How to use multiprocessing pool.map with multiple arguments

Asked 11 years, 8 months ago Modified 7 days ago Viewed 909k times

819

In the Python `multiprocessing` library, is there a variant of `pool.map` which supports multiple arguments?

```
import multiprocessing

text = "test"

def harvester(text, case):
    X = case[0]
    text + str(X)

if __name__ == '__main__':
    pool = multiprocessing.Pool(processes=6)
    case = RAW_DATASET
    pool.map(harvester(text, case), case, 1)
    pool.close()
    pool.join()
```

python multiprocessing python-multiprocessing

Share Edit Follow

edited Dec 15, 2021 at 17:12
Tomerikoo
16.9k 16 39 57

asked Mar 26, 2011 at 14:23
user642897
8,611 3 20 22

10 To my surprise, I could make neither `partial` nor `lambda` do this. I think it has to do with the strange way that functions are passed to the subprocesses (via `pickle`). – senderle Mar 26, 2011 at 15:27

13 @senderle: This is a bug in Python 2.6, but has been fixed as of 2.7: bugs.python.org/issue5228 – unutbu Mar 26, 2011 at 16:18

3 Just simply replace `pool.map(harvester(text,case),case, 1)` by: `pool.apply_async(harvester(text,case),case, 1)` – Tung Nguyen Jul 14, 2016 at 7:20

5 @Syrtis_Major , please don't edit OP questions which effectively skew answers that have been previously given. Adding `return` to `harvester()` turned @senderie 's response into being inaccurate. That does not help future readers. – Ricalsin Jan 29, 2017 at 0:46

3 I would say easy solution would be to pack all the args in a tuple and unpack it in the executing func. I did this when I needed to send complicated multiple args to a func being executed by a pool of processes. – H S Rathore Dec 12, 2019 at 6:31

23 Answers

Sorted by: Highest score (default)

736

> is there a variant of pool.map which support multiple arguments?

Python 3.3 includes `pool.starmap()` method:

```
#!/usr/bin/env python3
from functools import partial
from itertools import repeat
from multiprocessing import Pool, freeze_support

def func(a, b):
    return a + b

def main():
    a_args = [1,2,3]
    second_arg = 1
    with Pool() as pool:
        L = pool.starmap(func, [(1, 1), (2, 1), (3, 1)])
        M = pool.starmap(func, zip(a_args, repeat(second_arg)))
        N = pool.map(partial(func, b=second_arg), a_args)
        assert L == M == N

if __name__=="__main__":
    freeze_support()
    main()
```

For older versions:

```
#!/usr/bin/env python2
import itertools
from multiprocessing import Pool, freeze_support

def func(a, b):
    print a, b

def func_star(a_b):
    """Convert `f([1,2])` to `f(1,2)` call."""
    return func(*a_b)

def main():
    pool = Pool()
    a_args = [1,2,3]
    second_arg = 1
    pool.map(func_star, itertools.izip(a_args, itertools.repeat(second_arg)))

if __name__=="__main__":
    freeze_support()
    main()
```

Output

```
1 1
2 1
3 1
```

Notice how `itertools.izip()` and `itertools.repeat()` are used here.

Due to the bug mentioned by @unutbu you can't use `functools.partial()` or similar capabilities on Python 2.6, so the simple wrapper function `func_star()` should be defined explicitly. See also the workaround suggested by `uptimebox`.

Share Edit Follow

edited Dec 8, 2019 at 13:59

answered Mar 26, 2011 at 17:24
jfs
386k 185 950 1628

2 F.: You can unpack the argument tuple in the signature of `func_star` like this: `def func_star((a, b))`. Of course, this only works for a fixed number of arguments, but if that is the only case he has, it is more readable. – Björn Pollex Mar 26, 2011 at 21:01

2 @Space_C0wb0y: `f((a,b))` syntax is deprecated and removed in py3k. And it is unnecessary here. – jfs Mar 26, 2011 at 21:31

3 perhaps more pythonic: `func = lambda x: func(*x)` instead of defining a wrapper function – dylam Jul 17, 2015 at 8:34

