CoGrammar

The Stack and the Heap

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

Foundational Sessions Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(FBV: Mutual Respect.)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes. You can submit these questions here: **SE Open Class Questions or DS Open Class Questions**

Foundational Sessions Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: Feedback on Lectures

Reminders!

Guided Learning Hours

By now, ideally you should have 7 GLHs per week accrued. Remember to attend any and all sessions for support, and to ensure you reach 112 GLHs by the close of your Skills Bootcamp.

Progression Criteria

✅ **Criterion 1: Initial Requirements**

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

✅ **Criterion 2: Mid-Course Progress**

- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

✅ **Criterion 3: Post-Course Progress**

- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.

✅ **Criterion 4: Employability**

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.

Which of these is typically stored in the stack?

A. Objects instantiated from classes
B. Global variables
C. Local variables within a function
D. Data loaded from a database

What type of memory allocation is used for objects in Python?

A. Stack Allocation

B. Heap Allocation

C. Static Allocation

D. Register Allocation

In Python, what can lead to a Stack Overflow error?

A. Accessing an undefined variable
B. Excessive use of global variables
C. Deep recursion without a base case
D. Allocating large arrays in the heap

Recap of Graphs

CoGrammar

Graphs

- Non-linear data structure made up of **nodes** (vertices) and **edges**.

- Used to represent complex relationships between objects.

Types of Graphs

- **Undirected Graphs**: Edges have no direction.
- **Directed Graphs**: Edges have specified directions.
- **Weighted Graphs**: Edges carry weights.
- **Unlabelled vs Labelled Graphs:** Nodes or edges can carry additional information.

CoGrammar

Graph Terminology

- **Degree:** Number of edges connected to a node.
  - Directed graphs have **in-degrees** and **out-degrees** for each node.
- **Path:** Sequence of nodes connected by edges.
- **Cycle:** A path that **starts and ends at the same node**, visiting no node more than once.

CoGrammar

Implementing Graphs

- Using **dictionaries** for simple implementations.
- **Object-Oriented Programming (OOP)** for more control, using **Node and Graph classes**.
- **NetworkX library for comprehensive functionalities including visualization.**

The Stack and the Heap Topics

1. Memory Management in Python
2. The Stack
3. The Heap
4. Stack vs Heap Allocation
5. Recursion and Stack Overflow
6. Iterative vs Recursive Functions

CoGrammar

Memory Management Problem Statement

Imagine you are part of a software development team working on a machine learning project. Your task is to develop a Python application that **processes and analyzes a large dataset, potentially millions of rows, to predict customer behavior**.

➢ **Challenge:** The application needs to **handle various data transformations, algorithmic computations, and temporary data storage efficiently** without exhausting system memory or causing significant slowdowns.

- **Key Questions:**
  - How do you **manage memory** when working with such large datasets?
  - What strategies do you employ to **ensure your Python objects and variables are allocated and managed optimally**?
  - How do you mitigate issues like **memory leaks and stack overflows**, which are critical in maintaining the application's performance and reliability?

Tool for Memory Allocation Visualisation

➢ In this lecture we'll use PythonTutor in which the **stack is labelled as "Frames"** and the **heap is labelled as "Objects".** Python Tutor is useful because it visually abstracts memory management into stack and heap concepts for you to grasp how function calls and object lifetimes work in Python, even though **it is simplified** and does not show reference counting, garbage collection, and other over-complicating concepts involved.

