Recursion

Most modern programming languages support functional recursion using the identical mechanism that is used to support traditional forms of function calls. When one invocation of the function makes a recursive call, that invocation is suspended until the recursive call completes.

class Recursion:

    def factorial(self, num: int) -> int:

        if num == 0: # base case

            return 1

        else:

            return num \* self.factorial(num - 1)

A diagram of a number

Description automatically generated

In Python, each time a function (recursive or otherwise) is called, a structure known as an ***activation record*** or ***frame*** is created to store information about the progress of that invocation of the function. When the execution of a function leads to a nested function call, the execution of the former call is suspended and its

activation record stores the place in the source code at which the flow of control should continue upon return of the nested call. This process is used both in the standard case of one function calling a different function, or in the recursive case in which a function invokes itself. The key point is that there is a different activation record for each active call.

Understanding Recursion Limits

Python has a recursion limit (default is usually 1000) to prevent a stack overflow. This limit can be modified with **sys.setrecursionlimit()**, but it's crucial to understand the limitations of recursion in Python.

What is Recursion?

* **Definition**: A function that calls itself, creating a pattern of repetition.
* **Usage**: Effective for solving problems that can be decomposed into smaller, similar problems.
* **Example**: Navigating a file system with many branches.

Recursion vs. Iteration

* **Similar to Loops**: Recursion iterates by calling itself, similar to how a loop repeats code blocks.
* **Key Difference**: Unlike loops that use explicit iteration, recursion simplifies code by handling repetitive tasks internally through self-calls.

Coding with Recursion

* **Structure**: A recursive function usually has a base case (to stop recursion) and a recursive case (where it calls itself with modified arguments).
* **Base Case**: Prevents infinite loops by defining a condition under which recursion stops.
* **Recursive Case**: Involves the function calling itself with changed parameters to approach the base case.

Example: Factorial Calculation

* **Looping Solution**: Uses a for-loop to iteratively calculate the factorial of a number.
* **Recursive Solution**: Simpler and more compact; calculates factorial by calling itself, reducing the argument by 1 until it reaches the base case.

Understanding Recursion

* **Execution**: The function's argument is modified in each call until it meets the base case, at which point recursion stops, and the function unwinds.
* **Mechanism**: Each return statement holds a reference to its recursive call's result, contributing to the final outcome.

Advantages and Disadvantages

* **Advantages**:
  + Can simplify code for complex problems.
  + Breaks down tasks into smaller, manageable sub-problems.
  + Makes sequence generation more intuitive than using nested loops.
* **Disadvantages**:
  + Logic can be hard to follow.
  + Memory-intensive and sometimes less efficient due to stack usage.
  + Debugging and stepping through recursive code can be challenging.

Conclusion

* **Recursion** is a powerful tool for writing clean and concise code for problems that are naturally hierarchical or require repetitive computation.
* **Careful Use**: Essential to avoid infinite recursion and manage memory usage effectively.

Understanding and utilizing recursion allows for elegant solutions to problems that might be cumbersome or less intuitive with iterative approaches.

A screenshot of a computer code

Description automatically generated

Diagramming Recursions

Diagramming recursion involves visualizing the call stack and the sequence of recursive calls, which can help in understanding how a recursive function progresses through its base case(s) and recursive case(s). Here's a step-by-step guide on how to start diagramming recursion:

**1. Identify the Base Case and Recursive Case**

Before you start diagramming, understand the problem and identify the base case(s) that stop recursion and the recursive case(s) that continue the recursion.

**2. Start with the Initial Call**

Begin your diagram with the initial call to the recursive function. This is your starting point.

**3. Expand Recursive Calls**

For each recursive call:

* Draw a new level or branch in your diagram.
* Indicate the arguments passed to the recursive call.
* If the function has a return value, leave space to note what will be returned once the call completes.

**4. Mark the Base Case**

When a recursive call hits a base case:

* Clearly mark this on the diagram.
* Indicate the return value for the base case.

**5. Trace Back Up**

Once you hit a base case, trace back up through the recursive calls, filling in the return values based on your base case and any operations performed as you "return" from each recursive call.

**6. Use a Tree Structure**

For functions that make more than one recursive call (like in divide and conquer algorithms or tree traversals), use a tree structure to represent the branching of recursive calls.

**Example: Fibonacci Sequence**

Consider the Fibonacci sequence, where **Fib(n) = Fib(n-1) + Fib(n-2)** and the base cases are **Fib(0) = 0** and **Fib(1) = 1**.

To diagram **Fib(4)**, start with the initial call at the top:

Fib(4)

/ \

Fib(3) Fib(2)

/ \ / \

F(2) F(1) F(1) F(0)

/ \

F(1)F(0)

* **Base cases**: Mark **Fib(1)** and **Fib(0)** with their return values, 1 and 0, respectively.
* **Trace back up**: Combine the return values to determine the return value for each parent call until you reach the original call, **Fib(4)**.

**Tips for Diagramming Recursion**

* **Use Pencil and Paper**: This makes it easier to adjust your diagram as you go.
* **Simplify Parameters**: If the recursive function involves complex data structures, simplify them in your diagram to focus on the recursion logic.
* **Practice with Different Problems**: Try diagramming various recursive problems to get comfortable with the process.

Diagramming recursion is a powerful tool for understanding recursive algorithms, debugging, and explaining your logic to others. It can turn abstract recursive processes into tangible, visual representations that are easier to grasp and work with.