5. Data Structures

This chapter describes some things you’ve learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

list.**append**(*x*)

Add an item to the end of the list. Equivalent to a[len(a):] = [x].

list.**extend**(*iterable*)

Extend the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.

list.**insert**(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

list.**remove**(*x*)

Remove the first item from the list whose value is equal to *x*. It raises a [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError) if there is no such item.

list.**pop**([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.**clear**()

Remove all items from the list. Equivalent to del a[:].

list.**index**(*x*[, *start*[, *end*]])

Return zero-based index in the list of the first item whose value is equal to *x*. Raises a [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError) if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

list.**count**(*x*)

Return the number of times *x* appears in the list.

list.**sort**(*key=None*, *reverse=False*)

Sort the items of the list in place (the arguments can be used for sort customization, see [sorted()](https://docs.python.org/3/library/functions.html#sorted) for their explanation).

list.**reverse**()

Reverse the elements of the list in place.

list.**copy**()

Return a shallow copy of the list. Equivalent to a[:].

An example that uses most of the list methods:

>>>

**>>>** fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']

**>>>** fruits.count('apple')

2

**>>>** fruits.count('tangerine')

0

**>>>** fruits.index('banana')

3

**>>>** fruits.index('banana', 4) *# Find next banana starting a position 4*

6

**>>>** fruits.reverse()

**>>>** fruits

['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']

**>>>** fruits.append('grape')

**>>>** fruits

['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']

**>>>** fruits.sort()

**>>>** fruits

['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']

**>>>** fruits.pop()

'pear'

You might have noticed that methods like insert, remove or sort that only modify the list have no return value printed – they return the default None. [1](https://docs.python.org/3/tutorial/datastructures.html#id2) This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, [None, 'hello', 10] doesn’t sort because integers can’t be compared to strings and *None* can’t be compared to other types. Also, there are some types that don’t have a defined ordering relation. For example, 3+4j < 5+7j isn’t a valid comparison.

5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

>>>

**>>>** stack = [3, 4, 5]

**>>>** stack.append(6)

**>>>** stack.append(7)

**>>>** stack

[3, 4, 5, 6, 7]

**>>>** stack.pop()

7

**>>>** stack

[3, 4, 5, 6]

**>>>** stack.pop()

6

**>>>** stack.pop()

5

**>>>** stack

[3, 4]

5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use [collections.deque](https://docs.python.org/3/library/collections.html" \l "collections.deque" \o "collections.deque) which was designed to have fast appends and pops from both ends. For example:

>>>

**>>> from** **collections** **import** deque

**>>>** queue = deque(["Eric", "John", "Michael"])

**>>>** queue.append("Terry") *# Terry arrives*

**>>>** queue.append("Graham") *# Graham arrives*

**>>>** queue.popleft() *# The first to arrive now leaves*

'Eric'

**>>>** queue.popleft() *# The second to arrive now leaves*

'John'

**>>>** queue *# Remaining queue in order of arrival*

deque(['Michael', 'Terry', 'Graham'])

5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

>>>

**>>>** squares = []

**>>> for** x **in** range(10):

**...**  squares.append(x\*\*2)

**...**

**>>>** squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Note that this creates (or overwrites) a variable named x that still exists after the loop completes. We can calculate the list of squares without any side effects using:

squares = list(map(**lambda** x: x\*\*2, range(10)))

or, equivalently:

squares = [x\*\*2 **for** x **in** range(10)]

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

>>>

**>>>** [(x, y) **for** x **in** [1,2,3] **for** y **in** [3,1,4] **if** x != y]

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

and it’s equivalent to:

>>>

**>>>** combs = []

**>>> for** x **in** [1,2,3]:

**...**  **for** y **in** [3,1,4]:

**...**  **if** x != y:

**...**  combs.append((x, y))

**...**

**>>>** combs

[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

Note how the order of the [for](https://docs.python.org/3/reference/compound_stmts.html#for) and [if](https://docs.python.org/3/reference/compound_stmts.html#if) statements is the same in both these snippets.