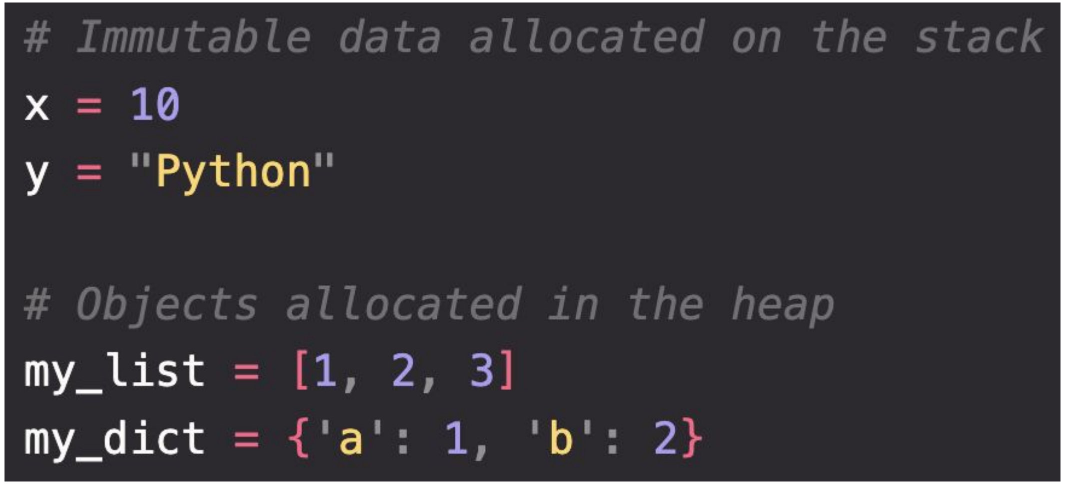
Memory Management

The function responsible for managing the computer's primary memory. It tracks the status of each memory location, either allocated or free, and decides how memory is allocated and deallocated among competing processes.

- In Python, memory is managed in two key areas: **the stack and the heap.**
- **The stack** is used for **static memory allocation**, which includes local variables and function calls.
- **The heap** is used for **dynamic memory allocation**, which is necessary for objects that need to persist outside the scope of a single function call.

CoGrammar

Example: Python Variables and Memory Allocation



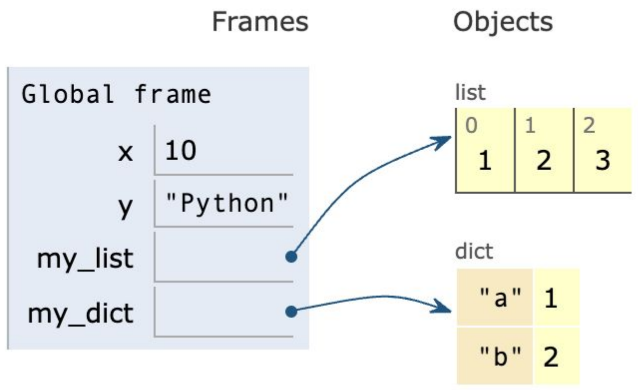
```
# Immutable data allocated on the stack
x = 10
y = "Python"

# Objects allocated in the heap
my_list = [1, 2, 3]
my_dict = {'a': 1, 'b': 2}
```

- In Python, when you create **variables** like integers or strings, the memory for these is typically allocated on the stack. **This space is automatically managed and is efficient for variables that have a short lifespan.**

- However, for more complex data structures such as **lists, dictionaries, and class instances**, Python allocates memory on the heap. **This is because these structures are usually larger and live longer than simple, stack-allocated variables.**

