

Mathematical Foundations

Recursion

CS51 – Fall 2025

I hope you all had a good Fall break! We are reaching the end of the course unit on Mathematical Foundations. Before we dive into recursion, I want to go over the feedback you provided last week.

Programs so far

- Contain control flow and iteration.
- We will see a new way of solving problems using **recursion**.

2

The programs you have written so far have probably contained control flow (if/else statements) and iteration in the form of for and while loops. Today, we will see a new way of solving problems using recursion.

Practice time

- The factorial of a non-negative integer n, denoted as $n! = 1 \times 2 \times 3 \times \cdots (n-1) \times n$
- For example, $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

- Write a function called `factorial` that takes a number as its single parameter and returns the factorial of that number.

3

Let's say you are being tasked with writing a Python function that calculates the factorial of a given number. The factorial of a non-negative integer n, denoted as $n!$, is the product of all positive integers less than or equal to n. For example, $5!$ (5 factorial) is $1 \times 2 \times 3 \times 4 \times 5 = 120$. the following question. By convention, $0!$ is defined as 1.
Can you work with the folks around you to write a function called `factorial` that takes a number as its single parameter and returns the factorial of that number.

Answer

```
def factorial_iterative(n):
    num = 1
    for i in range(2, n + 1):
        num *= i  # same as num = num * i
    return num
def factorial_iterative2(n):
    num = 1
    for i in range(n):
        num *= (i + 1)  # same as num = num * (i+1)
    return num
```

4

You probably went for one of these two options: starting at 1 and a loop from 2 up to n and multiplying the numbers. Or starting at 1 and multiplying with the next integer. Be careful with the arguments you pass to range

Recursion

- A recursive function is defined with respect to itself:
  - The function calls itself.
  - The recursive call should be on a "smaller" version of the problem.
- Can we write the factorial function recursively?
  - key idea: try and break down the problem into some computation, plus a smaller subproblem that looks similar.
    - $5! = 5 \times 4 \times 3 \times 2 \times 1$
    - $5! = 5 \times 4!$
  - How does this translate to Python code?

5

There is a third way we could write this function and it is by using recursion. A recursive function is defined with respect to itself: somewhere inside the function, the function calls itself, just like any other function call.
The recursive call should be on a "smaller" version of the problem. Can we write the factorial function recursively? The key idea is to break down the problem into some computation, plus a smaller subproblem that looks similar.
We can see that we could break 5! As 5*4!.

A first try at a recursive factorial function

```
def factorial(n):
        return n * factorial(n - 1)
```

- What happens if we call `factorial(5)`?

```
> 5*factorial(4)
  > 4*factorial(3)
    > 3*factorial(2)
      > 2*factorial(1)
        > 1*factorial(0)
          > 0*factorial(-1)
            > ….
```

  - At some point we need to stop. this is called the **base case** for recursion. When?
    - When `n == 1`.

6

A logical step would be to try writing the function factorial(n) by returning n*factorial(n-1). The problem is that our code would never terminate because we would keep calling factorial to even instances that don't make sense (-1, -2, etc). We need to stop at what is called the base case of the recursion, in our case, when n==1. Try to update the function to reflect this change

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

7

We will start exactly with this check and return 1 when we hit the base case. Otherwise, we can recursively call this function to n-1.

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

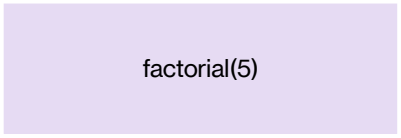
What happens in the call stack when we call `factorial(5)`?

8

What happens in the call stack when we call factorial(5)?

