Python Lists Reference

Operator `[ ]`

Usage

```
list[index]
list[:]
list[index:]
list[:index]
list[index1:index2]
```

Description

This operator can be used in the following contexts:

- Expression `list[index]` returns the element at position `index` in the given list.
- Expression `list[:]` returns a copy of the entire list.
- Expression `list[index:]` returns a new list containing all elements of the given list, starting at the position given in `index`.
- Expression `list[:index]` returns a new list containing all elements of the given list starting with the first and ending with the element at position `index` − 1.
- Expression `list[index1:index2]` returns a new list containing all elements between positions `index1` and `index2` − 1.

All these constructs can also be used on the left-hand side of an assignment, with the following considerations:

- Statement `list[index] = value` replaces the element at position `index` with the element given in `value`.
- Statement `list1[index1:index2] = list2`, as well as all variants that omit one or both of the indices between the square brackets, replaces subset of the list between `index1` and `index2` − 1 with the elements given in `list2`.

Indices can be expressed as positive or negative values. Index 0 represents the first element in the list; index 1 is the second element, etc. Index −1 represents the last element in the list; index −2 represents the second to last, etc.

Example

```
# Create list
x = ['John', 'Mary', 'Adam', 'Susan']

# Using positive indices
print(x[1])             # Prints 'Mary'
print(x[:])             # Prints ['John', 'Mary', 'Adam', 'Susan']
print(x[2:])             # Prints ['Adam', 'Susan']
print(x[:3])             # Prints ['John', 'Mary', 'Adam']
print(x[1:3])            # Prints ['Mary', 'Adam']

# Using negative indices
print(x[-1], x[-2])      # Prints 'Susan Adam'
print(x[-3:-1])          # Prints ['Mary', 'Adam']

# Updating list
x[1:3] = ['Lisa', 'Ed', 'Max']
print(x)                 # Prints ['John', 'Lisa', 'Ed', 'Max', 'Susan']
```

Operator +

Usage

```
list1 + list2
```

Description

When the `+` operator is surrounded with two lists, it serves the purpose of list concatenation. A new list is created containing all elements in *`list1`* followed by the elements in *`list2`*. Neither *`list1`* nor *`list2`* are modified in the process.

Example

```
men = ['John', 'Adam']
women = ['Mary', 'Susan']
persons = men + women
print(persons)   # Prints ['John', 'Adam', 'Mary', 'Susan']
```

Operator `*`

Usage

```
list * value
```

Description

When the `*` operator is used with a list on its left and an integer number on its right, it serves the purpose of concatenating *`list`* with itself, as many times as indicated by *`value`*. This expression returns a new list containing all elements in *`list`* repeated *`value`* times. The original list is not modified in the process.

Example

```
persons = ['John', 'Mary']
many_persons = persons * 3
print(many_persons)   # Prints ['John', 'Mary', 'John', 'Mary', 'John', 'Mary']
```

Operator `del`

Usage

```
del list[index]
del list
```

Description

When the `del` operator is followed by a list and a valid index in square brackets, it is used to delete the element in the list pointed to by that index.

When the `del` operator is followed by only a list name (or a variable name of any type, for that matter), it frees the entire list (or variable). Referencing that list (or variable) after that will cause a `NameError` exception.

Removing an element from the end of a list has a cost of $O(1)$. Removing an element from the beginning of a list has a cost of $O(n)$, where n is the number of elements currently present in the list. On average, assuming uniformly distributed indices, the cost of this operation is $O(n)$.

Example

```
# Create list
x = ['John', 'Mary', 'Susan']

# Remove element at index 1
del x[1]
print(x)         # Prints ['John', 'Susan']

# Remove entire list
del x
print(x)         # Raises 'NameError' exception
```

Built-in function `all()`

Usage

```
all(list)
```

Description

The `all` built-in function takes a list as an argument and returns a Boolean value indicating whether all elements in *`list`* are `True`. If any element in the list is not a Boolean value, it is first converted using the same rules as the `bool()` type conversion function uses, and without affecting the state of the list.

If the list is empty, the `all()` function returns `True`.

Example

```
print(all([ ]))                             # Prints True
print(all([True, True, True]))              # Prints True
print(all([True, False, True]))             # Prints False
print(all([1, 'x', True]))                  # Prints True
```

Built-in function `any()`

Usage

```
any(list)
```

Description

The `any` built-in function takes a list as an argument and returns a Boolean value indicating whether at least one of the elements in *`list`* is `True`. If any element in the list is not a Boolean value, it is first converted using the same rules as the `bool()` type conversion function uses, and without affecting the state of the list.

If the list is empty, the `any()` function returns `False`.

Example

```
print(any([ ]))                             # Prints False
print(any([True, False, False]))            # Prints True
print(any([False, False, False]))           # Prints False
print(any([0, '', False]))                  # Prints False
```

Built-in function `len()`

Usage

```
len(list)
```

Description

The `len()` built-in function returns the number of elements present in argument *`list`*.

Example

```
x = [2, 4, 6]
print(len(x))    # Prints 3
```

Built-in function `max()`

Usage

```
max(list)
```

Description

The `max()` built-in function returns the maximum value among all elements present in the list.

Example

```
x = [2, 5, 3]
print(max(x))    # Prints 5
```

Built-in function `min()`

Usage

```
min(list)
```

Description

The `min()` built-in function returns the minimum value among all elements present in the list.

