Основы и методология программирования, 1 модуль
ПМИ ФКН ВШЭ

Михаил Густокашин, 2016

mgustokashin@hse.ru

Оглавление

Лекция 1

Ввод-вывод, простые типы данных

1.1 Как изучать программирование

По нашему убеждению единственный способ научиться программировать — это много программировать. Возможно, вы слышали мнение, что научить программированию невозможно и единственный способ это самообразование. Так говорят люди, которые не умеют учить программированию. Мы умеем учить программированию.

Наш курс будет содержать в себе большое число несложных практических задач, чтобы «набить руку». В лекционных материалах также будут содержаться материалы для более глубокого понимания происходящих процессов.

Сдача задач займет достаточно большое время и некоторые из них действительно сложные, но через этот курс прошло уже большое количество людей и единственная причина, по которой математически одаренные люди не смогли пройти его — это их лень. Мы будем помогать вам бороться с ленью.

1.2 Язык Python

Язык Python (по-русски можно произносить как Пайтон или Питон) появился в 1991 году и был разработан Гвида ван Россумом. Язык назван в честь шоу «Летающий цирк Монти Пайтона». Одна из главных целей разработчиков — сделать язык забавным для использования.

Сейчас Питон регулярно входит в десятку наиболее популярных языков и хорошо подходит для решения широкого класса задач: обучение программированию, скрипты для обработки данных, машинное обучение, серверная веб-разработка и многое другое. Большинству из вас язык Питон потребуется для написания проектов и в ряде предметов третьего курса, а также в повседневном быту для автоматизации задач обработки данных.

Мы будем изучать язык Питон третьей версии.

1.3 Организация курса, выставление оценок и требования

Продолжительность курса — 1 модуль. Оценки выставляются за выполнение домашних заданий и экзамен. На каждом семинарском занятии вам будет предложено новое задание, которое оценивается максимум в 10 баллов (можно получить +2 дополнительных балла решая задачи «со звездочкой»). Раз в две недели будет проводится очная защита домашних заданий в виде контрольной работы по решению практических задач, при этом задачи будут похожи на задачи из ДЗ. Плохое написание контрольной работы снижает оценку за весь блок домашних заданий.

На решение задач ДЗ отводится одна неделя (168 часов) с момента начала семинара по соответствующей теме. В течение еще одной недели можно досдать задания с коэффициентом 0.5. Затем задания не оцениваются. Уважительной причиной для невыполнения ДЗ является длительная невозможность программировать, например, кома. Такие вопросы решаются индивидуально с преподавателем семинарских занятий.

Все задачи ДЗ, контрольных работ и экзамена проверяются автоматически и вручную. Проверка состоит из нескольких этапов. Во-первых, задача проверяется на заранее подготовленном наборе тестов, на которых ваша программа должна выдавать правильный ответ. Во-вторых, код проверяется на соответствие стандартам оформления PEP8 и другим требованиям предъявлемых преподавателями. В-третьих, программа проходит code review, где ваш преподаватель или ассистент может дать вам рекомендации по исправлению плохих, но не формализуемых явно вещей. В-четвертых, осуществляется проверка на списывание. Задача считается сданной, только если она прошла все четыре этапа проверки. Частичных оценок за задачу не предусмторено.

В соответствие со внутренними нормативными документами в отношение студентов, уличенных в списывании или двойной сдаче (совместное выполнение работы) будут применены следующие санкции:

1. предупреждение и обнуление балла за всю работу (даже если списана одна задача), в случае списывания на контрольной работе — обнуление оценок за весь блок домашних работ
2. выговор
3. отчисление

Наличие выговора не только ставит студента в рискованную ситуацию быть отчисленным за любое нарушение (например, за курение возле корпуса), но и мешает получению повышенных стипендий, переводу на другую специальность и многое другое.

В случае подлога (выполнения ненужной для себя работы за другого человека), например, решения контрольной работы человеку, у которого она происходит в другое время, применяются следующий санкции (номер — это количество выявленных нарушений):

1. отчисление

Обратите внимание, что указанные санкции применяются к обоим людям: как к списавшим, так и к давшим списать.

При этом мы крайне положительно относимся к конструктивному общению и взаимопомощи между студентами, ведь консультацию от преподавателя или ассистента не всегда можно получить столь же быстро и исчерпывающе. При оказании помощи отстающим товарищам соблюдайте следующие правила:

1. не показывать/пересылать код своих решений
2. не писать код за другого человека
3. убедиться, что отстающий разобрался в теме и способен решать подобные задачи самостоятельно

1.4 Программное обеспечение

Мы будем использовать на нашем курсе интерпретатор языка Питон под название CPython. Он установлен по умолчанию в современных операционных системах семейства Linux, для остальных операционных систем необходимо скачать его с официального сайта (смотри ссылки на вики-странице курса).

После установки нам потребуется IDE (Integrated Development Environment). Рекомендованы к использованию Wing IDE 101 и JetBrains PyCharm. Если вам говорят, что лучше пользоваться vim и голым интерпретатором — вы можете поверить (на свой страх и риск).

Указанное ПО должно быть установлено на ваших ноутбуках для выполнения ДЗ, а также оно установлено во всех компьютерных классах.

1.5 Типы данных и функция вывода

Программы на языке Питон представляют собой обычные текстовые файлы, в которых записана последовательность команд. Код легко читается и интуитивно понятен.

Например, программы выводящая Hello, world! записывается всего в одну строку:

```
print('Hello, world!')
```

В этой программе вызывается функция печати print, которой в качестве параметра передается строка, содержащая в себе фразу Hello, world!. Если мы хотим задать какую-то строку, то должны обрамлять её одинарными (') или двойными(") кавычками, иначе она будет интерпретироваться как код на языке Питон.

Кроме строк в сегодняшнем занятии мы рассмотрим целочисленный тип данных. Например, можно посчитать результат вычисления арифметического выражения $2 + 3$ и вывести его с помощью такой однострочной программы на языке Питон:

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

Такая программа выведет результат вычисления выражения, который будет равен 5. Если бы числа 2 и 3 были заключены в кавычки, то они интерпретировались бы как строки, а операция + проводила бы конкатенацию (склеивание) строк. Например, такой код:

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

выведет 23 — строку, состоящую из склеенных символов '2' и '3'.

Функция print может принимать и несколько параметров, тогда они будут выводиться через пробел, причем параметры могут иметь различные типы. Если мы хотим получить вывод вида 2 + 3 = 5, то можем воспользоваться следующей программой:

```
print('2 + 3 =', 2 + 3)
```

Обратите внимание, что в строке '2 + 3 =' нет пробела после знака =. Пробел появляется автоматически между параметрами функции print. Что же делать, если хочется вывести строку вида 2+3=5 (без пробелов)? Для этого понадобится именованный параметр sep (separator, разделитель) для функции print. Та строка, которая передается в качестве параметра sep будет подставляться вместо пробела в качестве разделителя. В этой задаче мы будем использовать пустую строку в качестве разделителя. Пустая строка задается двумя подряд идущими кавычками.

```
print('2+3=', 2 + 3, sep='')
```

В качестве параметра sep можно использовать любую строку, в том числе состоящую из нескольких символов. Если нам нужно сделать несколько разных разделителей для разных частей строк, то не остается другого выбора, кроме как использовать несколько подряд идущих функций print. Например, если мы хотим вывести строку вида 1 + 2 + 3 + 4 = 10, то можем попробовать воспользоваться следующим кодом:

```
print(1, 2, 3, 4, sep = ' + ')
print(' = ', 1 + 2 + 3 + 4, sep = '')
```

Однако, вывод такого кода нас огорчит. Он будет выглядеть как:

```
1 + 2 + 3 + 4
 = 10
```

Это связано с тем, что после каждой функции print по умолчанию осуществляется перевод строки. Для изменения того, что будет печататься после вывода всего, что есть в функции print можно использовать именованный параметр end. Например, в нашем случае после первого print мы не хотели бы печатать ничего. Правильный код выглядит следующим образом:

```
print(1, 2, 3, 4, sep=' + ', end='')
print(' = ', 1 + 2 + 3 + 4, sep='')
```

В качестве end также можно использовать абсолютно любую строку.

1.6 Переменные

В некоторых задачах вычисления удобно проводить используя вспомогательные переменные. Например, в школьных формулах по физики было удобно вычислять не гигантское выражение целиком, а запоминая результаты вычисления во вспомогательные переменные. Для примера решим задачу вычисления пройденного расстояния по известному времени и скорости:

```
speed = 108
time = 12
dist = speed * time
print(dist)
```

В этой программе мы создаем три переменные: speed для скорости, time для времени и dist для вычисленного расстояния. При использовании переменых в арифметическом выражении просто используется значение, которое лежит в переменной.

Для присваивания значения переменной используется знак =. Имя переменной должно быть записано слева от знака присваивания, а арифметическое выражение (в котором могут быть использованы числа и другие уже заданные переменные) — справа. Имя переменной должно начинаться с маленькой латинской буквы, должно быть осмысленным (анлийские слова или общеупотребимые сокращения) и не должно превышать по длине 10-15 символов. Если логичное имя переменной состоит из нескольких слов, то нужно записывать его с помощью camelTyping (каждое новое слово кроме первого должно быть записано с большой буквы).

Подбробнее о том, как осуществляется присваивание будет описано ниже.

1.7 Арифметические выражения

Мы уже использовали арифметические выражения в наших программах, в частности операции + и *. Также существует ряд других арифметических операций, которые приведены в таблице:

Обозначение	Операция	Операнд 1	Операнд 2	Результат
+	Сложение	11	6	17
-	Вычитание	11	6	5
*	Умножение	11	6	66
//	Целочисленное деление	11	6	1
%	Остаток от деления	11	6	5
**	Возведение в степень	2	3	8

Все операции инфиксные (записываются между операндами), т.е., например, для возведения 2 в степень 3 нужно писать 2**3.

Особо остановимся на операциях вычисления целой части и остатка от деления от числа.

Пусть заданы два числа A и B, причем $B > 0$. Обозначим за C целую часть от деления A на B, $C = A//B$, а за D — остаток от деления A на B, $D = A\%B$.

Тогда должны выполняться следующие утверждения:

$$A = B \times C + D$$

$$0 \leq D < B$$

Эти утверждения необходимы для понимания процесса взятия остатка от деления отрицательного числа на положительное. Нетрудно убедиться, что если -5 разделить на 2, то целая часть должна быть равна -3, а остаток равен 1. В некоторых других языках программирования остатки в такой ситуации могут быть отрицательными, что неправильно по математическим определениям.

В случае, если $B < 0$ выполняются следующие утверждения:

$$A = B \times C + D$$

$$B < D \leq 0$$

Например, при делении 11 на -5 мы получим целую часть равную -3, а остаток будет равен -4.

Если же разделить -11 на -5, то целая часть будет равна 2, а остаток будет равен -1.

Обратите внимание, что целые числа в Питоне не имеют ограничений на длину (кроме объема доступной памяти).

1.8 Операции над строками

Строки также можно сохранять в переменные и использовать в некотором ограниченном количестве выражений. В частности, можно склеивать две строки с помощью операции +:

```
goodByePhrase = 'Hasta la vista'
person = 'baby'
print(goodByePhrase + ', ' person + '!')
```

Также можно умножить строку на целое неотрицательное число, в результате получится исходная строка повторенная заданное число раз:

```
word = 'Bye'
phrase = word * 3 + '!'
print(phrase)
```

Вывод этой программы будет ByeByeBye!

Складывать число со строкой (и наоборот) нельзя. Но можно воспользоваться функцией str, которая по числу генерирует строку. Str — это сокращение от слова string, которое можно перевести на русский как «строка, которая представляет собой последовательность символов». Например, задачу про вывод $2 + 3 = 5$ можно решить и таким способом:

```
answer = '2 + 3 = ' + str(2 + 3)
print(answer)
```

1.9 Чтение данных

Программы, которые умеют только писать, но не умеют читать, редко представляют интерес для пользователей. Узнавать что-то из внешнего мира наши программы будут с помощью функции input(). Эта функция считывает строку из консоли, чтобы закончить ввод строки нужно нажать Enter. Под строкой в данном случае понимается английское слово line, что означает «строка, оканчивающаяся переводом строки». Например, если в такую программу:

```
name = input()
print('I love', name)
```

ввести слово Python, то она напечатает I love Python.

Во многих задачах нам требуется работать со введенными числами, а читать мы умеем только строки. Чтобы преобразовать строку, состоящую из цифр (и, возможно, знака − перед ними) в целое число можно воспользоваться функцией int (сокращение от английского integer, «целое число»). Например, решение задачи о сложении двух чисел будет выглядеть так:

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

Функция int может быть применена не только к результату, возвращаемому функцией input, но и к произвольной строке.

В строках могут быть не только буквы, цифры и прочие знаки препинания, но и, например, символы табуляции и перевода строки. Чтобы использовать эти символы в константной строке в коде программы необходимо записывать их как \t и \n соответственно. Использование бэкслеша перед символом называется экранирование. Также существуют и другие символы, которые требуют бэкслеша перед собой. Например, это кавычки \' и \" (использование бэкслеша просто необходимо, если в строке используются оба типа кавычек), а также, собственно, символ бэкслеша, который надо записывать как \\.

В случае считывания с помощью input символы в консоли экранировать не нужно.

1.10 Примеры решения задач

Рассмотрми несколько задач, решаемых с помощью арифметических операций, которые показывают некоторые идеи.

Пусть есть два товара, первый из них стоит A рублей B копеек, а второй — C рублей D копеек. Сколько рублей и копеек стоят эти товары вместе.

В задачах где есть несколько размерностей величин (например, рубли и копейки, километры и метры, часы и минуты) следует переводить все в наименьшую единицу измерени, осуществлять необходимые действия, а затем переводить обратно к нужным единицам.

В нашей задаче наименьшей единицей являются копейки, поэтому все цены следует перевести в них, затем сложить их, а затем перевести результат обратно в рубли и копейки. Код решения будет выглядеть так:

```
a = int(input())
b = int(input())
c = int(input())
d = int(input())
cost1 = a * 100 + b
cost2 = c * 100 + d
totalCost = cost1 + cost2
print(totalCost // 100, totalCost % 100)
```

Для определения количества рублей нужно разделить цену в копейках на 100 нацело, а для определения оставшегося числа копеек — посчитать остаток от деления на 100.

Следующая задача: Вася играет в Super Mario Bros. очень хорошо и получил N дополнительных жизней. Известно, что переменная, в которой хранится количество жизней может принимать значения от 0 до 255. В случае, если было 255 жизней и игрок получил дополнительную жизнь, счетчик обнуляется. Сколько жизней на счетчике?

В этой задаче достаточно посчитать остаток от деления введенного числа на 256. Такие действия часто требуются, например, при работе со временем (при переходе через сутки счетчик времени обнуляется). Решение задачи выглядит так:

```
n = int(input())
print(n % 256)
```

Следующая задача: вводится число N, необходимо отрезать от него K последних цифр. Например, при $N = 123456$ и $K = 3$ ответ должен быть 123.

Для решения этой задачи нужно понять, что происходит при целочисленном делении на 10 (основание системы счисления). Если мы разделим число на 10, то будет отброшена последняя цифра, независимо от того, какой она была. Если разделим число на 100 — будет отброшено две последние цифры. Исходя из этого получается решение задачи: необходимо просто разделить число N на 10^K:

```
n = int(input())
k = int(input())
print(n // 10**k)
```

1.11 Как переменные устроены внутри

В языке Питон все переменные являются ссылками на объекты. Каждый объект имеет тип (нам известны int и str) и содержимое, в нашем случае конкретное число или последовательность символов.

Переменные (ссылки) в языке Питон удобно представлять себе как ярлычки на веревочке, которые привязаны к какому-то объекту. Вообще говоря, к одному объекту может быть привязано сколь угодно много ярлыков. Различные переменные с одинаковым значением фактически являются ярлычками, привязанными к одному и тому же объекту.

Типы int и str в Питоне являются неизменяемыми. Любое присваивание в Питоне не может изменить неизменяемый тип, а может только изменить место, на которое указывает ссылка (и, при необходимости, сконструировать новый объект).

