Theory: Recursion in Python

33 minutes 0 / 5 problems solved

You've already learned the theory behind **recursion**. In this topic, we're gonna put this knowledge to use by learning how to write **recursive functions**. To make sure that we are on the same page, let's remind ourselves of the definition of **recursion**.

Recursion is a method of solving a problem by breaking it down into smaller instances of the same problem. The solutions to the sub-problems are then combined to get a solution to the original problem.

§1. How to construct a recursive function?

In programming, recursion is achieved when a function **calls itself** from its own code – sounds simple, doesn't it?

The main function can generally do only two things: either perform a simple task and return a value or call itself with new arguments (by doing so dividing the problem into smaller instances of itself). Take eating candies as an example. Let's say the goal is to eat all candies in a vase. We can do this by taking one candy at a time. Eating one candy is a recursive action (**recursive case**), after which we undertake the same action the next time. We do this for every candy, evaluating that we should take another one to reach the goal, until there are no candies left.

Here's an important thing: a recursive function has to terminate to be used in a program, otherwise it can lead to an infinite loop, and the program will continue to execute forever. A condition that stops the recursion is called the **base case**.

So, there are two obligatory steps in each recursive function:

1. A **base case** works as a stop sign: it is the smallest problem that can be solved without any further subdivision. It is some condition where a function just outputs a result, there is no need for further reduction of a problem.
2. A **recursive case,** also called **a reduction step,** is the part where the function calls itself to try and solve a smaller problem.

Let's construct a recursive function for finding the factorial of an integer number in Python!

First, let's make sure it can be solved recursively:

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * 4!$$

$$4! = 4 * 3 * 2 * 1 = 4 * 3!$$

etc.

This means that we can calculate factorial of a number by multiplying this number by the factorial of the previous number:

$$N! = N * (N-1)!$$

Now, to write a recursive function we need to define the recursive case and the base case.

Defining the recursive case. We need to define a step where we're trying to find the solution to simpler instances of our problem. With the factorial of ***n,*** this would be multiplying ***n*** by the factorial of ***n-1.***

```
def recursive_factorial(n):
    # Recursive case
    return n * recursive_factorial(n-1)
```

Current topic:

Recursion in Python

Topic depends on:

- ✓ Recursion basics
- ✓ Slicing
- ✓ While loop 11★
- ✓ Algorithms in Python

Topic is required for:

Quick sort in Python

Table of contents:

Defining the base case. To avoid our function calling itself infinitely, we need to set the stopping condition or give the solution to the smallest problem. In our case, the simplest problem is *0!*.

```
def recursive_factorial(n):
    # Base case
    if n == 0:
        return 1
    # Recursive case
    else:
        return n * recursive_factorial(n - 1)
```

This is the typical structure of a recursive algorithm. If the current problem represents a simple case, solve it. If not, divide it into subproblems and apply the same strategy to them.

§2. When not to loop?

An important thing to understand is that recursion and loops are basically very similar. If we can solve a problem using recursion, we can also solve it using loops. However, they have different purposes: if loops are designed to **repeat** a task, recursion is meant to **break down** a large task into smaller ones. It is sometimes just easier to solve a problem using recursion.

Let's consider a task connected with text processing. In Italian, vowels with diacritics (special symbols like *é* or *à*) are sometimes replaced with a vowel and an apostrophe – for simplicity in typing (e.g. *e'* and *a'*). Imagine that we want to find such cases and replace them with the proper vowels; recursion can help us find the index of such an apostrophe in the word. Let's also specify that it should be neither the first nor the last symbol in the given word (otherwise it might be a quotation mark).

Our recursive function will take a word where to look for an apostrophe and a number from where to start the search, initially, zero. It will return either the index of the apostrophe (the one that meets our conditions) or *-1* if there is no such apostrophe in the word:

```
vowels = {'a', 'e', 'i', 'o', 'u'}


def find_apostrophe(word, start):

    i = word.find("'", start)

    if i == -1:  # there are no apostrophes in the given word
        return -1




 if i == 0:  # found apostrophe is the first symbol in the word, it doesn't mee
t the conditions

        return find_apostrophe(word, 1)  # keep searching further



 elif i == len(word) - 1:  # the apostrophe is the last symbol, doesn't meet th
e conditions

    return -1  # we have reached the end of the word and haven't found a corre
ct apostrophe



    else:

        previous_char = word[i - 1]

        if previous_char in vowels:

            return i



    else:  # the found apostrophe does not meet the conditions, we keep search
ing further

            return find_apostrophe(word, i + 1)
```

Now, let's analyze what each line of code does.

`string.find()` is a string method that returns either the index of a specified string we are looking for or *-1* if it couldn't find any entries of the specified string. The search starts at the element with the index `start`.

Firstly, if no apostrophe is found, our function returns *-1*.

Then, if at this step we have found an apostrophe that is the first symbol in the word (i.e. its index = 0), we don't need such apostrophe according to our conditions. So we keep searching for another one starting from the next symbol onwards. That's where **recursion** comes to play: our function performs this very task, but this time, we start from the index = 1.

However, there are a few more cases we need to handle. If the found apostrophe's index equals the length of the word ("*-1*" in the condition is needed because indexing starts from 0), it means that we have reached the end of the word and haven't found a correct apostrophe.

And if everything was okay till now, we check the last condition: the previous character should be a vowel. If the condition is met – great, we have found an apostrophe we were looking for. And if not – we keep searching, calling our own function again, with another "start" parameter.

Let's now see **how it works**.

If we call our function with the arguments *ma'ma* and *0*, everything is simple: the first and only apostrophe meets all our conditions, and the function will return *2*.

```
print(find_apostrophe("ma'ma", 0))  # 2
```

However, if we pass the string *'wave'*, everything is not going to be that simple. In the beginning, the function will find the first apostrophe; but its index = 0, so it will have to look for the next apostrophe, starting from the index = 1. Then it will find the last apostrophe, but it's also not what we are looking for. And the function will return *-1* because we have reached the end of the given word.

```
print(find_apostrophe("'wave'", 0))  # -1
```

So, as you have seen, this task can be quite easily solved with the help of recursion, and solving it using loops would be much less natural.

§3. Summing up

Recursion can seem daunting at first but there are cases where it can be amazingly helpful. Likewise, loops can also be a better choice, depending on the scenario, so knowing how to do both effectively is a great tool for a programmer.

Report a typo

146 users liked this theory. **10** didn't like it. **What about you?**

Start practicing

Comments (22) Hints (0) Useful links (1) Show discussion