TASK

Recursion

Visit our website

Introduction

Recursion is a handy programming tool that, in many cases, enables you to develop a straightforward, simple solution to an otherwise complex problem. However, it is often difficult to determine how a program can be approached recursively. In this task, we explain the basic concepts of recursive programming and teach you to “think recursively”.

WHAT IS RECURSION?

When faced with a particularly difficult or complex problem, it is often easier to break the problem down into smaller, more manageable chunks that are easier to solve. This is the basic idea behind recursion. Recursive algorithms break down a problem into smaller pieces that you already know how to solve.

In simple terms, recursion is when a function calls itself. Normally a recursive function uses conditional statements to call the function recursively or not. The main benefit of using recursion is **compact and easy-to-understand** code that has **fewer variables**. Recursion and iteration (loops) can be used to achieve the same results. However, unlike loops, which work by explicitly specifying a repetitive structure, recursion uses continuous function calls to achieve repetition.

Recursion is a somewhat advanced topic and problems that can be solved with recursion can also most likely be solved by using simpler looping structures. However, recursion is a useful programming technique that, in some cases, can enable you to develop natural, straightforward, simple solutions to otherwise difficult problems.

The following guidelines will help you to decide which method to use depending on a given situation:

- **When to use recursion:** When compact, understandable, and intuitive code is required.
- **When to use iteration:** When there is limited memory and faster processing is required.

Russian dolls as an example of recursion (**Hadiseh Aghdam**, 2020)
CC BY-SA 4.0

We can visualise recursion using Russian dolls. In the image above the dolls are created in a manner in which each doll can contain a smaller doll. This repetition, or recursion, cannot continue indefinitely and at some point the creation of a doll that fits inside the smallest doll becomes impossible. The condition of the doll no longer being able to contain a smaller doll is what is known in recursion as the **base case**.

A recursive function, similarly, follows this pattern. It calls itself within its own code until a specified condition is met (the base case), at which point the recursion ends and a result from the computation is returned.

RECURSIVE FUNCTIONS

As mentioned previously, a recursive function is a function that calls itself. For example, let's say that you have a cake that you wish to share **equally** amongst several friends. To do so, you might start by cutting the cake in half and then cutting the resulting slices in half until there are enough slices for everyone. The code to implement such an algorithm might look something like this:

```python
def cut_cake(number_of_friends, number_of_slices):
    # Cut cake in half
    number_of_slices = number_of_slices * 2

    # Check if there are enough slices for everybody
    if number_of_slices >= number_of_friends:  # Base case
        # If there are enough slices - return the number of slices
        return number_of_slices

    else:
        # If there are not enough slices - cut the resulting
        # slices in half again
        return cut_cake(number_of_friends, number_of_slices)


print(cut_cake(11, 1))
```

The `cut_cake` function takes the number of friends (11) you wish to share the cake with and the number of slices of cake (initially 1 since the cake is not yet cut).

- Line 3 cuts the cake in half.
- Line 6 then checks if there are enough slices.
- If there are enough slices, the number of slices is returned (line 8).
- If there are not enough slices, the function calls itself again (line 13) to cut the cake in half one more time.
- The new number of slices after cutting the cake (2) is passed into the function on the second function call.

This is an example of a recursive function.

Here is a table to step through this process:

Current function call	Action taken after current function call
1st function call `cut_cake(11, 1)`	New `number_of_slices` value is passed as an argument in the next recursive function call (`number_of_slices = 2`) and `number_of_friends = 11` because the number of friends stays constant.
2nd function call because `number_of_slices < number_of_friends` after previous function call `cut_cake(11, 2)`	New `number_of_slices` value is passed as an argument in the next recursive function call (`number_of_slices = 4`) and `number_of_friends = 11` because the number of friends stays constant.
3rd function call because `number_of_slices < number_of_friends` after previous function call `cut_cake(11, 4)`	New `number_of_slices` value is passed as an argument in the next recursive function call (`number_of_slices = 8`) and `number_of_friends = 11` because the number of friends stays constant.
4th function call because `number_of_slices < number_of_friends` after previous function call `cut_cake(11, 16)`	New `number_of_slices = 16` is returned because the base case is met, i.e., the number of slices is now greater than the number of friends. The recursion has now ended and the value that is printed is `16`.