2 @zthomas.nc this question is about how to support multiple arguments for multiprocessing pool.map. If want to know how to call a method instead of a function in a different Python process via multiprocessing then ask a separate question (if all else fails, you could always create a global function that wraps the method call similar to `func_star()` above) – jfs Nov 21, 2016 at 21:21

4 I wish there were `starstarmap` . – Константин Ван Jan 23, 2019 at 6:25

492

The answer to this is version- and situation-dependent. The most general answer for recent versions of Python (since 3.3) was first described below by J.F. Sebastian.[1] It uses the `Pool.starmap` method, which accepts a sequence of argument tuples. It then automatically unpacks the arguments from each tuple and passes them to the given function:

```
import multiprocessing
from itertools import product

def merge_names(a, b):
    return '{} & {}'.format(a, b)

if __name__ == '__main__':
    names = ['Brown', 'Wilson', 'Bartlett', 'Rivera', 'Molloy', 'Opie']
    with multiprocessing.Pool(processes=3) as pool:
        results = pool.starmap(merge_names, product(names, repeat=2))
    print(results)

# Output: ['Brown & Brown', 'Brown & Wilson', 'Brown & Bartlett', ...
```

For earlier versions of Python, you'll need to write a helper function to unpack the arguments explicitly. If you want to use `with` , you'll also need to write a wrapper to turn `Pool` into a context manager. (Thanks to muon for pointing this out.)

```
import multiprocessing
from itertools import product
from contextlib import contextmanager

def merge_names(a, b):
    return '{} & {}'.format(a, b)

def merge_names_unpack(args):
    return merge_names(*args)

@contextmanager
def poolcontext(*args, **kwargs):
    pool = multiprocessing.Pool(*args, **kwargs)
    yield pool
    pool.terminate()

if __name__ == '__main__':
    names = ['Brown', 'Wilson', 'Bartlett', 'Rivera', 'Molloy', 'Opie']
    with poolcontext(processes=3) as pool:
        results = pool.map(merge_names_unpack, product(names, repeat=2))
    print(results)

# Output: ['Brown & Brown', 'Brown & Wilson', 'Brown & Bartlett', ...
```

In simpler cases, with a fixed second argument, you can also use `partial` , but only in Python 2.7+.

```
import multiprocessing
from functools import partial
from contextlib import contextmanager

@contextmanager
def poolcontext(*args, **kwargs):
    pool = multiprocessing.Pool(*args, **kwargs)
    yield pool
    pool.terminate()

def merge_names(a, b):
    return '{} & {}'.format(a, b)

if __name__ == '__main__':
    names = ['Brown', 'Wilson', 'Bartlett', 'Rivera', 'Molloy', 'Opie']
    with poolcontext(processes=3) as pool:
        results = pool.map(partial(merge_names, b='Sons'), names)
    print(results)

# Output: ['Brown & Sons', 'Wilson & Sons', 'Bartlett & Sons', ...
```

1. Much of this was inspired by his answer, which should probably have been accepted instead. But since this one is stuck at the top, it seemed best to improve it for future readers.

Share Edit Follow

edited Oct 10, 2017 at 16:11

answered Mar 26, 2011 at 14:36
senderle
141k 35 207 231

It seems to me that RAW_DATASET in this case should be a global variable? While I want the partial_harvester change the value of case in every call of harvester(). How to achieve that? – xgdgsc Sep 2, 2013 at 2:36

The most important thing here is assigning `=RAW_DATASET` default value to `case`. Otherwise `pool.map` will confuse about the multiple arguments. – Emerson Xu Jun 17, 2016 at 10:27

2 I'm confused, what happened to the `text` variable in your example? Why is `RAW_DATASET` seemingly passed twice. I think you might have a typo? – Dave Aug 22, 2016 at 23:16

not sure why using `with .. as ..` gives me `AttributeError: __exit__`, but works fine if i just call `pool = Pool();` then close manually `pool.close()` (python2.7) – muon Oct 10, 2017 at 15:44

2 @muon, good catch. It appears `Pool` objects don't become context managers until Python 3.3. I've added a simple wrapper function that returns a `Pool` context manager. – senderle Oct 10, 2017 at 15:56

