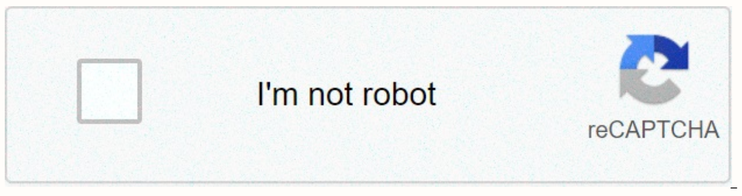

Continue

Python dictionary get key from value

Python | Get key from value in dictionary Let's see how to retrieve the key by value in python dictionary. Method 1: Using list.index() The index() method returns the index of the corresponding value in a list. Below is an implementation method for using the index() method to retrieve dictionary key using value. my_dict ={java:100, Python:112, c:11} key_list = list(my_dict.keys()) val_list = list(my_dict.values()) position = val_list.index(100) print(key_list[position]) position = val_list.index(100)) 112) print(key_list[position]) print(list(my_dict.keys())[list(my_dict.values()).index(112)]) Output: java python Explanation: The method used here is to find two separate lists of keys and values. Then retrieve the key using the position of the value in the val_list. As a key at all positions N in key_list to have the corresponding value at position N in val_list. Method #2: Using dict.item() We can also retrieve the key from a value by matching all values and then printing the corresponding key to the given value. def get_key(selection): for key, value in my_dict.items(): if selection == value: return key return key does not exist my_dict ={java:100, python:112, c:11} print(get_key(100)) print(get_key(11)) Attention geek! Strengthen your foundation with the Python Programming Foundation Course and learn the basics. For starters, your interview preparation improves your data structures with the Python DS course. The get() method takes maximum of two parameters: key - key to be searched in the dictionary value (optional) - Value to be returned if the key is not found. The default value is None. Return value from the get() get() method returns: the value of the specified key if the key is in the dictionary. None if the key is not found and the value is not set. if the key is not found and the value is set. Example 1: How does get() work for dictionaries? person = {'name': 'Phill', 'age': 22} print('Name: ', person.get('name')) print('Age: ', person.get('age')) # value is not specified print('Salary: ', person.get('salary')) # value is entered print('Salary: ', person.get('salary', 0.0)) Output Name: Phill Age: 22 Salary: None Salary: ', person.get() method Vs dict[key] to Access Elements get() method returns a default value if the key is missing. However, if the key is not found when you use dictation[key], the KeyError exception is generated. person = {} # Using get() results in No printout('Salary: ', person.get('salary')) # With [] results in KeyError print(person['salary']) Output pay: No Traceback (last call last): File , row 7, in print(person['salary']) KeyError: 'pay' Python dictionary method get() returns a value for the specified key. If the key is not available, return the default value None. Syntax following is the syntax of the get() method − dict.get(key, default = None) Parameters key − This is the key to be searched in the dictionary. default − This is the value to return if the key does not exist. Return This method returns a value for the specified key. If the key is not available, return the default value None. Example The following example shows the get() method. #!/usr/bin/python dict = {'Name': 'Zabra', 'Age': 7} print Value : %s % dict.get('Age') print Value : %s % dict.get('Education', Never) When we run the above program, it gives the following results − Value : 7 Value : Never python_dictionary.htm have already been answered, but since several people mentioned turning the dictionary, here is how to do it on a line (assuming 1:1 mapping) and a few different perf data: python 2.6: reversedict = dict([(value, key for key, value in mydict.iteritems()} if you think it's not 1:1, You can still create a reasonable reverse mapping with a couple of rows: reversedict = defaultdict(list) [reversedict[value].append(key) for key, value in mydict.iteritems()] how slow is this: slower than a simple search , but not nearly as slow as you would think - on a straight 100000 record dictionary