CoGrammar



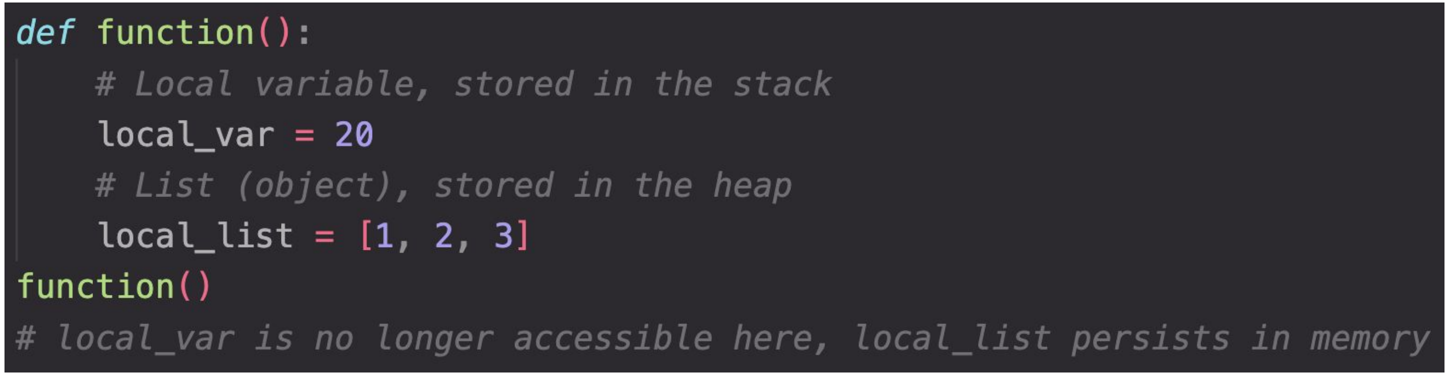
The **"Global frame"** in the visualization represents **the stack**, holding simple, quickly accessed variables like x and y, whose lifecycle is tied to their scope of declaration.

The **"Objects"** section signifies **the heap**, where complex objects such as my_list and my_dict are stored, referenced by variables in the stack and managed dynamically for persistent and flexible memory allocation.

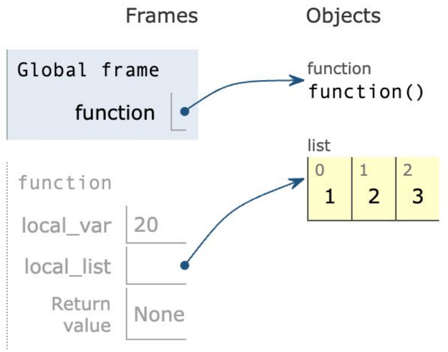
Visualising Local Variables vs Objects

- **Local variables have their space allocated on the stack.** This is ideal for **temporary information** that is only needed within the scope of a function call. **Once the function execution is over, these variables are automatically removed from the stack.**
- **Objects, however, are stored on the heap.** This memory is not managed automatically; **it persists until there are no more references to the object, at which point it can be garbage collected**. This allows for more complex data that outlives the function scope.

CoGrammar



```python
def function():
    # Local variable, stored in the stack
    local_var = 20
    # List (object), stored in the heap
    local_list = [1, 2, 3]
function()
# local_var is no longer accessible here, local_list persists in memory
```

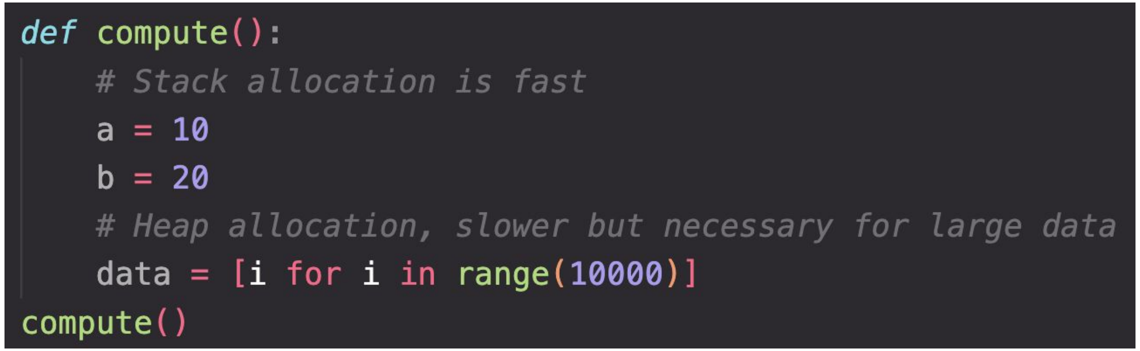


The call stack fades to indicate the completion of the function execution, leaving local_var out of scope, while the heap retains local_list, **preserving the object beyond the function's lifecycle**.