Например, команда x = 2, приведет сначала к созданию объекта типа «целое число» со значением 2 в памяти, а затем к созданию переменной x, которая будет являться ссылкой на этот объект.

Если после этого написать y = 2, то новый объект со значением 2 создаваться не будет, а создастся только новая ссылка с именем y показывающая на тот же самый объект, что и ссылка x.

Если теперь написать строку x = 3, то с объектом со значением 2 ничего не случится, ведь он не неизменяемый. Создастся новый объект со значением 3, ссылка x отвяжется от объекта со значением 2 и привяжется к новому объекту 3. При этом к объекту 2 останется привязана ссылка y.

Если изменить и значение переменной y, то у объекта 2 не останется ссылок на него. Поэтому он может быть безболезненно уничтожен при сборке мусора, ведь получить к

нему доступ уже невозможно — на него не ссылается ни одна переменная.

Константные значения в программе (например, явно заданные числа в исходном коде программы) также являются ссылками на объекты, содержимое которых совпадает со значением этих констант. Однако эти ссылки не могут быть изменены и не могут участвовать в присваивании с левой стороны от знака =.

1.12 Как решать задачи

У каждой задачи в вашем домашнем задании есть условие, формат входных и выходных данных и примеры. В условии содержится описание задачи, которую нужно решить. В формате входных данных сказано, какие числа вводятся и в каком порядке они даны, а также указаны ограничения на эти числа. Гарантируется, что чисел будет столько, сколько нужно, и они будут удовлетворять ограничениям. Вам нужно решить задачу только для указанных ограничений. Как программа будет работать для чисел, не удовлетворяющих ограничениям — абсолютно неважно, таких тестов не будет. В формате выходных данных указывается, что и в каком порядке программа должна выводить — ничего, кроме этого, в решении быть не должно.

Примеры к задаче нужны для лучшего понимания условия и первоначальной самопроверки. Естественно, программа должна работать не только на примерах, но и на любых других допустимых входных данных.

Если программа работает правильно, то она получит статус OK. Если программа получила другой статус — вам следует придумать тесты, удовлетворяющие ограничениям, и проверить работу своей программы. Обязательно найдётся ошибка. Наша тестирующая система работает правильно. Наши тесты правильные. Правильные ответы к нашим тестам правильные. Это точно. Мы проверяли много раз. И не только мы.

Лекция 2

Логический тип данных, условный оператор, цикл while, вещественные числа

2.1 Логический тип данных

Кроме уже известных нам целочисленных и строковых типов данных в Питоне существует также логический тип данных, который может принимать значения «истина» (True) или «ложь» (False).

По аналогии с арифметическими выражениями существуют логические выражения, которые могут быть истинными или ложными. Простое логическое выражение имеет вид <арифметическое выражение> <знак сравнения> <арифметическое выражение>. Например, если у нас есть переменные x и y с какими-то значениями, то логическое выражение $x + y < 3 * y$ в качестве первого арифметического выражения имеет $x + y$, в качестве знака сравнения $<$ (меньше), а второе арифметическое выражение в нём $3 * y$.

В логических выражениях допустимы следующие знаки сравнений:

Знак сравнения	Описание
==	равно
!=	не равно
<	меньше
>	больше
<=	меньше или равно
>=	больше или равно

В Питоне допустимы и логические выражения, содержащие несколько знаков сравнения, например $x < y < z$. При этом все сравнения обладают одинаковым приоритетом, который меньше, чем у любой арифметической операции.

Результат вычисления логического выражения можно сохранять в переменную, которая будет иметь тип bool. Переменные такого типа, как и числа и строки, являются неизменяемыми объектами.

Одним из примеров использования логического выражения является проверка на делимость. Например, чтобы проверить, является ли число четным, необходимо сравнить остаток от деления этого числа на два с нулём:

```
isEven = number % 2 == 0
```

Строки также могут сравниваться между собой. При этом сравнение происходит в лексикографическом порядке (как упорядочены слова в словаре).

2.2 Логические операции

Чтобы записать сложное логическое выражение, часто бывает необходимо воспользоваться логическими связками «и», «или» и «не». В Питоне они обозначаются как and, or и not соответственно. Операции and и or являются бинарными, т.е. должны быть записаны между операндами, например x < 3 or y > 2. Операция not — унарная и должна быть записана перед единственным своим операндом.

Все логические операции имеют приоритет ниже, чем операции сравнения (а значит, и ниже чем арифметические операции). Среди логических операций наивысший приоритет имеет операция not, затем идет and и наименьший приоритет имеет операция or. На порядок выполнения операций можно влиять с помощью скобок, как и в арифметических выражениях.

Для примера посмотрим задачу о пересечении двух длительных событий по времени. Оба события характеризуются двумя числами — годами начала и конца. Необходимо определить, пересекались ли события во времени, при этом если одно событие началось в тот год, когда закончилось другое — они считаются пересекающимися.

Первая идея заключается в том, чтобы рассмотреть все возможные варианты расположения событий и выделить следующий критерий пересечения: если начало или конец одного из событий лежит между началом и концом другого, то они пересекаются. В виде программы это можно записать так:

```
is1startIn2 = start2 <= start1 <= finish2
is1finisIn2 = start2 <= finish1 <= finish2
is1in2 = is1startIn2 or is1finishIn2
is2startIn1 = start1 <= start2 <= finish1
is2finisIn1 = start1 <= finish2 <= finish1
is2in1 = is2startIn1 or is2finishIn1
answer = is1in2 or is2in1
```

Если немного подумать, то можно придумать более короткий критерий для проверки такого пересечения: необходимо, чтобы начало первого события происходило не позже конца второго и начало второго события происходило раньше конца второго.

```
answer = start1 <= finish2 and start2 <= finish1
```

Такой способ значительно проще.

В Питоне, как и во многих других языках программирования, если результат вычисления выражения однозначно понятен по уже вычисленной левой части, то правая часть выражения даже не считается. Например, выражение True or 5 // 0 == 42, не будет вызывать ошибки деления на ноль, т.к. по левой части выражения (True) уже понятно, что результат его вычисления также будет True и арифметическое выражение в правой части даже не будет вычисляться.

2.3 Вывод и преобразование логических выражений

Если напечать результат вычисления логического выражения с помощью функции print, то будет напечатано слово True или False.

По аналогии с функциями int и str, существует функция bool, которая может приводить числа или строки к логическому типу. При преобразовании чисел, все числа, отличные от 0 преобразуются в True, а число 0 — в False.

Функция bool примененная к пустой строке дает результат False, а при применении к абсолютно любой другой строке дает результат True. Например, bool("False") даст результат True, поскольку строка "False" не пустая.

Если применить функцию int к логическому выражению, то True преобразуется в 1, а False — в 0.

Применение функции str к логическому выражению дает строки "True"и "False".

Некоторые приведения типов могут выполняться автоматически:

Тип операнда 1	Тип операнда 2	Операция	Тип результата
bool	int	Арифметическая	int
bool	bool	Арифметическая	int
bool	int	Логическая	bool
int	int	Логическая	bool

Некоторые операции допустимы и для переменных логического типа и строк, но их применение является экзотикой и их не следует использовать без явного приведения типов.

2.4 Условный оператор

Наиболее частое применение логические выражения находят в условных операторах.

Условный оператор позволяет выполнять действия в зависимости от того, выполнено условие или нет. Записывается условный оператор как "if <логическое выражение>:", далее следует блок команд, который будет выполнен только если логическое выражение приняло значение True. Блок команд, который будет выполняться, выделяется отступами в 4 пробела (в IDE можно нажимать клавишу tab).

Рассмотрим, например, задачу о нахождении модуля числа. Если число отрицательное, то необходимо заменить его на минус это число. Решение выглядит так:

```
x = int(input())
if x < 0:
    x = -x
print(x)
```

В этой программе с отступом записана только одна строка, x = -x. При необходимости выполнить несколько команд все они должны быть записаны с тем же отступом. Команда print записана без отступа, поэтому она будет выполняться в любом случае, независимо от того, было ли условие в if'е истинным или нет.

В дополнение к if можно использовать оператор else: (иначе). Блок команд, который следует после него, будет выполняться если условие было ложным. Например, ту же задачу о выводе модуля числа можно было решить, не меняя значения переменной x:

```
x = int(input())
```

```
if x > 0:
    print(x)
else:
    print(-x)
```

Все команды, которые выполняются в блоке else, должны быть также записаны с отступом. Else должен следовать сразу за блоком команд if, без промежуточных команд, выполняемых безусловно. Else без соответствующего if'а не имеет смысла.

Если после if записано не логическое выражение, то оно будет приведено к логическому, как если бы от него была вызвана функция bool. Однако, злоупотреблять этим не следует, т.к. это ухудшает читаемость кода.

Для подсчета модуля числа в Питоне существует функция abs, которая избавляет от необходимости каждый раз писать подсчет модуля вручную.

2.5 Вложенный условный оператор

Внутри блока команд могут находиться другие условные операторы. Посмотрим сразу на примере. По заданному количеству глаз и ног нужно научиться отличать кошку, паука, морского гребешка и жучка. У морского гребешка бывает более сотни глаз, а у пауков их восемь. Также у пауков восемь ног, а у морского гребешка их нет совсем. У кошки четыре ноги, а у жучка — шесть ног, но глаз у обоих по два. Решение:

```
eyes = int(input())
legs = int(input())
if eyes >= 8:
    if legs == 8:
        print("spider")
    else:
        print("scallop")
else:
    if legs == 6:
        print("bug")
    else:
        print("cat")
```

Если вложенных условных операторов несколько, то, к какому из них относится else, можно понять по отступу. Отступ у else должен быть такой же, как у if, к которому он относится.

2.6 Конструкция "иначе-если"

В некоторых ситуациях необходимо осуществить выбор больше чем из двух вариантов, которые могут быть обработаны с помощью if-else. Рассмотрим пример: необходимо вывести словом название числа 1 или 2 или сообщить, что это другое число:

```
number = int(input())
if number == 1:
    print('One')
```

```
elif number == 2:
    print('Two')
else:
    print('Other')
```

Здесь используется специальная конструкция elif, обозначающая «иначе, если», после которой записывается условие. Такая конструкция введена в язык Питон, потому что запись if-else приведет к увеличению отступа и ухудшению читаемости.

Конструкций elif может быть несколько, условия проверяются последовательно. Как только условие выполнено — запускается соответствующий этому условию блок команд и дальнейшая проверка не выполняется. Блок else является необязательным, как и в обычном if.

2.7 Цикл while

While переводится как «пока» и позволяет выполнять команды, до тех пор, пока условие верно. После окончания выполнения блока команд, относящихся к while, управление возвращается на строку с условием и, если оно выполнено, то выполнение блока команд повторяется, а если не выполнено, то продолжается выполнение команд, записанных после while.

С помощью while очень легко организовать вечный цикл, поэтому необходимо следить за тем, чтобы в блоке команд происходили изменения, которые приведут к тому, что в какой-то момент условие перестанет быть истинным.

Рассмотрим несколько примеров.

Есть число N. Необходимо вывести все числа по возрастанию от 1 до N. Для решения этой задачи нужно завести счётчик (переменную i), который будет равен текущему числу. Вначале это единица. Пока значение счетчика не превысит N, необходимо выводить его текущее значение и каждый раз увеличить его на единицу:

```
n = int(input())
i = 1
while i <= n:
    print(i)
    i = i + 1
```

Еще одна часто встречающаяся задача — поиск минимума (или максимума) в последовательности чисел. Пусть задана последовательность чисел, оканчивающаяся нулём. Необходимо найти минимальное число в этой последовательности. Эта задача может быть решена человеком: каждый раз когда ему называют очередное число, он сравнивает его с текущим запомненным минимумом и, при необходимости, запоминает новое минимальное число. В качестве первого запомненного числа нужно взять первый элемент последовательности, который должен быть считан отдельно до цикла.

```
now = int(input())
nowMin = now
while now != 0:
    if now < nowMin:
        nowMin = now
    now = int(input())
```

```
print(nowMin)
```

Также часто возникает задача о подсчете суммы последовательности. Для подсчета суммы чисел необходимо завести переменную, которая будет хранить накопленную на данный момент сумму и, при чтении очередного числа, прибавлять его к накопленной сумме:

```
now = int(input())
seqSum = 0
while now != 0:
    seqSum = seqSum + now
    now = int(input())
print(seqSum)
```

2.8 Инструкции управления циклом

Для управления поведением цикла можно использовать две инструкции, которые позволяют досрочно прерывать выполнение цикла или начинать выполнение инструкций цикла сначала.

Первая команда называется break. После её выполнения работа цикла прекращается (как будто не было выполнено условие цикла). Осмысленное использование конструкции break возможно, только если выполнено какое-то условие, то есть break должен вызываться только внутри if (находящегося внутри цикла). Использование break — плохой тон, по возможности, следует обходиться без него. Рассмотрим пример вечного цикла, выход из которого осуществляется с помощью break. Для этого решим задачу о выводе всех целых чисел от 1 до 100. Использовать break таким образом ни в коем случае не нужно, это просто пример:

```
i = 1
while True:
    print(i)
    i = i + 1
    if i > 100:
        break
```

Команда continue начинает исполнение тела цикла заново, начиная с проверки условия. Её нужно использовать, если начиная с какого-то места в теле цикла и при выполнении каких-то условий дальнейшие действия нежелательны.

Приведём пример использования continue (хотя при решении этой задачи можно и нужно обходиться без него): дана последовательность чисел, оканчивающаяся нулём. Необходимо вывести все положительные числа из этой последовательности. Решение:

```
now = -1
while now != 0:
    now = int(input())
    if now <= 0:
        continue;
    print(now)
```

В этом решении есть интересный момент: перед циклом переменная инициализируется заведомо подходящим значением. Команда вывода будет выполняться только в том случае, если не выполнится условие в if.

В языке Питон к циклу while можно написать блок else. Команды в этом блоке будут выполняться, если цикл завершил свою работу нормальным образом (т.е. условие в какой-то момент перестало быть истинным) и не будут выполняться только в случае, если выход из цикла произошел с помощью команды break.

2.9 Основы работы с вещественными числами

Кроме целых чисел в языке Питон также имеются вещественные числа. В отличие от целых чисел они имеют ограниченную точность и использовать их нужно только в ситуациях, когда целые числа использовать совершенно невозможно.

Для преобразования строки в вещественное число используется функция float. Вещественные числа можно использовать в арифметических выражениях вместе с целыми числами, при этом результат будет вещественным числом. Все арифметические операции с целыми числами также применимы и к вещественным числам с тем же смыслом.

Кроме деления в целых числах, существует также обычное деление, которое обозначается знаком "/". Результат такого деления всегда является вещественным числом, даже если оба операнда были целыми.

Более подробно вещественные числа будут изучены в следующей лекции.

Лекция 3

Вещественные числа и строки

3.1 Вещественные числа в памяти компьютера

В отличие от целых чисел, вещественные числа в языке Питон имеют ограниченную длину.

Подумаем, как хранить десятичную дробь в памяти. Поскольку вещественных чисел бесконечно много (даже больше, чем натуральных), то нам придется ограничить точность. Например, мы можем хранить только несколько первых значащих цифр, не храня незначащие нули. Будем отдельно хранить целое число с первыми значащами цифрами и отдельно хранить степень числа 10, на которую нужно умножить это число.

Например, число 5.972×10^{24} (это масса Земли в килограммах) можно сохранить как 5972 (цифры числа, мантисса) и 21 (на какую степень 10 нужно умножить число, экспонента). С помощью такого представления можно хранить вещественные числа любой размерности.

Примерно так и хранятся числа в памяти компьютера, однако вместо десятичной системы используется двоичные. На большинстве аппаратных систем в языке Питон для хранения float используется 64 бита, из которых 1 бит уходит на знак, 52 бита — на мантиссу и 11 бит — на экспоненту.

53 бита дают около 15-16 десятичных знаков, которые будут хранится точно. 11 бит на экспоненту также накладывает ограничения на размерность хранимых чисел (примерно от 10^{-1000} до 10^{1000}).