177

I think the below will be better:

```
def multi_run_wrapper(args):
   return add(*args)

def add(x,y):
    return x+y

if __name__ == "__main__":
    from multiprocessing import Pool
    pool = Pool(4)
    results = pool.map(multi_run_wrapper,[(1,2),(2,3),(3,4)])
    print results
```

Output

```
[3, 5, 7]
```

Share Edit Follow

edited Nov 3, 2021 at 17:04
Peter Mortensen
30.7k 21 104 125

answered Jan 15, 2014 at 6:01
imotai
1,936 1 11 9

28 Easiest solution. There is a small optimization; remove the wrapper function and unpack `args` directly in `add`, it works for any number of arguments: `def`

`add(args): (x,y) = args` – Ahmed Dec 16, 2016 at 23:20

2 you could also use a `lambda` function instead of defining `multi_run_wrapper(..)` – Andre Holzner Mar 2, 2017 at 9:39

4 hm... in fact, using a `lambda` does not work because `pool.map(..)` tries to pickle the given function – Andre Holzner Mar 2, 2017 at 11:48

3 How do you use this if you want to store the result of `add` in a list? – Vivek Subramanian Sep 16, 2019 at 19:56

2 please add `pool.close()` and `pool.join()` after getting **results = pool.map(...)**, else this might possibly runs forever – Sean William Apr 12 at 6:33

116

Using **Python 3.3+** with `pool.starmap()`:

```
from multiprocessing.dummy import Pool as ThreadPool

def write(i, x):
    print(i, "---", x)

a = ["1","2","3"]
b = ["4","5","6"]

pool = ThreadPool(2)
pool.starmap(write, zip(a,b))
pool.close()
pool.join()
```

Result:

```
1 --- 4
2 --- 5
3 --- 6
```

You can also zip() more arguments if you like: `zip(a,b,c,d,e)`

In case you want to have a **constant value** passed as an argument:

```
import itertools

zip(itertools.repeat(constant), a)
```

In case your function should **return** something:

```
results = pool.starmap(write, zip(a,b))
```

This gives a List with the returned values.

Share Edit Follow

edited Mar 6, 2021 at 18:19

answered Mar 10, 2015 at 22:24
user136036
10.2k 6 43 46

3 This is a near exact duplicate answer as the one from @J.F.Sebastian in 2011 (with 60+ votes). – Mike McKerns Apr 9, 2015 at 12:34

66 No. First of all it removed lots of unnecessary stuff and clearly states it's for python 3.3+ and is intended for beginners that look for a simple and clean answer. As a beginner myself it took some time to figure it out that way (yes with JFSebastians posts) and this is why I wrote my post to help other beginners, because his post simply said "there is starmap" but did not explain it - this is what my post intends. So there is absolutely no reason to bash me with two downvotes. – user136036 Apr 9, 2015 at 19:28

81

How to take multiple arguments:

```
def f1(args):
    a, b, c = args[0] , args[1] , args[2]
    return a+b+c

if __name__ == "__main__":
    import multiprocessing
    pool = multiprocessing.Pool(4)

    result1 = pool.map(f1, [ [1,2,3] ])
    print(result1)
```

Share Edit Follow

edited Jul 4, 2020 at 14:20
lpd11
133 1 7

answered Dec 25, 2017 at 11:44
Dane Lee
1,854 11 14

8 Neat and elegant. – Prav001 Aug 8, 2019 at 20:53

18 I don't understand why I have to scroll all the way over here to find the best answer. – toti Apr 27, 2020 at 11:35

This answer should literally have been at the top most. – Hammad Aug 2, 2021 at 13:09

Still, an explanation would be in order. E.g., what is the idea/gist? What languages features does it use and why? Please respond by editing (changing) your answer, not here in comments (***without*** "Edit:", "Update:", or similar - the answer should appear as if it was written today). – Peter Mortensen Oct 24, 2021 at 12:18

31

Having learnt about itertools in J.F. Sebastian's answer I decided to take it a step further and write a `parmap` package that takes care about parallelization, offering `map` and `starmap` functions in Python 2.7 and Python 3.2 (and later also) that can take *any number* of positional arguments.