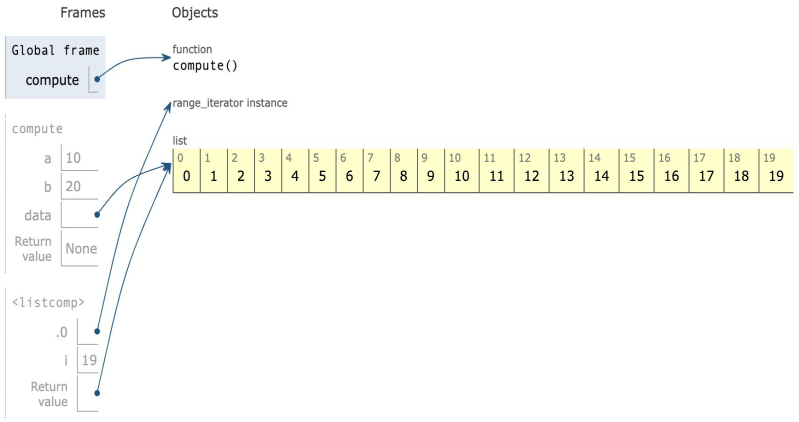
Memory Allocation: The Trade-Off Between Speed and Flexibility

- Stack memory allocation is a fast operation. **The stack works with a LIFO (Last-In-First-Out) principle, which allows for quick push and pop operations**. This makes it ideal for temporary data that has a well-defined lifespan.

- Heap memory, while more flexible, requires **dynamic memory allocation, which is a slower process. It involves finding a free block of memory large enough for the object**, which can lead to fragmentation over time.



```python
def compute():
    # Stack allocation is fast
    a = 10
    b = 20
    # Heap allocation, slower but necessary for large data
    data = [i for i in range(10000)]
compute()
```



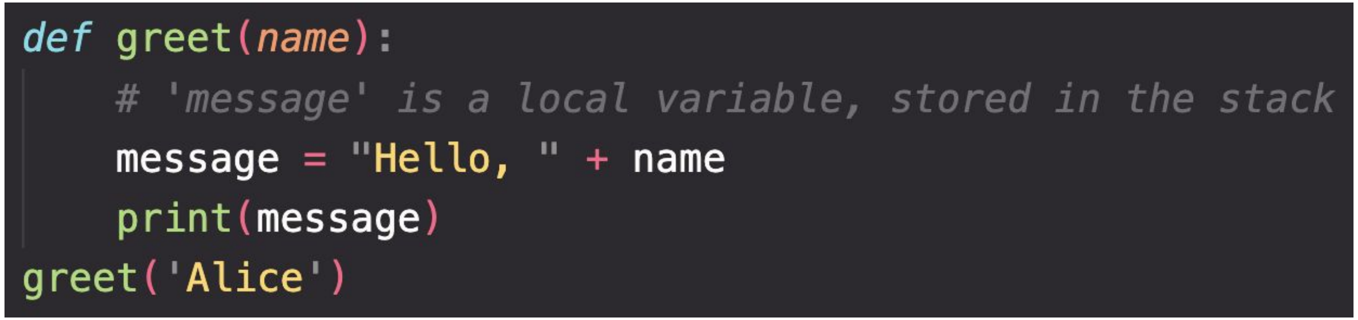
Notice how **much longer** it takes to store the data object when we walk through the storage process compared to the variables, and how **much simpler** the stack storage looks compared to the heap storage.

CoGrammar

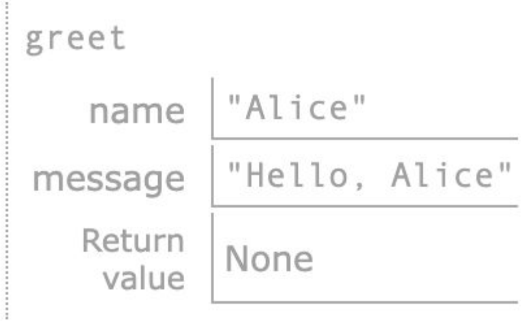
The Stack

The stack is a special area of a computer's memory which stores temporary variables created by a function.