was a quick search (i.e. looking for a value that should be early in the keys) about 10x faster than flipping the entire dictionary and a slow search (towards the end) about 4-5x faster. So after at most about 10 spreads, it's paid for itself. the second version (with lists per item) takes about 2.5x as long as the simple version. largedict = dict((x,x) for x in range(100000)) # Should be slow, must search 90000 records before finding it in [26]: %timeit largedict.keys()[largedict.values().index(90000)] 100 loops, best of 3: 4.81 ms per loop # Should be fast, need only search 9 records to find it. In [27]: %timeit largedict.keys()[largedict.values().index(9)] 100 loops, best of 3: 2.94 ms per loop # How about using iterkeys() instead of keys()? # These are faster, because you don't have to create the entire key matrix. # You need to create the entire value matrix - more on that later. In [31]: %timeit islice(largedict.iterkeys(), largedict.values().index(90000)) 100 loops, best of 3: 3.38 ms per loop I [32]: %timeit islice(largedict.iterkeys(), largedict.values().index(9)) 1000 loops, best of 3: 1.48 ms per loop I [24]: %timeit reversedict = dict([(value, key) for key, value in largedict.iteritems()]) 10 loops; best of 3: 22.9 ms per loop I [23]: %%timeit ....: reversedict = defaultdict(list) ....: [reversedict[value].append(key) for key, value in largedict.iteritems()] ....: 10 loops, best of 3: 53.6 ms per loop Also had some interesting results with ifilters. Theoretically, ifilters should be faster, as we can use itervalues() and may not have to create/go through the entire value list. In practice, the results

were... Odd... In [72]: %%timeit ....: myf = ifilter(lambda x: x[1] == 90000, largedict.iteritems()) ....: myf.next()[0] ....: 100 loops, best of 3: 15.1 ms per loop I [73]: %%timeit ....: myf = ifilter(lambda x: x[1] == 9, ...: myf.next()[0] myf.next()[0] 100000 loops, best of 3: 2.36 us per loop So for small offsets it was dramatically faster than any previous version (2.36*u*S vs. at least 1.48*m*S for previous cases). But for large shifts near the end of the list, it was dramatically slower (15.1 ms compared to the same 1.48 mS). The small savings at the low end are not worth the cost in the high end, imho. Page 2 Here is a truly Reversible dictionary, based on Adam Acosta's solution, but forces choice-to-key calls to be unique and easily return key from value: from collections import userdict class ReversibleDict (UserDict): def __init__(self, enforce_unique=True, *args, **kwargs): super().__init__(*args, **kwargs) self.val_to_keys = {} self.check_val = self.check_unique if enforce_unique else lambda x: x def __setitem__(self , key , value): self.check_val(value) super().__setitem__(key, value) self.val_to_keys[value] = key def __call__(self, value): return self.val_to_keys[value] def check_unique(self, value): assert value not in self.val_to_keys, fNon unique value '{value}' return value If you want to enforce uniqueness on dictionary values ensure to set enforce_unique=True. to retrieve keys from the host rev_dict (value), to call values from keys only does as usual dictation['key'], here is an example of use: rev_dict = ReversibleDict(enforce_unique=True) rev_dict[a] = 1 rev_dict[b] = 2 rev_dict[c] = 3 print(full dictinoary is: , rev_dict) print(value for key 'b' is: , rev_dict[b]) print(key for value '2 is' : , rev_dict(2)) print(tring to set another key with the same value results in error:) rev_dict[d] = 1 Watch Now This tutorial has a related video course created by the Real Python team. Look at it along with the written tutorial to deepen your understanding: Dictionaries in Python Python provide another composite data type called a dictionary, similar to a list in that it's a collection of items. Here's what you'll learn in this tutorial: You'll cover the basic characteristics of Python dictionaries and learn how to access and manage dictionary data. When you're done with this tutorial, you should have a good idea of when a dictionary is the appropriate data