Любое вещественное число на языке Питон представимо в виде дроби, где в числителе хранится целое число, а в знаменателе находится какая либо степень двойки. Например, 0.125 представимо как 1/8, а 0.1 как 3602879701896397/36028797018963968. Несложно заметить, что эта дробь не равно 0.1, т.е. хранение числа 0.1 точно в типе float невозможно, как и многих других «красивых» десятичных дробей.

3.2 Запись, ввод и вывод вещественных чисел

Для записи констант или при вводе-выводе может использоваться как привычное представление в виде десятично дроби, например 123.456, так и «инженерная» запись числа, где мантисса записывается в виде вещественного числа с одной цифрой до точки и некоторым количеством цифр после точки, затем следует буква "e" (или "E") и

экспонента. Число 123.456 в инженерной записи будет выглядеть как 1.23456e2, что означает, что 1.23456 нужно умножить на 10^2. И мантисса и экспонента могут быть отрицательными и записываются в десятичной системе.

Такая запись чисел может применяться при создании вещественных констант, а также при вводе и выводе. Инженерная запись удобна для хранения очень больших или очень маленьких чисел, чтобы не считать количество нулей в начале или конце числа.

Если хочется вывести число не в инженерной записи, а с фиксированным количеством знаков после точки, то следует воспользоваться методом format, который имеет массу возможностей. Нам нужен только вывод фиксированного количества знаков, поэтому воспользуемся готовым рецептом для вывода 25 знаков после десятичной точки у числа 0.1:

```
x = 0.1
print('{0:.25f}'.format(x))
```

Вывод такой программы будет выглядеть как 0.1000000000000000055511151, что еще раз подтверждает мысль о том, что число 0.1 невозможно сохранить точно.

3.3 Проблемы вещественных чисел

Рассмотрим простой пример:

```
if 0.1 + 0.2 == 0.3:
    print('All right')
else:
    print('WTF?!')
```

Если запустить эту программу, то можно легко убедиться в том, что $0.1 + 0.2$ не равно 0.3. Хотя можно было надеятся, что несмотря на неточное представление, оно окажется одинаково неточным для всех чисел.

Поэтому при использовании вещественных чисел нужно следовать нескольким простым правилам:

1. Если можно обойтись без использования вещественных чисел — нужно это сделать. Вещественные числа проблемные, неточные и медленные.

2. Две вещественных числа равны между собой, если они отличаются не более чем на ϵ. Число X меньше числа Y, если $X < Y - \epsilon$.

В целом будет полезно представлять себе вещественное число X как отрезок $[X - \epsilon; X + \epsilon]$. Как же определить величину ϵ?

Для этого нужно понять, что погрешность не является абсолютной, т.е. одинаковой для всех чисел, а является относительной. Упрощенно, аппаратную погрешность хранения числа X можно оценить как $X \times 2^{-54}$.

Чаще всего в задачах входные данные имеют определенную точность. Рассмотрим на примере: заданы два числа X и Y с точностью 6 знаков после точки (значит $\epsilon = 5{*}10^{-7}$) и по модулю не превосходящие 10^9. Оценить абсолютную погрешность вычисления $X \times Y$. Рассмотрим худший случай, когда X и Y равны 10^9 и отклонились на максимально возможное значение ϵ в одну сторону. Тогда результат вычисления будет выглядеть так:

$$(X+\epsilon)\times(Y+\epsilon) = XY + (X+Y)\times\epsilon + \epsilon^2$$

Величина ϵ^2 пренебрежимо мала, XY — это правильный ответ, а $(X+Y)\times\epsilon$ — искомое значение абсолютной погрешности. Подставим числа и получим:

$$2\times10^9\times5*10^{-7} = 10^3$$

Абсолютная погрешность вычисления составила 1000 (одну тысячу). Что довольно неожиданно и грустно.

Таким образом, становится понятно, что нужно аккуратно вычислять значение погрешности для сравения вещественных чисел.

Код для сравнения двух чисел, заданных с точностью 6 знаков после точки, выглядит так:

```
x = float(input())
y = float(input())
epsilon = 5 * 10**-7
if abs(x - y) < 2 * epsilon:
    print('Equal')
else:
    print('Not eaual')
```

В случае, если над числами совершались какие-то действия, то значения ϵ нужно вычислять как в приведенном выше примере. В учебных задачах это можно сделать не внутри программы, а один раз руками для худшего случая и применять вычисленное значение как константу.

3.4 Округление вещественных чисел

При использовании целых и вещественных чисел в одном выражении вычисления производятся в вещественных числах. Тем не менее, иногда возникает необходимость преобразовать вещественное число в целое. Для этого можно использовать несколько видов функций округления:

- int — округляет в сторону нуля (отбрасывет дробную часть)
- round — округляет до ближайшего целого, если ближайших целых несколько (дробная часть равно 0.5), то к четному
- floor — округляет в меньшую сторону
- ceil — округляет в большую сторону

Примеры для различных чисел:

Функция	2.5	3.5	-2.5
int	2	3	-2
round	2	4	-2
floor	2	3	-3
ceil	3	4	-3

Функции floor и ceil находятся в библиотеке math. Есть два способа получить воспользоваться ими в своей программе.

В первом способе импортируется библиотека math, тогда перед каждым вызовом функции оттуда нужно писать слово "math.", а затем имя функции:

```
import math

print(math.floor(-2.5))
print(math.ceil(-2.5))
```

Во втором способе из библиотеки импортируются некоторые функции и доступ к ним можно получить без написания "math.":

```
from math import floor, ceil

print(floor(-2.5))
print(ceil(-2.5))
```

Второй способ предпочтительно применять в случае, если какие-то функции используются часто и нет конфликта имен (функций с одинаковыми именами в нескольких подключенных библиотеках).

В библиотеке math также есть функция округления trunc, которая работает аналогично int.

3.5 Полезные функции библиотеки math

Кроме функций округления в библиотеке math находится масса полезных функций, которые приведены ниже:

sqrt(x)	Квадратный корень.
pow(a, b)	Возведение в степень, возвращает a^b.
exp(x)	Экспонента, возвращает e^x.
log(x)	Натуральный логарифм.
log(x, b)	Логарифм x по основанию b.
log10(x)	Десятичный логарифм.
e	Основание натуральных логарифмов $e \approx 2.71828$.
sin(x)	Синус угла, задаваемого в радианах.
cos(x)	Косинус угла, задаваемого в радианах.
tan(x)	Тангенс угла, задаваемого в радианах.
asin(x)	Арксинус, возвращает значение в радианах.
acos(x)	Арккосинус, возвращает значение в радианах.
atan(x)	Арктангенс, возвращает значение в радианах.
atan2(y, x)	Полярный угол (в радианах) точки с координатами (x, y).
hypot(a, b)	Длина гипотенузы прямоугольного треугольника с катетами a и b.
degrees(x)	Преобразует угол, заданный в радианах, в градусы.
radians(x)	Преобразует угол, заданный в градусах, в радианы.
pi	Константа π.

3.6 Срезы строк

Нам известны способы считывать, выводить и задавать константные строки, а также склеивать строки между собой и умножать строку на число.

Чтобы определить длину строки s можно воспользоваться функцией len(s) — она возвращает целое число, равное длине строки.

Срез — это способ извлечь из строки отдельные символы или подстроки. При применении среза конструируется новая строка, строка, к которой был применён срез, остается без изменений.

Простейший вид среза — это обращение к конкретному символу строки по номеру. Чтобы получить i-ый символ строки нужно написать s[i]. В результате этого будет сконструирована строка, содержащая только один символ — тот, который стоял на месте i. Нумерация символ идет с нуля, при попытке обратиться к символу с номером больше либо равном длине строки возникает ошибка.

В языке Питон присутствует и нумерация символов строки отрицательными числами. Последний символ строки имеет номер -1, предпослендий — -2 и так далее. При попытке обратиться к символу с номером, меньшим чем -len(s) возникает ошибка.

Нумерация символов в строке "String" представлена в таблице:

S	t	r	i	n	g
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Получить доступ, например, к символу n, можно двумя способами s[4] и s[-2].

Также существуют срезы с двумя параметрами: в результате применения среза s[a:b] будет сконструирована подстрока начиная с символа на позиции a и заканчивая символом на позиции $b-1$ (правая граница не включается). Каждый из индексов может быть как положительным, так и отрицательным.

Например, при s = "String s[1:5] будет равно "trin это можно было бы записать и как s[1:-1]. Если в качестве второго числа в срезе взять число, больше либо равное длине строки, то ошибки не возникнет и будут взяты все символы до конца строки.

В случае, если нужно взять все символы строки начиная с позиции a и до конца, то второй параметр можно опускать. Например, s[2:] будет равно "ring".

Если опустить первый параметр в срезе, то будет взята подстрока с начала, например s[:2] будет равно "St". Если же написать S[:] то будет взята вся строка от начала до конца.

Если первый параметр находится правее второго, то будет сгененрирована пустая строка.

Также существует срез с тремя параметрами, где третий параметр задает шаг, с которым нужно брать символы. Например, можно взять все символы с начала до конца с шагом 2, это будет выглядеть как s[::2], в результате чего получится строка "Srn". Естественно, первый и второй параметры можно не опускать. Если третий параметр не указан, т.е. в квадратных скобках записано только одно двоеточие, то шаг считается равным 1.

Шаг в срезе может быть и отрицательным, в таком случае первый параметр должен находится правеее второго. Например, s[5:1:-2] даст строку "gi-— 5-ый и 3-ий символы, а символ с номером 1 уже не входит. Развернутую строку можно получить срезом s[::-1] —все символы от «начала» до «конца» в обратном порядке. Если третий параметр отрицательный, то началом среза считается последний символ, а концом — позиция

перед нулевым символом.

3.7 Методы find и rfind

Методы — это функции, применяемые к объектам. Метод вызывается с помощью записи ИмяОбъекта.НазваниеМетода(Параметры). Методы очень похожи на функции, но позволяют лучшим образом организовывать хранение и обработку данных. Например, вы написали свою структуру данных и хотели бы, чтобы функция len возвращала длину вашей структуры. Чтобы это заработало, вам придется лезть в исходный код интерпретатора Питона и вносить изменения в функцию len. Если бы len было методом, то вы могли бы описать этот метод при создании структуры и никаких изменений в коде интерпретатора или стандартной библиотеки не потребовалось бы. Поэтому методы предпочтительнее для сложных структур, например, таких как строки.

У строк есть множество различных методов. В этом разделе мы рассмотрим методы поиска подстроки в строке. Метод find возвращает индекс первого вхождения подстроки в строку, а если она не нашлась — -1. Например, 'String'.find('ing') вернет 3 — индекс, с которого начинается вхождение подстроки ing.

Метод rfind ищет самое правое вхождение.

Существуют модификации этих методов с двумя параметрами. s.find(substring, from) будет осуществлять поиск в подстроке s[from:]. Например, 'String'.find('ing', 1) вернет 3 (нумерация символов остается как в исходной строке). По аналогии со срезами параметры могут быть и отрицательными.

Также есть модификации с тремя параметрами: они ищут подстроку в срезу s[a:b].

Часто возникает задача найти и вывести все вхождения подстроки в строку, включая накладывающиеся. Например, для строки 'ABABA' и подстроки 'ABA' ответ должен быть 0, 2. Ее решение выглядит так:

```
string = input()
substring = input()
pos = string.find(substring)
while pos != -1:
    print(pos)
    pos = string.find(substring, pos + 1)
```

3.8 Методы replace и count

Метод replace(old, new) позволяет заменить все вхождения подстроки old на подстроку new. При этом конструируется новая строка, где были произведены замены. Нужно обратить внимание, что метод replace заменяет вхождения подстрок без учета предыдущих совершенных замен. Если применить следующую операцию 'AAAAAA'.replace('AA', 'A'), то в результате получится строка 'AAA', а не 'A', как можно было бы ожидать.

Фактически, можно считать, что метод replace находит очередное вхождение подстроки old, осуществляет замену и продолжает поиск с позиции после всех замененных символов (без наложения и поиска в свежезамененной части).

Существует модификация replace(old, new, count), которая осуществляет не более count замен самых левых вхождений подстроки old.

Также для строк существует метод count, который позволяет подсчитать количество вхождений подстроки. По аналогии с методом find определены методы count с двумя и тремя параметрами.

Лекция 4

Функции и рекурсия

4.1 Создание простой функции

Функции — части программы, которые можно повторно вызывать с разными параметрами, чтобы не писать много раз одно и то же. Функции в программировании немного отличаются от математических функций. В математике функции могут только получить параметры и дать ответ, в программировании же функции умеют делать что-нибудь полезное, например, ничего не возвращать, но что-то печатать.

Функции чрезвычайно полезны, если одни и те же действия нужно выполнять несколько раз. Но некоторые логические блоки работы с программой иногда тоже удобно оформлять в виде функции. Это связано с тем, что человек может одновременно держать в голове ограниченное количество вещей. Когда программа разрастается, отследить все уже очень сложно. В пределах одной небольшой функции запутаться гораздо сложнее — известно, что она получает на вход, что должна выдать, а об остальной программе в это время можно не думать.

В программировании также считается хорошим стилем писать функции, умещающиеся на один экран. Тогда можно одновременно окинуть взглядом всю функцию и не нужно крутить текст туда-сюда. Поэтому, если получилось что-то очень длинное, то нужно нарезать его на кусочки так, чтобы каждый из них был логичным (делал какое-то определенное действие, которое можно назвать) и не превышал при этом 10-15 строк.

Мы уже использовали готовые функции, такие как print, len и некоторые другие. Эти функции описаны в стандартной библиотеке или других подключаемых библиотеках. Сегодня мы научимся создавать свои функции.

Рассмотрим, как создать свою функцию, на примере вычисления факториала. Текст программы без функции выглядит так:

```
n = int(input())
fact = 1
i = 2
while i <= n:
    fact *= i
    i += 1
print(fact)
```

Вычисление факториала можно вынести в функцию, тогда эта же программа будет выглядеть так:

```
def factorial(num):
    fact = 1
    i = 2
    while i <= num:
        fact *= i
        i += 1
    return fact

n = int(input())
print(factorial(n))
```

Описание функции должно идти в начале программы. На самом деле, оно может быть в любом месте, до первого вызова функции factorial.

Определение функции должно начинаться со слова def (сокращение от define, определить). Дальше идет имя функции, после которого в скобках через запятую перечисляются параметры (у нашей функции всего один параметр). После закрытия скобки должно стоять двоеточие.

Команды, выполняемые в функции должны записываться с отступом, как в блоках команд if или while.

В нашей функции num — это параметр, на его место подставляется то значение, с которым функция была вызвана. Действия внутри функции точно такие же, как в обычной программе, кроме дополнительной команды return. Команда return возвращает значение функции (оно должно быть записано через пробел после слова return) и прекращает её работу. Возвращенное значение подставляется на то место, где осуществлялся вызов функции.

Команда return может встречаться в любом месте функции. После того как она выполнится, работа функции будет прекращена. Здесь есть некоторая аналогия с командой break, применяемой для выхода из цикла.

4.2 Вызовы функций из функции

Функцию подсчета факториала можно использовать для подсчета биномиальных коэффициентов (числа сочетаний). Формула для подсчета числа сочетаний выглядит так: $\frac{n!}{k! \times (n-k)!}$.

Если бы мы не пользовались функциями, то нам потребовалось бы три раза записать почти одно и то же. С помощью функций вычисление выглядит намного проще:

```
print(factorial(n) // (factorial(k) * factorial(n-k)))
```

Подсчет биномиальных коэффициентов можно также оформить в виде функции с двумя параметрами:

```
def binomial(n, k):
    return factorial(n) // (factorial(k) * factorial(n - k))
```

4.3 Несколько return в функции

Как было сказано выше, выполнение функции прерывается по команде return. Для примера рассмотрим функцию поиска максимума из двух чисел, которые передаются ей в качестве параметров:

```
def max2(a, b):
    if a > b:
        return a
    else:
        return b
```

Её можно было бы записать и по-другому:

```
def max2(a, b):
    if a > b:
        return a
    return b
```

Если условие в if'е было истинным, то выполнится команда return a и выполнение функции будет прекращено — до команды return b выполнение просто не дойдет.