- Python operates within a **finite memory space** and must allocate memory efficiently between the stack and heap.
- The stack is generally reserved for **smaller, short-lived data, while the heap is used for larger, longer-lived objects**.
- Understanding how Python manages memory helps developers **write more efficient code**.



```python
def greet(name):
    # 'message' is a local variable, stored in the stack
    message = "Hello, " + name
    print(message)
greet('Alice')
```



Once again we see how faded the stack becomes once it is called, showcasing that the **function's execution is complete**, and its local variables, including message, are **out of scope and their memory is cleared from the stack.**

CoGrammar

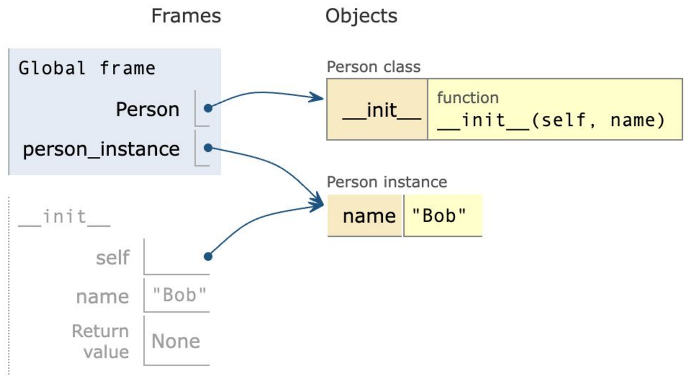
The Heap

The heap is a region of memory used by programming languages to store global variables and supports dynamic memory allocation.

- Unlike the stack, the heap is **not managed automatically by the CPU but by the Python memory manager.** When objects are created, they are placed in the heap and **remain until they are no longer needed**, at which point **the garbage collector reclaims the memory**.

CoGrammar

```python
class Person:
    def __init__(self, name):
        # 'name' attribute is stored in the heap as part of the object
        self.name = name
person_instance = Person('Bob')
```



The Person class instance **person_instance with its attribute name set to "Bob" is stored in the heap**, while the reference to this object is held in the stack within the global frame.

CoGrammar

Value Types and Reference Types

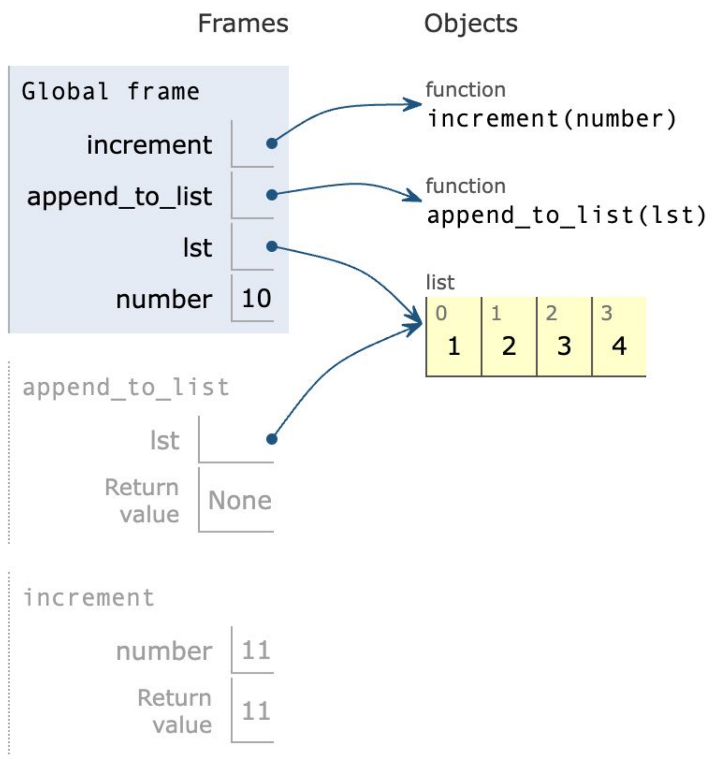
- **Value types in Python, such as numbers and booleans, are stored directly in the variable;** they are **copied** when the variable is assigned to another variable or passed to a function.

