Reduce

All Versions

Examples

Overview

```
# No import needed


# No import required...
from functools import reduce # ... but it can be loaded from the functools module


from functools import reduce # mandatory
```

`reduce` reduces an iterable by applying a function repeatedly on the next element of an `iterable` and the cumulative result so far.

```
def add(s1, s2):
    return s1 + s2

asequence = [1, 2, 3]

reduce(add, asequence)  # equivalent to: add(add(1,2),3)
# Out: 6
```

In this example, we defined our own `add` function. However, Python comes with a standard equivalent function in the `operator` module:

```
import operator
reduce(operator.add, asequence)
# Out: 6
```

`reduce` can also be passed a starting value:

```
reduce(add, asequence, 10)
# Out: 16
```

Cumulative product

```
import operator
reduce(operator.mul, [10, 5, -3])
# Out: -150
```

Using reduce

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                           arg2=s2,
                                           res=s1*s2))
    return s1 * s2

asequence = [1, 2, 3]
```

Given an `initializer` the function is started by applying it to the initializer and the first iterable element:

```
cumprod = reduce(multiply, asequence, 5)
# Out: 5 * 1 = 5
#      5 * 2 = 10
#      10 * 3 = 30
print(cumprod)
# Out: 30
```

Without `initializer` parameter the `reduce` starts by applying the function to the first two list elements:

```
cumprod = reduce(multiply, asequence)
# Out: 1 * 2 = 2
#      2 * 3 = 6
print(cumprod)
print(cumprod)
# Out: 6
```

Non short-circuit variant of any/all

reduce will not terminate the iteration before the iterable has been completly iterated over so it can be used to create a non short-circuit any() or all() function:

```
import operator
# non short-circuit "all"
reduce(operator.and_, [False, True, True, True]) # = False

# non short-circuit "any"
reduce(operator.or_, [True, False, False, False]) # = True
```

First truthy/falsy element of a sequence (or last element if there is none)

```
# First falsy element or last element if all are truthy:
reduce(lambda i, j: i and j, [100, [], 20, 10])    # = []
reduce(lambda i, j: i and j, [100, 50, 20, 10])    # = 10

# First truthy element or last element if all falsy:
reduce(lambda i, j: i or j, [100, [], 20, 0])     # = 100
reduce(lambda i, j: i or j, ['', {}, [], None])   # = None
```

Instead of creating a lambda -function it is generally recommended to create a named function:

```
def do_or(i, j):
    return i or j

def do_and(i, j):
    return i and j

reduce(do_or, [100, [], 20, 0])                   # = 100
reduce(do_and, [100, [], 20, 0])                  # = []
```

Syntax

```
reduce(function, iterable[, initializer])
```

Parameters

Parameter	Details
function	function that is used for reducing the iterable (must take two arguments). (*positional-only*)
iterable	iterable that's going to be reduced. (*positional-only*)
initializer	start-value of the reduction. (*optional* , *positional-only*)

Remarks

reduce might be not always the most efficient function. For some types there are equivalent functions or methods:

- sum() for the sum of a sequence containing *addable* elements (not strings):

  ```
  sum([1,2,3])                                 # = 6
  ```

- str.join for the concatenation of strings:

  ```
  ''.join(['Hello', ',', ' World'])            # = 'Hello, World'
  ```

- next together with a generator could be a short-circuit variant compared to reduce :

```
# First falsy item:
next((i for i in [100, [], 20, 0] if not i)) # = []
```