, как можно сохранить список фильмов для каждого из актеров:

```
films = {
   'Джонни Депп': [
      'Эдвард Руки-Ножницы',
      'Одинокий рейнджер',
      'Чарли и шоколадная фабрика'
      ],
   'Эмма Уотсон':
      ['Гарри Поттер и философский камень',
```

```
        'Красавица и Чудовище'
        ]
}
# Вывести список фильмов, в которых снималась Эмма Уотсон
print(films['Эмма Уотсон'])

# Проверить, снимался ли Джонни Депп в фильме «Чарли и шоколадная фабрика»
if 'Чарли и шоколадная фабрика' in films['Джонни Депп']:
    print('Снимался!')
```

Функции

С некоторыми функциями мы уже сталкивались ранее, но это были встроенные функции. Здесь мы узнаем, как создавать собственные функции. Также мы освоим основные приемы по использованию функций в программных кодах.

Когда мы говорим о функции, то имеем в виду фрагмент программного кода, который имеет название, и который по этому названию мы можем вызвать в любом (или практически любом) месте программы. Функции – очень полезный и эффективный инструмент программирования, поскольку позволяет значительно сократить объем программного кода, сделать его понятнее и эффективнее.

Создание функции

Перед тем, как функцию использовать, ее необходимо описать. Что подразумевает описание функции? Как минимум, это тот программный код, который следует выполнять при вызове функции, и, разумеется, название функции. Также функции при выполнении могут понадобиться некоторые параметры, которые передаются функции при вызове и без которых выполнение функции невозможно. Эти параметры мы будем называть *аргументами функции*.

Замечание

Нередко разделяют понятие параметра функции и аргумента функции.·О параметрах обычно говорят при описании функции. Об аргументах идет речь, когда уже описанная функция вызывается в программном коде . Мы , чтобы не усложнять себе жизнь во всех этих случаях будем использовать термин аргумент.

Итак, для описания функции в Python необходимо:

- указать (задать) имя функции;
- описать аргументы функции;
- описать программный код функции.

Описание функции начинается с ключевого слова `def`. После него указывают название функции (это должен быть уникальный идентификатор,

желательно со смысловой нагрузкой, состоящий из латинских букв, можно также цифр и символа подчеркивания – но с цифры имя функции начинаться не может). Затем перечисляются аргументы функции. Для аргументов просто указываются названия. Тип аргументов не указывается. Аргументы перечисляются после имени функции в круглых скобках. Если аргументов несколько, они разделяются запятыми. Даже если аргументов у функции нет (такое тоже бывает), круглые скобки все равно указываются. Заканчивается вся эта конструкция двоеточием, и далее с новой строки следует блок команд, которые собственно и определяют программный код функции. Тело функции, как и в операторе `if` или в операторе цикла, обязательно идет с отступом. Это нужно, чтобы интерпретатор Python знал, где заканчивается код функции.

Весь шаблон объявления функции выглядит следующим образом (жирным шрифтом выделены ключевые элементы):

```
def имя_ функции(аргументы):
    команды
```

То есть, всё достаточно просто и прозрачно.

Замечание

Для функции не указывается тип результата (хотя функция, разумеется, может возвращать результат). Это может стать сюрпризом для тех, кто изучал и знаком с другими языками программирования. Но это сюрприз только на первый взгляд. Если вспомнить, что в языке Python для переменных тип не указывается и определяется по значению, на которое ссылается переменная, то все выглядит вполне логичным: и отсутствие идентификатора типа для результата функции, и аргументы, для которых тип тоже не указывается.

Функция может возвращать результат, а может не возвращать результат (в последнем случае имело бы смысл говорить о процедуре – но в Python все называется одним словом *функция*). Если функция не возвращает результат, то ее можно рассматривать как некоторую последовательность команд, которые выполняются в том месте и в то время, что и вызов функции. Если функция возвращает результат (а это может быть значение практически любого типа), то инструкцию вызова функции можно использовать в выражениях так, как если бы это была некоторая переменная. В теле функции чтобы определить то значение, которое является результатом функции, обычно используют инструкцию **`return`**. Выполнение этой инструкции, во-первых; приводит к завершению выполнения программного кода функции, а во-вторых то значение, которое указано после инструкции **`return`**, возвращается функцией в качестве результата.

Замечание

Вызывается функция просто – указывается имя функции и аргументы, которые ей передаются. Ну и, разумеется, вызывать функцию можно только после того, как функция объявлена, но никак не раньше.

Далее мы рассмотрим простой пример, в котором объявляется несколько простых функций. После объявления эти функции вызываются в программном коде для выполнения несложных действий.

```
# Функция без аргументов
def your_name():
    # Отображается сообщение
    print("Добрый день!")
    # Запоминается введенный пользователем текст
    name = input("Как Вас зовут? ")
    # Результатфункции
    return name

# Функция с одним аргументом
def say_hello(txt):
    # Отображается сообщение
    print("Здравствуйте,", txt + "!")

# Вызываем функцию и результат записываем в переменную
my_name = your_name()
# Вызываем функцию с аргументом
say_hello(my_name)
```

В результате выполнения этого программного кода получаем следующий результат (жирным шрифтом выделено введенное пользователем значение):

```
Добрый день!
Как Вас зовут? Анастасия Орлова
Здравствуйте, Анастасия Орлова!
```

Код достаточно простой, но мы его все же прокомментируем. В программе объявляется две функции. Функция `your_name()` не имеет аргументов. При выполнении этой функции сначала командой `print("Добрый день!")` отображается приветствие. Затем пользователю предлагается ввести свое имя. Введенное пользователем текстовое значение запоминается в переменной `name`. Вся команда выглядит как `name = input("Как Вас зовут?")`.

После этого с помощью инструкции `return name` значение переменной `name` возвращается в качестве результата функции `your_name()`. Таким образом, у этой функции нет аргументов, но зато она возвращает результат. И ее результат – это то, что вводит пользователь (предполагается, что имя пользователя).

Еще одна функция `say_hello()` нужна для отображения приветствия. У функции один аргумент (обозначен как `txt`). Текст сообщения, которое отображается при вызове функции, формируется с учетом значения аргумента

txt, переданного функции. Результат функция не возвращает. В теле функции всего одна команда print("Здравствуйте,", txt + "!"), которой в консольное окно выводится сообщение.

Замечание

При вычислении выражения txt + "!" мы неявно предполагаем, что аргумент txt имеет текстовое значение. Если такой уверенности нет, то лучше перестраховаться и воспользоваться выражением str(txt) + "!", в котором выполняется явное приведение аргумента txt к текстовому типу.

Описанные функции используем следующим образом. Сначала командой my_name = your_name() вызываем функцию your_name() и результат вызова записываем в переменную my_name. Затем вызываем функцию say_hello(my_name), передав ей аргументом переменную my_name. Результат этих действий таков, как показано выше.

Отличие пользовательских функций от математических в том, что функция в Python может зависеть не только от аргументов, но и от внешних причин. Что для функции может быть внешними причинами? Например, действия пользователя.

Функция input, помимо пустого списка аргументов, получает ввод с клавиатуры пользователя. Некоторые функции читают файлы на жестком диске или в Интернете, а значит, их результат зависит от содержимого файла или веб-страницы. Есть функции, работа которых зависит от текущего времени. Есть функции, зависящие от датчика случайных чисел.

Кроме того, работа некоторых функций зависит от внешних (глобальных) переменных (об этом мы будем говорить на следующем занятии). Это еще одна особенность функций в Python: они могут изменять что-то снаружи функции – менять глобальные переменные, выводить текст на экран, записывать что-то в файлы.

Таким образом, функции можно разделить на функции с побочными эффектами и без них.

Функции без побочных эффектов

Функции без побочных эффектов и использования внешних источников данных (их еще называют «чистые функции») ведут себя в точности как математические функции. Их результат зависит только от аргументов, а вызов таких функций никак не влияет на ход остальной программы.

Большая часть известных вам встроенных функций Python ведет себя так: math.sqrt, math.cos, abs, int, str, len, min, max.

Функции с побочными эффектами

Функции с побочными эффектами – как правило, функции, предназначенные для общения с «внешним миром»: пользователем, файлами на жестком диске, другими программами или серверами в Интернете.

Пока что вы знаете две такие функции, встроенные в язык: `input` и `print`.

Побочные эффекты часто используются для изменения аргумента – как в методах sort и reverse ,которые изменяют данные в списке.

Любая функция, которая использует `input()` или `print()`, тоже имеет побочные эффекты. И любая функция, которая изменяет значения внешних переменных, тоже имеет побочные эффекты. На следующем занятии мы отдельно поговорим о том, как функции с побочными эффектами могут влиять на глобальные переменные, а как делать чистые функции и работать с ними, мы начнем говорить сейчас.

Возвращаемые значения

Ранее уже сообщалось, что для того чтобы функция вернула значение, используется оператор `return`. Рассмотрим подробнее действие этого оператора. Использовать его очень просто. Давайте напишем функцию `len_circle`, которая вычисляет длину окружности:

```
def len_circle(d):
    return 3.14*d
```

Эта функция получила число `d` в качестве аргумента (диаметр окружности), умножила его на `3.14` (число π) и вернула результат в основную программу. Значением, которое функция вернула, можно воспользоваться. Например, мы можем посчитать площадь боковой поверхности цилиндра с использованием этой функции:

```
height = 5
d = 4
square = height * len_circle(d)
```

Заметьте: функция `len_circle` ничего не выводит на экран. Она выполняет вычисления и сообщает их не пользователю, как мы делали раньше, а другой части программы.

Если нам потребуется не просто вернуть значение функции, а еще и вывести его на экран, лучше не добавлять `print` внутрь функции. Ведь если вы добавите `print` в функцию, уже никак не сможете вызвать эту функцию в «тихом» режиме, чтобы она ничего не печатала. Вместо этого лучше сначала вернуть результат, а потом уже распечатать его во внешней программе, если потребуется:

```
d = 4
```

```
print(len_circle(d))
```

А теперь рассмотрим чуть более сложный пример – вычисление суммы элементов списка:

```
def sum_arr(arr):
    result = 0
    for element in arr:
        result += element
    return result

print(sum_arr([21, 32, 43, 54]))
```

Здесь мы используем очень распространенный способ написания функций: создаем вспомогательную переменную, а затем возвращаем ее значение.

Множественные точки возврата из функции

Часто бывают ситуации, когда в зависимости от входных данных нужно выполнить различные наборы команд. Например, когда мы считаем модуль числа, в случае отрицательного числа нужно взять число со знаком минус, а в случае неотрицательного числа (положительное или ноль) мы берем само число.