Installation

```
pip install parmap
```

How to parallelize:

```
import parmap
# If you want to do:
y = [myfunction(x, argument1, argument2) for x in mylist]
# In parallel:
y = parmap.map(myfunction, mylist, argument1, argument2)

# If you want to do:
z = [myfunction(x, y, argument1, argument2) for (x,y) in mylist]
# In parallel:
z = parmap.starmap(myfunction, mylist, argument1, argument2)

# If you want to do:
listx = [1, 2, 3, 4, 5, 6]
listy = [2, 3, 4, 5, 6, 7]
param = 3.14
param2 = 42
listz = []
for (x, y) in zip(listx, listy):
        listz.append(myfunction(x, y, param1, param2))
# In parallel:
listz = parmap.starmap(myfunction, zip(listx, listy), param1, param2)
```

I have uploaded parmap to PyPI and to a GitHub repository.

As an example, the question can be answered as follows:

```
import parmap

def harvester(case, text):
    X = case[0]
    text+ str(X)
```

```
if __name__ == "__main__":
    case = RAW_DATASET  # assuming this is an iterable
    parmap.map(harvester, case, "test", chunksize=1)
```

Share Edit Follow

edited Nov 3, 2021 at 17:12
Peter Mortensen
30.7k 21 104 125

answered Jan 22, 2014 at 20:05
zeehio
3,874 2 37 46

19

There's a fork of `multiprocessing` called pathos (*note: use the version on GitHub*) that doesn't need `starmap` -- the map functions mirror the API for Python's map, thus map can take multiple arguments.

With `pathos` , you can also generally do multiprocessing in the interpreter, instead of being stuck in the `__main__` block. Pathos is due for a release, after some mild updating -- mostly conversion to Python 3.x.

```
Python 2.7.5 (default, Sep 30 2013, 20:15:49)
[GCC 4.2.1 (Apple Inc. build 5566)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def func(a,b):
...     print a,b
...
>>>
>>> from pathos.multiprocessing import ProcessingPool
>>> pool = ProcessingPool(nodes=4)
>>> pool.map(func, [1,2,3], [1,1,1])
1 1
2 1
3 1
[None, None, None]
>>>
>>> # also can pickle stuff like lambdas
>>> result = pool.map(lambda x: x**2, range(10))
>>> result
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
>>> # also does asynchronous map
>>> result = pool.amap(pow, [1,2,3], [4,5,6])
>>> result.get()
[1, 32, 729]
>>>
>>> # or can return a map iterator
>>> result = pool.imap(pow, [1,2,3], [4,5,6])
>>> result
<processing.pool.IMapIterator object at 0x110c2ffd0>
>>> list(result)
[1, 32, 729]
```

`pathos` has several ways that that you can get the exact behavior of `starmap` .

```
>>> def add(*x):
...   return sum(x)
...
>>> x = [[1,2,3],[4,5,6]]
>>> import pathos
>>> import numpy as np
>>> # use ProcessPool's map and transposing the inputs
>>> pp = pathos.pools.ProcessPool()
>>> pp.map(add, *np.array(x).T)
[6, 15]
>>> # use ProcessPool's map and a lambda to apply the star
>>> pp.map(lambda x: add(*x), x)
[6, 15]
>>> # use a _ProcessPool, which has starmap
>>> _pp = pathos.pools._ProcessPool()
>>> _pp.starmap(add, x)
[6, 15]
>>>
```

Share Edit Follow

edited Nov 3, 2021 at 17:07
Peter Mortensen
30.7k 21 104 125

answered Jan 20, 2014 at 20:37
Mike McKerns
32.2k 8 114 138

I want to note that this doesn't address the structure in the original question. [[1,2,3], [4,5,6]] would unpack with starmap to [pow(1,2,3), pow(4,5,6)], not [pow(1,4), pow(2,5), pow(3, 6)]. If you don't have good control over the inputs being passed to to your function, you may need to restructure them first. – Scott Apr 6, 2020 at 16:19

@Scott: ah, I didn't notice that... over 5 years ago. I'll make a small update. Thanks. – Mike McKerns Apr 7, 2020 at 17:27

