



< Previous



Next >

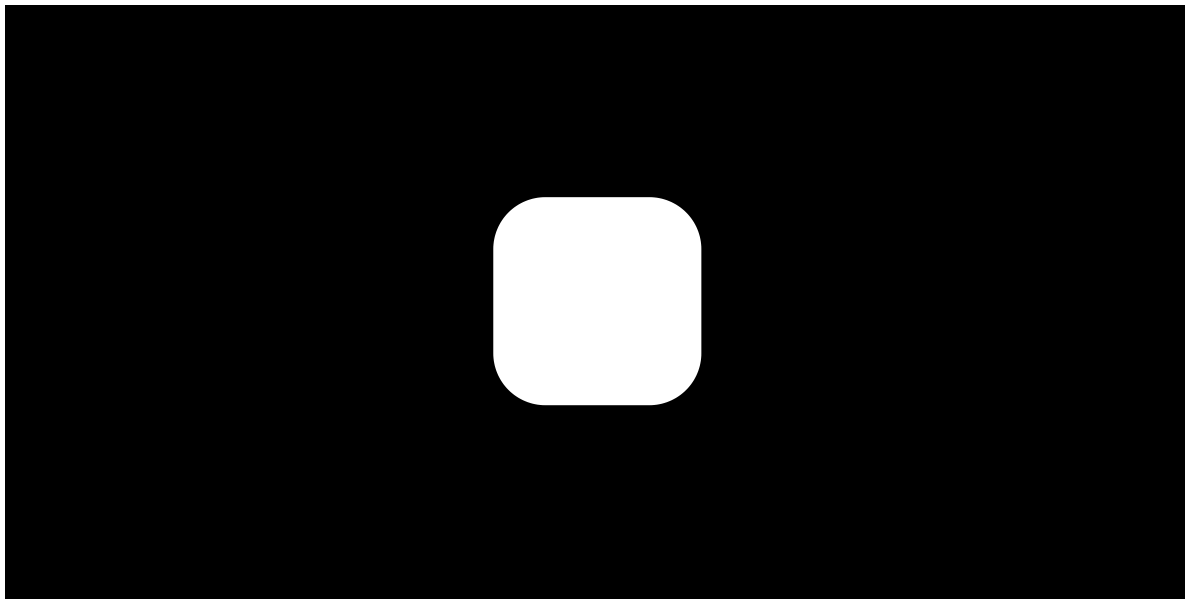
Part 3: Recursion

[Bookmark this page](#)

Part 3: Recursion

Recursion

[Start of transcript. Skip to the end.](#)



OK, now we're going to talk about recursion.

Recursion is considered a functional programming technique.

It grew up in the functional languages, the lisps, in particular.

A lot of people find it difficult to think about,

and difficult to use to solve real problems

Recursion is where a function or method calls itself, either directly or indirectly. When directly, the function simply calls itself from within itself. When indirectly, the more advanced scenario, it is called by some other function that it had already called; in other words, function a calls function b and then function b calls function a. In this tutorial, we will look at the first case, direct recursive calls.

Recursive algorithms naturally fit certain problems, particularly problems amenable to divide and conquer solutions. The general form is when a solution can be divided into an operation on the first member of a collection combined with the same operation on the remaining members of the collection.

A key element to a recursive solution involves the specification of a termination condition. The algorithm needs to know when to end and when to stop calling itself. Typically, this is when all of the members of the collection have been processed.

Python is not ideally suited to recursive programming for a few key reasons.

Python's workhorse data structure is the list and recursive solutions on list-like sequences can be attractive. However, Python lists are mutable and when mutable data structures are passed as arguments to functions they can be changed, affecting their value both inside and outside of the called function.

Clean and natural-looking recursive algorithms generally assume that values do not change between recursive calls and generally fail if they do. Attempts to avoid this problem, say by making copies of the mutable data structure to pass at each successive recursive call, can be expensive both computationally and in terms of memory consumption. Beware this scenario when designing and debugging recursive functions.

An astute observer might point out that by storing information on the stack, in successive stack frames, we are storing state, and that this is counter to functional programming's aversion to mutable state and its attraction to functional purity. Are we or are we not? The data stored on the stack during the execution of most recursive algorithms become the return values from and the arguments to successive function calls.

This results in a natural composition of functions, but rather than the composition of different functions, for instance $g(f(x))$ which is the way we normally think about functional composition, recursive algorithms represent the composition of a function with itself: $f(f(x))$. Provided we are using immutable data structures in our calls, or provided

composition of a function that returns `x(x)`. Provided we are using immutable data structures in our code, or provided we are careful not to mutate values between successive recursive calls, recursion should work.

[Previous](#)

[Next](#)

Stackframe Limits


The Python interpreter by default has its stackframe limit set to 1000. This value can be changed at runtime, but if you find you have large data sets to process, you may need to consider a non-recursive strategy. To increase the number of stack frames use `sys.setrecursionlimit` as follows:

© All Rights Reserved



© 2024 University of Washington | Seattle, WA. All rights reserved.

[Help Center](#) [Contact Us](#) [Privacy](#) [Terms](#)

Built on [OPENedX](#) by RACCOONGANG 

edX, Open edX and the edX and Open edX logos are trademarks or registered trademarks of edX Inc.