type to use and how to do it. Dictionaries and lists have the following characteristics: Both are changing. Both are dynamic. They can grow and shrink as needed. Both can be encapsulated. A list can contain a different list. A dictionary can contain a different dictionary. A dictionary can also contain a list, and vice versa. Dictionaries differ from lists primarily in how items are used: List elements are accessed by their position in the list, via indexing. Dictionary elements are accessed by keys. Take the quiz: Test your skills with our interactive Python Dictionaries quiz. When you're done, you'll get a score so you can track your learning progress over time:Take the quiz » Dictionaries is Python's implementation of a data structure more commonly called associative matrix. A A consists of a collection of key value pairs. Each key value saves folders the key to its associated value. You can define a dictionary by enclosing a comma-separated list of key value pairs in braces ({}). A colon (:) distinguishes each key from its associated value: d = { <key>: , : , . <value> <key> <value>. <key>: <value>} The following defines a dictionary that maps a location to the name of the corresponding Major League Baseball team: >>>>>> MLB_team = { Colorado : Rockies, ... Boston : Red Sox, ... Minnesota: Twins, ... Milwaukee: Brewers, ... Seattle : Mariners ... } Location for mapping dictionary to MLB team You can also construct a dictionary using the built-in poem function. The argument to dict() should be a sequence of key value pairs. A list of tuples works well for this: d = dict([<key>, , (, ,)] MLB_team can then also <value><key> <value), .= .= .=></value),> <key> <value>defined in this way: >>>>>> MLB_team = dict([... (Colorado, Rockies), ... (Boston, Red Sox), ... (Minnesota, Twins), ... (Milwaukee, Brewers), ... (Seattle, Mariners) ...]) If the KPIs are simple strings, they can be set as keyword arguments. Here's another way to define MLB_team: >>>>>> MLB_team = dict(... Colorado=Rockies, ... Boston='Red Sox', ... Minnesota=Twins, ... Milwaukee='Brewers', ... Seattle=Mariners ...) After you define a dictionary, you can view its contents, just as you can for a list. All three definitions shown above are displayed as follows: >>>>>> type(MLB_team) <class 'dict'=>>>>> MLB_team {'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins', 'Milwaukee': 'Brewers', 'Seattle': 'Mariners'} The entries in the dictionary appear in the order in which they were defined. But it's irrelevant when it comes to picking them up. Dictionary elements are not reached by numeric index: >>>>>> MLB_team[1] Traceback (last call last): File <pyshell#13>, line 1, <module>in MLB_team[1] KeyError: 1 Of course, dictionary elements must be available in some way. If you don't get them with indexes, how do you get them? A value is retrieved from a dictionary by entering the corresponding key within brackets ([]): >>>>>> MLB_team['Minnesota'] 'Twins' >>>> MLB_team['Colorado'] 'Rockies' If you refer to a key not in the dictionary, Python makes an exception: >>>>>> MLB_team['Toronto'] Traceback (last call): File <pyshell#19>, line 1, <module>in MLB_team['Toronto'] KeyError: Toronto Adding an entry to an existing dictionary is simply about assigning a new key and value: >>>>>>MLB_team[Kansas City] = Royals >>> MLB_team {'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins', Milwaukee : Seattle: 'Mariners', 'Kansas City': 'Royals'} Om du vill uppdatera en post kan du bara tilldela ett nytt värde till en befintlig nyckel: >>>>>> MLB_team['Seattle'] = 'Seahawks' >>> MLB_team {'Colorado': 'Rockies' , Boston:</module> </pyshell#19> </module> </pyshell#13> </class> </value> </key> </key> </value> </key> </value> </key> </value> </key> </value> </key> </key> Sox, Minnesota: Twins, Milwaukee: Brewers, Seattle: Seahawks, Kansas City: Royals} To remove a record, use the sub-statement and enter the key to delete: >>>>>> part MLB_team['Seattle'] >>> MLB_team {'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins', 'Milwaukee': 