Давайте запишем это в виде функции `my_abs(x)`.

```
def my_abs(x):
    if x >= 0:
        result = x
    else:
        result = -x
return result
```

Но если вдуматься: зачем ждать конца функции, когда мы уже вычислили результат и совершили все необходимые действия? Давайте завершим функцию сразу, ведь `return` именно для этого создан: он не только возвращает значение функции, но и возвращает нас из функции в основную программу. После вызова оператора `return` выполнение кода функции заканчивается. Раз так, давайте упростим программу. Перенесем `return` к тому месту, где результат получен:

```
def my_abs(x):
    if x >= 0:
        return x
    return -x
```

Можно обратить внимание, что здесь `else`-часть нам вообще не нужна.

Заметьте, что любую функцию можно написать с одним-единственным оператором `return`, но часто использовать несколько точек выхода из функции просто удобно.

Возврат из глубины функции

Множественные точки возврата из функции позволяют нам упростить обработку и более сложных структур, например, вложенных списков. Наша следующая программа будет проверять, есть ли в матрице элемент, отличающийся от искомого не больше, чем на число `eps`.

Матрица записывается как список списков. Мы предполагаем, что наша функция будет работать с большими матрицами, поэтому нам не хочется тратить лишнее время на проверку. Мы будем прекращать поиск, как только нашли подходящий элемент. Давайте для начала разберемся, как бы мы действовали без множественных операторов `return`.

```
def matrix_has_close_value(matrix, value, eps):
    found = False
    for row in matrix:
        for cell in row:
            if abs(cell-value)<= eps:
                found = True
                break
        if found:
            break
if found:
    return True
else:
    return False
```

Как видите, нам приходится прилагать некоторые усилия, чтобы выйти из нескольких уровней вложенности. Каждый уровень вложенности – дополнительное препятствие на пути к завершению функции. Ему мешают не только циклы, как в этом примере, но и условные операторы.

Перепишем теперь функцию с учетом того, что, как только мы нашли элемент, мы уже знаем, что ответ – `True` (т. е. элемент содержится в матрице). А если мы закончили перебор элементов и так и не нашли ни одного элемента, ответ – `False`.

```
def matrix_has_close_value(matrix, value, eps):
    for row in matrix:
        for cell in row:
            if abs(cell-value)<= eps:
                return True
return False
```

Хотя `return False` не заключен ни в какое условие, выполняется он только тогда, когда элемент не найден. Если элемент найден, мы сразу выходим из функции и до этой строки просто не доходим.

Оператор `return` очень удобен, когда нужно выйти из глубины функции.

Что можно возвращать из функции

В функциях, которые не возвращают значение, тоже можно использовать `return`. Если написать `return` без аргументов, функция просто сразу завершит свою работу (без `return` функция завершает работу, когда выполнит последнюю команду).

Как мы уже упоминали, результат возвращает любая функция, даже если в ней нет слова `return`. Результатом такой функции будет `None`.

Если в функции использован `return` без аргументов, это фактически эквивалентно `return None`.

Возврат нескольких значений

В задаче про корни квадратного уравнения у нас уже возникала необходимость вернуть несколько значений. Вы видели, что это можно сделать, вернув список значений.

То же самое можно сделать немного проще: если после `return` написать несколько значений через запятую, Python автоматически поместит эти значения в кортеж и вернет этот кортеж.

```
def get_coordinates():
    return 1, 2
print(get_coopdinates())
# => (1, 2)
```

Команда возврата нескольких значений

```
return 1, 2
```

практически идентична команде возврата кортежа с этими значениями

```
return (1, 2)
```

Вы можете пользоваться любой, как вам больше нравится.

Мы разобрались, как возвращать значения из функции. Но как программа их получает, когда значений несколько? Оказывается, есть несколько способов. Один вариант вы знаете, можно записать в переменную весь кортеж:

```
def get_coopdinates():
    return 1, 2


result = get_coordinates()
```

```
    print(pesult)
# => (1, 2)
```

Можно вместо этого воспользоваться множественным присваиванием, тогда значения автоматически будут распределены по разным переменным:

```
x, y = get coordinates()
print(x)  # => 1
print(y)  # => 2
```

Обратите внимание: `get_coordinates` в обоих случаях – одна и та же функция, которая используется немного по-разному.

Когда в функции используется несколько операторов `return`, позаботьтесь о том, чтобы каждый из операторов возвращал однотипные наборы значений. Ведь вызывающая функция заранее не знает, какой из операторов `return` выполнится, а значит, мы не сможем использовать множественное присваивание в полную силу.

В следующей функции (это, конечно, бесполезная функция, она нужна только для иллюстрации) мы не последовали этому совету и возвращаем координаты то на плоскости, то в трехмерном пространстве.

```
def get_coordinates(index):
    if index == 2:
        return 1.5, 2.5
    e1se:
        return 1.5, 2.5, 0
```

Теперь вызов `x, y = get_coordinates()` будет ломаться на нечетных индексах, а `x, y, z = get_coordinates()` – на четных индексах. В итоге мы не сможем записать ни один из вариантов так, чтобы он не сломался при каких-нибудь условиях.

Вы уже видели, как локальные переменные помогают интерпретатору Python не запутаться в именах переменных. Но не всегда бывает просто понять, что за переменная используется в функции: собственная локальная переменная или чужая – из внешней программы. Чтобы разобраться в этих тонкостях, на следующем занятии мы очень подробно обсудим, что такое область видимости переменных.

Локальные и глобальные переменные

Речь пойдет об областях видимости переменных. Занятие посвящено в первую очередь переменным и лишь во вторую – функциям. Эти знания нечасто нужны для написания кода, но совершенно необходимы для понимания того, как программа работает. Это поможет вам не гадать, как ведет себя программа, и значительно сократит время, которое вы тратите на отладку.

Разбиение программы на функции позволяет сильно упростить программу. С помощью них реализуется принцип модульности, обеспечивающий изоляцию частей программы. Так, программисту необходимо оперировать только небольшим набором переменных, а не всеми сразу.

Для того чтобы использовать только необходимые данные, придуманы области видимости, которые скрывают от одной части программы переменные, используемые в другой части программы. Хотя это делается автоматически, крайне важно понимать, как интерпретатор это делает, иначе неизбежны ошибки. А еще изредка вам будет нужно «обойти» эти ограничения и сделать переменную, доступную всем частям программы, и на такой случай вам придется познакомиться с глобальными переменными.

На прошлых занятиях мы уже неоднократно использовали аргументы функций. Пользоваться ими вроде бы просто, но за кажущейся простотой скрывается много тонкостей:

- С какими переменными может работать функция?
- Что если в двух функциях переменные называются одинаково?
- Может ли функция изменить значение аргумента?

Есть и много других вопросов. На этом занятии мы постараемся ответить на них.

Рассмотрим функцию, которая должна выводить на экран содержимое списка, печатая каждый элемент на своей строчке. Посмотрим на переданный аргумент функции.

```
def print_array(arr):
    for element in arr:
        print(element)

print_array(['Hello,', 'world!'])
print_array([123, 456, 789])
```

Результат:

```
Hello,
world!
123
456
789
```

Локальная переменная

Переменная `arr` будет локальной. Она существует только во время выполнения функции. Можно сказать, что имя аргумента функции – временное имя для переданного в функцию значения.

Мы создали список `['Hello,' 'world!']` и передали его в функцию. Список существует независимо от того, есть у него имя или нет, но, если у объекта нет имени, мы не можем с ним работать. Поэтому у переданного списка временно появляется имя – в тот самый момент, когда функция начинает работу. Имя существует на время «жизни» функции.

В вышеописанной программе имя `arr` не существует вне функции.

Что интересно, у каждого объекта в программе может быть несколько имен. Например, слегка модифицируем программу:

```
def print_array(arr):
    for element in arr:
        print(element)

words = ['Hello,', 'world!']
print_array(words)
```

Теперь у списка есть имя `words`. Но в момент выполнения функции у этого списка существуют сразу два имени: `words` – имя из «внешнего мира», из глобальной области видимости, и `arr` – локальное имя. В функции можно использовать любое из них, но внешнее – крайне не рекомендуется.

Давайте перепишем программу, используя внешнее имя, и разберемся, почему так делать не стоит:

```
def print_array(arr):
    for element in words:
        print(element)

words = ['Hello,', 'world!']
print_array(words)
```

Пока все работает аналогично предыдущему примеру. А теперь выполним такую команду (добавим строку к примеру):

```
print_array(['ab', 'cd', 'ef'])
```

Результат:

```
Hello,
world!
Hello,
world!
```

Мы рассчитывали, что будут распечатаны три строки: `ab`, `cd` и `ef`, но вместо этого снова распечатались `Hello,` и `world!`, которые находятся в списке `words`. Если ваша функция написана так, то, чтобы поменять ее

поведение, придется не просто передать ей список, а перезаписать используемую переменную `words`:

```
words = ['ab', 'cd', 'ef']
```

Еще один минус: раньше вы могли прочитать три строки функции `print_array` и точно понять, как она будет работать. Теперь вам необходимо прочитать всю программу (а в большой программе могут быть тысячи строк), чтобы разобраться, как ведет себя эта функция, а все потому, что она теперь не самодостаточна, а зависит от внешней переменной.

Все, что вы сейчас узнали об использовании внешних переменных в функции, необходимо в первую очередь для поиска ошибок. Если программа ведет себя странно, проверьте, не используете ли вы где-то внешние (глобальные) переменные.

Замечание

К внешним переменным прибегать допустимо, если переменная – константа. Константы обычно пишутся большими буквами, чтобы не перепутать их с обычными переменными.

Области видимости переменных

Сформулируем общее правило, касающееся видимости переменных:

- Внутри функции видны все переменные этой функции (локальные переменные и аргументы функции).
- Внутри функции видны переменные, которые определены снаружи этой функции.
- Снаружи не видны никакие переменные, которые определены внутри функции.

Но вот вопрос: что же произойдет, если имена переменных в функции и во внешней программе совпадут?

Оказывается, есть два варианта.