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```



The first stack frame for factorial(5) is added

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

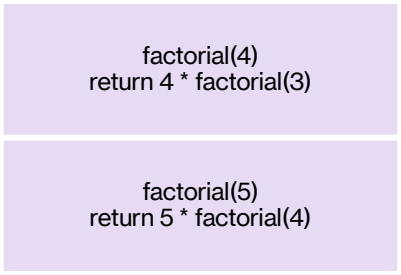
factorial(5)
return 5 * factorial(4)

10

n is substituded with 5 and the function has to return 5*factorial(4) which leads to factorial(4) being called

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

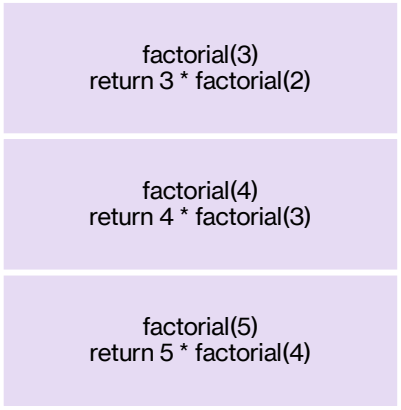


11

factorial(4) needs to call factorial(3) which adds a new stack frame

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

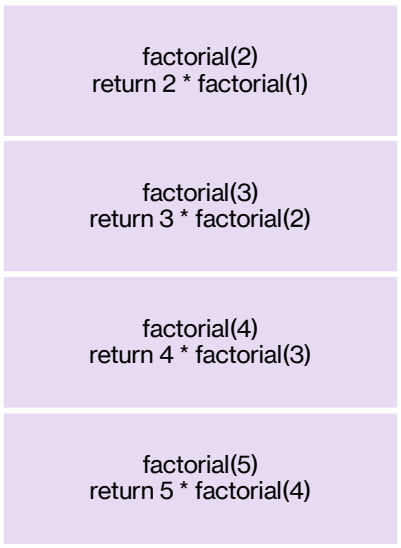


12

factorial(3) calls factorial(2)

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

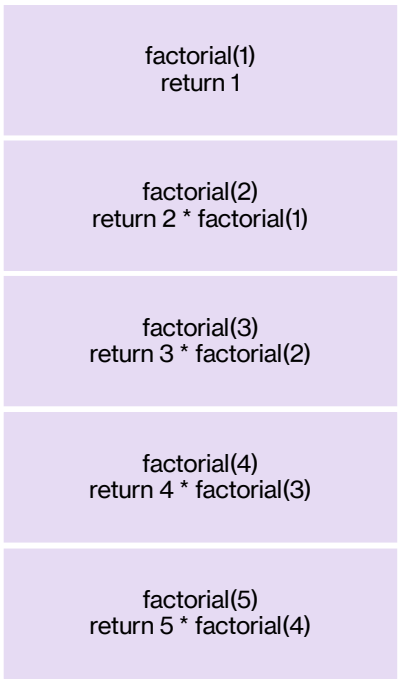


13

factorial(2) calls factorial(1) and our call stack keeps growing with each stake frame being pushed

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

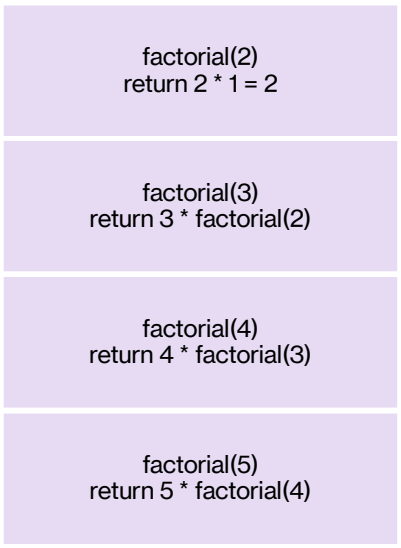


14

Our latest stack frame has been added for factorial(1) which at last gives us back 1!

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

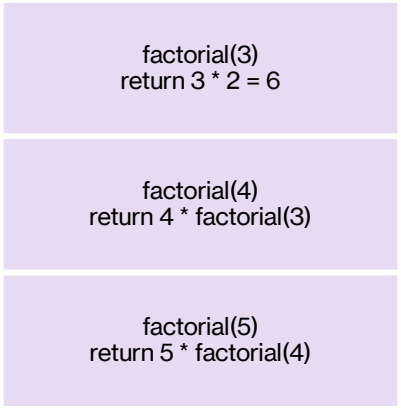


15

This results in the top of the stack being popped and factorial(2) returning2*1=2