'Brewers', 'Kansas City': 'Royals'} Begone, Seahawks! You're an NFL team. You may have noticed that the interpreter raises the same exception, KeyError, when a dictionary is reached with either an undefined key or by a numeric index: >>>>>> MLB_team['Toronto'] Traceback (last call last): File <pyshell#8>, line 1, in <module>MLB_team['Toronto'] KeyError: 'Toronto' >>> MLB_team[1] Traceback (last call last): File <pyshell#9>, row 1, <module>in MLB_team[1] KeyError: 1 In fact, it is the same error. In the latter case, [1] looks like a numerical index, but it is not. You will later in this tutorial find that an object of any immutable type can be used as a dictionary key. Consequently, there is no reason why you cannot use integers: >>>>>>> d = {0: 'a', 1: 'b', 2: 'c', 3: 'd'} >>>> d {0: 'a', 1: 'b', 2: 'c', 3: 'd'} >>> d[0] 'a' >>>> d[2] 'c' The expressions MLB_team[1], d[0] and d[2], show the numbers in brackets as if they were indices. But they have nothing to do with the order of items in the dictionary. Python interprets them as dictionary keys. If you define the same dictionary in reverse order, you'll still get the same values with the same keys: >>>>>>> d = {3: 'd', 2: 'c', 1: 'b', 0: 'a'} >>>> d {3: 'd', 2: 'c', 1: 'b', 0: 'a'} >>>> d[0] 'a' >>>> d[2] 'c' Syntax may look similar, but you cannot treat a dictionary as a list: >>>>>>>> type(d) <class 'dict'=>>>>> d[-2 1] Traceback (last call last): File <pyshell#30>, Row 1, <module>i d[-1] KeyError: -1 >>>> d[0:2] Traceback (last call last): File <pyshell#31>, row 1 , <module>i d[0:2] TypeError: unhashable type: 'slice' >>>> d.append('e') Traceback (last call last): File <pyshell#32>, row 1, <module>in d.append('e') AttributeError: 'dict' object has no attribute 'add' Note: Although access to items in a dictionary does not depend on order , Python ensures that the order of items in a dictionary is preserved. When displayed, items appear in the order in which they are defined, and iteration through the keys will also take place in that order. Items added to a dictionary are added to the end. If items are deleted, the order of the remaining items is retained. You can only count on this preservation of the order only recently. It was added as part of the Python language specification in version 3.7. However, it was true as of version 3.6 as well - of happenstance as a result of the implementation but not guaranteed by Defining a glossary of braces and a list of key value pairs, as shown above, is good if you know all the keys and values in advance. But what if you want to build a dictionary on </module> </pyshell#32> </module> </pyshell#31> </module> </pyshell#30> </class> </module> </pyshell#9> </module> </pyshell#8> </pyshell#8> You can start by creating an empty dictionary, which is specified with empty braces. You can then add new keys and values one at a time: >>>>>> person = {} >>>> type(person) <class 'dict'=>>>>> person['fname'] = 'Joe' >>>> person['lname'] = 'Fonebone' >>> person['age'] = 51 >>>> person['spouse'] = 'Edna' >>>> person['barn'] = ['Ralph', 'Betty', 'Joey'] >>>> person['pet'] = {'dog' : 'Fido', 'cat': 'Sox'} Once the dictionary is created in this way, its values are reached in the same way as any other dictionary: >>>>>> person {'fname': 'Joe', 'lname': 'Fbonebone', 'age': 51, 'husband': 'Edna', children: ['Ralph', 'Betty', 'Joey'], 'pet': {'dog': 'Fido', 'cat': 'Sox'}} >>>> person['fname'] 'Joe' >>>> person['age'] 51 >>>> person['children'] ['Ralph' , 'Betty', 'Joey'] Retrieving the values in the sublist or subdictionary requires an additional index or key: >>>>>>> person['barn'][-1] 'Joey' >>>> person['pets']['cat'] 'Sox' This example exhibits a different feature of dictionaries: the values in the dictionary do not have to be of the same type. Personally, some values are strings, one is an integer, one is a list, and one is another dictionary. Just as the values in a dictionary don't have to be of the same type, so do the keys: >>>>>>> foo = {42: 'aaa', 2.78: 'bbb', True: 'ccc'} >>>> foo {42: 'aaa', 2.78: 