FACTORIALS AND RECURSION

Many mathematical functions can be defined using recursion. A simple example is the **factorial function**. The factorial function, `n!`, describes the operation of multiplying a number by all positive integers less than or equal to itself. For example:

```
4! = 4*3*2*1
3! = 3*2*1
2! = 2*1
1! = 1
0! = 1
```

If you look closely at the examples above you might notice that `4!` can also be written as `4! = 4*3!`. In turn, `3!` can be written as `3! = 3*2!` and so on. Therefore, the factorial of a number `n` can be recursively defined as follows:

```
0! = 1
n! = n × (n − 1)! where n > 0
```

Assuming that you know `(n − 1)!`, you can easily obtain `n!` by using `n × (n − 1)!`. The problem of computing `n!` is, therefore, reduced to computing `(n − 1)!`. When computing `(n − 1)!`, you can apply the same idea recursively until `n` is reduced to `0`. The recursive function for calculating `n!` is shown below:

```python
def factorial(n):
    # Base case: if n is 0, return 1 because 0! is defined as 1
    if n == 0:
        return 1
    else:
        # Recursive case: calculate n! by multiplying n with the factorial
        # of (n - 1)
        return n * factorial(n - 1)


print(f"The factorial of 4 is {factorial(4)}")
```

If you call the function `factorial(n)` with `n = 0`, it immediately returns a result of `1`. This is the **base case** or the stopping condition. The base case of a function is the problem for which we already know the answer. In other words, it can be solved without any more recursive calls. The base case is what stops the recursion from continuing forever, and every recursive function must have at least one base case.

If you call the function `factorial(n)` with `n > 0`, the function reduces the problem into a subproblem for computing the factorial of `n - 1`. The subproblem is essentially a simpler or smaller version of the original version. Because the subproblem is the same as the original problem, you can call the function again, this time with `n - 1` as the argument. This is referred to as a recursive call. A **recursive call** can result in many more recursive calls because the function keeps on dividing a subproblem into new subproblems. For a recursive function to terminate, the problem must eventually be reduced to a base case.

In summary, there are two main requirements for a recursive function:

- **Base case:** The function returns a value when a certain condition is satisfied, without any other recursive calls.
- **Recursive call:** The function calls itself with an input that is a step closer to the base case.

RECURSION IN PYTHON

It is worth noting that recursion cannot run forever as any computer has a limited memory capacity. When a function is called in Python, the interpreter creates a new local namespace. This is so that names defined within the function do not conflict with names defined elsewhere.

Because of this, each function call in the recursive function creates a new namespace for the variables in the current function. This increases the memory used for each function call required.

To mitigate this, Python institutes a maximum limit for function calls at 1000 function calls. Imagine a recursive function that has no base case and would theoretically run forever if there was infinite memory.

In Python, you will reach the 1000-call limit and the following error will be raised and can be read in the traceback:

```
RecursionError: maximum recursion depth exceeded.
```

It is possible in Python to see the current recursion limit as well as to set the recursion limit.

```python
# To view the recursion limit, do the following:
from sys import getrecursionlimit

print(getrecursionlimit())  # Value printed would be 1000 by default
```

```python
# To change the recursion limit, do the following:
from sys import getrecursionlimit, setrecursionlimit

setrecursionlimit(2000)
print(getrecursionlimit())  # Value printed will now be 2000
```

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects. In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task 1

Follow these steps:

1. Create a file named **sum_recursion.py**.
2. Define a function which takes two arguments:
    a. A list of integers.
    b. A single integer that represents an index point.
3. The single integer will represent the index point up to which the function should sum all the numbers in the list.
    a. **Note:** List indices start at 0. The number at the specified index should be included in the calculation.
4. The function is required to sum all the numbers in the list up to and including the given index point.
5. The function should calculate the sum using recursion as opposed to using loops.

Examples of input and output:

```
adding_up_to([1, 4, 5, 3, 12, 16], 4)
=> 25
```
=> adding the numbers all the way up to index 4 (1 + 4 + 5 + 3 + 12)

```
adding_up_to([4, 3, 1, 5], 1)
=> 7
```
=> adding the numbers all the way up to index 1 (4 + 3)

Auto-graded task 2

Follow these steps:

1. Create a file named **largest_number.py**.
2. Define a function that takes a single argument:
    a. A list of integers.
3. Within the function, implement logic to find the largest number in the list.
4. The function should return the largest number found in the list.
    a. **Note:** The problem must be solved using recursion without using loops.
    b. **Additional note:** The solution should not use built-in functions such as `max()`.

Examples of input and output:

```
largest_number([1, 4, 5, 3])
=> 5
largest_number([3, 1, 6, 8, 2, 4, 5])
=> 8
```

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.