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

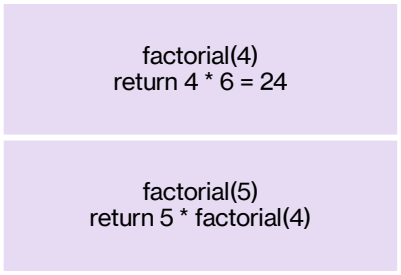


16

This results in the new top of the stack being popped and factorial(3) returning3*2=6

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```



17

This results in the new top of the stack being popped and factorial(4) returning 4*6=24

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

factorial(5)
return 5 * 24 = 120

18

This results in the new top of the stack being popped and factorial(5) returning 5*24=120

A recursive factorial function

```
def factorial(n):
    # check the base case
    if n == 1:
        return 1
    else:
        # the recursive case
        return n * factorial(n - 1)
```

19

Ultimately, there is nothing left in the stack and the function terminates returning to whomever called it 120.

Writing recursive functions

1. Define what the function the name and parameters of the function are.
2. Define the recursive case
3. Define the base case
4. Put it all together

20

To write our recursive functions we will follow four basic steps. 1) we will start by defining the header of the function, that is writing its name and parameters. 2) We will proceed with defining the recursive case. Most of the times, it will be enough to pretend that we have a working version of the function, but the function works only on smaller versions of our problem. (When I say smaller versions, I mean smaller numbers, shorter strings, shorter, lists, etc). You might also want to try writing things in plain English or even thinking in mathematical terms. 3) We then proceed with defining our base case which is the smallest/simplest problem. Putting it all together, we start by checking the base case where we return or do something for the recursive case. For the recursive case (that is, when the base case is not true), we calculate the problem using the recursive definition and return the answer.

Recursion is similar to induction

- Proof by principle of induction:
  - 1. show something works for $P(0)$ (base case).
  - 2. create an induction hypothesis for $P(k)$
  - 3. show it will work for $P(k+1)$ (recursive case)
  - 4. therefore, it must work for every n.

21

Recursion might sound familiar to induction if you have taken a proof-based course. In induction you would show something works the first time (this is your base case), you assume that it works for some time, and show that it will work for the next time (recursive case), therefore it must work for all times.

Practice time

- Write a recursive function called `rec_sum` that takes a positive number as a parameter and calculates the sum of the numbers from 1 up to and including that provided number.
- Remember the steps:

1. Define what the function the name and parameters of the function are.
2. Define the recursive case
3. Define the base case
4. Put it all together

22

Over the next slides, we will follow the same recipe. I will provide you with problems that you will solve writing recursive functions for, following the four steps we just saw. Our first problem asks us to write a recursive function that takes a positive number and calculates the sum of numbers from 1 up to and including that number.

Answer

1. Define what the function the name and parameters of the function are.
   - `def rec_sum(n)`
2. Define the recursive case
   - `rec_sum(n) = n + rec_sum(n-1)`
   - i.e. the sum of the numbers 1 through n, is n plus the sum of the numbers 1 through n-1
3. Define the base case
   - in each case, the number is getting smaller. What's the smallest number we would ever want to have the sum of?
     - 0. What's the answer when it's 0? 0!
4. Put it all together

23

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

24

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

What happens in the call stack when we call `rec_sum(3)`?

rec_sum(3)

25

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

rec_sum(3)

return 3 + rec_sum(2)

26

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

rec_sum(2)

return 2 + rec_sum(1)

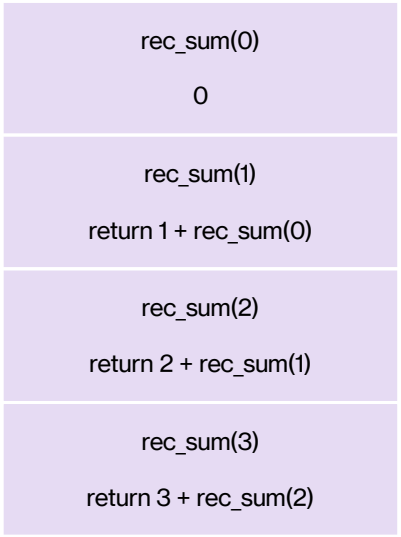
rec_sum(3)

return 3 + rec_sum(2)