Should zip input vectors. More understandable than transposing and array, don't you think? – pauljohn32 Jul 15, 2020 at 3:56

The array transpose, while possibly less clear, should be less expensive. – Mike McKerns Jul 15, 2020 at 10:54

10

A better solution for Python 2:

```
from multiprocessing import Pool
def func((i, (a, b))):
    print i, a, b
    return a + b
pool = Pool(3)
pool.map(func, [(0,(1,2)), (1,(2,3)), (2,(3, 4))])
```

Output

```
2 3 4

1 2 3

0 1 2

out[]:

[3, 5, 7]
```

Share Edit Follow

edited Oct 24, 2021 at 12:19
Peter Mortensen
30.7k 21 104 125

answered May 23, 2017 at 10:11
xmduhan
830 11 14

10

A better way is using a **decorator** instead of writing a **wrapper function** by hand. Especially when you have a lot of functions to map, a decorator will save your time by avoiding writing a wrapper for every function. Usually a decorated function is not picklable, however we may use `functools` to get around it. More discussions can be found here.

Here is the example:

```
def unpack_args(func):
    from functools import wraps
    @wraps(func)
    def wrapper(args):
        if isinstance(args, dict):
            return func(**args)
        else:
            return func(*args)
    return wrapper

@unpack_args
def func(x, y):
    return x + y
```

Then you may map it with zipped arguments:

```
np, xlist, ylist = 2, range(10), range(10)
pool = Pool(np)
res = pool.map(func, zip(xlist, ylist))
pool.close()
pool.join()
```

Of course, you may always use `Pool.starmap` in Python 3 (>=3.3) as mentioned in other answers.

Share Edit Follow

edited Nov 3, 2021 at 17:21
Peter Mortensen
30.7k 21 104 125

answered May 29, 2016 at 1:17
Syrtis Major
3,583 1 27 39

Results are not as expected: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18] I would expect: [0,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9,10,2,3,4,5,6,7,8,9,10,11, ... – Tedo Vrbanec Oct 12, 2018 at 23:58

@TedoVrbanec Results just should be [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]. If you want the later one, you may use `itertools.product` instead of `zip` . – Syrtis Major Oct 13, 2018 at 4:38

`starmap` was the answer I was looking for. – root-11 Apr 23 at 15:26

10

Another way is to pass a list of lists to a one-argument routine:

```
import os
from multiprocessing import Pool

def task(args):
    print "PID =", os.getpid(), ", arg1 =", args[0], ", arg2 =", args[1]

pool = Pool()

pool.map(task, [
        [1,2],
        [3,4],
        [5,6],
        [7,8]
    ])
```

One can then construct a list lists of arguments with one's favorite method.

Share Edit Follow

edited Nov 3, 2021 at 17:13
Peter Mortensen
30.7k 21 104 125

answered Mar 13, 2014 at 21:55
Adobe
12.4k 8 83 123

This is an easy way, but you need to change your original functions. What's more, some time recall others' functions which may can't be modified. – WeizhongTu Aug 28, 2015 at 13:14

I will say this sticks to Python zen. There should be one and only one obvious way to do it. If by chance you are the author of the calling function, this you should use this method, for other cases we can use imotai's method. – nehem Oct 2, 2015 at 1:02

My choice is to use a tuple, And then immediately unwrap them as the first thing in the first line. – nehem Oct 2, 2015 at 1:03

What do you mean by *"a list lists of arguments"* (seems incomprehensible)? Preferably, please respond by editing (changing) your answer, not here in comments (***without*** "Edit:", "Update:", or similar - the answer should appear as if it was written today). – Peter Mortensen Nov 3, 2021 at 17:15

9

You can use the following two functions so as to avoid writing a wrapper for each new function:

```
import itertools
from multiprocessing import Pool

def universal_worker(input_pair):
    function, args = input_pair
    return function(*args)

def pool_args(function, *args):
    return zip(itertools.repeat(function), zip(*args))
```

Use the function `function` with the lists of arguments `arg_0`, `arg_1` and `arg_2` as follows:

```
pool = Pool(n_core)
list_model = pool.map(universal_worker, pool_args(function, arg_0, arg_1,
arg_2)
pool.close()
pool.join()
```