Example

```
x = [2, 5, 3]
print(min(x))    # Prints 2
```

Built-in function `sum()`

Usage

```
sum(list)
```

Description

Obtain the sum of all elements in the list. If the list is empty, this function returns 0.

Example

```
print(sum([2, 5, 3]))    # Prints 10
print(sum([]))           # Prints 0
```

Method `append()`

Usage

```
list.append(value)
```

Description

Append `value` at the end of `list`.

In most cases, this operation has a cost of $O(1)$. Occasionally, Python must internally increase the storage currently devoted to the list elements, for which it must allocate a larger region of memory and copy the entire list. In this situation, the cost of appending a new element is $O(n)$, where n is the number of elements currently present in the list.

Example

```
x = [10, 11, 12]
x.append(20)
print(x)          # Prints [10, 11, 12, 20]
```

Method `clear()`

Usage

```
list.clear()
```

Description

Clears the content of a list and turns it into an empty list.

Example

```
x = [10, 11, 12]
x.clear()
print(x)          # Prints []
```

Method `count()`

Usage

```
list.count(value)
```

Description

Return the number of occurrences of *`value`* in *`list`*. If the given value is not present in the list, this method returns 0.

Example

```
x = ['Adam', 'John', 'Adam', 'Mary']
print(x.count('Adam'))     # Prints 2
print(x.count('Susan'))    # Prints 0
```

Method `copy()`

Usage

```
list.copy()
```

Description

Return a new list containing all elements present in *`list`*.

Example

```
x = [10, 11, 12]
y = x                   # Variable 'y' points to the same list
y[0] = 'hello'          # Changes in 'y' affect 'x'
print(x)                # Prints ['hello', 11, 12]
print(y)                # Prints ['hello', 11, 12]

x = [10, 11, 12]
y = x.copy()            # Variable 'y' points to a new list
y[0] = 'hello'          # Changes in 'y' are separate
print(x)                # Prints [10, 11, 12]
print(y)                # Prints ['hello', 11, 12]
```

Method `extend()`

Usage

```
list1.extend(list2)
```

Description

Append all elements present in *`list2`* at the end of *`list1`*. List *`list2`* is not modified in the process.

Example

```
x = [10, 11]
y = [12, 13]
x.extend(y)
print(x)       # Prints [10, 11, 12, 13]
print(y)       # Prints [12, 13]
```

Method `index()`

Usage

```
list.index(value, [start], [end])
```

Description

Return the position of the first occurrence of `value` in `list`.

Arguments `start` and `end` are both optional. If they are given, the search space is limited between index `start` and `end` – 1.

If `value` is not present in the list within the search space (or in the entire list if `start` and `end` are not given), this function raises a `ValueError` exception.

Example

```
names = ['Alice', 'Kevin', 'Susan', 'Kevin', 'Taylor']
print(names.index('Kevin'))           # Prints 1
print(names.index('Susan', 1, 4))     # Prints 2
print(names.index('Taylor', 1, 4))    # Raises 'ValueError' exception
```

Method `insert()`

Usage

```
list.insert(index, value)
```

Description

Insert `value` in `list` before the element currently occupying position `index`. If `index` is equal to the number of elements currently present in the list, the new element is inserted at the end of the list.

Example

```
x = [10, 11, 12]
x.insert(3, 13)          # 'x' is now [10, 11, 12, 13]
x.insert(1, 'hello')     # 'x' is now [10, 'hello', 11, 12, 13]
```

Method `pop()`

Usage

```
list.pop([index])
```

Description

Remove the element at position `index` and return it. Argument `index` is optional. If it is not given, the function removes and returns the last element in the list.

If `index` does not represent a valid position in the list, or if the list is empty, an `IndexError` exception is raised.

Example

```
names = ['John', 'Mary', 'Adam']
print(names.pop(1))          # Prints 'Mary', list is now ['John', 'Adam']
print(names.pop())           # Prints 'Adam', list is now ['John']
print(names.pop())           # Prints 'John', list is now [ ]
print(names.pop())           # Raises 'IndexError'
```

Method `remove()`

Usage

```
list.remove(value)
```

Description

Remove the first occurrence of *`value`* in *`list`*. If the given value is not present in the list, this function raises a `ValueError` exception.

Example

```
animals = ['dog', 'elephant', 'tiger', 'dog']
animals.remove('dog')        # The list is now ['elephant', 'tiger', 'dog']
animals.remove('dog')        # The list is now ['elephant', 'tiger']
animals.remove('dog')        # Raises 'ValueError' exception
```

Method `reverse()`

Usage

```
list.reverse()
```

Description

Reverse the order of the element in `list`.

Example

```
x = [10, 11, 12, 13]
x.reverse()
print(x)     # Prints [13, 12, 11, 10]
```

Method `sort()`

Usage

```
list.sort(reverse = False)
```

Description

Sort `list` in place. If optional argument `reverse` is given and it is set to `True`, the list is sorted in reverse order.

This function uses the *Timesort* algorithm, which has a worst-case temporal cost of $O(n \log n)$.

Example

```
x = [12, 10, 13, 11]
x.sort()                    # The list is now [10, 11, 12, 13]
x.sort(reverse = False)     # The list is now [13, 12, 11, 10]
```