Первый вариант мы уже видели: ситуация, когда внутри функции используется внешняя переменная. В следующем примере – это переменная PI.

```
def print_circle_length(radius):
    dlina = 2*PI*radius
    print('Длина окружности', dlina)

PI = 3.14
print_circle_length(10)
```

Второй вариант – ситуация, когда внутри функции используется переменная с тем же именем, что и в основной программе, но перед использованием ей присваивается новое значение.

```
area = 'Красная площадь'
```

```
def print_square_area(length, width):
    area = length * width
    print('Площадь прямоугольника:', area)

print ('Место встречи:', area)
print_square_area(300, 80)
print('Повторно - место встречи:', area)
```

Результат:

```
Место встречи: Красная площадь
Площадь прямоугольника: 24000
Повторно - место встречи: Красная площадь
```

Если попытаться мысленно выполнить программу, кажется, что функция должна испортить внешнюю переменную `area`, записав в нее вместо строки «Красная площадь» число, равное площади прямоугольника со сторонами `length` и `width`. Но этого не происходит. Когда после выполнения функции мы повторяем печать, оказывается, что переменная `area` все так же содержит строку «Красная площадь», хотя внутри функции там точно было число.

Что же произошло?

Оказывается, `area` внутри функции и снаружи – это две совершенно разные переменные. Если внутри функции переменной что-то присваивается (в любом месте функции), интерпретатор не позволит вам работать с внешней переменной.

Вместо этого он создает новую переменную с тем же именем и присваивает значение уже ей. При этом внутри функции к внешней переменной вы обратиться уже не можете.

Как иногда говорят, внутренняя переменная с тем же именем ее перекрыла.

Переменные в области видимости

Это сделано для того, чтобы обезопасить вас от ошибок. Если бы функция могла менять внешние переменные, в программе происходило бы много странных и непонятных вещей.

Например, представьте, что вы написали функцию, которая вычисляет площадь и записывает что-то в переменную `area`. При этом в основной программе переменной `area` у вас не было, поэтому все работало хорошо. Потом вы в какой-то момент ввели переменную `area` – теперь функция, которая работала хорошо и безопасно, стала менять значение `area` во внешней программе. Вам бы пришлось долго искать в программе ошибку, из-за которой вы встречаетесь на площади 24000.

Именно поэтому области видимости так важны. Они позволяют изолировать небольшой кусочек программы, в котором ваши изменения на что-то влияют. Благодаря областям видимости не окажется так, что вы поменяли строку в одной функции, а сломалась другая.

Для того чтобы не вносить в программу лишние глобальные переменные, часто весь код с внешнего уровня программы (т. е. все, что не находится внутри функции) переносят в отдельную функцию `main`. После этого остается только вызвать ее. Все переменные, которые были глобальными, таким образом, становятся локальными для функции `main` и не загрязняют область видимости других функций.

```
def print_square_area(length, width):
    area = length * width
    print('Площадь прямоугольника:', area)


def main():
    area = 'Красная площадь'
    print ('Место встречи:', area)
    print_square_area(300, 80)
    print('Повторно - место встречи:', area)

main()
```

Результат:

```
Место встречи: Красная площадь
Площадь прямоугольника: 24000
Повторно - место встречи: Красная площадь
```

Использование глобальных переменных

Впрочем, если у вас есть глобальная переменная и вам зачем-то очень нужно повлиять на эту внешнюю переменную, это можно сделать.

Для изменения глобальной переменной перед тем, как присваивать внутри функции этой переменной какое-то значение, нужно указать, что она глобальная, т. е. относится к внешней области видимости.

Например, напишем функцию, которая при вызове будет обновлять счетчик, считающий, сколько раз вызвана эта функция.

```
ask_number = 0

def ask_again():
    global ask_number
    ask_number += 1
```

```
    print(ask_number, '-й вызов функции', sep='')

ask_again()
ask_again()
ask_again()
ask_again()
```

Результат:

```
1-й вызов функции
2-й вызов функции
3-й вызов функции
4-й вызов функции
```

Хотя переменной `ask_number` и присваивается значение, она является внешней переменной за счет строчки `global ask_number`.

Мы уже не раз говорили, но повторим еще раз: о глобальных переменных нужно помнить и знать, но использовать их в своих программах крайне не рекомендуется.

Менять внутри функции значение внешней переменной еще хуже, чем использовать внешние переменные, не изменяя их. Старайтесь писать функции, которые минимально зависят от всего, что происходит снаружи.

Одна из немногих причин использовать глобальные переменные – возможность сохранить какие-то данные между вызовами функции. Функция не может ничего сохранить «на будущее» в локальных переменных, поскольку они исчезнут, после того как интерпретатор вернется из функции. Зато, если функция при первом вызове сохранит что-либо в глобальную переменную, при следующих вызовах эти данные будут доступны для чтения (и для изменения). Но не забудьте, что глобальная переменная может быть изменена не только из этой функции, но и в любом другом месте программы, так что вы должны внимательно следить за тем, чтобы значение вашей переменной не испортилось между вызовами функции.

Аргументы функций как локальные переменные

Мы не рассмотрели еще один случай (точнее, даже два) использования переменных с одинаковыми именами.

Это случай, когда имя локальной переменной совпадает с именем аргумента функции, и случай, когда имя внешней переменной совпадает с именем аргумента.

Первая программа:

```
def greet(name):
    print('Привет,', name)
    name = 'товарищ'
```

```
    print('Здравствуй,', name)

greet('Вася')
```

Результат:

```
Привет, Вася
Здравствуй, товарищ
```

Вторая программа:

```
name = 'Саша'
def greet(name):
    print('Привет,', name)

greet('Вася')
print(name)
```

Результат:

```
Привет, Вася
Саша
```

Работу обеих программ очень легко объяснить. Они выполняются именно так, потому что аргумент функции – по сути обыкновенная локальная переменная. В момент вызова функции происходит присваивание значения «Вася» этой локальной переменной. А дальше программист волен ее использовать и изменять, как ему вздумается.

Аргумент функции всегда является локальной переменной, а значит, будет иметь приоритет над внешней переменной с тем же именем.

Отличие между переменной и значением

Хотя мы много времени уделили изучению работы переменных с функциями, это еще не все, что нужно знать о переменных. На этом занятии выясним, чем переменная отличается от значения переменной.

Внесём небольшие изменения в рассмотренный выше первый пример:

```
name = 'Вася'

def greet(name):
    print('Привет,', name)
    name = 'товарищ'
    print('Здравствуй,', name)

greet(name)
```

```
print('name:', name)
```

Результат:

```
Привет, Вася
Здравствуй, товарищ
name: Вася)
```

Из этого примера ясно, что аргумент функции `name` является локальной переменной и перекрывает переменную `name` из внешней области видимости. Внешняя переменная `name` не изменяется. В функцию вообще не передается переменная! Туда передается значение, которое она хранила.

В языке Python имя переменной связывается со значением. В тот момент, когда мы присваиваем переменной новое значение, старое значение никуда не исчезает. Оно просто теряет связь с прежним именем.

Имена локальных переменных и имена аргументов можно менять как угодно, это не влияет на работу программы. Например, приведенную выше программу совершенно безопасно можно было бы переписать так:

```
name = 'Вася'

def greet(person):
    print('Привет,', person)
    person = 'товарищ'
    print('Здравствуй,', person)

greet(name)
print('name:', name)
```

Стоит отметить, что, прежде чем функция будет вызвана, все ее аргументы должны быть вычислены. Таким образом, сначала выполняются инструкции, которые вычисляют значение переменной, и лишь затем полученные значения передаются в функцию, только после этого начинает выполняться код функции.

Функции, изменяющие значение аргумента

Как же быть, если вам хочется изменить значение объекта? Один из способов – использовать глобальные переменные, но мы уже объяснили, чем он плох.

Оказывается, есть еще несколько способов. Вариант, который мы рассмотрим сейчас, тоже неидеален, но все же гораздо лучше, чем использование глобальных переменных.

Рассмотрим функцию, которая принимает список чисел и возводит каждое число в квадрат.

```
def convert_squares(array):
    for i in range(len(array)):
        array[i] = array[i]**2

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
convert_squares(my_list)
print(my_list)
```

Результат:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Эта программа выведет список квадратов чисел от 1 до 9. Заметьте, мы не присваивали ничего переменной `my_list`, это все тот же объект, что был. Но его содержимое поменялось. Ключевой момент: объект тот же, а его наполнение другое.

Рассмотрим подобную функцию:

```
def convert_squares(array):
    new_array = []
    for i in range(len(array)):
        new_array.append(array[i]**2)
    array = new_array

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
convert_squares(my_list)
print(my_list)
```

Результат:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Она выводит на экран не квадраты чисел, а просто числа от одного до девяти. В этом примере переменная `array` приняла адрес локальной переменной `new_array`, и поэтому потерялась связь с внешней переменной `my_list`.

Объекты: одни и те же или одинаковые?

Представьте теперь, что у вас есть задача заполнить холодильник едой. В первом случае вы покупаете еду и кладете ее в свой холодильник. Холодильник с едой – это тот же холодильник, что у вас был. Это ровно тот же объект.

Другой способ решения проблемы – купить новый холодильник, уже заполненный едой. Но первый холодильник при этом так и остался незаполненным. Внешняя программа ничего не знает про новый холодильник, но знает, что старый вы не заполняли, и выводит вам пустое содержимое.

Итак, мы обнаружили, что есть два способа изменить значение переменной: присвоить переменной новый объект или оставить старый объект, но поменять его содержимое. Однако, оказывается, не у любого объекта можно поменять содержимое.

Когда вам кажется, что вы изменяете, например, число или строку, на самом деле вы создаете новое число или строку и связываете ее с переменной со старым именем.

Это легко проверить.

Оператор *is*

Проверить, связаны ли две переменные с одним и тем же объектом, можно с помощью оператора **is**.

`x is y` возвращает `True`, когда объект `x` и объект `y` – один и тот же объект. Кроме того, в Python есть встроенная функция **id**, которая выдает уникальный номер объекта. У двух разных объектов **id** разный, а у одного и того же объекта – одинаковый.

Посмотрим, что происходит с переменными при попытке их изменить.

Изменяемость и неизменяемость объектов

На самом деле в языке Python не так много встроенных объектов, которые можно поменять. Сейчас из ***изменяемых объектов*** вы знаете ***списки, множества и словари***.

А ***числа, булевы значения, строки*** и даже ***кортежи*** менять нельзя. Их содержимое всегда неизменно с момента создания. Такие объекты называются ***иммутабельными***, то есть ***неизменяемыми***.

Числа

```
x = 1
print(id(x))    # id объекта ”число 1”
x += 1
print(id(x))    # id изменился, x – это объект другого числа
```

Числа – неизменяемые объекты. Поэтому каждая попытка изменить их создает новый объект.