Share Edit Follow

answered Jun 27, 2014 at 7:42
M. Toya
595 7 23

9

Another simple alternative is to wrap your function parameters in a tuple and then wrap the parameters that should be passed in tuples as well. This is perhaps not ideal when dealing with large pieces of data. I believe it would make copies for each tuple.

```
from multiprocessing import Pool

def f((a,b,c,d)):
    print a,b,c,d
    return a + b + c +d

if __name__ == '__main__':
    p = Pool(10)
    data = [(i+0,i+1,i+2,i+3) for i in xrange(10)]
    print(p.map(f, data))
    p.close()
    p.join()
```

Gives the output in some random order:

```
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
7 8 9 10
```

```
6 7 8 9
8 9 10 11
9 10 11 12
[6, 10, 14, 18, 22, 26, 30, 34, 38, 42]
```

Share Edit Follow

answered Nov 21, 2016 at 7:16
Alex Klibisz
1,284 1 14 20

Indeed it does, still looking for a better way :(– Fábio Dias Feb 13, 2018 at 22:22

7

Here is another way to do it that IMHO is more simple and elegant than any of the other answers provided.

This program has a function that takes two parameters, prints them out and also prints the sum:

```
import multiprocessing

def main():

    with multiprocessing.Pool(10) as pool:
        params = [ (2, 2), (3, 3), (4, 4) ]
        pool.starmap(printSum, params)
    # end with

# end function

def printSum(num1, num2):
    mySum = num1 + num2
    print('num1 = ' + str(num1) + ', num2 = ' + str(num2) + ', sum = ' +
str(mySum))
# end function

if __name__ == '__main__':
    main()
```

output is:

```
num1 = 2, num2 = 2, sum = 4
num1 = 3, num2 = 3, sum = 6
num1 = 4, num2 = 4, sum = 8
```

See the python docs for more info:

https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.pool

In particular be sure to check out the `starmap` function.

I'm using Python 3.6, I'm not sure if this will work with older Python versions

Why there is not a very straight-forward example like this in the docs, I'm not sure.

Share Edit Follow

edited Jan 24, 2020 at 19:53

answered Jan 24, 2020 at 19:37
cdahms
3,094 9 44 71

5

From Python 3.4.4, you can use multiprocessing.get_context() to obtain a context object to use multiple start methods:

```
import multiprocessing as mp

def foo(q, h, w):
    q.put(h + ' ' + w)
    print(h + ' ' + w)

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,'hello', 'world'))
    p.start()
    print(q.get())
    p.join()
```

Or you just simply replace

```
pool.map(harvester(text, case), case, 1)
```

with:

```
pool.apply_async(harvester(text, case), case, 1)
```

Share Edit Follow

edited Nov 3, 2021 at 17:17
Peter Mortensen
30.7k 21 104 125

answered May 27, 2016 at 10:52
Tung Nguyen
1,438 2 17 13

3

There are many answers here, but none seem to provide Python 2/3 compatible code that will work on any version. If you want your code to *just work*, this will work for either Python version:

```
# For python 2/3 compatibility, define pool context manager
# to support the 'with' statement in Python 2
if sys.version_info[0] == 2:
    from contextlib import contextmanager
    @contextmanager
    def multiprocessing_context(*args, **kwargs):
        pool = multiprocessing.Pool(*args, **kwargs)
        yield pool
        pool.terminate()
else:
    multiprocessing_context = multiprocessing.Pool
```

After that, you can use multiprocessing the regular Python 3 way, however you like. For example:

```
def _function_to_run_for_each(x):
       return x.lower()
with multiprocessing_context(processes=3) as pool:
    results = pool.map(_function_to_run_for_each, ['Bob', 'Sue', 'Tim'])
print(results)
```

will work in Python 2 or Python 3.

