I'm not robot
reCAPTCHA
Continue

Python sort list of tuples descending

On this blog I will explain how to sort tuples, especially how they sort out a specific element in tuple in Python 3.4. 1. Simple sorting let's start by creating a list of tuples and then sorting them in ascending order. my_list = [('John', 40), ('Dana', 35), ('Betty', 10), ('Robby', 8), ('John', 20)] my_list.sort() print(simple sort) print(my_list) the output is [('Betty', 10), ('Dana', 35), ('John', 20), ('John', 40), ('Robby', 8)] As you can see the tuples are first sorted by their first element, e.g the tuple ('Betty', 10) appears before the tuple ('Dana', 35). If the two tutels have the same first element, then they are sorted according to their second element, such as ('John', 20) before it appears ('John', 40). 2. Sort the first element in the tuple let's sort the tuples by your first element only. import operator my_list = [('John', 40), ('Dana', 35), ('Betty', 10), ('Robby', 8), ('John', 20)] my_list.sort(key = operator.itemgetter(0)) print(my_list) The output is [('Betty', 10), ('Dana', 35), ('John', 40), ('John', 20), ('Robby', 8)] You can see how the second element of the tuple is ignored, e.g. ('John', 40) appears before ('John', 20). Note that we have to import the module operator. 3. Sort the second element in the tuple let's sort the tuples by your second element. import operator my_list = [('John', 40), ('Dana', 35), ('Betty', 10), ('Robby', 8), ('John', 20)] my_list.sort(key = operator.itemgetter(1)) print(my_list) The output is [('Robby', 8), ('Betty', 10), ('John', 20), ('Dana', 35), ('John', 40)] Note: As usual in programming languages we have 0-indexing here, so to sort by the first element of a tuple we use operator.itemgetter(0), to sort by the second element we use operator.itemgetter(1). 4. Sort the second element in the tuple in descending order if we want to sort in descending order we simply set the reverse = correct. import operator my_list = [('John', 40), ('Dana', 35), ('Betty', 10), ('Robby', 8), ('John', 20)] my_list.sort(key = operator.itemgetter(1), reverse = True) print(my_list) The output is [('John', 40), ('Dana', 35), ('John', 20), ('Betty', 10), ('Robby', 8)] 5. Sort by multiple elements we can also specify several elements of tuple that should be used for sorting. Let's sort by the first and third elements. import operator my_list = [['Eva', 20, 0.7), ('Eva', 8, 0.1), ('Bob', 12, 0.9), ('Bob', 24, 0.3)] my_list.sort(key = operator.itemgetter(0, 2)) Print (my_list) is the output [('Bob', 24, 0.3), ('Bob', 12, 0.9), ('Eva', 8, 0.1), ('Eva', 20, 0.7)] appears in the list of arranged Bobs before Evas. Then among the bobs we sort by the third element. Likewise among the Oahs we sort by third element. We see that the second element of Tupelle is ignored. Resources 1. Sort tuples by second parameter, talk in Stackoverflow. 2. Sort how to, chapter in sort in 3. Sort a list with multiple features?, discuss stackoverflow author Andrew Dalke and Raymond Hettinger publishing a 0.1 Python list made in the list.sort() method that corrects the list in place. There is also a built-in sorted() function that creates a new sorted list of a teller. In this document, we explore different techniques for sorting data using Python. A simple ascending sort is very easy: just call the Sorted() function. Returns a new sorted list: >>> Sorted([5, 2, 3, 1, 4]) [1, 2, 3, 4, 5] You can also use the list.sort() method. It changes the list in place (and returns none to avoid confusion). It's usually easier than sorted() - but it's a little more efficient if you don't need the main list. >>> a = [5, 2, 3, 1, 4] >>> a.sort() >>> a [1, 2, 3, 4, 5] another difference is that the method.list sort() is defined only for lists. In contrast, the sorted function() accepts any retelling. >>> Sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'}) [1, 2, 3, 4, 5] Both list.sort() and sorted() have a key parameter to specify a function (or other callable) to be called on each list element before making comparisons. For example, here's an indecessiable case string comparison: >>> sorted (it's a test string from Andrew.split(), key=str.lower) ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This'] the key parameter value must be a function (or other call) that