Строки

```
s = 'Привет'
print(id(s))    # id строки 'Привет'
s += ', друг'
print(id(s))    # id другого объекта –строки 'Привет, друг'
```

Строки – тоже неизменяемые объекты, и ведут себя так же, как и числа.

Списки

```
a = [1, 2, 3]
print(id(a))  # id списка
a += [4, 5, 6]
print(id(a)) # тот же id - того же списка, с измененным содержимым
```

Сложение списков создало новый список на базе старого.

Списки – изменяемые.

Следует отметить еще один важный момент. Равенство объектов не означает, что это один и тот же объект (или, как еще говорят, объекты идентичны). Это особенно важно для изменяемых объектов-контейнеров, таких как списки. Но и с неизменяемыми кортежами, и со строками бывают ситуации, когда отличие между равенством и идентичностью оказывается важным.

Вернемся к аналогии с холодильниками. Если у вас в холодильнике те же продукты, что у вашего друга, это не значит, что у вас с ним один и тот же холодильник. Если вы что-нибудь добавите в свой холодильник, холодильник вашего друга автоматически никак не изменится.

```
my_fridge = ['молоко', 'яйца', 'овощи']
my_friends_fridge = ['молоко', 'яйца', 'овощи']
print(my_fridge == my_friends_fridge)
print(id(my_fridge) == id(my_friends_fridge))
```

Содержимое холодильников одинаковое, что показывает сравнение списков при помощи оператора ==. Но их уникальные `id` разные.

Совсем другая ситуация, когда у вас есть две переменные, ссылающиеся на один и тот же холодильник. Например, ваш холодильник является одновременно холодильником ваших родителей.

```
my_fridge = ['молоко', 'яйца', 'овощи']
my_parents_fridge = my_fridge
print(my_fridge == my_parents_fridge)
print(id(my_fridge) == id(my_parents_fridge))
```

Строка `my_parents_fridge = my_fridge` сделала холодильник ваших родителей не просто таким же, как ваш, а ровно тем же. Их `id` равны.

Теперь, если ваши родители положат что-то в свой холодильник, ваш холодильник тоже изменится, так как это тот же самый холодильник.

```
my_parents_fridge += ['мясо']
print(my_fridge)
```

Если вы раньше программировали на Паскале или СИ, обратите особое внимание на этот пример. Хотя вы никак не меняли переменную `my_fridge`, она изменилась. Вернее, изменился объект, на который она ссылается.

Бывают ситуации, когда такое поведение неудобно. Иногда нам хочется получить копию объекта, а не просто вторую ссылку на тот же объект. В общем

случае это не так просто сделать, ведь объект (например, список) может содержать в себе другие объекты, которые тоже нужно скопировать. Если вы хотите узнать об этом подробно, вам лучше обратиться к документации в Интернете.

Копирование одномерного списка

Напомним о самом распространенном случае. Для того чтобы сделать копию одномерного списка `arr`, можно сделать срез, содержащий все элементы:

```
arr = [1, 2, 3]
arrCopy = arr[:]
arrCopy[0] = 9
print(arr)      # => [1,2,3]
print(arrCopy)  # => [9,2,3]
```

Но будьте аккуратны: ведь если список содержал вложенные списки, копия внешнего списка содержит те же самые вложенные списки (с их адресами), а не их копии. А значит, изменения одного списка отразятся на другом. Проще всего это увидеть на примере:

```
arr = [[1], [2], [3]]
arrCopy = arr[:]
arrCopy[0].append(9)
print(arr)      # => [[1,9],[2],[3]]
print(arrCopy)  # => [[1,9],[2],[3]]
```

С кортежами дело обстоит аналогично.

Кортежи

Кортеж – объект неизменяемый. В него нельзя, например, добавить элемент или заменить существующий элемент новым объектом. Но его элементы вполне могут быть изменяемыми; если среди элементов кортежа есть изменяемые элементы, поменяв их содержимое, вы фактически измените содержимое кортежа.

По этой причине иногда бывает недостаточно следить за типом переменной. Всегда думайте заодно о типах содержимого контейнеров (контейнер – объект, содержащий другие объекты).

Напомним также, что в случае списков оператор `a1 += a2` ведет себя не совсем как `a1 = a1 + a2`

В первом случае изменяется сам список (к его концу дописываются все элементы списка `a2`), во втором – создается новый. В случае чисел, строк и кортежей, которые изменяться не могут, две эти формы записи полностью эквивалентны.

Вопрос для самопроверки 1:

Что выведет на экран следующая программа и почему?

```
arr = [2, 96, 5]
print(arr, arr.sort(), arr, sep='\n')
```

Напоминаем, что метод `sort` сортирует элементы в списке (и ничего не возвращает).

Вопрос для самопроверки 2:

Что выведет на экран следующая программа и почему?

```
x = 1
def double_x():
    global x
    x *= 2
print(x, double_x(), x, sep='\n')
```

Вопрос для самопроверки 3:

Можете ли вы придумать три различных примера, которые продемонстрируют, что список и его полный срез – различные объекты?

Рекурсия

При описании функций иногда удобно использовать рекурсию. Рекурсия подразумевает, что в программном коде функции используется вызов этой же самой функции (но обычно с другим аргументом). Лучше всего сказанное проиллюстрировать на примере вычисления факториала.

Посмотрим на определение факториала и обратим внимание на фрагмент, выделенный жирным шрифтом.

$$n! = \begin{cases} 1, n = 0 \\ \mathbf{1 * 2 * 3 * \ldots * (n - 1)} * n, n > 0 \end{cases}$$

Можно увидеть, что 1·2·3·…·(n-1) не что иное, как факториал числа n-1. Поэтому определение факториала можно записать в сокращённом виде:

$$n! = \begin{cases} 1, \ n = 0 \\ \mathbf{(n - 1)!} * n, \ n > 0 \end{cases}$$

Рассмотрим функцию вычисления факториала с применением рекурсии. Известно, что 0! = 1, 1! = 1. А как вычислить значение n! для большого n? Если бы мы могли вычислить (n-1)!, тогда мы легко вычислим n!, поскольку

n! = n · (n-1)!

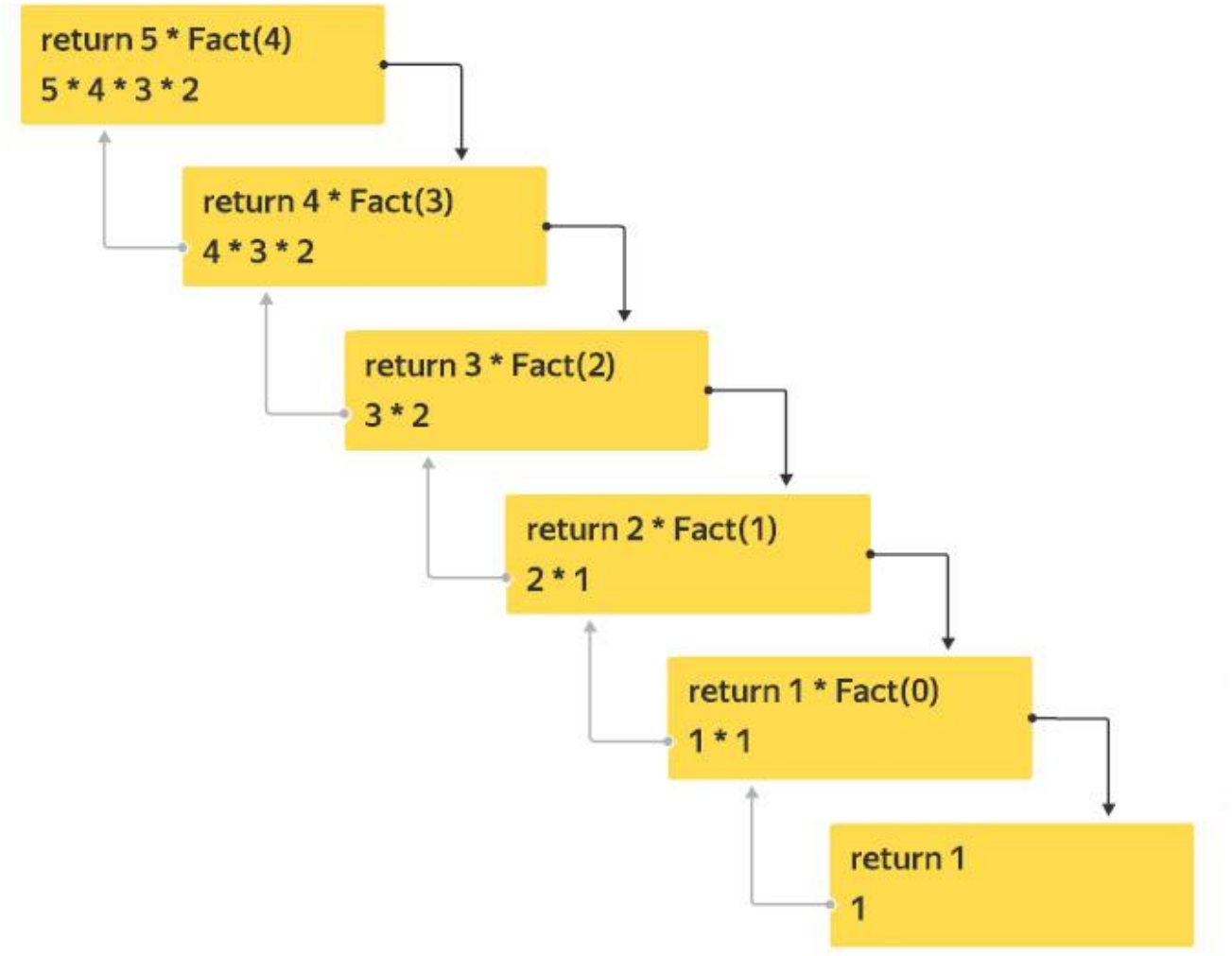
Но (n-1)! = (n-1) · (n-2)! И далее: (n-2)! = (n-2) · (n-3)!

В конце концов мы дойдём до величины 0!, которая равна 1. Таким образом, для вычисления факториала мы можем использовать значение факториала для меньшего числа. Напишем соответствующую функцию:

```
#  Рекурсивное вычисление факториала
def re_factorial(number):
    if number == 0:
        return 1
    else:
        return number * re_factorial(number - 1)
```

Логическая сложность рекурсивных функций заключается в изменении параметров и особенностях получения результатов при последовательном обращении программы к себе. Выполняются две серии шагов. Первая серия – шаги рекурсивного погружения подпрограмм в себя до тех пор, пока выбранный параметр не достигнет граничного значения (***глубина рекурсии***). Вторая серия – шаши рекурсивного выхода до тех пор, пока значение выбранного параметра не достигнет начального. Она, как правило, и обеспечивает получение промежуточных и конечных результатов.

