

# Forum:[Hack Forums](#)

Thread: [Recursion help.](#)

**Lindros** - January 12, 2020, 2:12 pm

I'm trying to understand recursion more deeply.

Can somebody walk me line by line through the execution of this function? It really isn't making sense to me for some reason.

Code stuff = [2, 3, 4]

```
def prod_2(things):  
    if len(things) == 0:  
        return 1  
    return things[0] * prod_2(things[1:])
```

```
print(prod_2(stuff))
```

output

Code 24

I understand it is summing the numbers, I understand this can be done with a loop. I understand what slicing/indexing is/does. My confusion begins with the returns and how/when the 1 is multiplied by each number in the list in the recursive calls. It's my understanding that the function slices off a number until the list is empty and then goes backwards multiplying each of the numbers by 1, I could be wrong because I am confused by it.

**Vanagander** - January 12, 2020, 6:25 pm

(Jan 12, 2020 03:12 PM)ZietzSeV Wrote: [\[CITE\]](#) -snip-Edit: This became much longer than I intended but I hope it clears up the confusion you're experiencing, if not then let me know and I'll try to help.

We'll start with the return:

As you've stated already: "recursion can be used interchangeably with loops" and as such, like loops, it needs to have a stop condition or else it will just keep on going like a while(True) loop. So the if statement that checks the length of the list is the stop condition and if the length is 0 it returns 1, stopping the recursion. We can see the recursion stops since it doesn't call the function again for another go around (Maybe a bit obvious but I'd rather explain too much than too little).

Alright, now on to the multiplication. First and foremost, the

Code return 1 when list length is 0 is multiplied with the rest but since  $X * 1 = X$  it doesn't affect the result.

Second, when you first call prod\_2 and pass your list as an argument the function takes the element at index 0. It then multiplies that element with the return value of the (next) function call where the list has been sliced so the previous index 1 is now index 0.

With a walking-through-door-metaphor:

You have a notepad with 3 pages, each page contains a number (this is the list). Each time you walk through a door, you call the function prod\_2. Starting in a hallway and open the first door, carrying the initial list with you. You then check if your list length is 0, it is not so you rip out the first page (index 0, element 2) and place it on the ground. Afterwards you open another door, carrying

the same but modified list with you (prod\_2 call). You check the length again and rip out a page (element 3), placing it on the floor. Open the next door, check the length and place the ripped out page (element 4) on the ground. You open the next door and check your list length, it is now 0 and it is time to stop.

Going back the way you came is the function returning something. So you take the number 1 with you back through the most recent door and multiply it with the number on the page that's on the ground (element 4). You now have  $1 \times 4$ . Going back through the next door, picking up the number 3 from the ground and multiplies  $1 \times 4$  with 3. Next door, picking up number 2 from the ground and multiplies  $1 \times 4 \times 3$  with 2. Going through the last door left (original function call) you return  $1 \times 4 \times 3 \times 2$  which equals 24.

This is the same function but with a loop:

Code prod\_2(list):

```
temp_var = 0
for i in list:
    temp_var *= i
return temp_var
```

Lindros - January 12, 2020, 7:24 pm

(Jan 12, 2020 07:25 PM)Vanagander Wrote: [\[CITE\]](#) Edit: This became much longer than I intended but I hope it clears up the confusion you're experiencing, if not then let me know and I'll try to help.

We'll start with the return:

As you've stated already: "recursion can be used interchangeably with loops" and as such, like loops, it needs to have a stop condition or else it will just keep on going like a while(True) loop. So the if statement that checks the length of the list is the stop condition and if the length is 0 it returns 1, stopping the recursion. We can see the recursion stops since it doesn't call the function again for another go around (Maybe a bit obvious but I'd rather explain too much than too little).

Alright, now on to the multiplication. First and foremost, the

Code return 1 when list length is 0 is multiplied with the rest but since  $X \times 1 = X$  it doesn't affect the result.

Second, when you first call prod\_2 and pass your list as an argument the function takes the element at index 0. It then multiplies that element with the return value of the (next) function call where the list has been sliced so the previous index 1 is now index 0.

With a walking-through-door-metaphor:

You have a notepad with 3 pages, each page contains a number (this is the list). Each time you walk through a door, you call the function prod\_2. Starting in a hallway and open the first door, carrying the initial list with you. You then check if your list length is 0, it is not so you rip out the first page (index 0, element 2) and place it on the ground. Afterwards you open another door, carrying the same but modified list with you (prod\_2 call). You check the length again and rip out a page (element 3), placing it on the floor. Open the next door, check the length and place the ripped out page (element 4) on the ground. You open the next door and check your list length, it is now 0 and it is time to stop.