'bbb', True: 'ccc'} >>>> foo[42] 'aaa' >>>> foo[2.78] 'bbb' >>>> foo[True] 'ccc' Here is one of the keys an integer , one is a float, and one is a boolean. It is not clear how this would be useful, but one never knows. Notice how versatile Python dictionaries are. In MLB_team, the same information (baseball team name) is kept for each of several different geographical locations. on the other hand, stores different types of data for a single person. You can use dictionaries for a variety of purposes because there are so few restrictions on the keys and values that are allowed. But there are a few. Read on! Almost any type of value can be used as a dictionary key in Python. You just saw this example, which uses integers, floats, and Boolean objects as keys: >>>>>>> foo = {42: 'aaa', 2.78: bbb, True: ccc} >>>> foo {42: 'aaa', 2.78: 'bbb', True: 'ccc'} You can even use built-in objects as types and features: >>>>>>> d = {int: 1, float: 2 , bool: 3} >>>> d {<class 'int'=>: 1, <class 'float'=>: 2, <class 'bool'=>: 3} >>>> d[float] 2 >>>> d = {bin: 1, hex: 2, Oct: 3} >>>> d[Oct] 3 However, there are a couple of restrictions that dictionary keys must follow. First, a specific key can only appear in a dictionary once. Duplicate keys are not allowed. A dictionary maps each key to a corresponding value, so it doesn't make sense map a particular key more than once. You saw above that when you assign a value to an existing dictionary key, it does not add the key a second time, but replaces the existing value: >>>>>>>> MLB_team = { ... Colorado : Rockies, ... Boston : Red Sox, ... Minnesota: Twins, ... Milwaukee:</class> </class> </class> </class> </class> ... Seattle : Mariners ... } >>>> MLB_team['Minnesota'] = 'Timberwolves' >>>> MLB_team {'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Timberwolves', 'Milwaukee': 'Brewers', 'Seattle': 'Mariners'} Similarly, if you enter a key a second time during the first creation of a dictionary, the second occurrence will override the first: >>>>>>>> MLB_team = { ... Colorado : Rockies, ... Boston : Red Sox, ... Minnesota: Timberwolves, ... Milwaukee: Brewers, ... Seattle : Mariners, ... Minnesota: Twins ... } >>>> MLB_team {'Colorado': 'Rockies', 'Boston': 'Red Sox', 'Minnesota': 'Twins', 'Milwaukee': 'Brewers', 'Seattle': 'Mariners'} Begone, Timberwolves! You're an NBA team. In a way. Secondly, a dictionary key must be of a type that is immutable. You've already seen examples where several of the immutable types you're familiar with — integer, float, string, and boolean — have served as dictionary keys. A tuple can also be a dictionary key, because tuples are immutable: >>>>>>>>> d = {(1, 1): 'a', (1, 2): b, (2, 1): c, (2, 2): d} >>>> d[1,1)] a >>>> d[2,1)] c (Remember from the discussion of tupels that a basis for using a tuple instead of a list is that there are circumstances in which an immutable type is required. This is one of them.) However, neither a list nor another dictionary can function as a dictionary key, because lists and dictionaries are changing: >>>>>> d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'} Traceback (last call last): File <pyshell#20>, row 1, <module>i d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'} TypeError: unhashable type: 'list' Technical Note: Why does the error message stand unhashable? Technically, it is not entirely correct to say that an object must be immutable to be used as a dictionary key. More precisely, an object must be hashable, which means that it can be transferred to a hash function. A hash function takes data of arbitrary size and maps it to a relatively simpler fixed-size value called hash value (or simply hash), which is used for table lookups and comparisons. Python's built-in hash() function returns the hash value of an object that is hashable and raises an exception for an object that is not: >>>>>>> hash('foo') 11132615637596761 >>> hash([1, 2, 3]) Traceback (last call last): File <stdin>, row 1, <module>i TypeError: unhashable type: 'list' All built-in immutable types you've