**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:

|  |  |
| --- | --- |
| 1 | myList = list(map(int, input().split())) |
| 2 | myList.sort() |
| 3 | 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:

|  |  |
| --- | --- |
| 1 | myList = list(map(int, input().split())) |
| 2 | sortedList = sorted(myList) |
| 3 | 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 reverse. Например, это будет выглядеть как myList.sort(reverse=True) или sorted(myList, reverse=True).

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

|  |  |
| --- | --- |
| 1 | print(sorted((1, 3, 2))) |
| 2 | print(sorted(range(10, -1, -2))) |
| 3 | 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.

**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.

**Именованный параметр key**

Пусть 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.

|  |  |
| --- | --- |
| 1 | n = int(input()) |
| 2 | peopleList = [] |
| 3 | for i in range(n): |
| 4 | tempManData = input().split() |
| 5 | manData = (int(tempManData[0]), tempManData[1]) |
| 6 | peopleList.append(manData) |
| 7 | peopleList.sort() |
| 8 | for manData in peopleList: |
| 9 | 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.

|  |  |
| --- | --- |
| 1 | n = int(input()) |
| 2 | peopleList = [] |
| 3 | for i in range(n): |
| 4 | tempManData = input().split() |
| 5 | manData = (-int(tempManData[0]), tempManData[1]) |
| 6 | peopleList.append(manData) |
| 7 | peopleList.sort() |
| 8 | for badManData in peopleList: |
| 9 | manData = (-badManData[0], badManData[1]) |
| 10 | print(' '.join(map(str, manData))) |

Этот код малопонятен и плох.

**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:

|  |  |
| --- | --- |
| 1 | n = int(input()) |
| 2 | strings = [] |
| 3 | for i in range(n): |
| 4 | strings.append(input()) |
| 5 | 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):

|  |  |
| --- | --- |
| 1 | def dist(point): |
| 2 | return point[0] \*\* 2 + point[1] \*\* 2 |
| 3 |  |
| 4 | n = int(input()) |
| 5 | points = [] |
| 6 | for i in range(n): |
| 7 | point = tuple(map(int, input().split())) |
| 8 | points.append(point) |
| 9 | points.sort(key=dist) |
| 10 | for point in points: |
| 11 | 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.

Последнее изменение: Thursday, 20 September 2018, 22:04

**"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:

|  |  |
| --- | --- |
| 1 | class Man: |
| 2 | height = 0 |
| 3 | name = '' |
| 4 |  |
| 5 | def manKey(man): |
| 6 | return (-man.height, man.name) |
| 7 |  |
| 8 | n = int(input()) |
| 9 | peopleList = [] |
| 10 | for i in range(n): |
| 11 | tempManData = input().split() |
| 12 | man = Man() |
| 13 | man.height = int(tempManData[0]) |
| 14 | man.name = tempManData[1] |
| 15 | peopleList.append(man) |
| 16 | peopleList.sort(key=manKey) |
| 17 | for man in peopleList: |
| 18 | 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.

**Лямбда-функции**

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

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:

|  |  |
| --- | --- |
| 1 | lambda x: x\*\*2 |

Что эквивалентно привычной записи функции:

|  |  |
| --- | --- |
| 1 | def sqr(x): |
| 2 | 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:

|  |  |
| --- | --- |
| 1 | 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:

|  |  |
| --- | --- |
| 1 | n = int(input()) |
| 2 | points = [] |
| 3 | for i in range(n): |
| 4 | point = tuple(map(int, input().split())) |
| 5 | points.append(point) |
| 6 | points.sort(key=lambda point: point[0]\*\*2 + point[1]\*\*2) |
| 7 | for point in points: |
| 8 | 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 обычно применяется один параметр.

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

|  |  |
| --- | --- |
| 1 | def traditionalSqr(x): |
| 2 | return x\*\*2 |
| 3 |  |
| 4 | lambdaSqr = lambda x: x\*\*2 |
| 5 | print(traditionalSqr(3)) |
| 6 | 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.

**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:

|  |  |
| --- | --- |
| 1 | def printList(myList, sep=' '): |
| 2 | print(sep.join(map(str, myList))) |
| 3 |  |
| 4 | printList([1, 2, 3]) |
| 5 | 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:

|  |  |
| --- | --- |
| 1 | def mySum(\*args): |
| 2 | nowSum = 0 |
| 3 | for now in args: |
| 4 | nowSum += now |
| 5 | return nowSum |
| 6 |  |
| 7 | print(mySum(1, 2)) |
| 8 | 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:

|  |  |
| --- | --- |
| 1 | def myMin(first, \*others): |
| 2 | nowMin = first |
| 3 | for now in others: |
| 4 | if now < nowMin: |
| 5 | nowMin = now |
| 6 | return nowMin |
| 7 |  |
| 8 | print(myMin(1)) |
| 9 | print(myMin(3, 1, 2)) |

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

**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.

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

|  |  |
| --- | --- |
| 1 | inFile = open('input.txt', 'r', encoding='utf8') |
| 2 | outFile = open('output.txt', 'w', encoding='utf8') |
| 3 | lines = inFile.readlines() |
| 4 | for line in lines: |
| 5 | print(line[-2::-1], file=outFile) |
| 6 | inFile.close() |
| 7 | 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 (это особенно актуально для больших файлов). Решение без запоминания всего файла можно было реализовать так:

|  |  |
| --- | --- |
| 1 | inFile = open('input.txt', 'r', encoding='utf8') |
| 2 | outFile = open('output.txt', 'w', encoding='utf8') |
| 3 | for line in inFile: |
| 4 | print(line[-2::-1], file=outFile) |
| 5 | inFile.close() |
| 6 | 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 \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 (например, для перебора строк консоли можно написать for line in sys.stdin). Ввести признак конца файла в консоли можно, нажав Ctrl+Z в Windows или Ctrl+D в Unix-системах. В среде программирования такой способ может не работать. Более подробно в этой теме: <https://www.coursera.org/learn/python-osnovy-programmirovaniya/discussions/all/threads/DLrJXKbWEeiD2QqOJunOiA/replies/fJUPEK4gEeix7A7XXq2u1g>.

Еще об одном способе работы с файлами с помощью конструкции with .. as .., что позволяет автоматически вызывать метод close(), можно почитать здесь: <http://book.pythontips.com/en/latest/open_function.html>.

**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:

|  |  |
| --- | --- |
| 1 | marks = map(int, input().split()) |
| 2 | cntMarks = [0] \* 11 |
| 3 | for mark in marks: |
| 4 | cntMarks[mark] += 1 |
| 5 | for nowMark in range(11): |
| 6 | 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.

**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 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.