Вот как работает рекурсивная функция вычисления факториала (5!):



В общем случае рекурсия тяготеет к декларативному стилю программирования. Если в двух словах: когда мы пишем императивную функцию (как делали всё время до этого), отвечаем на вопрос, **как** достигнуть необходимого результата, а когда создаём декларативную – на вопрос, **что** такое наш результат.

Поэтому в любой рекурсивной функции должно быть как минимум **две** ветки развития процесса:

1. Основная
2. Точка выхода

Сама функция при этом получается декларативной: она повторяет практически один в один определение факториала

Опасность использования рекурсивных алгоритмов

Наиболее распространённой ошибкой при использовании рекурсии является бесконечная рекурсия, когда цепочка вызовов функций никогда не завершается и продолжается, пока не кончится свободная память в компьютере. При разработке рекурсивной функции необходимо прежде всего оформлять условия завершения рекурсии и думать, почему рекурсия когда-либо завершит работу.

Ещё одна проблема, связанная с использованием рекурсивных функций, – нетривиальность задачи оценки сложности и эффективности алгоритма. Сложность этих алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Рекурсивная программа может выглядеть достаточно простой, но она может серьёзно усложнить программу, многократно вызывая себя.

Функции с переменным числом аргументов

Распаковка и запаковка значений

Ранее, рассматривая возврат значений из функции, мы коснулись темы возврата нескольких значений и множественного присваивания получившихся значений нескольким переменным.

```
def get_coordinates():
    return 1, 2
x, y = get_coordinates()
print(x)   # => 1
print(y)   # => 2
```

Хотя этот прием с присваиванием результата нескольким значениям часто используется именно в применении к функциям, на самом деле никакого отношения к механике работы функций не имеет.

В приведенном примере функция просто возвращает кортеж, а всю дальнейшую работу делает механизм множественного присваивания, а точнее, процедура «распаковывания». Вы с ней сталкивались, когда обсуждали кортежи. Сейчас мы посмотрим на возможности множественного присваивания внимательнее.

Итак, вы можете написать:

```
x, y = (1.5, 2.5)
```

В момент присваивания кортеж будет распакован, его первый элемент будет записан в **x**, второй – в **y**.

Распаковать можно не только кортежи, но и списки: `x,y = [1.5, 2.5]` будет работать точно так же.

Если при множественном присваивании (когда слева больше одной переменной) число элементов слева и справа не совпадает, возникает ошибка времени исполнения. Вы это видели, когда разбирали запись трех координат в две переменные или двух – в три.

Распаковка значений

Есть способ собрать сразу несколько значений в одну переменную. Это делается при помощи звездочки перед именем переменной:

```
x, y, *rest = 1, 2, 3, 4, 5
```

В этом случае в `x` будет записана единица, в `y` – 2, а в `rest` – список, состоящий из всех аргументов, которые не попали в обычные переменные. В данном случае `rest` будет равен `[3, 4, 5]`

Учтите, что `rest` всегда будет списком, даже когда в него попадает лишь один элемент или даже ноль:

```
x, y, *rest = 1, 2, 3
print(rest)  # => [3]


x, y, *rest = 1, 2
print(rest)  # => []
```

Звездочка может быть только у одного аргумента, но необязательно у последнего:

```
names, surname = 'Анна Мария Луиза Медичи '.split()
print(names) = ['Анна', 'Мария', 'Луиза']
print(surname) # => 'Медичи'
```

Такой аргумент может стоять и посередине:

```
Li = ['Дейнерис',
```

```
        'королева андалов, ройнаров и Первых Людей',
        'королева Миэрина',
        'кхалиси Великого Травяного моря',
        'Неопалимая',
        'Бурерожденная',
        'Матерь Драконов',
        'Разрушительница оков',
        'Низвергательница Колдунов',
        'Таргариен']
    name, *titles, surname = Li
    print(name)
    print(titles)
    print(surname)
```

Результат:

```
Дейнерис
['королева андалов, ройнаров и Первых Людей',
'королева Миэрина', 'кхалиси Великого Травяного моря',
'Неопалимая', 'Бурерожденная', 'Матерь Драконов',
'Разрушительница оков','Низвергательница Колдунов']
Таргариен
```

Также есть возможность ***распаковывать вложенные списки***. Например:

```
a, (b, c), d = [1, [2, 3], 4]
```

запишет в переменные `a`, `b`, `c`, `d` значения 1, 2, 3 и 4 соответственно.

Если вы хотите распаковать единственное значение в кортеже, после имени переменной должна идти запятая:

```
a = (1,)
b, = (2,)
print(a)   # => (1,)
print(b)   # =>  2
```

Помимо распаковывания, есть и операция ***запаковывания***. Она выполняется всегда, когда справа от знака равенства стоит больше одного значения. Например, можно написать: `values = 1, 2, 3`. Тогда значения в правой части автоматически будут запакованы в кортеж `(1, 2, 3)`.

Запаковывание можно комбинировать с распаковыванием:

```
a, *b = 1, 2, 3
print(a, b)   #  => 1, [2, 3]
```

Общее правило такое: при любых нестандартных присваиваниях сначала происходит запаковывание значений в правой части, а затем распаковка их в переменные, стоящие в левой части.

Вообще, лучший способ понять операции с запаковыванием и распаковыванием значений – поэкспериментировать с ними.

Техника запаковывания и распаковывания переменных со звездочкой используется не только в операциях присваивания. Похожий механизм применяется для написания функций, которые могут принимать переменное число аргументов. И синтаксис для этого используется похожий на тот, который применяется в множественном присваивании. В списке аргументов функции один из аргументов может быть помечен звездочкой, тогда в него попадут все значения на соответствующей позиции, которые еще не присвоены другим аргументам.

Но есть и отличие. При ***множественном присваивании*** переменная со звездочкой получает ***список*** значений. А когда аргумент функции указан со звездочкой, он получает ***кортеж*** значений.

Например, функция `print` принимает сколько угодно аргументов и дает таким образом возможность выводить на экран неограниченное число значений.

Разработчики языка Python могли поступить иначе и сделать функцию, принимающую всегда ровно один аргумент-список, и выводить на экран элементы этого списка. С точки зрения функциональности результат был бы аналогичным, но такую функцию было бы не так удобно использовать.

Мы сделаем свою функцию для вычисления произведения всех аргументов.

```
def product(first, *rest):
    result = first
    for value in rest:
        result *= value
    return result

print(product(2, 3, 5, 7))    # => 210
```

Эта функция принимает как минимум один аргумент – `first`. Это не позволяет вызвать функцию без аргументов, что было бы бессмысленно. Все аргументы, кроме первого, попадают в кортеж `rest`.

Не всем функциям необходимо произвольное число элементов. Бывает так, что функции требуется просто разное число аргументов. В этом случае можно поступить следующим образом: поставить звездочку, которая формально позволяет использовать любое число аргументов, а внутри функции вручную проверять, что число переданных элементов – правильное.

Звездочку можно использовать не только для того, чтобы запаковать аргументы. Распаковать их тоже можно. Если при вызове функции вы поставите

звездочку перед переданным аргументом-списком, список раскроется и как бы «потеряет границы». Элементы списка станут аргументами функции.

```
arr = ['cd', 'ef', 'gh']
# Здесь мы передаем просто список как один аргумент
print(arr)   # ['cd', 'ef', 'gh']
# А здесь мы раскрыли список и
# функция print получила три отдельных аргумента
print(*arr)  # => cd ef gh
# Это аналогично вызову
print('cd', 'ef', 'gh')   # => cd ef gh
# Раскрытие списка можно комбинировать с любыми аргументами
print('ab', *arr, 'yz')   # => ab cd ef gh yz
# При раскрытии может быть несколько аргументов со звездочкой
print(*arr, *arr)  # cd ef gh cd ef gh
```

Такую технику применяют, когда вам надо передать в функцию заранее неизвестное число аргументов. Вы делаете отдельную переменную, хранящую список аргументов, а затем просто раскрываете ее при помощи звездочки.

Аргументы по умолчанию

Бывает так, что какой-то параметр функции часто принимает одно и то же значение.

Например, хорошо известная вам функция `int` принимает два параметра: строка, которую нужно преобразовать в число, а также основание системы счисления. Это позволяет ей считывать числа в различных системах счисления, например, двоичное число 101 мы можем считать так:

```
int('101', 2)
```

Но чаще всего эта функция используется для считывания из строки чисел, записанных в десятичной системе счисления. Было бы неудобно каждый раз писать 10 вторым аргументом. На такой случай Python позволяет задавать некоторым аргументам значения по умолчанию. У функции `int` второй аргумент по умолчанию равен 10, и потому можно вызывать эту функцию с одним аргументом. Значение второго подставится автоматически.

Для того чтобы определить аргумент по умолчанию, в списке аргументов функции достаточно после имени переменной написать знак равенства и нужное значение. Аргументы, имеющие значение по умолчанию, должны идти в конце, ведь иначе интерпретатор не смог бы понять, какой из аргументов указан, а какой пропущен (и значит, для него нужно использовать значение по умолчанию)

Именованные аргументы

Еще одна проблема функций заключается в том, что программист вынужден помнить (или каждый раз узнавать в документации) порядок аргументов. В некоторых случаях тяжело угадать логичный порядок аргументов. Чтобы не запоминать эти малозначительные детали, можно передавать аргументы в функцию с указанием имени аргумента, в таком случае порядок аргументов неважен. Вы уже сталкивались с именованными аргументами. Встроенная функция `print` часто используется с несколькими такими параметрами: `sep=' '` – для разделения аргументов при выводе (по умолчанию – пробелами) и `end='\n'` для того, чтобы в конце добавлялся символ перевода строки. Так, нам не надо беспокоиться о том, какой параметр – `sep` или `end` – указывать первым.

Позиционные и именованные аргументы

Аргументы, которые передаются без указания имен, называются позиционными, потому что по положению аргумента понятно, какому параметру он соответствует. Аргументы, которые передаются с именами, называются именованными.

Чтобы вашу функцию можно было вызывать, используя именованные аргументы, буквально ничего не нужно делать. Все функции, которые вы писали на предыдущих занятиях, уже можно вызывать, передавая им именованные аргументы.