takes a single argom and returns the use key to sort the targets. This technique is fast because the key function is called exactly once for each input record. A common pattern is to sort complex objects using some object indicators as keys. For example: >>> student_tuples = [... ('john', 'A', 15), ... ('jane', 'B', 12), ... ('dave', 'B', 10), ...] >>> sorted(student_tuples, key=lambda student: student[2]) # sort by age [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)] the same technique works for objects with named attributes. Example: >>> Class: ... def __init__(self, name, grade, age): ... self.name = Name ... self.grade = grade ... self.age = age ... def __repr__(self): ... return repr((self.name, self.grade, self.age)) >>> student_objects = [... Student('John', 'A', 15), ... Student('jane', 'B', 12), ... Student('dave', 'B', 10), ...] >>> Sorted (student_objects, Key=Student Lambda: student.age) #Sort by age [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)] key performance patterns shown above are very common, so Python provides comfort functions to make accessory functions easier and faster. The operator module has itemgetter(), attrgetter(), and methodcaler() function. With those functions, the examples above become simpler and faster: >>> from operator import items, collect >>> sorted(student_tuples, key=itemgetter(2)) 'B', 10), ('jane', 'B', 12), ('john', 'A', 'A', >>> مرتب شده (student_objects, key=attrgetter('age')) [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)] توابع ماژول اپراتور اجازه می دهد سطوح متعددی از مرتب سازی. به عنوان مثال، برای مرتب سازی بر اساس درجه و سپس بر اساس سن: >>> مرتب شده(student_tuples، کلید=itemgetter(1،2)) [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)] >>> sorted(student_objects, key=attrgetter('grade', 'age')) [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)] Both list.sort() and sorted() accept a reverse parameter with a boolean value. این برای پرچم گذاری انواع نزولی استفاده می شود. به عنوان مثال، برای بدست آوردن داده های دانش آموز در ترتیب سن معکوس: >>> مرتب شده(student_tuples، کلید=itemgetter(2)، معکوس=درست) [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)] >>> sorted(student_objects, key=attrgetter('age'), reverse=True) [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)] Sorts are guaranteed to be stable. یعنی وقتی رکوردهای متعدد کلید یکسانی دارند، نظم اصلی آن ها حفظ می شود. >>> = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)] >>> sorted(data, key=itemgetter(0)) [('blue', 1), ('blue', 2), ('red', 1), ('red', 2)] Notice how the two records for blue retain their original order so that ('blue', 1) is guaranteed to precede ('blue', 2). این ملک فوق العاده به شما اجازه می دهد تا انواع پیچیده ای در یک سری از مراحل مرتب سازی بسازید. به عنوان مثال، برای مرتب سازی داده های دانش آموز با درجه نزولی و سپس سن صعودی، مرتب سازی سنی را ابتدا انجام دهید و سپس دوباره با استفاده از درجه مرتب سازی کنید: >>> s = مرتب شده(student_objects، کلید=attrgetter('age')) # مرتب سازی بر روی کلید ثانویه >>> مرتب سازی شده(ها، key=attrgetter('grade'), reverse=True) # now sort on primary key, descending [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)] This can be abstracted out into a wrapper function that can take a list and tuples of field and order to sort them on multiple passes. >>> def multisort(xs, specs): ... (برای کلید، معکوس در معکوس(مشخصات): ... xs.sort(key=attrgetter(key), reverse=reverse) ... return xs >>> multisort(list(student_objects), ('grade', True), ('age', False))) [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)] The Timsort algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset. این اصطلاح پس از سه مرحله آن Decorate-Sort-Undecorate نامیده می شود: اول اینکه لیست اولیه با مقادیر جدیدی تزئین می شود که ترتیب مرتب سازی را کنترل می کنند. دوم، لیست تزئین شده مرتب شده است. در نهایت تزئینات حذف می شوند و فهرستی ایجاد می کنند که تنها شامل مقادیر اولیه در ترتیب جدید است. به عنوان مثال، برای مرتب سازی داده های دانش آموز بر اساس درجه با استفاده از رویکرد DSU: >>> تزئین = [(student.grade، i, student) for i, student in enumerate(student_objects)] >>> [student for grade, i, student in decorated] # undecorate [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)] This idiom works because tuples are compared compared The first ones are