- **Reference types, such as lists and class instances, store a reference to the object's memory address.** When assigning or passing these, **only the reference is copied**, not the object itself.

```python
# Value type: Copied when passed to a function
def increment(number):
    number += 1
    return number

# Reference type: The reference is passed, original list can be modified
def append_to_list(lst):
    lst.append(4)

lst = [1, 2, 3]
append_to_list(lst)
number = 10
increment(number)
```



Notice how the **number did not change after the function was called**, because it was copied into the function as a new variable.

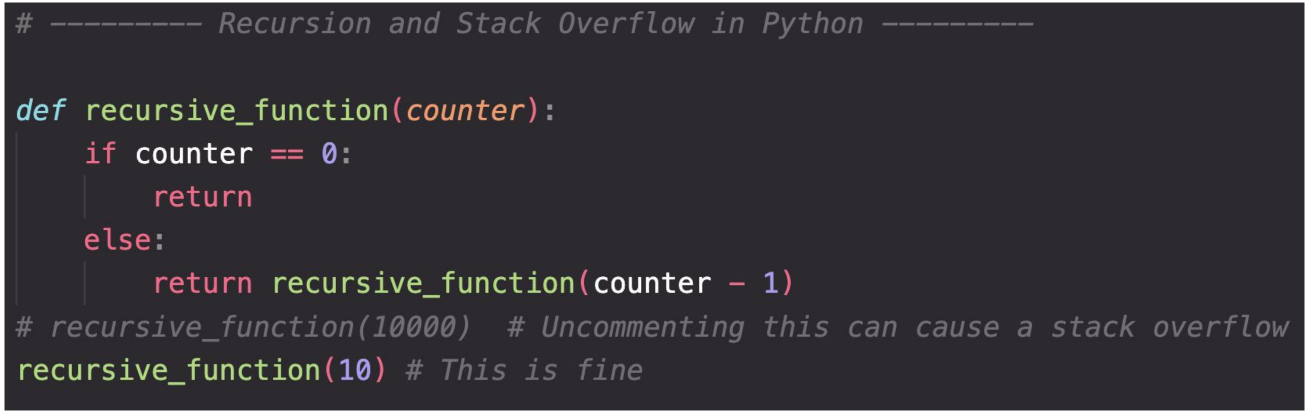
On the other hand, the reference to lst was passed into the append_to_list function, which means **the original object was changed.**

Recursion and Stack Overflow

- **Recursion** is a common technique in Python where a function calls itself.

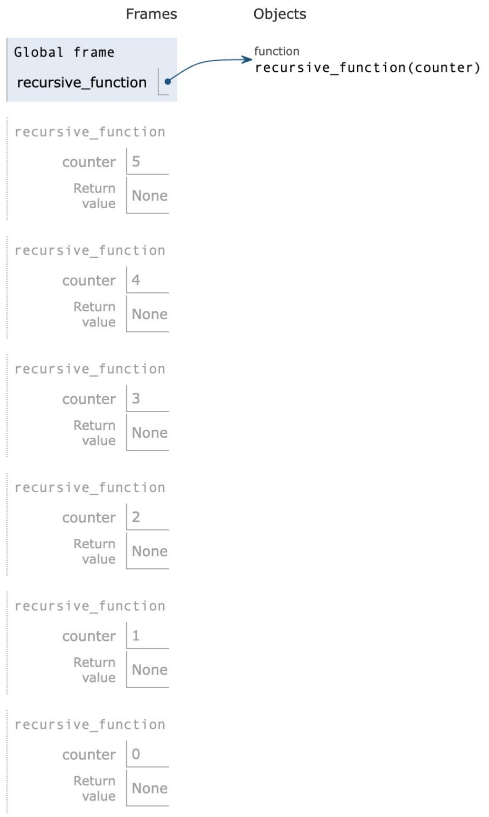
- Each call adds a new frame to the stack, which can result in a **stack overflow** if the recursion goes too deep. This is because there's a limit to the size of the stack, which, when exceeded, **causes the program to crash**.