Share Edit Follow

answered Feb 16, 2019 at 4:56

cgnorthcutt
3,672 33 39

3

In the official documentation states that it supports only one iterable argument. I like to use apply_async in such cases. In your case I would do:

```
from multiprocessing import Process, Pool, Manager

text = "test"
def harvester(text, case, q = None):
 X = case[0]
 res = text+ str(X)
 if q:
  q.put(res)
 return res


def block_until(q, results_queue, until_counter=0):
 i = 0
 while i < until_counter:
  results_queue.put(q.get())
  i+=1

if __name__ == '__main__':
 pool = multiprocessing.Pool(processes=6)
 case = RAW_DATASET
 m = Manager()
 q = m.Queue()
 results_queue = m.Queue() # when it completes results will reside in this
queue
 blocking_process = Process(block_until, (q, results_queue, len(case)))
 blocking_process.start()
 for c in case:
  try:
   res = pool.apply_async(harvester, (text, case, q = None))
   res.get(timeout=0.1)
  except:
   pass
 blocking_process.join()
```

Share Edit Follow

answered Apr 2, 2017 at 6:34
roj4s
231 2 8

You mean `c` instead of `case` here, right?: `res = pool.apply_async(harvester, (text, case, q = None))` – Michael Silverstein Aug 5, 2021 at 13:58

2

This might be another option. The trick is in the `wrapper` function that returns another function which is passed in to `pool.map`. The code below reads an input array and for each (unique) element in it, returns how many times (ie counts) that element appears in the array, For example if the input is

```
np.eye(3) = [ [1. 0. 0.]
              [0. 1. 0.]
              [0. 0. 1.]]
```

then zero appears 6 times and one 3 times

```
import numpy as np
from multiprocessing.dummy import Pool as ThreadPool
from multiprocessing import cpu_count
```

```
def extract_counts(label_array):
    labels = np.unique(label_array)
    out = extract_counts_helper([label_array], labels)
    return out

def extract_counts_helper(args, labels):
    n = max(1, cpu_count() - 1)
    pool = ThreadPool(n)
    results = {}
    pool.map(wrapper(args, results), labels)
    pool.close()
    pool.join()
    return results

def wrapper(argsin, results):
    def inner_fun(label):
        label_array = argsin[0]
        counts = get_label_counts(label_array, label)
        results[label] = counts
    return inner_fun

def get_label_counts(label_array, label):
    return sum(label_array.flatten() == label)

if __name__ == "__main__":
    img = np.ones([2,2])
    out = extract_counts(img)
    print('input array: \n', img)
    print('label counts: ', out)
    print("========")

    img = np.eye(3)
    out = extract_counts(img)
    print('input array: \n', img)
    print('label counts: ', out)
    print("========")

    img = np.random.randint(5, size=(3, 3))
    out = extract_counts(img)
    print('input array: \n', img)
    print('label counts: ', out)
    print("========")
```

You should get:

```
input array:
 [[1. 1.]
 [1. 1.]]
label counts:  {1.0: 4}
========
input array:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
label counts:  {0.0: 6, 1.0: 3}
========
input array:
 [[4 4 0]
 [2 4 3]
 [2 3 1]]
label counts:  {0: 1, 1: 1, 2: 2, 3: 2, 4: 3}
========
```

Share Edit Follow

edited Nov 15, 2020 at 14:28

answered Nov 15, 2020 at 13:26

Aenaon

2,929 4 28 56

2

This is an example of the routine I use to pass multiple arguments to a one-argument function used in a pool.imap fork:

```
from multiprocessing import Pool

# Wrapper of the function to map:
class makefun:
    def __init__(self, var2):
        self.var2 = var2
    def fun(self, i):
        var2 = self.var2
        return var1[i] + var2

# Couple of variables for the example:
var1 = [1, 2, 3, 5, 6, 7, 8]
var2 = [9, 10, 11, 12]

# Open the pool:
pool = Pool(processes=2)

# Wrapper loop
for j in range(len(var2)):
    # Obtain the function to map
    pool_fun = makefun(var2[j]).fun

    # Fork loop
    for i, value in enumerate(pool.imap(pool_fun, range(len(var1))), 0):
        print(var1[i], '+' ,var2[j], '=', value)

# Close the pool
pool.close()
```