Если у функции есть аргументы, при вызове можно использовать имена параметров, которые вы использовали в определении функции (исключение составляют списки аргументов неопределенной длины, в которых используются аргументы со звездочкой). Это еще один повод давать аргументам значащие, а не однобуквенные имена. Можно вспомнить или догадаться, что функция `matrix_has_value` имеет параметры `matrix` и `value`, но совершенно невозможно будет вспомнить про имена параметров, такие как `a, b` или `t, v`.

Именованные аргументы можно использовать вместе со значениями по умолчанию.

Именованные и позиционные аргументы не всегда хорошо ладят друг с другом. При вызове функции позиционные аргументы должны обязательно идти перед именованными аргументами. Достаточно сложно сформулировать точные правила поведения аргументов функции при использовании одновременно аргументов со звездочкой и именованных аргументов. Чем запоминать точные правила, в таких случаях лучше пользоваться здравым смыслом.

Общие рекомендации

- Если вам приходится долго думать о том, как оформить список аргументов, чтобы он работал корректно, лучше использовать более

простую версию. Ведь код, который тяжело писать, с большой вероятностью будет тяжело читать

- Если ваш вызов функции не работает, попробуйте прочитать его глазами интерпретатора. Однозначно ли он читается или вы можете придумать несколько вариантов разложить переданные в вызове функции параметры по аргументам? Если вы можете трактовать код несколькими способами, с большой вероятностью интерпретатор столкнется с теми же трудностями. В ситуации, когда код неоднозначен, интерпретатор Python не пытается угадать, что программист имел в виду, а сообщает об ошибке. Часто это считается синтаксической ошибкой, и ошибка возникает еще до того, как программа начинает выполняться

Инструкция *pass*. Согласованность аргументов

В языке Python есть эталонно бесполезная инструкция `pass`. Инструкция `pass` – инструкция-заглушка, которая не делает ничего. Дело в том, что синтаксис языка Python не позволяет в некоторых местах обойтись без команд.

Например, не может быть функции с пустым телом. Ветвь условного оператора или тело цикла тоже должны выполнять какие-либо действия, но иногда программист хочет отложить их написание и ставит такую заглушку.

```
if game_is_over:
    pass   # надо написать вывод итогового результата
```

Давайте теперь напишем функцию, которая при вызове не делает ничего. Наша первая попытка будет такой:

```
def nop():
    pass

nop()
```

Было бы удобно, если можно было завести функцию, которая принимает любые аргументы и, игнорируя их все, не делает ничего. Для захвата произвольного числа параметров, воспользуемся аргументом со звездочкой:

```
def nop(*rest):
    pass

nop()
nop('Любое', 'сказанное', 'вами слово',
     'будет проигнорировано')
nop(1050, None, [1, 2, 3, 4])
```

Чтобы написать функцию, которая игнорирует любой список аргументов, необходимо разрешить ей принимать произвольное число позиционных аргументов и произвольное число именованных аргументов:

```
def nop(*rest,     **kwargs)
```

```
    pass

nop(1,[2,3], debug=True, file='debug.log')
```

*****kwargs***

Аргумент с двумя звездочками ** kwargs – специальный аргумент, который может перехватить все «лишние» именованные аргументы, переданные в функцию. Лишними аргументами будут все именованные аргументы в команде вызова функции, для которых нет соответствующего параметра в определении функции.

Этот аргумент, как и аргумент с одной звездочкой, захватывающий «лишние» позиционные аргументы, можно использовать в комбинации с обычными аргументами. Например, сделаем и вызовем функцию, распечатывающую информацию о пользователе:

```
def profile(name, surname, city, *children, **add_info):
    print('Имя:', name)
    print('Фамилия:', surname)
    print('Город проживания:', city)
    if len(children) > 0:
        print('Дети:', ', '.join(children))
    print(add_info)

profile('Сергей', 'Михалков', 'Москва', 'Никита Михалков',
    'Андрей Кончаловский', occupation='writer', diedIn=2009)
```

Результат:

```
Имя: Сергей
Фамилия: Михалков
Город проживания: Москва
Дети: Никита Михалков, Андрей Кончаловский
{'occupation: 'writer', 'diedIn': 2009}
```

Как вы уже знаете, параметр `children` будет списком лишних позиционных аргументов. А вот `add_info` будет словарем лишних именованных аргументов. В последней строке мы распечатали переданный словарь, он выглядит так:

```
{'occupation: 'writer', 'diedIn': 2009}
```

Вы уже знаете, что звездочка может не только запаковывать аргументы, но и распаковывать, чтобы передать в функцию список со звездочкой перед ним:

```
print(['Массив', 'из', 'четырех', 'аргументов'])
```

```
# => ['Массив', 'из', 'четырех', 'аргументов']
print(*['Просто', 'три', 'аргумента'])
# => Просто три аргумента
```

Две звездочки также позволяют не только запаковывать именованные аргументы в словарь, но и распаковывать словарь в набор именованных аргументов.

Часто в функции используется запаковывание, а затем распаковывание. Например, сделаем собственную функцию `pepforated_print`, которая будет делать всё то же самое, что функция `print`, но при этом будет печатать горизонтальную линию над распечатанным текстом и под ним. Мы хотим использовать все опции функции `print`, но не хотим их самостоятельно обрабатывать. Поэтому мы перехватываем все опции (`sep`, `end` и т.п.), переданные в нашу функцию `perforated_print`, а затем передаем их без изменений в функцию `print`. С позиционными аргументами поступаем так же:

```
def perforated_print(*args, **kwargs):
    print(*args, **kwargs)
    print('-'*30)

perforated_print('Теперь текст выводится с линией перфорации')
perforated_print('и', 'можно', 'использовать', 'любые',
                 'опции', end=':\n')
perforated_print('end', 'sep', 'прочие', sep=', ', end='!\n')
```

Вы можете использовать распаковывание только что запакованных аргументов для того, чтобы усложнять поведение функции, подобно тому, как мы добавили черту к функции `print`.

Результат:

```
Теперь текст выводится с линией перфорации
------------------------------
и можно использовать любые опции:
------------------------------
end, sep, прочие!
------------------------------
```

Функция как объект

В языке Python все является объектом. Даже функция. До сих пор мы рассматривали функции как совершенно отдельный элемент языка со своим синтаксисом и механизмами работы. Но, оказывается, функция – что-то вроде особого типа объектов. Бывают числа, бывают строки, бывают списки. А бывают – функции. У каждого из этих типов есть свои операции, свой синтаксис, но все

они являются объектами. Например, есть объект, который умеет печатать текст на экране. У него есть имя `print`. Можно считать, что `print` – это что-то вроде имени переменной, которая хранит объект функции. Давайте посмотрим, насколько далеко идет эта аналогия.

Первым делом, давайте получим объект функции. Для этого достаточно написать имя функции без скобок. Это аналогично тому, что происходит со списками или строками: если вы пишете после имени списка квадратные скобки с индексом – выполняется операция взятия элемента, не пишете скобки – получаете сам список. Пишете после функции круглые скобки с аргументами – она вызывается, не пишете – получаете саму функцию как объект.

`input` – объект функции чтения из стандартного ввода. Давайте мы этот объект напечатаем:

```
print(input)
```

Python выдает нам текстовое представление этой функции: строку `built-in function input`, которая поясняет, что `input` – встроенная функция языка.

Вопросы для самопроверки:

- Что сделает `print(input())` и почему это отличается от `print(input)`?
- Как вывести на экран функцию печати?

Раз мы можем получить какой-то объект, мы его можем записать в переменную. Давайте попробуем!

```
vyvod = print
```

Это сработало. Теперь у функции `print` есть псевдоним на транслите. Эту новую переменную можно вызвать, как и обычную функцию, посредством круглых скобок:

```
vyvod('Привет, мир!')
```

Возможность записать функцию в переменную позволяет нам гибко управлять тем, какую функциональность мы хотим использовать. В одну и ту же переменную мы можем записать разные варианты поведения и менять их при необходимости. При этом нам не только не нужно будет менять код по всей программе, но не придется даже изменять код функций. Достаточно переменной присвоить вместо одной функции – другую.

Напишем программу, которая печатает списки либо просто через запятую, либо в «коробочках», в зависимости от значения переменной `formatting`.

```
def print_boxed(arr):
    arr_stringified = [str(element) for element in arr]
    mid = ' | '.join(arr_stringified)
    bar = '-'*(2+len(mid))
    print(' ' + bar + ' ')
    print('| ' + mid + ' |')
    print(' ' + bar + ' ')
```

```
def print_simple(arr):
    arr_stringified = [str(element) for element in arr]
    print(', '.join(arr_stringified))

formatting = 'boxed'
if formatting == 'boxed':
    print_formatted = print_boxed
else:
    print_formatted = print_simple
```

Дальше в программе можно использовать `print_formatted` повсюду:

```
print_formatted([1, 1, 2, 3, 5, 8, 13, 21])
print_formatted([1, 2, 4, 8, 16, 32, 64, 128])
print_formatted(['abc', 'def', 'ghi'])
```

Вывод программы:

```
 --------------------------------
| 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |
 --------------------------------
 ----------------------------------
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
 ----------------------------------
 -----------------
| abc | def | ghi |
 -----------------
```

Здесь мы один раз проверили условие и указали, как должна вести себя функция `print_formatted` во всей программе.

Функции высшего порядка

Объект функции можно не только записать в переменную, но и передать в качестве аргумента в другую функцию и даже вернуть из функции.

Функции, которые принимают или возвращают другие функции, называются ***функциями высшего порядка***.

Часто функции высшего порядка используются для обработки наборов данных. Например, из раза в раз встречается такая задача: взять список элементов и оставить среди них только небольшую часть, согласно какому-то критерию.

Эту задачу можно встретить в самых разных формах. В списке товаров найти только дешевые. Отобрать все слова, в которых ровно три слога, для генератора рифм. Найти среди кораблей в игре «Морской бой» все подбитые.

Это разные списки и разные критерии, а задача одна и та же – отфильтровать элементы списка. В языке Python для этой цели есть встроенная функция `filter`.

Функция `filter`

Функция `filter` принимает критерий отбора элементов, а затем сам список элементов. Возвращает она список из элементов, удовлетворяющих критерию.