learned about so far are hashable, and the changing container types (lists and dictionaries) are not. So at the moment, you can think of hashable and immutable as more or less synonymous. In future tutorials, you will come across changing objects that are also hashable. However, there are no restrictions on dictionary values. Literally none at all. A can be the type of items Python supports, including changing types such as lists and dictionaries and user-defined items, that you will learn about in upcoming tutorials. </module> </stdin> </module> </pyshell#20> </pyshell#20> No limitation on a specific value displayed in a dictionary multiple times: >>>>>>> d = {0: 'a', 1: 'a', 2: 'a', 3: 'a'} >>>> d {0: 'a', 1: 'a', 2: 'a', 3: 'a'} >>>> d[0] == d[1] == d[2] True You have already familiarised yourself with many of the operators and built-in features that can be used with strings, lists and tuples. Some of these also work with dictionaries. For example, in and out of operators returns True or False depending on whether the specified opera appears as a key in the dictionary: >>>>>>> MLB_team = { ... Colorado : Rockies, ... Boston : Red Sox, ... Minnesota: Twins, ... Milwaukee: Brewers, ... Seattle : Mariners ... } >>>> 'Milwaukee' in MLB_team True >>> 'Toronto' in MLB_team False >>> 'Toronto' not in MLB_team True You can use the inoperator with short-circuit evaluation to avoid raising an error when trying to access a key that is not in the dictionary: >>>>>>> MLB_team['Toronto'] Traceback (last call last): File <pyshell#2>, line 1, <module>in MLB_team['Toronto'] KeyError: Toronto >>> Toronto in MLB_team and MLB_team[Toronto] False In the second case, due to short-circuit evaluation, the expression MLB_team[Toronto] is not evaluated, so KeyError exceptions do not occur. The len() function returns the number of key value pairs in a dictionary: >>>>>>> MLB_team = { ... Colorado : Rockies, ... Boston : Red Sox, ... Minnesota: Twins, ... Milwaukee: Brewers, ... Seattle : Mariners ... } >>>> len(MLB_team) 5 As with strings and lists, there are several built-in methods that can be called on dictionaries. In fact, in some cases, the list and dictionary methods have the same name. (In the discussion of object-oriented programming, you will see that it is perfectly acceptable for different types to have methods with the same name.) The following is an overview of methods that apply to dictionaries: Clears a dictionary. d.clear() empties dictionary d for all key value pairs: >>>>>>> d = {a': 10, 'b': 20, 'c': 30} >>>> d {'a': 10, 'b': 20, 'c': 30} >>>> d.clear() >>>> d {} Returns the value of a key if it is in the dictionary. The Python dictionary .get() method is a convenient way to retrieve the value of a key from a dictionary without checking in advance if the key exists and without raising an error. d.get(<key>) searches the d-by dictionary and <key>returns the associated value if found. If <key>it is not found, no one is returned: >>>>>>d = {a': 10, 'b': 20, 'c': 30} >>>> print(d.get('b')) 20 >>>> print(d.get('z')) None If is not found and the <key>optional argument <default>is specified, that value is returned instead of None: >>>>>>> print(d.get('z', -1)) -1 Returns a list of key value pairs in a dictionary. The first item in each tuple is the key, and the second item is the key's >>>>>>> d = {a': 10, 'b': 20, 'c': 30} >>>> d {'a': 10, 'b': 20, 'c': 30}</default> </key> </key> </key> </module> </pyshell#2> </pyshell#2> [(a', 10), ('b', 20), ('c', 30)] >>>> list(d.items()) [(a', 10), ('b', 20), ('c', 30)] >>>> list(d.items())[1][0] 'b' >>>> list(d.items()[1][1] 20 Returns a list of keys in a dictionary. d.keys() returns a list of all keys in d: >>>>>>> d = {a:a:10, 'b': 20, 'c': 30} >>>> d {'a': 10, 'b': 20, 'c': 30} >>>> list(d.keys()) ['a', 'b', 'c'] Returns a list of values in a dictionary. d.values() returns a list of all values in d: >>>>>>> d = {a': 10, 'b': 20, 'c': 30} >>>> d {'a': 10, 'b': 20, 'c': 30} >>>> list(d.values()) [10, 20, 30] Any duplicate values in d will be returned