Map Function

All Versions

Examples

Basic use of map, itertools.imap and future_builtins.map

The map function is the simplest one among Python built-ins used for functional programming. map() applies a specified function to each element in an iterable:

```
names = ['Fred', 'Wilma', 'Barney']
```

Python 3.x ≥3.0

```
map(len, names)  # map in Python 3.x is a class; its instances are iterable
# Out: <map object at 0x00000198B32E2CF8>
```

A Python 3-compatible map is included in the future_builtins module:

Python 2.x ≥2.6

```
from future_builtins import map  # contains a Python 3.x compatible map()
map(len, names)                  # see below
# Out: <itertools.imap instance at 0x3eb0a20>
```

Alternatively, in Python 2 one can use imap from itertools to get a generator

Python 2.x ≥2.3

```
map(len, names)   # map() returns a list
# Out: [4, 5, 6]

from itertools import imap
imap(len, names)  # itertools.imap() returns a generator
# Out: <itertools.imap at 0x405ea20>
```

The result can be explicitly converted to a list to remove the differences between Python 2 and 3:

```
list(map(len, names))
# Out: [4, 5, 6]
```

map() can be replaced by an equivalent *list comprehension* or *generator expression* :

```
[len(item) for item in names] # equivalent to Python 2.x map()
# Out: [4, 5, 6]

(len(item) for item in names) # equivalent to Python 3.x map()
# Out: <generator object <genexpr> at 0x00000195888D5FC0>
```

Mapping each value in an iterable

For example, you can take the absolute value of each element:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # the call to `list` is unnecessary in 2.x
# Out: [1, 1, 2, 2, 3, 3]
```

Anonymous function also support for mapping a list:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Out: [2, 4, 6, 8, 10]
```

or converting decimal values to percentages:

```
def to_percent(num):
    return num * 100

list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
# Out: [95.0, 75.0, 101.0, 10.0]
```

or converting dollars to euros (given an exchange rate):

```
from functools import partial
```

```
from functools import partial
from operator import mul

rate = 0.9  # fictitious exchange rate, 1 dollar = 0.9 euros
dollars = {'under_my_bed': 1000,
           'jeans': 45,
           'bank': 5000}

sum(map(partial(mul, rate), dollars.values()))
# Out: 5440.5
```

`functools.partial` is a convenient way to fix parameters of functions so that they can be used with `map` instead of using `lambda` or creating customized functions.

Mapping values of different iterables

For example calculating the average of each `i`-th element of multiple iterables:

```
def average(*args):
    return float(sum(args)) / len(args)  # cast to float - only mandatory for python 2.x

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117, 91, 102]
measurement3 = [104, 102, 95, 101]

list(map(average, measurement1, measurement2, measurement3))
# Out: [102.0, 110.0, 95.0, 100.0]
```

There are different requirements if more than one iterable is passed to `map` depending on the version of python:

- The function must take as many parameters as there are iterables:

  ```
  def median_of_three(a, b, c):
      return sorted((a, b, c))[1]

  list(map(median_of_three, measurement1, measurement2))
  ```

  > TypeError: median_of_three() missing 1 required positional argument: 'c'

  ```
  list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
  ```

  > TypeError: median_of_three() takes 3 positional arguments but 4 were given

Python 2.x ≥2.0.1

- `map` : The mapping iterates as long as one iterable is still not fully consumed but assumes `None` from the fully consumed iterables:

  ```
  import operator

  measurement1 = [100, 111, 99, 97]
  measurement2 = [102, 117]

  # Calculate difference between elements
  list(map(operator.sub, measurement1, measurement2))
  ```

  > TypeError: unsupported operand type(s) for -: 'int' and 'NoneType'

- `itertools.imap` and `future_builtins.map` : The mapping stops as soon as one iterable stops:

  ```
  import operator
  from itertools import imap

  measurement1 = [100, 111, 99, 97]
  measurement2 = [102, 117]

  # Calculate difference between elements
  list(imap(operator.sub, measurement1, measurement2))
  # Out: [-2, -6]
  list(imap(operator.sub, measurement2, measurement1))
  # Out: [2, 6]
  ```

Python 3.x ≥3.0.0

- The mapping stops as soon as one iterable stops:

  ```
  import operator
  ```

```
measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Calculate difference between elements
list(map(operator.sub, measurement1, measurement2))
# Out: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Out: [2, 6]
```

Transposing with Map: Using "None" as function argument (python 2.x only)

```
from itertools import imap
from future_builtins import map as fmap # Different name to highlight differences

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

list(map(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Out: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]
list(map(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8), (3, None, 9)]  # Fill missing values with None
list(fmap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # ignore columns with missing values
list(imap(None, *image2))
# Out: [(1, 4, 7), (2, 5, 8)]                # dito
```

Python 3.x ≥3.0.0

```
list(map(None, *image))
```

TypeError: 'NoneType' object is not callable

But there is a workaround to have similar results:

```
def conv_to_list(*args):
    return list(args)

list(map(conv_to_list, *image))
# Out: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Series and Parallel Mapping

map() is a built-in function, which means that it is available everywhere without the need to use an 'import' statement. It is available everywhere just like print() If you look at Example 5 you will see that I had to use an import statement before I could use pretty print (import pprint). Thus pprint is not a built-in function

Series mapping

In this case each argument of the iterable is supplied as argument to the mapping function in ascending order. This arises when we have just one iterable to map and the mapping function requires a single argument.

Example 1

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # the function defined by f is executed on each item of the iterabl
```

results in

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Example 2

```
print(list(map(len, insects))) # the len function is executed each item in the insect list
```

results in

```
[3, 3, 6, 10]
```

Parallel mapping

In this case each argument of the mapping function is pulled from across all iterables (one from each iterable) in parallel. Thus the number of iterables supplied must match the number of arguments required by the function.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']

def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Example 3

```
# Too many arguments
# observe here that map is trying to pass one item each from each of the four iterables to len.
# it is being fed too many arguments
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

results in

```
TypeError: len() takes exactly one argument (4 given)
```

Example 4

```
# Too few arguments
# observe here that map is suppose to execute animal on individual elements of insects one-by-or
# it only gets one argument, whereas it was expecting four.
print(list(map(animals, insects)))
```

results in

```
TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z'
```

Example 5

```
# here map supplies w, x, y, z with one value from across the list
import pprint
pprint.pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

results in

```
 ['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
 'Ant, tiger, moose, and dove ARE ALL ANIMALS',
 'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
 'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Syntax

```
map(function, iterable[, *additional_iterables])
```

```
future_builtins.map(function, iterable[, *additional_iterables])
```

```
itertools.imap(function, iterable[, *additional_iterables])
```

Parameters

Parameter	Details
function	function for mapping (must take as many parameters as there are iterables) (*positional-only*)

Parameter	Details
iterable	the function is applied to each element of the iterable (*positional-only*)
*additional_iterables	see iterable, but as many as you like (*optional* , *positional-only*)

Remarks

Everything that can be done with `map` can also be done with comprehensions :

```
list(map(abs, [-1,-2,-3]))    # [1, 2, 3]
[abs(i) for i in [-1,-2,-3]]  # [1, 2, 3]
```

Though you would need `zip` if you have multiple iterables:

```
import operator
alist = [1,2,3]
list(map(operator.add, alist, alist))  # [2, 4, 6]
[i + j for i, j in zip(alist, alist)]  # [2, 4, 6]
```

List comprehensions are efficient and can be faster than `map` in many cases, so test the times of both approaches if speed is important for you.