If the expression is a tuple (e.g. the (x, y) in the previous example), it must be parenthesized.

>>>

**>>>** vec = [-4, -2, 0, 2, 4]

**>>>** *# create a new list with the values doubled*

**>>>** [x\*2 **for** x **in** vec]

[-8, -4, 0, 4, 8]

**>>>** *# filter the list to exclude negative numbers*

**>>>** [x **for** x **in** vec **if** x >= 0]

[0, 2, 4]

**>>>** *# apply a function to all the elements*

**>>>** [abs(x) **for** x **in** vec]

[4, 2, 0, 2, 4]

**>>>** *# call a method on each element*

**>>>** freshfruit = [' banana', ' loganberry ', 'passion fruit ']

**>>>** [weapon.strip() **for** weapon **in** freshfruit]

['banana', 'loganberry', 'passion fruit']

**>>>** *# create a list of 2-tuples like (number, square)*

**>>>** [(x, x\*\*2) **for** x **in** range(6)]

[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

**>>>** *# the tuple must be parenthesized, otherwise an error is raised*

**>>>** [x, x\*\*2 **for** x **in** range(6)]

File "<stdin>", line 1, in <module>

[x, x\*\*2 for x in range(6)]

^

SyntaxError: invalid syntax

**>>>** *# flatten a list using a listcomp with two 'for'*

**>>>** vec = [[1,2,3], [4,5,6], [7,8,9]]

**>>>** [num **for** elem **in** vec **for** num **in** elem]

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

List comprehensions can contain complex expressions and nested functions:

>>>

**>>> from** **math** **import** pi

**>>>** [str(round(pi, i)) **for** i **in** range(1, 6)]

['3.1', '3.14', '3.142', '3.1416', '3.14159']

5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

>>>

**>>>** matrix = [

**...**  [1, 2, 3, 4],

**...**  [5, 6, 7, 8],

**...**  [9, 10, 11, 12],

**...** ]

The following list comprehension will transpose rows and columns:

>>>

**>>>** [[row[i] **for** row **in** matrix] **for** i **in** range(4)]

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

As we saw in the previous section, the nested listcomp is evaluated in the context of the [for](https://docs.python.org/3/reference/compound_stmts.html#for) that follows it, so this example is equivalent to:

>>>

**>>>** transposed = []

**>>> for** i **in** range(4):

**...**  transposed.append([row[i] **for** row **in** matrix])

**...**

**>>>** transposed

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

which, in turn, is the same as:

>>>

**>>>** transposed = []

**>>> for** i **in** range(4):

**...**  *# the following 3 lines implement the nested listcomp*

**...**  transposed\_row = []

**...**  **for** row **in** matrix:

**...**  transposed\_row.append(row[i])

**...**  transposed.append(transposed\_row)

**...**

**>>>** transposed

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

In the real world, you should prefer built-in functions to complex flow statements. The [zip()](https://docs.python.org/3/library/functions.html#zip) function would do a great job for this use case:

>>>

**>>>** list(zip(\*matrix))

[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

See [Unpacking Argument Lists](https://docs.python.org/3/tutorial/controlflow.html#tut-unpacking-arguments) for details on the asterisk in this line.

5.2. The del statement

There is a way to remove an item from a list given its index instead of its value: the [del](https://docs.python.org/3/reference/simple_stmts.html#del) statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

>>>

**>>>** a = [-1, 1, 66.25, 333, 333, 1234.5]

**>>> del** a[0]

**>>>** a

[1, 66.25, 333, 333, 1234.5]

**>>> del** a[2:4]

**>>>** a

[1, 66.25, 1234.5]

**>>> del** a[:]

**>>>** a

[]

[del](https://docs.python.org/3/reference/simple_stmts.html#del) can also be used to delete entire variables:

>>>

**>>> del** a

Referencing the name a hereafter is an error (at least until another value is assigned to it). We’ll find other uses for [del](https://docs.python.org/3/reference/simple_stmts.html#del) later.