27

Answer

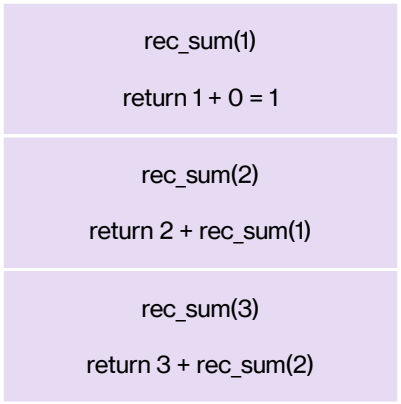
```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```



28

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```



29

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

rec_sum(2)

return 2 + 1 = 3

rec_sum(3)

return 3 + rec_sum(2)

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

rec_sum(3)

return 3 + 3 = 6

31

Answer

```
def rec_sum(n):
if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

32

Ultimately, the call stack is empty and rec_sum(3) returns 6 to whomever called it.

Practice time

- Write a recursive function `rec_binary` that takes a positive number in decimal form and returns the string representation of its binary form (unsigned).
- For example, `rec_binary(13)` should return the string '1101'
- Remember the steps:

1. Define what the function the name and parameters of the function are.
2. Define the recursive case
3. Define the base case
4. Put it all together

Answer

1. Define what the function the name and parameters of the function are.
   - `def rec_binary(n)`
2. Define the recursive case
   - `rec_binary(n) = rec_binary(n//2) + str(n%2)`

- 3. Define the base case
  - if n == 0, then the string would be '0'
  - if n == 1, then the string would be '1'

Answer

- 4. Put it all together

```
def rec_binary(n):
    if n == 0:
        return '0'
    elif n == 1:
        return '1'
    return rec_binary(n // 2) + str(n % 2)
```

35

Answer

How does this work? Let's look at an example:

```
>> rec_binary(5)
  >> rec_binary(2) + '1'
     >> rec_binary(1) + '0'
         >> '1'
      >> '1' + '0' = '1'
  >> '10' + '1' = '101'
>> '101'
```

```
def rec_binary(n):
    if n == 0:
        return '0'
    elif n == 1:
        return '1'
    return rec_binary(n // 2) + str(n % 2)
```

Practice time

- Write a recursive function called `power` that takes a base and an exponent as parameters and returns $base^{exponent}$.
- That is, it calculates base**exponent without using the ** operator. You can assume a positive exponent.
- Remember the steps:

1. Define what the function the name and parameters of the function are.
2. Define the recursive case
3. Define the base case
4. Put it all together

37

Finally, we will write a simple function that calculates the power of two numbers.

Answer

1. Define what the function the name and parameters of the function are.
   - `def power(base, exponent)`
2. Define the recursive case
   - $base^{exponent} = base^{exponent-1} * base$

3. Define the base case
   - in each case, the exponent is getting smaller.
   - Eventually, the exponent will be 0.
     - $base^0 = 1$

4. Put it all together

```
def power(base, exponent):
    if exponent == 0:
        return 1
    else:
        return base * power(base, exponent - 1)
```

38

Practice time

- Write a recursive function called `sum_of_digits` that takes a number and returns the sum of its digits.
- For example, `sum_of_digits(1234)` would return 10.

39

Finally, we will write a simple function that calculates the power of two numbers.

Answer

```
def sum_of_digits(n):
    if n < 10:
        return n
    else:
        return (n % 10) + sum_of_digits(n // 10)
```

Answer

```
def sum_of_digits(n):
    if n < 10:
        return n
    else:
        return (n % 10) + sum_of_digits(n // 10)
```

How does this work? Let's look at an example:

```
>> sum_of_digits(1234)
     >> 4 + sum_of_digits(123)
           >> 3 + sum_of_digits(12)
                 >> 2 + sum_of_digits(1)
                       >> 1
                 >> 2 + 1 = 3
           >> 3 + 3 = 6
     >> 4 + 6 = 10
>> 10
```

41