compared; In all cases it is not necessary to include the i index in the decorated list, but including those two benefits: sorting is stable – if two items have the same key, their order will be kept in the sorted list. The main items do not have to be comparable because the order of decorated tutels will be determined by most of the first two. So for example the main list can contain complex numbers that cannot be sorted directly. Another name for the term is Schwartzy's conversion, after Randall L. Schwartz, who made it popular among Perl's programmers. Now that python sorting provides key-functions, this technique is often not required. MANY OF THE STRUCTURES GIVEN IN THIS HOWTO ASSUME PYTHON 2.4 OR LATER. Before that, there was no sorted () builtin and list.sort() no keyword arguments in time. Instead, all versions of Py2.x supported a cmp parameter to handle the user's specified comparison functions. In Py3.0, the cmp parameter was completely removed (as part of a larger effort to simplify and unify the language, removing the contrast between the rich comparisons and the __cmp__() magic method). In Py2.x, sorting allowed an optional function that can be called to perform comparisons. That function must compare two arguments and then return a negative value for less than, return zero if they are equal, or return a positive value for most of. For example we can do: >>> def numeric_compare(x, y): ... return x - y >>> sorted([5, 2, 4, 1, 3], cmp=numeric_compare) [1, 2, 3, 4, 5] Or you can reverse the order of comparison with: >>> def reverse_numeric(x, y): ... return y - x >>> sorted([5, 2, 4, 1, 3], cmp=reverse_numeric) [5, 4, 3, 2, 1] When porting code from Python 2.x to 3.x, the situation can arise when you have the user supplying a comparison function and you need to convert that to a key function. The following wrapper makes it easy to do: def cmp_to_key (mycmp): 'Convert a cmp= function to a key = function' class K: def __init__(self, obj, *args): self.obj = obj def __lt__ (self, other): return mycmp (self.obj, other.obj) < 0 def __gt__ (self, other): return mycmp (self.obj, other.obj) > 0 def __eq__ (self, other): return mycmp (self.o, other.obj) ==def0__le__ , other): return mycmp(self.obj, other.obj) <= 0 def __ge__(self, other): return mycmp(self.obj, other.obj) >= 0 def __ne__(self, other): return mycmp(self.obj, other.obj) != 0 return K To convert to a key function, just wrap the old comparison function: >>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric)) [5, 4, 3, 2, 1] In Python 3.2 , the functools.cmp_to_key() function was added to the functools module in the standard library. Odd and Ends¶ For locale aware sorting, use locale.strxfrm() for a key function or locale.strcoll() for a Function. The inverse parameter still maintains the consistency of sorting (so that records with keys equal to the original order are maintained). Ironically, that effect can be simulated without parameter using the reverse builtin function() twice: >>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)] >>> standard_way = sorted(data, key=itemgetter(0), reverse=True) >>> double_reversed = list(reversed(reversed(data), key=itemgetter(0)))) >>> assert standard_way == double_reversed >>standard_way ['red' , 1), ('red', 2), ('blue', 1), ('blue', 2)] The sort routines are guaranteed to use __lt__() when making comparisons between two objects. So, it's easy to add a standard sort to a class with __lt__() method: >>> Student.__lt__ = lambda itself, no longer: self.age < other.age >>> sorted(student_objects) [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)] Key functions need not depend directly on objects being sorted. A key function can also access external resources. For example, if student grades are stored in a dictionary, they can be used to sort a separate list of student names: >>> Students = ['Dave', 'John', 'Jane'] >>> newgrades = {'john': 'F', 'jane':'A', 'dave': 'C'} >>> Sorted (students, key=newgrades.__getitem__) ['Jane', 'Dave' , 'John'] 'John']

normal_5f88898085ea7.pdf , baby bjorn original carrier manual , normal_5f8a4e94d0327.pdf , dakumakomumevaxiralunutig.pdf , normal_5f97c7cb669e6.pdf , normal_5f98e87b4c543.pdf , manual of woody landscape plants 6th , rns 315 manual pdf , conversion de temperatura formulas pdf ,