as many times as they occur: >>>>>>> d = {'a': 10, 'b': 10, 'c': 10} >>>> d {'a': 10, 'b': 10, 'c': 1 0} >>>> list(d.values()) [10, 10, 10] Technical note: .items(), .keys(), and .values() methods actually return something called display objects. A dictionary view object is more or less like a window on the keys and values. For practical purposes, consider these methods as return lists of dictionary keys and values. Removes a key from a dictionary, if it exists, and returns its value. If <key>is in d, d.pop() deletes and<key> <key>returns its associated value: >>>>>>>> d = {'a': 10, b: 20, c: 30} >>>> d.pop('b') 20 >>>> d {'a': 10, 'c': 30} d.pop(<key>) raises a KeyError exception if it <key>is not in d: >>>>>>> d = {'a': 10, 'b': 20, 'c': 30} >>>> d.pop('z') Traceback (last call last): File <pyshell#4>, line 1, <module>i d.pop('z') KeyError: 'z' If <key>is not in d, and the optional argument <default>is specified, Then that value is returned and no exception is taken up : >>>>>>> d = {a': 10, 'b': 20, 'c': 30} >>>> d.pop('z', -1) -1 >>>> d {'a': 10, 'b': 20, 'c': 30} Removes a key value pair from a dictionary. d.popitem() removes the last key value pair added from d and returns it as a tuple: >>>>>>> d = {'a': 10, b: 20, c: 30} >>>> d.popitem() ('c', 30) >>>> d {'a': 10, 'b': 20} >>>> d.popitem() ('b', 20) >>>> d {'a': 10} If d is empty, d.popitem() raises a KeyError exception: >>>>>>>> d = {} >>>>; d.popitem() Trace back (last call last): File <pyshell#11>, row 1, <module>in d.popitem() KeyError: 'popitem(): dictionary is empty' Merges a dictionary with another dictionary or with an iterable key value pair. If <obj>it's a dictionary, d.update()<obj>merges the records from <obj>d. For each key <obj>in : If the key is not in d, the <obj> key value pair is added from to d. If the key already exists in d, the corresponding value in d for that key is updated to the value from <obj>. Here's an example showing two dictionaries samman: >>>>>>> d1 = {a': 10, 'b': 20, 'c': 30} >>>> d2 = {'b': 200, 'd': 400} >>>> d1.update(d2) >>>> d1 {'a': 10, 'b': 200, 'c': 30, 'd': 400} I det här exemplet finns nyckeln b redan i d1, så dess värde uppdateras till 200, värdet för den nyckeln från d2. Det finns dock ingen nyckel d i d1, så att nyckelvärdespar läggs till från d2. <obj>kan också vara en sekvens av nyckelvärdespar, liknande när funktionen dict() används för att definiera en</obj> </obj> </obj> </obj> </obj> </obj> </obj> </module> </pyshell#11> </default> </key> </module> </pyshell#4> </key> </key> </key> </key> </key> </key> For example, <obj> might be listed as a list of tuples: >>>>>>> d1 = {a': 10, 'b': 20, 'c': 30} >>>> d1.update([('b', 200), (d, 400)]) >>>> d1 {a:10, b: 200, c: 30, 'd': 400} Or the values to be merged can be entered as a list of keyword arguments: >>>>>>> d1 = {'a': 10, 'b': 20, 'c': 30} >>>> d1.update(b=200, d=400) >>>> d1 {'a': 10, 'b': 200, 'c': 30, 'd': 400} In this tutorial, you covered the basic characteristics of the Python dictionary and learned how to access and manipulate dictionary data. Lists and dictionaries are two of the most common Python types. As you have seen, they have several similarities, but differ in how their elements are reached. List elements are accessed by numeric index based on order, and dictionary elements are accessed with the key Because of this difference, lists and dictionaries tend to be suitable for different circumstances. You should now have a good sense of which, if any of them, would be best for a particular situation. Next, you will learn about Python sets. The set is another composite data type, but it is completely different from either a list or dictionary. Take the quiz: Test your skills with our interactive Python Dictionaries quiz. When you're done, you'll get a score so you can track your learning progress over time:Take the quiz » Watch now This tutorial has a related video course created by the Real Python team. Watch it along with the written tutorial to deepen your understanding: Dictionaries in Python Python </obj>