Share Edit Follow

answered Jan 23, 2019 at 11:53

A. Nodar
21 2

2

```
import time
from multiprocessing import Pool


def f1(args):
    vfirst, vsecond, vthird = args[0] , args[1] , args[2]
    print(f'First Param: {vfirst}, Second value: {vsecond} and finally third
value is: {vthird}')
    pass


if __name__ == '__main__':
    p = Pool()
    result = p.map(f1, [['Dog','Cat','Mouse']])
    p.close()
    p.join()
    print(result)
```

Share Edit Follow

answered Jul 9, 2021 at 14:47
Dipankar Biswas
141 1 3

An explanation would be in order. E.g., what is the idea/gist? Please respond by editing (changing) your answer, not here in comments (***without*** "Edit:", "Update:", or similar - the answer should appear as if it was written today). – Peter Mortensen Oct 24, 2021 at 12:16

2

```
text = "test"

def unpack(args):
    return args[0](*args[1:])

def harvester(text, case):
    X = case[0]
    text+ str(X)

if __name__ == '__main__':
    pool = multiprocessing.Pool(processes=6)
    case = RAW_DATASET
    # args is a list of tuples
    # with the function to execute as the first item in each tuple
    args = [(harvester, text, c) for c in case]
    # doing it this way, we can pass any function
    # and we don't need to define a wrapper for each different function
    # if we need to use more than one
    pool.map(unpack, args)
    pool.close()
    pool.join()
```

Share Edit Follow

answered Oct 15, 2018 at 23:21
Jaime
21 1

0

For me,Below one was short and simple solution:

```
from multiprocessing.pool import ThreadPool
from functools import partial
from time import sleep
from random import randint
```

```
def dosomething(var,s):
    sleep(randint(1,5))
    print(var)
    return var + s

array = ["a", "b", "c", "d", "e"]
with ThreadPool(processes=5) as pool:
    resp_ = pool.map(partial(dosomething,s="2"), array)
    print(resp_)
```

Output:

```
a
b
d
e
c
['a2', 'b2', 'c2', 'd2', 'e2']
```

Share Edit Follow

answered Nov 28 at 10:58

Sachin

1,126 13 21

0

Store all your arguments as an *array of tuples*.

The example says normally you call your function as:

```
def mainImage(fragCoord: vec2, iResolution: vec3, iTime: float) -> vec3:
```

Instead pass one tuple and unpack the arguments:

```
def mainImage(package_iter) -> vec3:
    fragCoord = package_iter[0]
    iResolution = package_iter[1]
    iTime = package_iter[2]
```

Build up the tuple by using a loop beforehand:

```
package_iter = []
iResolution = vec3(nx, ny, 1)
for j in range((ny-1), -1, -1):
    for i in range(0, nx, 1):
        fragCoord: vec2 = vec2(i, j)
        time_elapsed_seconds = 10
        package_iter.append((fragCoord, iResolution, time_elapsed_seconds))
```

Then execute all using map by passing the *array of tuples*:

```
array_rgb_values = []

with concurrent.futures.ProcessPoolExecutor() as executor:
    for val in executor.map(mainImage, package_iter):
        fragColor = val
        ir = clip(int(255* fragColor.r), 0, 255)
        ig = clip(int(255* fragColor.g), 0, 255)
        ib = clip(int(255* fragColor.b), 0, 255)
        array_rgb_values.append((ir, ig, ib))
```

I know Python has `*` and `**` for unpacking, but I haven't tried those yet.

Also better to use the higher-level library concurrent futures than the low level multiprocessing library.

Share Edit Follow

edited Oct 24, 2021 at 12:14
Peter Mortensen
30.7k 21 104 125

answered Jun 19, 2021 at 15:16
Omar Khan
101 3

-2

For Python 2, you can use this trick

```
def fun(a, b):
    return a + b

pool = multiprocessing.Pool(processes=6)
b = 233
pool.map(lambda x:fun(x, b), range(1000))
```

Share Edit Follow

edited Oct 24, 2021 at 12:17
Peter Mortensen
30.7k 21 104 125

answered May 18, 2018 at 4:06
Hz Shang
95 1 8

why b=233. defeats the purpose of the question – as - if Aug 22, 2019 at 21:30