5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see [Sequence Types — list, tuple, range](https://docs.python.org/3/library/stdtypes.html#typesseq)). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

>>>

**>>>** t = 12345, 54321, 'hello!'

**>>>** t[0]

12345

**>>>** t

(12345, 54321, 'hello!')

**>>>** *# Tuples may be nested:*

**...** u = t, (1, 2, 3, 4, 5)

**>>>** u

((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

**>>>** *# Tuples are immutable:*

**...** t[0] = 88888

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

**>>>** *# but they can contain mutable objects:*

**...** v = ([1, 2, 3], [3, 2, 1])

**>>>** v

([1, 2, 3], [3, 2, 1])

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are [immutable](https://docs.python.org/3/glossary.html#term-immutable), and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of [namedtuples](https://docs.python.org/3/library/collections.html" \l "collections.namedtuple" \o "collections.namedtuple)). Lists are [mutable](https://docs.python.org/3/glossary.html#term-mutable), and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

>>>

**>>>** empty = ()

**>>>** singleton = 'hello', *# <-- note trailing comma*

**>>>** len(empty)

0

**>>>** len(singleton)

1

**>>>** singleton

('hello',)

The statement t = 12345, 54321, 'hello!' is an example of *tuple packing*: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

>>>

**>>>** x, y, z = t

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4. Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the [set()](https://docs.python.org/3/library/stdtypes.html#set) function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

>>>

**>>>** basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

**>>>** print(basket) *# show that duplicates have been removed*

{'orange', 'banana', 'pear', 'apple'}

**>>>** 'orange' **in** basket *# fast membership testing*

True

**>>>** 'crabgrass' **in** basket

False

**>>>** *# Demonstrate set operations on unique letters from two words*

**...**

**>>>** a = set('abracadabra')

**>>>** b = set('alacazam')

**>>>** a *# unique letters in a*

{'a', 'r', 'b', 'c', 'd'}

**>>>** a - b *# letters in a but not in b*

{'r', 'd', 'b'}

**>>>** a | b *# letters in a or b or both*

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

**>>>** a & b *# letters in both a and b*

{'a', 'c'}

**>>>** a ^ b *# letters in a or b but not both*

{'r', 'd', 'b', 'm', 'z', 'l'}

Similarly to [list comprehensions](https://docs.python.org/3/tutorial/datastructures.html#tut-listcomps), set comprehensions are also supported:

>>>

**>>>** a = {x **for** x **in** 'abracadabra' **if** x **not** **in** 'abc'}

**>>>** a

{'r', 'd'}

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see [Mapping Types — dict](https://docs.python.org/3/library/stdtypes.html#typesmapping)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the [in](https://docs.python.org/3/reference/expressions.html#in) keyword.

Here is a small example using a dictionary:

>>>

**>>>** tel = {'jack': 4098, 'sape': 4139}

**>>>** tel['guido'] = 4127

**>>>** tel

{'jack': 4098, 'sape': 4139, 'guido': 4127}

**>>>** tel['jack']

4098

**>>> del** tel['sape']

**>>>** tel['irv'] = 4127

**>>>** tel

{'jack': 4098, 'guido': 4127, 'irv': 4127}

**>>>** list(tel)

['jack', 'guido', 'irv']

**>>>** sorted(tel)

['guido', 'irv', 'jack']

**>>>** 'guido' **in** tel

True

**>>>** 'jack' **not** **in** tel

False

The [dict()](https://docs.python.org/3/library/stdtypes.html" \l "dict" \o "dict) constructor builds dictionaries directly from sequences of key-value pairs:

>>>

**>>>** dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'sape': 4139, 'guido': 4127, 'jack': 4098}

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

>>>

**>>>** {x: x\*\*2 **for** x **in** (2, 4, 6)}

{2: 4, 4: 16, 6: 36}

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

>>>

**>>>** dict(sape=4139, guido=4127, jack=4098)

{'sape': 4139, 'guido': 4127, 'jack': 4098}

5.6. Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

>>>

**>>>** knights = {'gallahad': 'the pure', 'robin': 'the brave'}

**>>> for** k, v **in** knights.items():

**...**  print(k, v)

**...**

gallahad the pure

robin the brave

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) function.