С помощью функции max2 можно реализовать функцию max3, возвращающую максимум из трех чисел:

```
def max3(a, b, c):
    return max2(max2(a, b), c)
```

Эта функция дважды вызывает max2: сначала для выбора максимума среди чисел a и b, а затем для выбора максимума между найденным значением и оставшимся числом c.

Здесь нужно обратить внимание, что в качестве аргумента функции может передаваться не только переменная или константное значение, но и результат вычисления любого арифметического выражения. Например, результат, возвращенной другой функцией.

Наши функции max2 и max3 будут работать не только для чисел, но и для любых сравнимых объектов, например, для строк.

4.4 Локальные и глобальные переменные

Все переменные, которыми мы пользовались до сегодняшнего дня, были глобальными. Глобальные переменные видны во всех функциях программы.

Например, такой код:

```
def f():
    print a
a = 1
f()
```

напечатает 1 и выполнится без ошибок. Переменная a — глобальная, поэтому мы можем смотреть на её значение из любой функции. На момент вызова функции f переменная a уже создана, хотя описание функции и идет раньше присваивания.

Если же инициализировать переменную внутри функции, то использовать её вне функции невозможно. Например, такой код:

```
def f():
    a = 1
f()
print(a)
```

завершится с ошибкой "builtins.NameError: name 'a' is not defined"(переменная a не определена). Переменные, значения которых изменяются внутри функции по умолчанию считаются локальными, т.е. доступными только внутри функции. Как только функция заканчивает свою работу, то переменная уничтожается.

Таким образом, если в функции происходило присваивание какой-то переменной, то эта переменная считается локальной. Если присваиваний не происходило, то переменная считается глобальной.

Локальные переменные можно называть такими же именами, как и глобальные. Например, вывод такого кода:

```
def f():
    a = 1
    print(a, end='␣')
a = 0
f()
print(a)
```

Будет "1 0". Сначала произойдет вызов функции f, в которой будет создана локальная переменная a со значением 1 (получить доступ к глобальной переменной a из функции теперь нельзя), затем функция закончит свою работу и будет выведена глобальная переменная a, со значением которой ничего не случилось.

Переменная считается локальной даже в случае, если её присваивание происходило внутри условного оператора (даже если он никогда не выполнится):

```
def f():
    print(a)
    if False:
        a = 0
a = 1
f()
```

Эта программа завершится с ошибкой builtins.UnboundLocalError: local variable 'a' referenced before assignment (обращение к переменной до инициализации). Любое присваивание значения переменной внутри тела функции делает переменную локальной.

С помощью специальной команды global можно сделать так, что функция сможет изменить значение глобальной переменной. Для этого нужно записать в начале функции слово global, а затем через запятую перечислить имена глобальных переменных, которые функция сможет менять. Например, такой код:

```
def f():
    global a
    a = 1
    print(a, end='␣')
```

```
a = 0
f()
print(a)
```

выведет "1 1 т.к. значение глобальной переменной будет изменено внутри функции.

Все параметры функции являются локальными переменными со значениями, которые были переданы в функцию. Параметры также можно изменять и это никак не повлияет на значения переменных в том месте, откуда была вызвана функция (если тип объектов-параметров был неизменяемым).-

Использование глобальных переменных как на чтение, так и на запись внутри функций — очень плохой тон. Это связано с тем, что другие люди могут захотеть использовать некоторые отдельные функции из вашего кода, которые не будут работать вне вашей программы в случае использования глобальных переменных.

Поэтому использование глобальных переменных внутри функций в нашем курсе **строго запрещено**. Все нужное для работы функции должно передаваться в качестве параметров.

4.5 Возврат нескольких значений функцией

Рассмотрим случай, когда функция должна вернуть несколько значений на примере функции, упорядочивающей два числа. Чтобы вернуть несколько значений, достаточно записать их в return через запятую. Аналогично, через запятую должны быть перечислены переменные, в которые будут попадать вычисленные значения.

```
def sort2(a, b):
    if a < b:
        return a, b
    else:
        return b, a
a = int(input())
b = int(input())
minimum, maximum = sort2(a, b)
print(minimum, maximum)
```

На самом деле, при перечислении значений через запятую, формируются объекты типа «кортеж», их подробное изучение будет на следующей лекции. Имеющихся знаний достаточно для использования функций, возвращающих несколько значений.

4.6 Возврат логических значений

Иногда удобно оформлять даже простые вещи в виде функций, чтобы повысить читаемость программы. Например, если нужно проверить число на четность, то гораздо понятнее будет каждый раз вызывать функцию isEven(n), а не писать каждый раз n

Такая функция может выглядеть так:

```
def isEven(n):
    return n % 2 == 0
```

Результатом работы этой функции будет истина или ложь. Теперь функцию очень удобно применять в if'ах:

```
if isEven(n):
    print("EVEN")
else:
    print("ODD")
```

Если есть сложное логическое выражение, то лучше оформить его в виде функции с говорящим названием — так программу будет легче читать, а вероятность ошибок в ней резко снизится.

4.7 Механизм запуска функций

Работая с функциями, нужно различать две сущности: последовательность команд, которые выполняются в функции, и локальные переменные конкретного экземпляра функции. Кроме этого каждый экземпляр функции помнит, куда он должен вернуться после завершения работы.

Этим свойством помнить, с какого места нужно продолжить работу функции мы пользуемся при любом запуске функции — выполнение на время уходит в функцию, а после выполнения всех команд управление возвращается ровно на то место, где был произведен вызов функции.

4.8 Рекурсия

Мы уже пробовали запускать функцию из другой функции и все работало. Ничто не мешает запустить из функции саму себя — тогда просто создастся новый экземпляр функции, которые будет выполнять те же команды. Такой процесс называется рекурсией.

Представим себе, что у нас есть миллиард человек (это будущие экземпляры функции), сидящих в ряд, и у каждого из них есть листочек для записи (это его локальная память). Нам нужно произносить числа и написать инструкцию для людей так, чтобы они в итоге сказали все числа из последовательности в обратном порядке. Пусть каждый из них будет записывать на своем листочке только одно число. Тогда инструкция для человека будет выглядеть так:

1. Запиши названное число
2. Если число не последнее — потереби следующего за тобой человека, пришла его очередь работать
3. Когда следующий за тобой человек сказал, что он закончил – назови записанное число
4. Скажи тому, кто тебя теребил (предыдущий человек), что ты закончил

Формализуем задачу. Пусть задается последовательность натуральных чисел, заканчивающаяся нулем. Необходимо развернуть ее с помощью рекурсии.

```
def rec():
    n = int(input())
    if n != 0:
        rec()
        print(n)
rec()
```

Эта функция осуществляет действие (вывод числа) на рекурсивном спуске, т.е. после рекурсивного вызова.

Рассмотрим задачу, где действия выполняются как на рекурсивном подъёме, так и на рекурсивном спуске. Пусть дана последовательность, которая оканчивается нулём. Необходимо вывести все чётные члены последовательности в прямом порядке, а затем все нечётные члены последовательности в обратном порядке. Её решение будет выглядеть так:

```
def rec():
    n = int(input())
    if n != 0:
        if n % 2 == 0:
            print(n)
        rec()
        if n % 2 != 0:
            print(n)
rec()
```

Каждый экземпляр функции считывает в свою локальную переменную n число, если оно чётное, то сразу выводит его и запускает следующий экземпляр. После того, как все последующие экземпляры функции окончили работу (и вывели нечётные числа в обратном порядке), функция выводит число, если оно было нечетным.

Рассмотрим еще один пример: подсчитать факториал числа, не пользуясь циклами:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
n = int(input())
print(factorial(n))
```

Тем, кто знаком с методом математической индукции, будет довольно просто осознать рекурсию. Как и в математической индукции, в рекурсии должна быть база (момент, когда функция не вызывает другую рекурсивную функцию) и переход (правило, по которому считается результат по известному результату для меньшего параметра). Наша функция подсчета факториала делает только свою работу, но пользуется результатами чужого труда. Например, если функция получила на вход параметр 4, то должна вернуть 4 умноженное на 3! (который будет посчитан другими функциями). В случае факториала аналогом «базы индукции» может выступать 0! – по определению он равен единице.

Эти примеры иллюстрируют общую схему написания рекурсивных функций: сначала проверяется условие, когда функция должна закончиться, а дальше делается все

остальное. При этом параметр должен сходиться к значению базы. Обычно это означает, что при каждом следующем вызове рекурсии параметр должен уменьшаться.

Лекция 5

Кортежи, цикл for, списки

5.1 Кортежи

По аналогии со строками, которые могут хранить в себе отдельные символы, в языке Питон существуют тип кортеж, который позволяет хранить в себе произвольные элементы.

Кортеж может состоять из элементов произвольных типов и является неизменяемым типом, т.е. нельзя менять отдельные элементы кортежа, как и символы строки. Константные кортежи можно создавать в программе, записывая элементы через запятую и окружая скобками. Например, testTuple = (1, 2, 3). В случае, если кортеж является единственным выражением слева или справа от знака присваивания, то скобки могут быть опущены. Во всех остальных случаях скобки опускать не следует— это может привести к ошибкам.

Многие приемы и функции для работы со строками также подходят и для кортежей, например, можно складывать два кортежа:

```
a = (1, 2, 3)
b = (4, 5, 6)
print(a + b)
```

В результате применения этой операции будет выведено (1, 2, 3, 4, 5, 6). В случае сложения создается новый кортеж, который содержит в себе элементы сначала из первого, а затем второго кортежа (точно также как и в случае со строками). Также кортеж можно умножить на число, результат этой операции аналогичен умножению строки на число.

Кортеж можно получить из строки, вызвав функцию tuple от строки. В результате каждая буква станет элементом кортежа. К кортежу можно применять функцию str, которая вернет текстовое представление кортежа (элементы, перечисленные через запятую с пробелом и разделенные пробелами).

5.2 Работа с кортежами

К кортежу можно применять функцию len и обращаться к элементам по индексу (в том числе по отрицательному) также как и к строкам.

В одном кортеже могут храниться элементы различных типов, например, строки, числа и другие кортежи вперемешку. Например, в кортеже myTuple = (('a', 1, 3.14),

'abc', ((1), (2,))), myTuple[0] будет кортежем ('a', 1, 3.14), myTuple[1] строкой 'abc', а myTuple[2] кортежем состоящим из числа 1 и кортежа из одного элемента (2,). Числа, записанные в скобках, интерпретируются как числа, в случае возникновения необходимости создать кортеж из одного элемента необходимо после значения элемента написать запятую. Если вывести myTuple[2][1], то напечатается (2,), а если вывести myTuple[2][1][0], то будет напечатано число 2.

Кортеж, содержащий в себе один элемент называется синглтоном. Как и к строкам, к кортежам можно применять операцию среза с тем же смыслом параметров. Если в срезе один параметр, то будет возвращена ссылка на элемент с соответствующим номером. Например, print((1, 2, 3)[2]) напечатет 3. Если же в срезе более одного параметра, то будет сконструирован кортеж, даже если он будет синглтоном. Например, в случае вызова print((1, 2, 3)[1:]) будет напечатно (2, 3), а в случае вызова print((1, 2, 3)[2:]) будет напечатан синглтон (3,).

Кортежи, обычно, предназначаются для хранения разнотиповых значений, доступ к которым может быть получен в результате обращения по индексу или с помощью операции распаковки.

Распаковкой называется процесс присваивания, в котором кортеж, составленный из отдельных переменных находится в левой части выражения. В таком выражении справа должен находится кортеж той же длины. Например, в результате выполнения такого кода:

```
manDesc = ("Ivan", "Ivanov", 28)
name, surname, age = manDesc
```

В переменной name кажется "Ivan", в surname — "Ivanov", а в переменной age число 28. На английском распаковка кортежа называется tuple unpacking.

Процесс создания кортежа называется упаковкой кортежа. Если в одном выражении присваивания происходит и упаковка и распаковка кортежа, то сначала выполняется упаковка, а затем распаковка кортежа. Так, в результате работы программы:

```
a, b, c = 1, 2, 3
a, b, c = c, b, a
print(a, b, c)
```

будет выведено 3 2 1. Обратите внимание, что функции print передается в качестве параметра не кортеж, а три целых числа.

Главное что нужно понять, что записать вида (a, b, c) = (c, b, a) не эквивалентна цепочке присваиваний вида a = c; b = b; c = a. Такая цепочка присваиваний привела бы к тому, что в переменных a, b, c оказались бы значения 3, 2, 3.

5.3 Функция range

В языке Питон есть функция range, которая позволяет генерировать объекты типа iterable (к элементам которых можно получать последовательный доступ) состоящие из целых чисел.

Для вывода объектов типа iterable мы будем пользоваться функцией tuple, которая позволяет сделать кортеж, состоящий из всех элементов iterable, записанных последовательно.

Например, есл запустить программу

```
print(tuple(range(10)))
```

то будет напечатано (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Функция range с одним параметром n генерирует iterable, содержащий последовательные числа от 0 до n-1.

Существует вариант range с двумя параметрами, range(from, to) сгенерирует iterable со всеми числами от $from$ до $to - 1$ включительно.

Также существует range с тремя параметрами range(from, to, step), который сгенерирует iterable с числами от $from$, не превышающие to с шагом изменения $step$. Если шаг отрицателен, то $from$ должен быть больше to. Например, range(10, 0, -2) сгенерирует последовательность чисел 10, 8, 6, 4, 2. 0 не будет входить в эту последовательность.

Во многом параметры range напоминают значения параметров в срезах строк.

5.4 Цикл for

Цикл for позволяет поочередно перебрать элементы из чего-нибудь итерируемого (iterable или tuple). Например, мы можем перебрать названия цветов яблок таким способом:

```
for color in ('red', 'green', 'yellow'):
    print(color, 'apple')
```

В результате выполнения этой программы будет напечатано:
red apple
green apple
yellow apple

На место переменной color будут поочередно подставляться значения из кортежа. В общем случае цикл for выглядит так for имяПеременной in нечтоИтерируемое:

Все действия, которые должны выполняться в for, должны выделяться отступом, как и в if или while. Работа цикл for может быть прервана с помощью команды break или может быть осуществлен переход к следующей итерации с помощью continue. Эти команды имеют тот же эффект, что и при работе с циклом while.

Часто for спользуется вместе с функцией range. Например, с помощью for можно напечатать нечетные числа от 1 до 100:

```
for i in range(1, 100, 2):
    print(i)
```

Внутри for может быть расположен и другой for. Вот так выглядит код для вывода таблиы умножения всех чисел от 1 до 10 (не очень красивой):

```
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end='␣')
    print()
```

Как вы можете заметить, при использовании функции range в for мы не преобразовывали iterable в tuple. Это связано с тем, что for как раз хочет получать последовательный доступ, который умеет давать iterable. Tuple умеет намного больше, но здесь его использование приведет к ненужным затратам времени и памяти.

5.5 Списки

Список в Питоне является аналогом массивов в других языках программирования. Список — это набор ссылок на объекты (также как и кортеж), однако он является изменяемым.

Конастнантные списки записываются в квадратных скобках, все остальное в них аналогично кортежам. Например можно создать список с числами от 1 до 5: myList = [1, 2, 3, 4, 5].

Списки и кортежи легко преобразуются друг в друга. Для преобразования списка в кортеж надо использовать уже известную нам функцию tuple, а для преобразования кортежа в строку нужна функция list. Также функцию list можно применить к строке. В результате этого получится список, каждым элементом которого будет буква из строки. Так list('abc') будет выглядеть как ['a', 'b', 'c'].

К спискам также применима функция len и срезы, которые работают также как в кортежах.

Главным отличием списка от кортежа является изменяемость. То есть можно взять определенный элемент списка и изменить его (он может быть в левой части операции присваивания).

Например, в результате выполнения такого кода:

```
myList = [1, 2, 3]
myList[1] = 4
print(myList)
```

будет напечатано [1, 4, 3].

Изменение символа (или элемента в кортеже) можно было реализовать, сделав два среза и конкатенация первой части строки, нового символа и «хвоста» строки. Это очень медленная операция, время ее выполнения пропорционально длине строки. Замена элемента в списке осуществляется за $O(1)$, т.е. не зависит от длины списка.

5.6 Изменение списков

Список, как и другие типы в языке Питон, является ссылкой на список ссылок. При этом список является изменяемым объектом, т.е. содержимое по этой ссылке может поменяться. рассмаотрим такой пример:

```
a = [1, 2]
b = a
b[0] = 3
print(a)
```

В результате выполнения этой программы будет напечатано [3, 2]. Это связано с тем, что присваивание в Питоне — это просто «привязывание» нового «ярлычка» объекту. После присваивания b = a обе ссылки начинают показывать на один и тот же объект и если он измененен по одной ссылке, то по второй ссылке он тажке будет доступен в измененном состоянии.

Если же написать такой код:

```
a = [1, 2]
b = [1, 2]
```

```
a[0] = 3
print(b)
```

то будет выведено [1, 2]. Несмотря на то, что объекты имеют одинаковое значение из-за их мутабельности (изменяемости) для каждого значения будет создан отдельный объект и ссылки a и b будут показывать на разные объекты. Изменение одного из них, естетственно, не приводит к изменению другого.

В результате выполнения такого кода:

```
a = [1, 2]
b = a
a = [3, 4]
print(b)
```

будет выведено [1, 2]. Сначала в памяти создается объект [1, 2] и к нему привязывается ссылка a, затем к тому же объекту привязывается ссылка b, а затем создается новый объект [2, 3], к которому привязывается ссылка a (отвязавшись от своего предыдущего значения). При этом ссылка b не изменилась (она может измениться только если b будет участвовать в левой части присваивания) и по-прежнему показывает на [1, 2].

Если списки переданы в функцию в качестве параметров, то их содержимое также может быть изменено этой функцией:

```
def replaceFirst(myList):
    myList[0] = 'x'

nowList = list('abcdef')
replaceFirst(nowList)
print(nowList)
```

Выводе этой программы будет ['x', 'b', 'c', 'd', 'e', 'f'].

Однако, сама ссылка внутри функции не может быть изменена, если она передана как параметр функции. Рассмотрим пример:

```
def reverseList(funcList):
    funcList = funcList[::-1]

mainList = list('abc')
reverseList(mainList)
print(mainList)
```

Эта программа не развернет список, т.е. вывод будет ['a', 'b', 'c'].

Здесь в основной программе конструируется объект ['a', 'b', 'c'] и к нему привязывается ссылка mainList. При передаче mainList в качестве параметра в функцию создастся еще одна ссылка funcList, показывающая на объект ['a', 'b', 'c']. В результате применения среза создастся новый объект ['c', 'b', 'a'] и ссылка funcList начнет указывать на него. Однако, значение ссылки mainList при этом не изменится и со значениями по ссылке mainList также ничего не произойдет (напомним, что операция среза создает новый объект, не изменяя старый).

5.7 Методы split и join, функция map

Строки имеют два полезных метода, которые пригодятся при работе со списками.

Метод split позволяет разрезать строку (string) на отдельные слова («токены»). В качестве разделителя может выступать пробел, символ табуляции или перевода строки. Этот метод не изменяет строку и возвращает список строк-токенов.

Например, если заупстить такую программу

```
print('red green        blue'.split())
```

то будет напечатано ['red', 'green', 'blue']. Количество разделителей между токенами не играет роли.

Чтобы научиться читать числа из одной строки нужно научиться еще одной функции — map. Функция map принимает два параметра: первый это функции, а второй — iterable элементов, к которому нужно применить эту функцию. В результате получается iterable с результатом применения функции к каждому элементу списка параметра.

Например, такой код:

```
print(list(map(len, ['red', 'green', 'blue'])))
```

напечатает [3, 5, 4] — список с результатом применения функции len к списку ['red', 'green', 'blue'].

Метод split в сочетании с функцией map удобно использовать для считывания списка чисел, записанных в одну строку и разделенных пробелами. Такое считывание будет выглядеть так:

```
numList = list(map(int, input().split()))
```

Сначала осуществляется считывание строки, затем выполняется метод split, который создает список токенов, состоящих из цифр, а затем к каждому токену применяется функция int. В результате этого получается список цифр.

Метод join позволяет объединить iterable строк, используя ту строку, к которой он применен, в качестве разделителя. Например, такой код:

```
print(', '.join(['Veni', 'Vidi', 'Vici']))
```

выведет Veni, Vidi, Vici. Строка ', ' будет выступать в качестве разделителя который будет вставляться после каждой строки из списка-параметра (кроме последней).

Метод join позволяет быстро и коротко выводить списки чисел. Проблема в том, что он умеет принимать в качестве параметра только iterable строк. Но с помощью функции map мы можем легко получить iterable из списка чисел, применив к каждому элементу функцию str. Вывод списка чисел numList разделенных пробелами будет выглядеть так:

```
numList = [1, 2, 3]
print(' '.join(map(str, numList)))
```

5.8 Другие полезные методы для работы со списками

К переменным типа список можно применять методы, перечислим некоторые из них:

Методы, не изменяющие список и возвращающие значение:

count(x) — подсчитывает число вхождений значения x в список. **Работает за время O(N)**

index(x) — находит позицию первого вхождения значения x в список. **Работает за время O(N)**

index(x, from) — находит позицию первого вхождения значения x в список, начиная с позиции from. **Работает за время O(N)**

Методы, не возвращающие значение, но изменяющие список:

append(x) — добавляет значение x в конец списка

extend(otherList) — добавляет все содержимое списка otherList в конец списка. В отличие от операции + изменяет объект к которому применен, а не создает новый

remove(x) — удаляет первое вхождение числа x в список. **Работает за время O(N)**

insert(index, x) — вставляет число x в список так, что оно оказывается на позиции index. Число, стоявшее на позиции index и все числа правее него сдвигаются на один вправо. **Работает за время O(N)**

reverse() — Разворачивает список (меняет значение по ссылке, а не создает новый список как myList[::-1]). **Работает за время O(N)**

Методы, возвращающие значение и изменяющие список:

pop() — возвращает последний элемент списка и удаляет его

pop(index) — возвращает элемент списка на позиции index и удаляет его. **Работает за время O(N)**

5.9 Почему создание нового лучше удаления

Рассмотрим такую задачу: необходимо выбрать все нечетные элементы списка myList и удалить их из него.

Попробуем решить задачу в лоб — просто будем перебирать все позиции в строке и, если на этой позиции стоит нечетное число, будем удалять его.

```
numbers = list(map(int, input().split()))
for i in range(len(numbers)):
    if numbers[i] % 2 != 0:
        numbers.pop(i)
print(' '.join(map(str, numbers)))
```

Такое решение будет работать неправильно в ситуации, когда в списке есть хоть одно нечетное число. Это связано с тем, что объект без названия с типом iterable и значением range(len(numbers)) сгенерируется один раз, когда интерпретатор впервые дойдет до этого места и уже никогда не изменится. Если в процессе мы выкинем из списка numbers хоть одно значение, то в процессе перебора всех индексов выйдем за пределы нашего списка. range, используемый в for, не будет менять свое значение если в процессе работы изменились параметры функции range.

Решение можно переписать с помощью while:

```
numbers = list(map(int, input().split()))
i = 0
while i < len(numbers):
    if numbers[i] % 2 != 0:
        numbers.pop(i)
```

```
    else:
        i += 1
print(' '.join(map(str, numbers)))
```

Такое решение будет работать, но оно не очень эффективно. Каждый раз при удалении элемента нам придется совершать количество операций, пропорциональное длине списка. Итоговое количество операций в худшем случае будет пропорционально квадарту количества элементов в списке.

В случае, если нет очень строгого ограничения в памяти, в задачах, где нужно удалить часть элементов списка гораздо проще создать новый список, в который нужно добавлять только подходящие элементы.

```
numbers = list(map(int, input().split()))
newList = []
for i in range(len(numbers)):
    if numbers[i] % 2 == 0:
        newList.append(numbers[i])
print(' '.join(map(str, newList)))
```

Сложность такого решения пропорциональна длине исходного списка, что намного лучше.

Лекция 6

Сортировка, лямбда-функции, именованные параметры

6.1 Сортировка списков

В программировании очень часто удобнее работать с отсортированными данными. В языке Питон существует возможность отсортировать списки двумя способами. Рассмотрим их на примере решения простой задачи о сортировке последовательности чисел по неубыванию (это как по возрастанию, но, возможно, с одинаковыми числами). Вот первый способ упорядочить его:

```
myList = list(map(int, input().split()))
myList.sort()
print(' '.join(map(str, myList)))
```

В этом примере используется метод sort, применяемый к списку. Этот метод изменяет содержимое списка — после применения метода sort элементы в списке становятся упорядоченными. Такой метод определен только для объектов типа список, его нельзя применить к кортежу или iterable или строке.

Второй способ состоит в применении функции sorted, которая возвращает отсортированный список, но не изменяет значение своего параметра:

```
myList = list(map(int, input().split()))
sortedList = sorted(myList)
print(' '.join(map(str, sortedList)))
```

Использование функции sorted оправдано в случае, если исходные данные нужно сохранить в неизменном виде с какой-то целью. Например, sorted можно использовать внутри своей функции для создания отсортированной копии, чтобы не портить переданный нам список.

Чтобы отсортировать список по невозрастанию (убыванию), необходимо передать в метод или фукнцию именованный параметр reversed. Например, это будет выглядеть как myList.sort(reversed=True) или sorted(myList, reversed=True).

Функция sorted может принимать в качестве параметра не только список, но и что угодно итерируемое: кортежи, iterable или строки:

```
print(sorted((1, 3, 2)))
print(sorted(range(10, -1, -2)))
```

```
print(sorted("cba"))
```

При этом sorted всегда возвращает список, т.е. вывод этой программы будет такой:

```
[1, 2, 3]
[0, 2, 4, 6, 8, 10]
['a', 'b', 'c']
```

Сортировку можно применять к спискам, все элементы которых сравнимы между собой. Обычно это однородные списки (состоящие из элементов одного типа) или, в редких случаях, целые и вещественные числа вперемешку.

6.2 Сравнение кортежей и списков

Два кортежа или списка можно также сравнивать между собой. Например, выражение $(1, 2, 3) < (2, 3, 4)$ будете истинным, а $[1, 2, 3] < [1, 2]$ ложным. Сравнение кортежей и списков происходит поэлементно, как и сравнение строк. Как только на каких-то позициях кортежа или списка встретились различные элементы, то взаимный порядок кортежей такой же, как у этих элементов. Если же различий найдено не было, то меньше тот кортеж, который короче. Всё в точности как при сравнении строк.

Естественно, сравниваемые кортежи или списки должны содержать на соответствующих позициях сравнимые элементы. Попытка сравнить кортеж (1, 2) с кортежем ("Some text 42) приведет к ошибке (а сравнение (1, 2) с (42, "Some text") к ошибке не приведет). Обычно, всё же, сравниваются кортежи, состоящие из элементов одинакового типа.

Это свойство кортежей можно использовать для решения сложных задач на сортировку. Например, для каждого человека задан его рост и имя, необходимо определить упорядочить список людей по росту, а в случае одинакого роста — в алфавитном порядке. При решении этой задачи достаточно хранить описание каждого человека в виде кортежа, где первым элементом будет рост, а вторым — фамилия.

Рассмотрим пример: для каждого человека задан рост и его имя. Необходимо упорядочить их по возрастанию роста, а в случае одинакового роста — в алфавитном порядке.

```
n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    manData = (int(tempManData[0]), tempManData[1])
    peopleList.append(manData)
peopleList.sort()
for manData in peopleList:
    print(' '.join(map(str, manData)))
```

В этом примере нам повезло и удалось составить кортеж, который содержит параметры сравниваемых людей ровно в нужном порядке. Часто встречаются более неприятные ситуации. Рассмотрим ту же задачу, но теперь людей нужно упорядочить по убыванию роста, но в случае одинакового роста они по-прежнему должны быть упорядочены по алфавиту. Простое использование reversed=True не приведет к желаемому результату: люди с одинаковым ростом будут стоять в неправильном порядке.

Здесь можно применить хитрость и превратить рост каждого человека в отрицательное число, модуль которого будет равен исходному росту. После этого список можно просто упорядочить по возрастанию — самые высокие люди будут иметь наименьший отрицательный «рост», по которому происходит сравнение в первую очередь. Перед выводом необходимо превратить рост обратно в положительное число.

```
n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    manData = (-int(tempManData[0]), tempManData[1])
    peopleList.append(manData)
peopleList.sort()
for badManData in peopleList:
    manData = (-badManData[0], badManData[1])
    print(' '.join(map(str, manData)))
```

Этот код малопонятен и плох.

6.3 Параметр key в функции sort

Для реализации нестандартных сортировок лучше не уродовать исходные данные, а использовать параметр key, передающийся в функцию сортировки.

Значением этого параметра должна быть функция, которая применяется к каждому элементу списка и затем сравнение элементов происходит по значению этой функции (оно называется ключом).

Рассмотрим такой пример: необходимо упорядочить введённые строки по длине, а в случае равной длины оставить их в том порядке, как они шли во входном файле. Например, для входных строк "c", "abb", "b" правильным ответом должно быть "c", "b", "abb" ("c" идет раньше "b", т.к. они имеют равную длину, а "c" стояло во входных данных раньше "b").

К счастью, сортировка, используемая в Питоне обладает свойством устойчивости (stable), т.е. для элементов с равным ключом сохраняется их взаимный порядок.

Решение этой задачи будет выглядеть следующим образом:

```
n = int(input())
strings = []
for i in range(n):
    strings.append(input())
print('\n'.join(sorted(strings, key=len)))
```

В качестве еще одного примера рассмотрим задачу о сортировке точек на плоскости, заданных парой целых координат x и y по неубыванию расстояния от начала координат. В данном случае в качестве функции для генерации ключа, по которому будут сравниваться элементы, мы напишем свою функцию, которая будет возвращать квадрат расстояния от точки до начала координат. Квадрат расстояния мы используем для того, чтобы оставаться в целых числах и избавится от необходимости считать квадратный корень (медленно и неточно):

```
def dist(point):
    return point[0] ** 2 + point[1] ** 2

n = int(input())
points = []
for i in range(n):
    point = tuple(map(int, input().split()))
    points.append(point)
points.sort(key=dist)
for point in points:
    print(' '.join(map(str, point)))
```

Здесь каждый элемент списка — кортеж из двух чисел. Именно такой параметр принимает наша функция. Возможно, вы хотели бы использовать функцию hypot из библиотеки math, чтобы не писать свою функцию подсчета ключа, однако это невозможно — она ожидает на вход два числовых параметра, а не кортеж.

6.4 «Структуры» в Питоне

Для хранения сложных записей во многих языках есть специальные типы данных, такие как struct в C++ или record в Паскале.

Переменная типа структура содержит в себе несколько именованных полей. Например, возвращаясь к задаче сортировки людей по убыванию роста, нам было бы удобно хранить описание каждого человека в виде структуры с двумя полями: ростом и именем.

В чистом виде типа данных «структура» в стандарте языка Питон нет. Есть несколько способов реализации аналога структур: namedtuple из библиотеки collections, использование словарей (будет рассмотрено в следующих лекциях) или использование классов в качестве структур. Рассмотрим на примере последний способ.

Напомним условие задачи: людей нужно упорядочить по убыванию роста, но в случае одинакового роста они должны быть упорядочены по фамилии. Решение с использованием классов в качестве структур будет выглядеть так:

```
class Man:
    height = 0
    name = ''

def manKey(man):
    return (-man.height, man.name)

n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    man = Man()
    man.height = int(tempManData[0])
    man.name = tempManData[1]
    peopleList.append(man)
```