Чтобы этой функцией воспользоваться, нужно сообщить функции `filter` критерий, который говорит, брать элемент в результирующий список или нет. Давайте напишем простую функцию, которая проверяет, что слово длиннее шести букв, и затем отберем с ее помощью длинные слова.

```
def is_word_long(word):
    return len(word) > 6

words = ['В', 'новом', 'списке', 'останутся', 'только',
         'длинные', 'слова']
for word in filter(is_word_long, words):
    print(word)

# => останутся
# => длинные
```

С методом `filter` вам не нужно вручную создавать и заполнять список, достаточно указать условие отбора.

Итераторы

Если вы попробуете распечатать результат функции `filter` при помощи функции `print` (а не перебирая элементы по одному в цикле `for`), удивитесь: будет выведен не список, а специальный объект `<filter object at 0x..>`

Он похож на список тем, что его можно перебирать циклом `for`, т. е. ***итерировать***. Такие объекты называют ***итераторами***.

Чтобы получить из итератора список, можно воспользоваться функцией `list`:

```
long_words = list(filter(isWordLong, words)
```

Описанный способ отфильтровать список пока далек от удобного, поскольку нам приходится заводить функцию для каждой проверки, что занимает две лишних строки кода. Для каждой такой маленькой функции приходится придумывать имя (и загромождать пространство имен).

Для того чтобы создавать такие короткие функции, «на один раз», в языке Python есть специальный синтаксис.

Лямбда-функции

Часто в качестве аргумента для функций высшего порядка мы хотим использовать совсем простую функцию. Причем нередко такая функция нужна в программе только в одном месте, поэтому ей необязательно даже иметь имя.

Такие короткие безымянные (анонимные) функции можно создавать инструкцией

```
lambda <аргументы>: <выражение>
```

Такая инструкция создаст функцию, принимающую указанный список аргументов и возвращающую результат вычисления выражения.

В языке Python тело лямбда-функции имеет ровно одно выражение. Инструкция `return` подразумевается, писать ее не требуется, да и нельзя. Скобки вокруг аргументов не пишутся, аргументы от выражения отделяет двоеточие.

Теперь мы можем записать функцию, проверяющую длину слова, следующим образом:

```
lambda word: len(word) > 6
```

И список длинных слов теперь извлечь очень просто:

```
long_words = list(filter(lambda word: len(word) > 6, words))
```

Лямбда-функция – полноценная функция. Ее можно использовать в составе любых конструкций. Например, если вы хотите использовать ее несколько раз, но не хотите определять функцию с помощью `def`, вы можете присвоить созданную лямбда-функцию какой-либо переменной.

```
add = lambda x, y: x + y
add(3, 5)  # => 8
add(1, add(2, 3))  # => 6
```

А теперь рассмотрим вариант, что вам нужно взять все элементы списка, но в программе уже стоит `filter` и вам не хочется его удалять. В этой ситуации вам поможет функция с особенно простым выражением:

```
lambda x: True
```

Покажем:

```
def print_some_primes(criterion):
    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
    for number in filter(criterion, primes):
        print(number)

print_some_primes(lambda x: True)
# => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Лямбда-функция принимает аргумент, хотя и не использует его. Функция `filter` всегда передает в критерий элемент, который проверяет.

Чтобы функция `filter` не казалась магической, напишем свой упрощенный аналог. Наша функция `simple_filter` будет принимать критерий и список и возвращать новый список.

```
def simple_filter(criterion, arr):
    result = []
    for element in arr:
        if criterion(element):
            result.append(element)
    return result
```

Мы передали критерий как функцию, а потому можем его вызвать, что мы и сделали в условном операторе. Так как для перечисления элементов мы использовали конструкцию `for`, мы можем вместо списка в функцию передать любой итерируемый объект. Например, строку (элементами будут отдельные символы) или интервал `range`.

Например, найдем все числа от 1 до 99, которые при делении на 12 дают 7 в остатке.

```
simple_filter(lambda x: x % 12 == 7,  range(1, 100))
# => [7, 19, 31, 43, 55, 67, 79, 91]
```

Функция `map`

Другая популярная функция высшего порядка называется **`map`**.

Функция **`map`** преобразует каждый элемент списка по некоторому общему правилу и в результате создает список (вернее, как и `filter`, специальный итерируемый объект, похожий на список) из преобразованных значений.

Функция, которую `map` принимает, – преобразование одного элемента. Зная, как преобразуется один элемент, `map` выполняет превращение целых списков.

Например, возьмем набор из чисел от 1 до 10 и применим к ним функцию возведения в квадрат.

```
list(map(lambda x: x**2, range(1, 10)))
# => [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Еще немного о списочных выражениях

Изучая списочные выражения, мы разбирали конструкции, которые очень похожи на действие функции `map`. Например, список квадратов цифр можно посчитать так:

```
[x**2 for x in pange(10)]
# => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Оказывается, для фильтрации списка тоже есть специальный вид списочного выражения: достаточно приписать `if <условие>` в конце выражения. Такая конструкция отберет только те элементы, которые удовлетворяют условию.

Например, сгенерируем список четных цифр, не делящихся на 3:

```
[x for x in range(10) if x % 2==0 and x %3 !=0]
# => [2, 4, 8]
```

Списочное выражение можно рассматривать как комбинацию фильтрации и трансформации: сначала выполняется фильтрация, затем – трансформирование. Возьмем, например, список слов, оставим только длинные слова и преобразуем их в слова, написанные большими буквами:

```
words = ['В', 'новом', 'списке', 'останутся', 'только',
        'длинные', 'слова']
long_words = list(map(lambda word: word.upper(),
            filter(lambda word: len(word) > 6, words)))

#  => ['ОСТАНУТСЯ', 'ДЛИННЫЕ']

# => или

long_words = [word.upper() for word in words if len(word) > 6]

#  => ['ОСТАНУТСЯ', 'ДЛИННЫЕ']
```

Как видно, оба способа позволяют добиться результата, но списочные выражения выглядят немного проще. В зависимости от ситуации, бывает удобно использовать либо одну форму, либо другую.

Комбинирование функций

Посмотрим, как функции высшего порядка комбинируются для решения более сложных задач.

Разберем, как можно было бы при помощи этих функций решить следующую задачу: найти слова, имеющие три слога и более.

Мы можем разбить эту задачу на такие этапы: выделить список слов, посчитать число гласных букв и в зависимости от их числа взять слово или отбросить. Перед тем как выделять слова мы еще дополнительно отбросим пунктуацию.

Каждый этап обернем в одну небольшую функцию.

```
ENGLISH_ALPHABET = set([chr(c) for c in range(ord('a'),
       ord('z') + 1)])
RUSSIAN_ALPHABET = set([chr(c) for c in range(ord('а'),
       ord('я') + 1)] + ['ё'])
ALPHABET = ENGLISH_ALPHABET | RUSSIAN_ALPHABET
```

```
ENGLISH_VOWELS = {'a', 'e', 'i', 'o', 'u'}
RUSSIAN_VOWELS = {'а','е','ё','и','о','у','ы','э','ю','я'}
VOWELS = ENGLISH_VOWELS | RUSSIAN_VOWELS

def remove_punctuation(text):
    return ''.join(filter(lambda c: c in ALPHABET ^ {' '},
text))

def get_words(text):
    return remove_punctuation(text).split()

def number_of_vowels(word):
    return len(list(filter(lambda c: c in VOWELS, word)))

def long_words(text):
    return filter(lambda word: number_of_vowels(word) >= 3,
            get_words(text))

def print_long_words(text):
    for word in long_words(text):
        print(word.lower())

# Проверка:

txt = 'Как хорошо в стране советской жить'
print_long_words(txt)
txt = 'Commands execute without debug'
print_long_words(txt)
```

Результат:

```
хорошо
советской
execute
without
```

Функцию `filter` мы используем трижды:

- Чтобы оставить только буквы и пробелы
- Чтобы выделить в строке только гласные
- Чтобы из всех слов отобрать только нужные

Функции получились не очень сложные и (что важно) их легко комбинировать друг с другом.

Обратите внимание: не всегда требуется создавать лямбда-функцию при вызове, поскольку иногда нужная функция уже существует. Например, если мы хотим преобразовать список слов в список длин слов, мы можем использовать любой из двух вариантов (но, конечно, удобнее использовать более короткий):

```
words = 'the quick brown fox jumps over the lazy dog'.split()
list(map(lambda word: len(word), words))
# => [3, 5, 5, 3, 5, 4, 3, 4, 3]
list(map(len, words))
# => [3, 5, 5, 3, 5, 4, 3, 4, 3]
```

Еще один пример – считывание списка чисел с клавиатуры:

```
numbers = list(map(float, input().split()))
```

Аналогично можно использовать в качестве передаваемой функции методы объектов. Но при этом нужно указать не только название метода, но и название типа объекта, к которому эта функция относится. То есть, для метода строк это будет `str`, для списков – `list` и т. д.

И ещё один простой пример – преобразовать каждое слово в списке к верхнему регистру:

```
words = ['list', 'of', 'several', 'words']
list(map(lambda word: word.upper(), words))
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
list(map(str.upper, words))
# => ['LIST', 'OF', 'SEVERAL', 'WORDS']
```

Существует еще множество полезных функций, которые принимают функцию в качестве аргумента. Оказывается, что даже давно известные вам функции `min`, `max`, `sort` могут принимать функцию одним из аргументов, что изменяет их поведение.

Обработка коллекций

В Python встроено множество функций, которые помогают перебирать и комбинировать данные любыми способами. Ранее мы познакомились с функциями высшего порядка и даже попробовали их комбинировать. В Python нередко можно сложное вычисление свести к одной строке, если правильно подобрать порядок преобразования данных. Здесь мы будем изучать арсенал имеющихся инструментов и учиться их использовать.

Почему `filter` и `map` возвращают не список

Прежде чем обсуждать новые функции, нужно немного поговорить об уже изученных функциях `map` и `filter`. Вы, возможно, помните, что эти функции

принимают любую коллекцию (список, кортеж, строку символов и т. д.). Возвращают эти функции уже не список, а специальный объект, который можно затем передать в список, в цикл `for` и в некоторые другие функции. Давайте разберемся, как это работает и почему так сделано.

Для начала поймем, почему эти функции возвращают не список. Представьте, что вы работаете с очень большим списком. Например, списком из миллиарда чисел (он занимает не меньше 4 гигабайтов памяти). Если вам требуется как-то обработать набор квадратов этих чисел, есть несколько вариантов.

Первый – перебирать элементы обычным циклом `for` и отказаться от комбинирования операций, которое вы научились делать при помощи `map` и `filter`. Этот вариант, наверное, самый простой, но не слишком удобный. Особенно учитывая, что, помимо `map` и `filter`, вы познакомитесь со множеством других удобных функций, работающих аналогично.

Второй вариант – сделать список квадратов, затем работать уже с ним. Это удобно, но придется потратить еще несколько гигабайтов оперативной памяти, даже если чисел меньше миллиарда, вы вряд ли захотите, чтобы программа тратила лишнюю память.

Итерируемые объекты

Функция `map` использует гибридный метод. Ее результат позволяет перебирать не числа, а их квадраты – как мы и хотели. При этом квадраты чисел нигде не хранятся и не занимают память! Объекты, которые возвращают функции `map`, `filter` и подобные, называются итерируемыми объектами. Это означает, что они позволяют перебирать значения по очереди и последовательно.

В нашем примере функция `map` в любой момент времени хранит только то единственное число, с которым работает, а не весь миллиард квадратов исходных чисел. Вы не создаете огромный промежуточный список и не тратите лишнюю память.

Эффект легко увидеть своими глазами. Откройте диспетчер задач и следите за потреблением памяти интерпретатором Python при запуске двух разных команд:

```
# Версия, создающая промежуточный список.
# Осторожно: при запуске этой команды, Python сначала
# занимает несколько сотен мегабайт оперативной памяти,
# а затем, когда список станет не нужен – освобождает память.

sum([x**2 for x in range(50*1000*1000)])
# => 416666654166666750000000

# Версия, работающая при помощи итератора, который
```

```
# не хранит промежуточный список.
# она занимает минимум дополнительной памяти.

sum(map(lambda x: x**2, range(50*1000*1000)))
# => 41666665416666675000000
```

Упрощенно говоря, есть два типа итерируемых объектов:

- Итераторы, которые позволяют перебирать элементы. Они не хранят все значения элементов, им нужно помнить только начало промежутка, его конец и текущий элемент
- Коллекции (списки, строки, словари и т. д.), которые позволяют создать итератор по своим элементам

Большинство функций Python, которые работают с итераторами, умеют работать и с коллекциями.

Поэтому слова «итерируемый объект» и «итератор» мы будем использовать как синонимы. Кроме того, за неимением лучшего названия, мы часто будем называть итераторами функции, которые возвращают итератор (такие как `range`, `map`, `filter` и мн. др.).

Функции `max/min/sorted` и использование ключа сортировки

Рассмотрим еще один полезный специальный синтаксис в Python, позволяющий избавиться от промежуточных итераторов, которые исходно нам не даны и не нужны в итоговом результате. Так мы сможем сократить число неуклюжих конструкций, в которых сначала создается сложная структура, а потом эта структура упрощается обратно.

Параметр `key`

У функций вроде `min/max/sorted` есть опциональный (необязательный) параметр `Key` Параметр `Key` принимает функцию, по значению которой будут сравниваться элементы.

Например, пусть у нас есть набор слов, который мы хотим отсортировать:

```
words = ['мир', 'и', 'война']
```

Отсортировать слова можно различными способами. Если мы применим функцию `sorted` без аргумента `Key`, слова будут отсортированы как в словаре (это называется лексикографически):

```
sorted(words)    # =>  ['война', 'и', 'мир']
```

Теперь давайте вызовем функцию `sorted` следующим образом:

```
sorted(words, key=lambda s: len(s))
#  => ['и', мир', 'война']
```

Мы указали, что в качестве ключа для сортировки должны использоваться не сами строки (встроенное в Python сравнение строк – лексикографическое), а их длины. Таким образом, мы получаем список, отсортированный по возрастанию длины слова.

Очень удобно использовать ключ сортировки, если нам надо отсортировать список упорядоченных коллекций (списков, кортежей, строк). Например, у нас есть список, элементами которого тоже являются списки, которые содержат название фильма, его возрастное ограничение и рейтинг по отзывам критиков. И мы хотим отсортировать его сначала по возрастному ограничению, затем по оценке критиков (по убыванию) и только в конце по названию. В этом случае нам поможет вот такой код:

```
Li = [
['Crawl',     'R', 61],
['Stubep',    'R', 42],
['Midsommar', 'R', 73],
['Yesterday', 'PG-13', 56],
['Annabelle Comes Home', 'R', 53],
['Child''s Play', 'R', 48],
['Anna', 'R', 40],
['Toy Story 4', 'G', 84],
['Shaft', 'R', 40],
['Men in Black: International', 'PG-13', 38]
]

print(*sorted(Li, key=lambda x: (x[1],-x[2],x[0])), sep='\n')
```

Результат:

```
['Toy Story 4', 84]
['Yesterday', 'PG-13', 56]
['Men in Black: International', 'PG-13', 38]
['Midsommar', 'R', 73]
['Crawl', 'R',  61]
['Annabelle Comes Home', 'R', 53]]
['Chi1d''s P1ay', 'R', 48]
['Stuber', 'R', 42]
['Anna', 'R', 40],
['Shaft', 'R', 40]
```

Помимо функции `sorted`, параметр `key` принимают функции `max` и `min`. Вызов `max (values, key )` позволяет найти значение из набора `values`, наибольшее по ключу `key`.

Проверка коллекций: `all`, `any`

При работе с коллекциями часто приходится определять, выполняется ли некоторое условие одновременно для всех элементов коллекции или хотя бы для одного.

`all` и `any`

Для этих целей существуют две встроенные функции: **`all` и `any`**. Первая проверяет, что все элементы переданного ей итерируемого набора значений истинны (приводятся к `True`). Вторая проверяет, что есть хотя бы один такой элемент. В терминах математической логики эти функции – кванторы общности и существования.

В качестве единственного аргумента **`all`** и **`any`** принимают что-нибудь перечисляемое – например, список, кортеж или итератор.

Итак, **`all`** вернет `True` в том случае, если все элементы аргумента `True` или приводятся к `True` (или если коллекция пустая):

```
print(all([1, 2, 3, 4, 5, 6, 7, 8, 9]))
#  True – так как все элементы ненулевые

print(all([1, 2, 3, 4, 5, 6, 7, 8, 0]))
#  False – есть 0

print(all([1, 2, 3, 4, 5, 6, [], set()]))
#  False – есть пустые вложенные коллекции

print(all([]))    # True
```

Функция **`any`** вернет `True`, если истинен хотя бы один элемент аргумента. **`any`** возвращает `False` для пустых коллекций:

```
print(any((set(), [], {}, 0, True)))
# True – есть True среди элементов

print(any([set(), [], {}, 0, [1,2,3]]))
# True – непустой список приводится к True

print(any([set(), [], {}, 0, False]))
# False – все элементы приводятся к False

print(any([]))  # False
```

Функции **`all`** и **`any`** могут быть особенно полезны в комбинации с функцией **`map`**, которая для каждого элемента коллекции проверит некоторое условие и вернет итератор, в котором будут перечисляться результаты этих проверок. Так, например, можно проверить, все ли числа в списке четные:

```
data = [1, 2, 3, 4, 5]
print(all(x % 2 == 0 for x in data))
# => False
```

А так – узнать, есть ли среди слов хотя бы одно, длиной 5 букв или более:

```
words = "Ехал грека через реку".split()
print(any(map(lambda w: len(w) >= 5, words)))
# => True
```

Потоковый ввод `stdin`

В Python есть очень полезный встроенный итерируемый объект: `sys.stdin`. Это – итератор так называемого потока ввода.

Поток ввода (`stdin`) – специальный объект в программе, куда попадает весь текст, который ввел пользователь. Потоком его называют потому, что данные хранятся там до тех пор, пока программа их не считала. Данные поступают в программу и временно «складируются» в потоке ввода, а программа может «забрать» их оттуда, например, при помощи функции `input()`. В момент прочтения они пропадают из потока ввода: он хранит данные «до востребования».

`sys.stdin`

`sys.stdin` – пример итератора, который невозможно перезапустить. Как и любой итератор, он может двигаться только вперед. Но если для списка можно сделать второй итератор, который начнет чтение с начала списка, то с потоком ввода такое не пройдет. Как только данные прочитаны, они удаляются из потока ввода безвозвратно.

Элементы, которые выдает этот итератор, – строки, введенные пользователем. Если пользовательский ввод закончен, итератор тоже прекращает работу. Пока пользователь не ввел последнюю строку, мы не знаем, сколько элементов в итераторе.

Чтобы работать с `sys.stdin`, прежде всего необходимо подключить модуль `sys` командой `import sys`. Напишем небольшую программу, которая печатает каждую введенную пользователем строку:

```
import sys
for line in sys.stdin:
    print(line.rstrip('\n'))
# rstrip('\n') "отрезает" от строки line крайний правый символ
# перевода строки, ведь print сам переводит строку
```

Что происходит?

Пока есть данные в потоке `sys.stdin` (то есть пока пользователь их вводит), программа будет получать вводимые строки в переменную `line`, убирать справа символы перевода строки и выводить их на печать.

Но если вы запустите эту программу, она будет работать вечно. Чтобы показать, что ввод закончен, пользователю недостаточно нажать Enter – компьютер не знает, завершил пользователь работу или будет еще что-то вводить (при этом Enter превратится в пустую строку). Вместо этого вы должны нажать Ctrl + D (если работаете в консоли Linux или IDE PyCharm) либо Ctrl + Z, затем Enter (если работаете в консоли Windows).

Если вы работаете в IDE Wing, кликните правой кнопкой мыши и выберите Send EOF, затем нажмите Enter. Это запишет в поток ввода специальный символ EOF (end of file), который отмечает конец ввода.

Хочется обратить ваше внимание на один интересный факт, который удобнее показать на примере. Напишите простую программу:

```
x, y = input(), input()
```

Запустите программу и введите одну строку (не забудьте нажать Enter). Вместо второй строки введите EOF тем способом, которым это делается в вашей системе. Вы увидите ошибку EOFError – это означает, что `input()` пытается считать данные из потока, который закончился.

Поэтому, если вы не знаете, в какой момент надо прекратить ввод, воспользоваться функцией `input()` не удастся. В таких случаях остается только работать с `sys.stdin`.

Ввод в одну строку

С помощью `sys.stdin` можно в одну строку прочитать весь ввод (о количестве строк которого мы ничего не знаем) в список. Реализуется это, например, так:

```
data = list(map(str.strip, sys.stdin))
```

Кроме того, можно считать все строки (с сохранением символов перевода строки) в список вот таким образом:

```
data = sys.stdin.readlines()
```

А считать многострочный текст из стандартного потока ввода в текстовую переменную можно вот так:

```
str_data = sys.stdin.read()
```