>>>

**>>> for** i, v **in** enumerate(['tic', 'tac', 'toe']):

**...**  print(i, v)

**...**

0 tic

1 tac

2 toe

To loop over two or more sequences at the same time, the entries can be paired with the [zip()](https://docs.python.org/3/library/functions.html#zip) function.

>>>

**>>>** questions = ['name', 'quest', 'favorite color']

**>>>** answers = ['lancelot', 'the holy grail', 'blue']

**>>> for** q, a **in** zip(questions, answers):

**...**  print('What is your *{0}*? It is *{1}*.'.format(q, a))

**...**

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the [reversed()](https://docs.python.org/3/library/functions.html#reversed) function.

>>>

**>>> for** i **in** reversed(range(1, 10, 2)):

**...**  print(i)

**...**

9

7

5

3

1

To loop over a sequence in sorted order, use the [sorted()](https://docs.python.org/3/library/functions.html#sorted) function which returns a new sorted list while leaving the source unaltered.

>>>

**>>>** basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

**>>> for** f **in** sorted(set(basket)):

**...**  print(f)

**...**

apple

banana

orange

pear

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

>>>

**>>> import** **math**

**>>>** raw\_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]

**>>>** filtered\_data = []

**>>> for** value **in** raw\_data:

**...**  **if** **not** math.isnan(value):

**...**  filtered\_data.append(value)

**...**

**>>>** filtered\_data

[56.2, 51.7, 55.3, 52.5, 47.8]

5.7. More on Conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

The comparison operators in and not in check whether a value occurs (does not occur) in a sequence. The operators is and is not compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, a < b == c tests whether a is less than b and moreover b equals c.

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired composition.

The Boolean operators and and or are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but B is false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

>>>

**>>>** string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'

**>>>** non\_null = string1 **or** string2 **or** string3

**>>>** non\_null

'Trondheim'

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the [walrus operator](https://docs.python.org/3/faq/design.html#why-can-t-i-use-an-assignment-in-an-expression) :=. This avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

5.8. Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

(1, 2, 3) < (1, 2, 4)

[1, 2, 3] < [1, 2, 4]

'ABC' < 'C' < 'Pascal' < 'Python'

(1, 2, 3, 4) < (1, 2, 4)

(1, 2) < (1, 2, -1)

(1, 2, 3) == (1.0, 2.0, 3.0)

(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)

Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) exception.

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [clear()](https://www.w3schools.com/python/ref_dictionary_clear.asp) | Removes all the elements from the dictionary |
| [copy()](https://www.w3schools.com/python/ref_dictionary_copy.asp) | Returns a copy of the dictionary |
| [fromkeys()](https://www.w3schools.com/python/ref_dictionary_fromkeys.asp) | Returns a dictionary with the specified keys and value |
| [get()](https://www.w3schools.com/python/ref_dictionary_get.asp) | Returns the value of the specified key |
| [items()](https://www.w3schools.com/python/ref_dictionary_items.asp) | Returns a list containing a tuple for each key value pair |
| [keys()](https://www.w3schools.com/python/ref_dictionary_keys.asp) | Returns a list containing the dictionary's keys |
| [pop()](https://www.w3schools.com/python/ref_dictionary_pop.asp) | Removes the element with the specified key |
| [popitem()](https://www.w3schools.com/python/ref_dictionary_popitem.asp) | Removes the last inserted key-value pair |
| [setdefault()](https://www.w3schools.com/python/ref_dictionary_setdefault.asp) | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| [update()](https://www.w3schools.com/python/ref_dictionary_update.asp) | Updates the dictionary with the specified key-value pairs |
| [values()](https://www.w3schools.com/python/ref_dictionary_values.asp) | Returns a list of all the values in the dictionary |

Top of Form

Test Yourself With Exercises

Bottom of Form