```
peopleList.sort(key=manKey)
for man in peopleList:
    print(man.height, man.name)
```

Для того чтобы пользоваться классами как структурами мы создаем новый тип данных Man. В описании класса мы перечисляем имена всех полей и их значения по-умолчанию.

В дальнейшем мы можем создавать объекты класса Man (это делается строкой man = Man()), которые сначала проинициализируют свои поля значениями по умолчанию. Доступ к полям класса осуществляется через точку.

Функция сравнения принимает объект класса и генерирует ключ, по которому эти объекты будут сравниваться при сортировке.

Использование структур для описания сложных объектов намного предпочтительнее, чем использование кортежей. При количестве параметров больше двух использование кортежей запутывает читателя и писателя кода, т.к. совершенно невозможно понять что хранится в badNamedTuple[13] и легко понять что хранится в goodNamedStruct.goodNamedField.

6.5 Лямбда-функции

В ряде случаев функции, используемые для получения ключа сортировки, так просты, что не хочется оформлять их стандартным образом, а хочется написать их прямо на месте и даже не давать им имени.

Это можно осуществить с помощью лямбда-функций, которые могут заменить собой функции, содержащие в своем теле только оператор return. Запись лямбда-функции, возводящей число в квадрат может выглядеть так:

```
lambda x: x**2
```

Что эквивалентно привычной записи функции:

```
def sqr(x):
    return x**2
```

Отличие лямбда-функции также заключается в том, что у неё нет имени и, следовательно, вызов её по имени невозможен. Пока единственным применением лямбда-функций для нас может служить их передача в качестве параметра в такие функции как sort или map. Например, с помощью лямбда-функции мы можем вывести список квадратов всех чисел от 1 до 100 всего в одну строку:

```
print(' '.join(map(lambda x: str(x**2), range(1, 101))))
```

В этой программе лямбда функция принимает в качестве параметра число, а возвращает строковое представление его квадрата.

Вернемся к задаче сортировки точек по удаленности от начала координат. Эта задача также может быть решена с использованием лямбда-функции:

```
n = int(input())
points = []
for i in range(n):
    point = tuple(map(int, input().split()))
    points.append(point)
```

```
points.sort(key=lambda point: point[0]**2 + point[1]**2)
for point in points:
    print(' '.join(map(str, point)))
```

Лямбда-функция может принимать несколько параметров (тогда после слова lambda нужно записать их имена через запятую), однако при использовании их в sort или map параметр должен быть всегда один.

В языке Питон функция также является объектом и мы можем создать ссылку на объект типа функция. Например, две записи функции возведения в квадрат эквивалентны:

```
def traditionalSqr(x):
    return x**2

lambdaSqr = lambda x: x**2
print(traditionalSqr(3))
print(lambdaSqr(3))
```

Такой подход позволяет переиспользовать лямбда-функции, но в подавляющем большинстве случаев стоит пользоваться стандартным объявлением функции — это упрощает чтение и отладку программы.

6.6 Именованные параметры и неопределенное число параметров

Мы уже много раз пользовались функциями, которые могут принимать (или не принимать) именованные параметры. Например, это необязательные именованные или неименованные параметры sep и end для функции print или параметр key для метода sort и функции sorted.

Сейчас мы научимся создавать функции, которые принимают именованные параметры. Например, напишем функцию, печатающую что угодно итерируемое, состоящее из чего угодно приводимого к строке, с именованным параметром sep, по-умолчанию равным пробелу:

```
def printList(myList, sep=' '):
    print(sep.join(map(str, myList)))

printList([1, 2, 3])
printList([3, 2, 1], sep='\n')
```

Именованный параметр в объявлении функции должен идти после основных параметров. В списке параметров записывается его имя, а затем значение по-умолчанию (т.е. то значение, которое будет подставляться на место соответствующего параметра, если он не был передан при вызове функции).

Также мы пользовались функциями, которые умеют принимать произвольное количество параметров. Например, в функцию print можно передать любое количество параметров. Можно написать собственные функции, которые будут принимать произвольное количество параметров. При этом параметры функции будут упакованы в

список. Например, функция подсчета суммы всех переданных параметров может выглядеть так:

```
def mySum(*args):
    nowSum = 0
    for now in args:
        nowSum += now
    return nowSum

print(mySum(1, 2))
print(mySum(1, 2, 3, 4))
```

Функция принимает один параметр, перед которым написана звездочка — это признак того, что аргументы будут упакованы в список.

Можно писать функции, которые принимают не менее определенного количества параметров. Например, мы можем написать функцию поиска минимума среди неопределенного числа аргументов, но в нее должно быть передано не менее одного аргумента:

```
def myMin(first, *others):
    nowMin = first
    for now in others:
        if now < nowMin:
            nowMin = now
    return nowMin

print(myMin(1))
print(myMin(3, 1, 2))
```

Параметр со звездочкой всегда должен быть последним, за исключением ситуации, когда в функции также определены именованные параметры.

6.7 Чтение до конца ввода

Во многих задачах заранее неизвестно, сколько данных нам предстоит считать. Особенно яркий пример — это обработка текста, когда мы заранее не знаем, сколько строк нам будет введено.

Наиболее удобно работать с такими данными не пользуясь функцией input, используя методы чтения файла (или ввода с консоли) целиком или построчно.

Расммотрим простой пример: считать все строки файла input.txt и вывести каждую строку развернутой в файл output.txt:

```
inFile = open('input.txt', 'r', encoding='utf8')
outFile = open('output.txt', 'w', encoding='utf8')
lines = inFile.readlines()
for line in lines:
    print(line[-2::-1], file=outFile)
inFile.close()
outFile.close()
```

Для открытия файла используется функция open, принимающая два параметра: имя файла и режим открытия ("r" для чтения и "w" для записи), а также именованный параметр encoding (значение кодировки "utf8" подходит для большинства современных текстовых файлов). Эта функция возвращает ссылку на объект типа файл.

Для чтения всех строк из файла используется метод readlines, который возвращает список всех строк (в смысле lines) файла. Обратите внимание, что строки попадают в список вместе с символом перевода строки, в нашей программе это учитывается при создании среза (этот символ последний в строке). В тестирующей системе все входные файлы имеют перенос строки после последней строки, в реальной жизни это может оказаться не так и тогда программа будет работать неверно.

Для печати в файл мы пользуемся стандартной функцией print, которой передается именованный параметр file с указанием, в какой файл печатать.

После окончания работы с файлами нужно вызвать для них методы close.

В этой задаче, на самом деле, можно было обойтись без запоминания всего файла в памяти (что особенно актуально для больших файлов). Решение без запоминания всего файла можно было реализовать так:

```
inFile = open('input.txt', 'r', encoding='utf8')
outFile = open('output.txt', 'w', encoding='utf8')
for line in inFile:
    print(line[-2::-1], file=outFile)
inFile.close()
outFile.close()
```

Переменные типа файл являются iterable и умеют возвращать очередную строку из файла, не храня его целиком в памяти.

Также существует метод read, который позволяет считать все содержимое файла в одну строковую переменную (при этом содержащую в себе переводы строки $\backslash n$).

В принципе, читать до конца ввода можно и из консоли. Для этого нужно подключить библиотеку sys и использовать определенный в ней файловый дескриптор stdin в качестве файла. Ввести признак конца файла в консоли можно нажав Ctrl+Z в Windows или Ctrl+D в Unix-системах.

6.8 Сортировка подсчетом

В ряде задач возможные значения в сортируемом списке сильно ограничены. Например, если мы хотим отсортировать оценки от 0 до 10, то может оказаться эффективнее подсчитать, сколько раз встречалась каждая из оценок и затем вывести её столько раз.

Реализация такого подхода очень проста:

```
marks = map(int, input().split())
cntMarks = [0] * 11
for mark in marks:
    cntMarks[mark] += 1
for nowMark in range(11):
    print((str(nowMark) + ' ') * cntMarks[nowMark], end='')
```

В этой программе мы создали список, состоящий из 11 нулей в одну строку. Этот приём часто пригождается и в других задачах.

6.9 Связь задач поиска и сортировки

Во многих задачах линейного поиска (например, поиск минимального элемента) возникает соблазн воспользоваться сортировкой.

С этим соблазном следует бороться, т.к. сложность сортировки в языке Питон составляет $O(NlogN)$, т.е. для сортировки списка из N элементов нужно совершить порядка $NlogN$ действий.

При этом алгоритмы линейного поиска работают за $O(N)$, что заметно асимптотически быстрее, чем сортировка. Поэтому в задачах линейного поиска (даже для поиска третьего по величине элемента) следует реализовывать линейный поиск, а не пользоваться сортировкой.

По иронии судьбы, сортировка в интерпретаторе CPython может оказаться быстрее рукописного линейного поиска (из-за того, что она реализовано максимально эффективно и на языке Си). Но это досадное недоразумение не должно побороть в вас желание писать линейный поиск руками.

Лекция 7

Множества, словари, полезные методы для строк

7.1 Множества и хеш-функции

В языке Питон множества имеют тот же смысл, что и в математике: набор объектов без определенного порядка. В множество можно добавлять и удалять объекты, проверять принадлежность объекту множества и перебирать все объекты множества.

Также над множествами можно совершать групповые операции, например, пересекать и объединять два множества.

Проверка принадлежности элемента множеству, а также операции удаления и добавления элементов, осуществляются за $O(1)$ (если бы мы хранили элементы в списке и хотели бы проверить принадлежность элемента списку, то нам потребовалось бы $O(N)$ операций, где N — длина списка).

Такая скорость достигается использованием хеш-таблиц. Хеш-таблица — это массив достаточно большого размера (назовем этот размер K). Каждому неизменяемому объекту можно сопоставить по некоторому правилу число M от 0 до K и поместить этот объект в ячейку списка с индексом M. Например, для целых чисел таким правилом сопоставления может быть просто подсчет остатка от деления целого числа на K. Операцию взятия остатка будет нашей хеш-функцией.

Теперь если нам нужно проверить, принадлежит ли некоторое число множеству, мы просто считаем хеш-функцию от него и проверяем, лежит ли в ячейке с индексом, равным результату вычисления хеш-функции наш объект или нет. Для других типов данных можно применить такой подход: любой объект так или иначе является последовательностью байт. Будем интерпретировать эту последовательность байт как число и подсчитаем хеш-функцию для этого числа.

Естественно, может оказаться, что несколько объектов дают один и тот же хеш (отображение между огромным множеством различных объектов и скромным размером множества допустимых хешей не может быть биективным). Такие проблемы можно разрешить, не ухудшая асимптотическую сложность. Подробнее такие методы вы будете изучать на курсе алгоритмов.

Поскольку, например, числа, могут быть достаточно длинными, то операция подсчета хеш-функции при каждой операции с этим объектов в множестве может быть очень медленной. Поэтому каждый неизменяемый объект в Питоне имеет заранее насчитанный хеш, который подсчитывается один раз при его создании. Кстати, с помощью этих

же хешей можно понимать, есть ли уже объект в памяти и не создавать новых объектов, а просто подвешивать еще одну ссылку на уже существующий объект.

Изменяемые типы, такие как список, не имеют заранее насчитанных хешей. Изменение всего одного элемента в списке привело бы к полному пересчету хеша для всего списка, что катастрофически замедлило бы работу со списками. Поэтому у изменяемых объектов нет хеша и они не могут быть добавлены в множество.

Само множество также является изменяемым объектом и не может быть, например, элементом другого множества.

Существуют также неизменяемые множества, которые создаются с помощью функции frozenset.

7.2 Создание множеств

Множество в теле программы может быть создано с помощью записи элементов через запятую в фигурных скобках:

```
mySet = {3, 1, 2}
print(mySet)
```

Вывод с помощью print осуществляется в том же формате. Порядок элементов в множестве может быть случайным, т.к. хеш функция не гарантирует, что если $A > B$, то $h(A) > h(B)$.

Если при задании множества присутствовало несколько одинаковых элементов, то они попадут в множество в единственном экземпляре:

```
firstSet = {1, 2, 1, 3}
secondSet = {3, 2, 1}
print(firstSet == secondSet)
```

Эта программы выведет True (множества можно сравнивать на равенство).

Также множества можно создавать с помощью функции set, которая может принимать в качестве параметра что угодно итерируемое:

```
setFromList = set([1, 2, 3])
print(setFromList)
setFromTuple = set((4, 5, 6))
print(setFromTuple)
setFromStr = set("lol")
print(setFromStr)
setFromRange = set(range(2, 22, 3))
print(setFromRange)
setFromMap = set(map(abs, (1, 2, 3, -2, -4)))
print(setFromMap)
setFromSet = set({1, 2, 3})
print(setFromSet)
```

Вывод этой программы такой:

```
{1, 2, 3}
{4, 5, 6}
```

```
{'l', 'o'}
{2, 5, 8, 11, 14, 17, 20}
{1, 2, 3, 4}
{1, 2, 3}
```

Множество также является итерируемым объектом (еще раз: объекты идут в «случайном» порядке, не по возрастанию!).

Множество может содержать в себе объекты разных типов:

```
mixedSet = {1, 3.14, (1, 2, 3), "i have no idea why i'm here"}
print(mixedSet)
```

По аналогии со строками, списками и кортежами, количество элементов в множестве можно узнать с помощью функции len.

Из множества можно сделать список или кортеж с помощью функций list и tuple соответственно. Применение функции str к множеству даст нам текстовое представление (элементы в фигурных скобках, разделенные запятыми).

Частой операцией является вывод упорядоченных элементов множества. Это можно сделать, применив функцию sorted сразу к множеству (ведь оно итерируемо):

```
mySet = {'abba', 'a', 'long string'}
print(', '.join(mySet))
print(', '.join(sorted(mySet)))
```

Вывод этой программы будет:

```
long string, a, abba
a, abba, long string
```

К множеству можно применять функцию map.

7.3 Работа с элементами множеств

Чтобы создать пустое множество нужно написать:

```
emptySet = set()
```

Писать пустые фигурные скобки нельзя (во второй части лекции узнаете почему).

Добавление элемента в множество осуществляется с помощью метода add, если элемент уже был в множестве, то оно не изменится.

Перебрать элементы множества можно с помощью for (for умеет ходить по любым итерируемым объектам):

```
mySet = {1, '2', 2, '1'}
for elem in mySet:
    print(elem, end=' ')
```

Вывод такой программы будет "1 1 2 2", но упорядоченность является чистой случайностью.

Чтобы проверить, входит ли элемент X в множество A достаточно написать X in A. Результатом этой операции будет True или False. Чтобы проверить, что элемент не лежит в множестве можно писать not X in A, или, более по-человечески X not in A.

```
mySet = {1, 2, 3}
if 1 in mySet:
    print('1 in set')
else:
    print('1 not in set')
x = 42
if x not in mySet:
    print('x not in set')
else:
    print('x in set')
```

Вывод этой программы будет:

```
1 in set
x not in set
```

Чтобы удалить элемент из множества, можно воспользоваться одним из двух методов: discard или remove. Если удаляемого элемента в множестве не было, то discard не изменит состояния множества, а remove выпадет с ошибкой.

7.4 Групповые операции над множествами

В Питоне можно работать не только с отдельными элементами множеств, но и с множествами в целом. Например, для множеств определены следующие операции:

Операция	Описание
A | B	Объединение множеств
A & B	Пересечение множеств
A - B	Множество, где элементы входят в A, но не входят в B
A ^ B	Элементы входят в A | B, не не входят в A & B

В результате этих операций создается новое множество, однако для них определена и сокращенная запись: |=, &=, -= и ^=. Такие операции изменяют множество, находящееся слева от знака операции.

Для множеств также определены операции сравнения:

Операция	Описание
A == B	Все элементы совпадают
A != B	Есть различные элементы
A <= B	Все элементы A входят в B
A >= B	Все элементы B входят в A
A > B	A >= B и A != B
A < B	A <= B и A != B

Все групповые операции и сравнения проводятся над множествами за время, пропорциональное количеству элементов в множествах.

7.5 Словари

В жизни нередко возникает необходимость сопоставить ключу значение. Например, в англо-русском словаре английскому слову сопостовляется одно или несколько русских слов. Здесь английское слово является ключам, а русское — значением.

В языке Питон есть структура данных словарь, которая позволяет реализовывать подобные операции. При этом объекты-ключи уникальны и каждому из них сопоставлен некоторый объект-значение. Ограничения на ключи такие же, как на элементы множества, а вот значения могут быть и изменяемыми.

По-сути, словарь является множеством, где каждому элементу-ключу сопоставлен еще и объект-значение.

Создать словарь в исходном тексте программы можно записав в фигурных скобках пары ключ-значения через запятую, а внутри пары ключ отделяется от значения двоеточием:

```
countries = {'Russia' : 'Europe', 'Germany' : 'Europe', 'Australia' : 'Austra
```

Добавлять пары ключ значение в словарь очень просто: это делается по аналогии со списками:

```
sqrs = {}
sqrs[1] = 1
sqrs[2] = 4
sqrs[10] = 100
print(sqrs)
```

Пустой словарь можно создать, написав пустые фигурные скобки (это будет словарь, а не множество).

Словарь также можно конструировать из других объектов с помощью функции dict:

```
myDict = dict([['key1', 'value1'], ('key2', 'value2')])
print(myDict)
```

На вход функции должен подавать iterable, каждый элемент которого, в свою очередь, является iterable строго с двумя элементами — ключом и значением.

Узнавать значение по ключу можно также с помощью записи ключа после имени словаря в квадратных скобках:

```
phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
print(phones['police'])
```

Если такого ключа в словаре нет, то возникнет ошибка.

Удаление элемента из словаря делается специальной командой del. Это не функция, после слова del ставится пробел, затем пишется имя словаря, а затем, в квадратных скобках, удаляемый ключ:

```
phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
del phones['police']
print(phones)
```

Проверка принадлежности ключа словарю осуществляется с помощью операции key in dictionary (точно также, как проверка принадлежности элемента множеству).

Словарь является iterable и возвращает ключи в случайном порядке. Например, такой код напечатает содержимое словаря:

```
phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
for service in phones:
    print(service, phones[service])
```

Также существует метод items, который возвращает iterable, содержащий в себе кортежи ключ-значение для всевозможных ключей.