Going back the way you came is the function returning something. So you take the number 1 with you back through the most recent door and multiply it with the number on the page that's on the ground (element 4). You now have  $1 \times 4$ . Going back through the next door, picking up the number 3 from the ground and multiplies  $1 \times 4$  with 3. Next door, picking up number 2 from the ground and multiplies  $1 \times 4 \times 3$  with 2. Going through the last door left (original function call) you return  $1 \times 4 \times 3 \times 2$  which equals 24.

This is the same function but with a loop:

```
Code prod_2(list):
    temp_var = 0
    for i in list:
        temp_var *= i
    return temp_var
```

Yeah that makes sense.

Also I believe in your loop function you have to use 1 and not 0 for product.

**Oscura** - January 13, 2020, 7:41 am

The function takes the product of the elements of the list. So let's walk through execution:

```
Code prod_2([2, 3, 4]) =
    2 * prod_2([3, 4])
    |
    3 * prod_2([4])
    |
    4 * prod_2([])
    |
    1
    = base case -> no further recursion
```

So after reaching the base case, the stack collapses back as such:

```
Code prod_2([2, 3, 4]) =
    |
    2 * prod_2([3, 4])
    |
    3 * prod_2([4])
    |
    4 * prod_2([])
    |
    1
```

```
-->
prod_2([2, 3, 4]) =
    |
    2 * prod_2([3, 4])
    |
    3 * prod_2([4])
    |
    4 * 1
```

```
-->
prod_2([2, 3, 4]) =
    |
    2 * prod_2([3, 4])
    |
    3 * 4
```

```
-->
prod_2([2, 3, 4]) =
    |
    2 * 12
-->
prod_2([2, 3, 4]) = 24
```

I assume you have knowledge about the stack and stack frames. Does this make it a bit clearer?

**Lindros** - January 13, 2020, 7:57 am

(Jan 13, 2020 08:41 AM)Oscura Wrote: [\[CITE\]](#) The function takes the product of the elements of the list. So let's walk through execution:

```
Code prod_2([2, 3, 4]) =  
  2 * prod_2([3, 4])  
  |  
  3 * prod_2([4])  
  |  
  4 * prod_2([])  
  |  
  1  
= base case -> no further recursion
```

So after reaching the base case, the stack collapses back as such:

```
Code prod_2([2, 3, 4]) =  
  |  
  2 * prod_2([3, 4])  
  |  
  3 * prod_2([4])  
  |  
  4 * prod_2([])  
  |  
  1
```

```
-->  
prod_2([2, 3, 4]) =  
  |  
  2 * prod_2([3, 4])  
  |  
  3 * prod_2([4])  
  |  
  4 * 1
```

```
-->  
prod_2([2, 3, 4]) =  
  |  
  2 * prod_2([3, 4])  
  |  
  3 * 4
```

```
-->  
prod_2([2, 3, 4]) =  
  |  
  2 * 12
```

```
-->  
prod_2([2, 3, 4]) = 24
```

I assume you have knowledge about the stack and stack frames. Does this make it a bit clearer?

Yes, I understand now. That illustration is helpful. I just wanted to internalize a deeper knowledge of what was happening.

**Oscura** - January 13, 2020, 8:05 am

(Jan 13, 2020 08:57 AM)ZietzSeV Wrote: [\[CITE\]](#) Yes, I understand now. That illustration is helpful. I

just wanted to internalize a deeper knowledge of what was happening.

Great. While this toy example could give you your first insights into recursion, convince yourself of the fact that implementing the function with recursion is less memory efficient than without recursion and recursion does not give any benefits of clarity to the implementation. An iterative approach would be better (in fact, any recursive program can be turned into an iterative program and visa versa, however for more complex programs that doesn't seem always straightforward).

I would suggest checking out some algorithms for which the use of recursion is actually really intuitive/natural to get a deeper understanding of recursion with some practical/heavily used algorithms. Examples are binary search, merge sort and depth/breadth first search.

**Lindros** - January 13, 2020, 8:12 am

(Jan 13, 2020 09:05 AM)Oscura Wrote: [\[CITE\]](#) Great. While this toy example could give you your first insights into recursion, convince yourself of the fact that implementing the function with recursion is less memory efficient than without recursion and recursion does not give any benefits of clarity to the implementation. An iterative approach would be better (in fact, any recursive program can be turned into an iterative program and visa versa, however for more complex programs that doesn't seem always straightforward).

I would suggest checking out some algorithms for which the use of recursion is actually really intuitive/natural to get a deeper understanding of recursion with some practical/heavily used algorithms. Examples are binary search, merge sort and depth/breadth first search.

Oh yeah I know, that was our first in class example. They explained that these problems shouldn't be done with recursion and it is just trying to show us recursion in general.

I actually used recursion before I even understood what it meant, the double returns is what was initially confusing me.