```
phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
for service, phone in phones.items():
    print(service, phone)
```

7.6 Когда нужно использовать словари

1. По прямому назначению: сопоставление ключа значению (названия дней недели, переводы слов и т.д.).

2. Для подсчета числа объектов. При очередной встрече объекта счетчик увеличивается на единицу. Это похоже на сортировку подсчётом.

3. Для хранения разреженных массивов. Например, если мы хотим хранить цену 92, 95 и 98 бензина, то могли бы создать массив из 99 элементов и хранить в нём. Но большая часть элементов не нужны — массив разреженный. Словарь здесь подходит больше.

Для подсчета числа элементов удобно использовать метод get. Он принимает два параметра: ключ для которого нужно вернуть значения и значение, которое будет возвращено, если такого ключа нет. Например, подсчитать сколько раз входит в последовательность каждое из чисел можно с помощью такого кода:

```
seq = map(int, input().split())
countDict = {}
for elem in seq:
    countDict[elem] = countDict.get(elem, 0) + 1
for key in sorted(countDict):
    print(key, countDict[key], sep = ' : ')
```

7.7 Полезные методы строк

It's dangerous to go alone! Take this.

isalpha — проверяет, что все символы строки являются буквами.

isdigit — проверяет, что все символы строки являются цифрами.

isalnum — проверяет, что все символы строки являются буквами или цифрами.

islower — проверяет, что все символы строки являеются маленькими (строчными) буквами.

isupper — проверяет, что все символы строки являеются большими (заглавными, прописными) буквами.

lstrip — обрезает все пробельные символы в начале строки.

rstrip — обрезает все пробельные символы в конце строки.

strip — обрезает все пробельные символы в начале и конце строки.

7.8 Пример решения сложной задачи на словари

Рассмотрим такую задачу: словарь задан в виде набора строк, в каждой строке записано слово на английском языке, затем следует символ -, затем, через запятую, перечислены возможные переводы слова на латынь.

Требуется составить латино-английский словарь и вывести его в том же виде. Все слова должны быть упорядочены по алфавиту. Возможные переводы одного слова должны быть также упорядочены по алфавиту.

Например, для ввода:

```
3
apple - malum, pomum, popula
fruit - baca, bacca, popum
punishment - malum, multa
```

Вывод должен выглядеть так:

```
7
baca - fruit
bacca - fruit
malum - apple, punishment
multa - punishment
pomum - apple
popula - apple
popum - fruit
```

Идея решения заключается в следующем: разрежем каждую строку на английское и латинские слова. Каждое из латинских слов возьмем в качестве ключа и добавим к его значениям английское слово (переводов может быть несколько). Затем пройдем по сортированным ключам и для каждого ключа выведем отсортированный список переводов:

```
n = int(input())
latinEnglish = {}
for i in range(n):
    line = input()
    english = line[:line.find('-')].strip()
    latinsStr = line[line.find('-') + 1:].strip()
    latins = map(lambda s : s.strip(), latinsStr.split(','))
    for latin in latins:
        if latin not in latinEnglish:
            latinEnglish[latin] = []
        latinEnglish[latin].append(english)
print(len(latinEnglish))
for latin in sorted(latinEnglish):
    print(latin, '-', ', '.join(sorted(latinEnglish[latin])))
```

Лекция 8

Элементы функционального программирования

8.1 Парадигмы программирования

Языки программирования предлагают различные средства для декомпозиции задачи. Существует несколько парадигм программирования:

- Императивное (структурное, процедурное) программирование: программы являются последовательностью инструкций, которые могут читать и записывать данные из памяти. При изучении предыдущих тем мы пользовались, в основном, императивной парадигмой. Ряд языков, такие как Паскаль (не Object Pascal) или C являются яркими предтавителями императивных языков.

- Декларативное программирование: описывается задача и ожидаемый результат, но не описываются пути её решения. Ярким представителем является язык запросов к базам данных SQL: большая часть внутреннего устройства скрыта в СУБД, программист описывает только структуру базы данных и ожидаемый результат запросов.

- Объектно-ориентированное программирование: программы манипулируют наборами объектов, при этом объекты обладают сохраняющимся во времени состоянием и методами для изменения этого состояния (или создания новых объектов). Мы познакомимся с ООП подробнее на одной из следующих лекций. Примером языка с объектно-ориентированной парадигмой является Java, ООП также поддерживается в Питоне и C++.

- Функциональное программирование: задача разбивается на набор функций. В идеале, функции только принимают параметры и возвращают значения, не изменяя состояния объектов или программы. Представителем функциональных языков является Haskell.

Иногда, также, выделяют и другие парадигмы программирования.

Некоторые языки предназначены, в основном, для написания программ в рамках одной парадигмы, другие же поддерживают несколько парадигм. Например, Питон и C++ поддерживают различные парадигмы программирования. Разные части программы можно писать в разных парадигмах, например, использовать функциональный

стиль для обработки больших данных, объектно-ориентированный подход для реализации интерфейса и императивное программирование для промежуточной логики.

8.2 Функциональное программирование

Несмотря на то, что в функциональном стиле писать достаточно сложно и непривычно, функциональное программирование имеет массу плюсов:

- Достаточно легко доказать формальную корректность алгоритмов. Хотя доказательство, даже для функциональных программ, часто намного длиннее, чем сама программа, но для фундаментальных алгоритмов, которые широко используются, лучше иметь формальное доказательство, чтобы не попадать в глупые ситуации. Строить формальные доказательства императивных программ намного сложнее.

- Для программ, написанных в функциональном стиле, легко проводить декомпозицию, отладку и тестирование. Когда вся программа разбита на функции, выполняющие элементарные действия, то их разработка и проверка занимает намного меньше времени.

- Для функциональных программ легко автоматически проводить распараллеливание, векторизацию и конвейеризацию.

Задача распараллеливания выполнения программ крайне актуальна сейчас, когда скорости ядер процессоров практически перестали расти. Единственным способом ускорить выполнение программ является их параллельное вычисление на нескольких устройствах.

Последние успехи в решении задач машинного обучения связаны с использованием большого количества простых вычислительных устройств, например, рассчетов на видеокарте.

В функциональных программах не принято вносить изменения в объекты (и, вообще говоря, желательно, чтобы все объекты были неизменямыми). Поэтому, если нам нужно посчитать результат вычисления двух функций, то во многих случаях можно делать это параллельно на разных ядрах процессора. Такое распараллеливание легко сделать автоматически.

Векторизация — это когда одни и те же действия выполняются над большим набором данных. Тогда данные можно нарезать на куски и раскидать по разным вычислительным устройствам, а затем собрать результат вычислений в один объект.

Конвейеризация — это разбиение вычислений на несколько этапов, причём данные поступают на следующий этап обработки по времени готовности. Например, если нам нужно попарно перемножить значения в векторах A и B, а затем сложить со значениями в векторе C, то мы можем посчитать несколько первых результатов подсчета произведения и уже выполнять с ними сложение, не дожидаясь подсчета остальных значений. Это позволяет быстрее получить первые результаты и занять еще большое количество вычислительных устройств параллельно.

8.3 Итераторы и генераторы

В Питоне итераторы — это объекты, которые имеют внутреннее состояние и метод `__next__` для перехода к следующему состоянию. Например, можно сконструировать итератор от списка и перебрать все значения:

```
myList = [1, 2, 3]
for i in iter(myList):
    print(i)
for i in myList:
    print(i)
```

В этой программе два цикла эквивалентны.

В Питоне можно создавать и свои итераторы. Их разумно использовать в том случае, если нужно перебрать большое количество значений и существует правило, по которому можно получить следующее значение, однако хранений всех этих значений не имеет смысла, т.к. они пригодятся только один раз.

Для создания итераторов в Питоне используется специальный вид функций, называемых генераторами. В обычной функции return прекращает работу функции. В генераторе вместо return используется оператор yield, который также возвращает значение, но не прекращает выполнение функции, а приостанавливает его до тех пор, пока не потребуется следующее значение итератора. При этом работа функции продолжится с того места и в том состоянии, в котором она находилась на момент вызова yield. Посмотрим, как может быть реализован генератор, аналогичный стандартному range с одним параметром:

```
def myRange(n):
    i = 0
    while i < n:
        yield i
        i += 1
for i in myRange(10):
    print(i)
```

Генераторы могут иметь и сложную рекурсивную структуру. Например, мы можем написать генератор, который будет выдавать все числа заданной длины, цифры в которых не убывают и старшая цифра не превосходит заданного параметра:

```
def genDecDigs(cntDigits, maxDigit):
    if cntDigits > 0:
        for nowDigit in range(maxDigit + 1):
            for tail in genDecDigs(cntDigits - 1, nowDigit):
                yield nowDigit * 10**(cntDigits - 1) + tail
    else:
        yield 0

print(*genDecDigs(2, 3))
```

Вывод этой программы будет выглядеть так: 0 10 11 20 21 22 30 31 32 33

В этой программе рекурсивный генератор перебирал все допустимые цифры в качестве той, которая должна стоять на заданной позиции и генерировал все возможные

последующие цифры.

Также в этой программе мы использовали одну особенность функции print: если переди именем iterable (а результат, возвращаемый генератором является iterable) поставить *, то будут напечатны все значения через пробел.

Результат работы генератора можно сохранить, например, в список с помощью функции list, как мы уже делали это с результатом работы range.

8.4 Встроенные функции для работы с последовательностями

В языке Питон есть много функций, которые принимают в качестве параметра iterable и могут сделать что-то полезное. С некоторыми из них, такими как sorted или map мы уже немного знакомы. Рассмотрим еще некоторые из них:

sum — находит сумму всех элементов iterable.

min, max — находит минимум и максимум в последовательности iterable.

map — умеет принимать более двух параметров. Например, такая запись map(f, iterA, iterB) вернет iterable со значениями f(iterA[0], iterB[0]), f(iterA[1], iterB[2]), ...

filter(predicate, iterable) — применяет функцию predicate ко всем элементам iterable и возврщает iterable, который содержит только те элементы, которые удовлетворяли предикаты (т.е. функция predicate вернула True). Например, так может выглядеть решение задачи о поиске минимального положительного элемента в списке:

```
print(min(filter(lambda x: x > 0, map(int, input().split()))))
```

Здесь в качестве предиката использована лямбда-функция, которая возвращает True при значении параметра больше 0, а в качестве входного iterable — результат вызова map для функции int и нарезанного на слова ввода. Функция max применена к тому iterable, который был возвращен функцией filter.

enumerate — возвращает кортежи из номера элемента (при нумерации с нуля) и номера очередного элемента. С помощью enumerate, например, удобно перебирать элементы iterable (доступ по индексу в которых невозможен) и выводить номера элементов, которые обладают некоторым свойством:

```
f = open('data.txt', 'r', encoding='utf8')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line', i)
```

any, all — возвращают истину, если хотя бы один или все элементы iterable истинны соответствнно. Например, так можно проверить, не превышают ли все члены последовательности 100 по модулю:

```
print(all(map(lambda x: abs(int(x)) <= 100, input().split())))
```

zip(iterA, iterB, ...) — конструирует кортежи из элементов (iterA[0], iterB[0], ...), (iterA[1], iterB[1], ...), ...

С помощью этих функция можно решить достаточно сложные задачи, без использования циклов и условных операторов. Например, задачу из домашнего задания по сортировкам про такси: в первой строке задано количество людей и автомобилей такси,

в следующих двух строках расстояние в километрах для каждого человека и цена за километр для каждого такси. Необходимо сопоставить каждому человеку номер такси, чтобы суммарная цена поездок была минимальна. Идея решения заключается в том, чтобы люди, которым ехать дальше ехали на более дешевых такси:

```
n = int(input())
peopleList = map(int, input().split())
peoples = sorted(list(enumerate(peopleList), key=lambda x: x[1]))
taxiList = map(int, input().split())
taxis = sorted(list(enumerate(taxiList)), key=lambda x: x[1], reverse=True)
ans = sorted(zip(peoples, taxis), key=lambda x: x[0][0])
print(*map(lambda x: x[1][0] + 1, ans))
```

8.5 Генерация комбинаторных объектов itertools

В Питоне есть библиотека itertools, которая содержит много функций для работы с итераторыми. С этими функциями можно ознакомиться в официальной документации к языку.

Нам наиболее интересны функции, генерирующие комбинаторные объекты.

itertools.combinations(iterable, size) — генерирует все подмножества множества iterable размером size в виде кортежей. Это может быть использовано вместо вложенных циклов при организации перебора. Например, мы можем неэффективно решить задачу о поиске трех чисел в последовательности, дающих наибольшее произведение:

```
from itertools import combinations

nums = list(map(int, input().split()))
combs = combinations(range(len(nums)), 3)
print(max(map(lambda x: nums[x[0]] * nums[x[1]] * nums[x[2]], combs)))
```

itertools.permutations(iterable) — генерирует все перестановки iterable. Существует вариант функции с двумя параметрами, второй параметр является размером подмножества. Тогда генерируются все перестановки всех подмножеств заданного размера.

itertools.combinations_with_replacement(iterable, size) — генерирует все подмножества iterable размером size с повторениями, т.е. одно и то же число можно входить в подмножество несколько раз.

8.6 partial, reduce, accumulate

Модуль functools содержит некоторые функции, которые могут полезны для обработки последовательностей и не только.

Функция functools.partial предназначена для оберачивания существующих функций с подстановкой некоторых параметров. Например, мы можем создать функцию для печати в файл, чтобы каждый раз не указывать какие-то параметры. Например, существует вариант функции int с двумя параметрами: первый — это переменная, которую необходимо преобразовать в число, а второй — система счисления в которой записано число. С помощью partial мы можем создать функцию-обёртку, преобразующую строки из 0 и 1 в числа:

```
from functools import partial

binStrToInt = partial(int, base=2)
print(binStrToInt('10010'))
```

В модуле functools также содержатся функции для обработки последовательностей.

functools.reduce(func, iterable) позволяет применить функцию ко всем элементам последовательности, используя в качестве первого аргумента накопленный результат. Например, если в последовательности были элементы myList = [A, B, C], то результатом применения reduce(f, myList) будет f(f(A, B), C). С помощью reduce, например, можно найти НОД всех чисел в iterable:

```
from functools import reduce

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print(reduce(gcd, map(int, input().split())))
```

itertools.accumulate(func, iterable), которая возвращает iterable со всеми промежуточными значениями, т.е. для списка [A, B, C] accumulate вернет значения A, f(A, B), f(f(A, B), C). Например, можно узнать максимальный элемент для каждого префикса (некоторого количества первых элементов) заданной последовательности:

```
from itertools import accumulate

print(*accumulate(map(int, input().split()), max))
```

Лекция 9

Объектно-ориентированное программирование

9.1 Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) является одной из парадигм программирования, созданной на базе императивного программирования для более удобного повторного использования кода и облегчения читаемости программ.

ООП не является «серебряной пулей», которая решает все задачи наиболее удобным образом. Императивное программирование удобно для решения простых задач, использующих стандартные объекты. Повторное использование кода в императивной парадигме обеспечивается с помощью циклов и функций, написанных в императивном стиле.

Функциональное программирование удобно для задач, где существует ярко выраженный поток данных (data flow), который «перетекает» из одной функции в другую, изменяясь по пути.

ООП же позволяет обеспечить повторное использование кода за счет того, что обрабатываемые программой объекты имеют много общего и лишь незначительные отличия в своем поведении.

ООП основано на трёх концепциях: инкапсуляция, наследование, полиморфизм.

Инкапсуляция — это помещение в «капсулу»: логическое объединение данных и функций для работы с ними, а также сокрытие внутреннего устройства объекта с предоставлением интерфейса взаимодействия с ним (публичные методы).

Наследование — это получение нового типа объектов на основе уже существующего с частично или полностью заимствованный у родительского типа функциональностью.

Полиморфизм — это предоставление одинаковых средств взаимодействия с объектами разной природы. Например, операция + может работать как с числами, так и со строками, несмотря на разную природу этих объектов.

Классом в Питоне называется описание структуры объекта (полей структуры) и методов (функций) для работы с данными в этой структуре.

Объектом называется экземпляр класса, где поля заполнены конкретными значениями. Объекты находятся в памяти программы и могут изменять своё состояние или выполнять какие-то действия с помощью вызова методов класса для этого объекта.

9.2 Инкапсуляция и конструкторы

Мы уже использовали ключевое слово class для создания структур — набора именованных полей, совокупность которых описывает объект. Однако, мы пользовались для их обработки отдельно лежащими функциями или кусками кода.

Было бы намного удобнее, если бы описание структуры объекта и методов работы с ним лежало рядом, для удобства изучения, модификации и использования.

Мы будем рассматривать элементы ООП на примере комплексных (ударение на «е») чисел. Это забавный математический объект, который состоит из действительной (real) и мнимой (imaginary) части. Запись это числа выглядит как $re + im \times i$, где i это $\sqrt{-1}$. Глубокое математическое понимание комплексных чисел нам не понадобится: достаточно понимать, что это структура с двумя полями re и im, где оба эти поля — вещественные числа.

Для создания новых объектов класса используется специальный метод, который называется «конструктор». Методы класса записываются внутри описания класса как функции, конструктор должен называться `__init__`. В качестве первого параметра они должны принимать переменную self — конкретный объект класса, с которым они работают.

Рассмотрим класс для вещественного числа, вызов конструктора и печать полей объекта:

```
class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im

a = Complex(1, 2)
b = Complex(3)
c = Complex()
print(a.re, a.im)
print(b.re, b.im)
print(c.re, c.im)
```

Здесь конструктор содержит три параметра: self — пустой объект класса Complex, re и im по умолчанию равные нулю. Вызов конструктора осуществляется с помощью написания названия класса, в скобках указываются параметры конструктора (все, кроме self). Названия классов принято записывать с большой буквы, а объекты — с маленькой.

Вывод этой программы будет:

```
1 2
3 0
0 0
```

Обратите внимание, что мы меняем переменные конкретного объекта класса, который передан в качестве параметра self. Если мы создали какие-то переменные в описании класса, то их значения были бы доступны во всех объектах этого класса и их можно было бы даже изменить, перечислив их имена в начале метода после выражения nonlocal. Такие переменные называются статическими, обычно они предназначены для хранения каких-то констант (что очень удобно если вы, например, описываете какой-то класс для физических вычислений). Их изменение может понадобится в экзотических

ситуациях, например, при подсчете количества объектов класса. Мы не будем заострять на них внимание.

9.3 Определение методов и стандартные функции

Некоторые стандартные функции языка Питон являются всего-лишь обертками над вызовом метода для передаваемого параметра. Например, функция str вызывает метод `__str__` для своего параметра. Если мы опишем такой метод для нашего класса, то можно будет применять к нему функцию str явно и неявно (например, она автоматически вызовется при вызове print для объекта нашего класса).

Мы бы хотели, чтобы `__str__` возвращал текстовое представление нашего комплексного числа. Например, число с действительной частью 1 и мнимой 2 должно быть представлено в виде строки "1+2i", а число с действительной частью 3 и мнимой -4.5 — как "3-4.5i". Полное описание класса с добавленным методом будет выглядеть так:

```
class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep

a = Complex(1, 2)
print(a)
b = Complex(3, -4.5)
print(b)
```

9.4 Переопределение операторов

В языке Питон можно переопределить и поведение операторов. Например, если у нас есть два числа x и y, то запись x + y реально преобразуется в вызов метода `x.__add__(y)`. Значок операции + является всего лишь удобным для человека переопределением вызова метода add.

Для вещественных чисел логично определена операция сложения: это сложение отдельно действительных и отдельно мнимых частей. В результате вызова метода для сложения двух чисел должен конструироваться новый объект класса Complex, а переданные в качестве параметров объекты не должны изменяться. Действительно, когда мы выполняем операция z = x + y для обычных чисел, то ожидаем, что сконструируется новый объект, к которому привяжется ссылка z, а x и y останутся без изменения.

Будем придерживаться этой же логики при реализации метода для сложения двух комплексных чисел. Наш метод `_add` должен принимать два параметра, каждый из которых является комплексным числом.

```
class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)

a = Complex(1, 2)
b = Complex(3, -4.5)
print(a + b)
```

Переопределять метод add имеет смысл только в тех ситуациях, когда программисту, использующему ваш класс, будет очевиден смысл операции +. Например, если бы вы создали класс для описание некоторых характеристик человека, то операция + для двух объектов-людей воспринималась разными пользователями вашего класса совершенно по-разному, в зависимости от развитости фантазии читателя. Такого неоднозначного понимания лучше избегать и вовсе не переопределять операцию +, если результат её работы не очевиден.

9.5 Проверка класса объекта

Переопределим для комплексных чисел ещё одну операцию — умножение. При этом мы хотим уметь умножать комплексные числа как на целые или действительные, так и на другие комплексные числа.

При умножении комплексного числа вида $a+b\times i$ на целое или вещественное число x результатом будет комплексное число $a\times x+b\times x\times b$.

Переменожение двух комплексных чисел производится аналогично умножению двух многочленов первой степени:

$$(a+b\times i)\times(c+d\times i)=a\times c+(a\times d+b\times c)\times i+b\times d\times i^2$$

Мы знаем, что $i^2=-1$. Значит окончательно результат умножения будет выглядеть так:

$$a\times c-b\times d+(a\times d+b\times c)\times i$$

Нам осталось понять, как определить, передано ли в наш метод комплексное или не комплексное число.

В языке Питон существует функция isinstance, которая в качестве первого параметра принимает объект, а в качестве второго — название класса. Она возвращает истину,

если объект относится к данному классу и ложь в противном случае. Эта функция позволит нам добиться нужной функциональности от метода `__mul__`, умножающего числа.

```
class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.re * other.im
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        return Complex(newRe, newIm)
    __rmul__ = __mul__

a = Complex(1, 2)
b = Complex(3, -4.5)
print(a * b)
print(a * 2)
```

Кроме добавленного метода `__mul__` внимания также заслуживает строка `__rmul__ = __mul__`. Это присваивание одного метода (функции) другому, т.е. при вызове метода `__rmul__` будет вызываться тот же самый метод `__mul__`.

В языке питон операция `a * b` заменяется на вызов метода `a.__mul__(b)`. Если a было комплексным числом, a b — вещественным, то вызовется метод `__mul__` для объекта a нашего класса Complex.

Однако, если a было вещественным числом, a b — комлексным, то произойдет попытка вызвать метод `__mul__` для объекта класса float. Естественно, разработчики стандартной библиотеки языка Питон не могли предположить, что вы когда-нибудь напишите класс Complex и будете пытаться умножить на него вещественное число, поэтому метод `__mul__`, где в качестве параметра передается нечто неизвестное, будет заканчивать свою работу с ошибкой. Чтобы избежать таких ситуаций в языке Питон после неудачной попытки совершить `a.__mul__(b)` просходит попытка совершить действие `b.__rmul__(a)` и в нашем случае она заканчивается успехом.

9.6 Обработка ошибок

Время от времени в программах возникают ошибочные ситуации, которые не могут быть обработаны в том месте, где возникла ошибка, а должны быть обработаны тем или иным образом в более внешней части программы.

Например, если на этапе выполнения промежуточной логики обнаружилось, что вы пытаетесь записать строку в то, что должно быть числом, то вы ничего не можете с этим сделать. В таком случае нужно вываливаться из стека вызовов функций или методов промежуточной логики до тех пор, пока мы не дойдем до фронтенда, который сообщит пользователю о том, что он ввел недопустимое значение и попросит, например, ввести его заново.

Наверняка вы сталкивались с ситуацией заполнения огромной формы, попытка отправить которую приводила к тому, что появлялось окошко со словом «ошибка» без каких-либо уточнений. Это плохой стиль, сообщение об ошибке должно быть информативным, чтобы позволить быстро её найти и исправить. Таким образом, при создании ошибки нужно передавать исчерпывающую инфорамацию о ней.

В нашем примере с вещественными числами мы можем рассмотреть такой пример ошибки: умножение комплексного числа на что-то, отличное от целого, вещественного или комплексного числа. Когда мы дошли до этапа умножения — мы уже ничего не можем предпринять для исправления этой ошибки, кроме как просигнализировать о ней в то место, откуда была вызвана наша операция умножения.

При этом на этапе вызова операции умножения мы уже можем предпринять какие-то действия. Например, сообщить пользователю о том, какую фигню он ввел и попросить ввести все-таки комплексное число. Или, если мы обрабатываем последовательность, из которой нужно вычленить и перемножить комплексные числа — просто перейти к следующему элементу последовательности. На этапе когда мы дошли до неудачного выполнения операция умножения мы не знаем и не можем знать как должна себя вести конкретная программа при возникновении такой ошибки.

Когда мы дойдем до ошибочной операции мы можем сконструировать специальный класс, содержащий подробное описание ошибки и выбросить его в то место, которое способно его обработать.

Выбрасывается ошибка с помощью команды raise, а ловится блоком try-except. Для случая умножения комплексного числа на мусор мы можем сконструировать класс ошибки, содержащий в себе ссылку как на комплексное число, так и на второй аргумент метода умножения.

Класс для ошибки должен быть наследником стандартного класса BaseError. Пока для нас это значит только то, что при создании описания класса ошибки мы должны написать в скобках после его названия BaseError. Потенциально ошибочные действия должны выполняться в блоке try, а команды для обработки ошибки должны быть в блоке except. Пример с обработкой ошибки будет выглядеть так:

```
class ComplexError(BaseException):
    def __init__(self, Complex, other):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __init__(self, re=0, im=0):
```

```
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.re * other.im
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        else:
            raise ComplexError(self, other)
        return Complex(newRe, newIm)
    __rmul__ = __mul__

a = Complex(1, 2)
try:
    res = a * 'abcd'
except ComplexError as ce:
    print('Error in mul with args:', ce.arg1, ce.arg2)
```

Вывод этой программы будет:

Error in mul with args: 1+2i abcd

По нему легко понять, что ошибка возникает при операции умножения и увидеть, что было передано в неё в качестве аргументов.

После команды except мы можем указать имя класса ошибки, который он должен обрабатывать, затем написать "as" и указать имя переменной в которую попадет объект с описанием конкретной ошибки.

Блоков except может быть несколько для обработки ошибок разных типов. Првоерки выполняются последовательно, будет выполнен тот блок команд, у которого имя класса совпадает с именем класса ошибки или является его предком в дереве иерархии наследования.

9.7 Наследование и полиморфизм

Операции сложения и умножения на вещественное или целое число для комплексных чисел очень похожи на поведение свободных векторов на плоскости. Они соответствуют сложению векторов или умножение вектора на число, где действительная часть

комплексного числа является x-координатой вектора, а мнимая — y-координатой.

Кроме того, свободный вектор обладает некоторыми операциями, которые характерны только для него, но не для вещественного числа. Например, это может быть метод length, вычисляющий длину вектора. Нам не хотелось бы засорять код для описания комплексного числа методами для работы со свободным вектором, но с другой стороны не хотелось бы заново переписывать методы сложения и умножения для свободных векторов.

В такой ситуации разумно создать новый класс для описания свободного вектора (или точки на плоскости, что то же самое), который унаследовал бы все методы комплексных чисел и добавил бы новый метод length.

Мы уже знаем, что для того, чтобы пронаследовать класс от другого достаточно в описании класса указать в круглых скобках, от кого он наследуется. Таким образом, мы можем записать наше описание класса Point с определенным методом length так:

```
class ComplexError(BaseException):
    def __init__(self, Complex, other):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.re * other.im
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        else:
            raise ComplexError(self, other)
        return Complex(newRe, newIm)
    __rmul__ = __mul__

class Point(Complex):
    def length(self):
```

```
        return (self.re**2 + self.im**2)**(1/2)

a = Point(3, 4)
b = Complex(1, 2)
print(a.length())
c = a + b
print(c)
```

Вывод этой программы будет
5.0
4+6i

Здесь мы не только убедились в том, что свежесозданный метод работает, но и сделали довольно странную вещь: сложили точку на плоскости с комплексным числом. Дело в том, что объекта класса Point также одновременно является и объектом типа Complex, т.к. Point пронаследован от Complex. Point лишь расширяет и дополняет Complex, а значит Point может смело быть интерпретирован как Complex, но не наоборот.

В этом примере будут истинны выражения isinstance(a, Point) и isinstance(a, Complex), но будет ложно выражение isinstance(b, Point).

9.8 Переопределение методов

Если мы попытаемся напечатать объект типа Point, то он напечатается как комплексное число. Нам хотелось бы, чтобы точки на плоскости печатались в виде (x, y), а не x+yi.

В языке Питон любой метод можно переопределить в наследнике, что мы и сделаем для метода `__str__` для класса Point:

```
class ComplexError(BaseException):
    def __init__(self, Complex, other):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
```

```
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.re * other.im
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        else:
            raise ComplexError(self, other)
        return Complex(newRe, newIm)
    __rmul__ = __mul__

class Point(Complex):
    def length(self):
        return (self.re**2 + self.im**2)**(1/2)
    def __str__(self):
        return str((self.re, self.im))

a = Point(3, 4)
print(a)
```

Наш метод обязан возвращать строку, но мы можем воспользоваться функцией str от кортежа, которая выдаст нам нужный результат.

9.9 Проектирование структуры классов

Структура описания классов представляет собой дерево (на самом деле в Питоне — ациклический граф), пронаследованный от единого корня — базового пустого класса.

С помощью грамотно спроектированной структуры классов можно добиться легкой читаемости и максимального повторного использования кода, обеспечения единого интерфейса и множество других радостей.

Однако, внесение фичи или изменение на высоком уровне иерархии классов может привести к необходимости выполнить огромное количество работы по модификации всех потомков этого класса, а также к полной несовместимости с предыдущей версией. Многочисленные изменения такого рода приводят к уродливым конструкциям, которые невозможно понимать и отлаживать.

В то же время, закладывание перспективных фичей в структуру классов ведет к переусложнению и сводит на нет все повторное использование кода за счет громоздкости конструкций. Кроме того, перспективные фичи могут быть недостаточно обдуманы и приведут к еще большему уродству, когда дело дойдет до их реальной реализации (если дойдет).

Таким образом, грамотное проектирование системы классов требует не только хорошего знания паттернов проектирования, но и большого практического опыта. На начальном этапе стоит обучаться проектированию систем, в которые не планируется внесение изменений.