CoGrammar



```python
# --------- Recursion and Stack Overflow in Python ---------

def recursive_function(counter):
    if counter == 0:
        return
    else:
        return recursive_function(counter - 1)
# recursive_function(10000)  # Uncommenting this can cause a stack overflow
recursive_function(10) # This is fine
```



No stack overflow occurs because the recursion depth is small, but if the commented out code with recursive_function(10000) were executed, it would **likely cause a stack overflow due to too many recursive calls exceeding the stack size limit.**

Preventing Stack Overflow

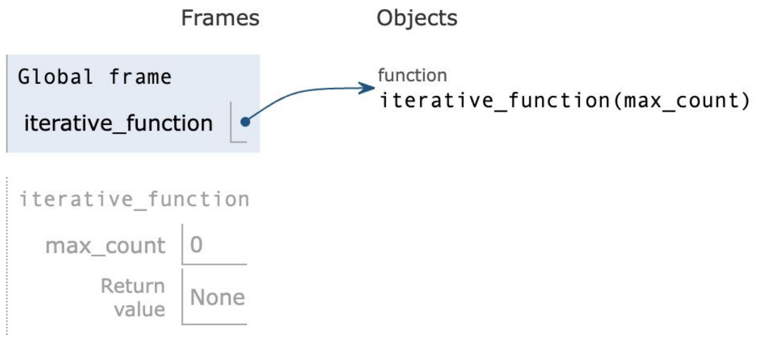
- Stack overflow can be mitigated by **increasing the maximum recursion depth using sys.setrecursionlimit()**, though this is often just a band-aid solution.

- A more robust approach is to **convert recursive algorithms into their iterative counterparts, thereby using the heap instead of the stack and avoiding the risk of overflow.**

```
import sys
sys.setrecursionlimit(10000)  # Increase the maximum recursion depth

def iterative_function(max_count):
    while max_count > 0:
        max_count -= 1

iterative_function(10000)  # No risk of stack overflow
```



Notice how the stack is not used and instead the heap is used which **takes up much less space** to prevent stack overflow.

Worked Example

You're tasked with **optimizing a Python script** that analyzes large datasets. The current script uses **recursive functions and stores processed data in a global list,** leading to performance degradation and memory issues.

1. How would you **resolve a stack overflow** caused by a recursive data processing function, and what would you do to **manage the global list results more effectively**?

2. If a large list is created within a function for temporary calculations, what steps would you take to ensure it **doesn't consume memory unnecessarily after the function's execution**?

CoGrammar

Worked Example

You're tasked with **optimizing a Python script** that analyzes large datasets. The current script uses **recursive functions and stores processed data in a global list,** leading to performance degradation and memory issues.

1. How would you **resolve a stack overflow** caused by a recursive data processing function, and what would you do to **manage the global list results more effectively**?

Convert the recursive function to an iterative one to prevent stack overflow and either clear the global list after processing or use a local variable for better memory management.

2. If a large list is created within a function for temporary calculations, what steps would you take to ensure it **doesn't consume memory unnecessarily after the function's execution**?

Use the list locally within the function, and if it must be returned, implement a generator to yield items instead of returning the whole list.

CoGrammar

Summary

Memory Management in Python

★ Python's memory management involves two key areas: **the stack for static memory allocation and the heap for dynamic memory allocation**, with mechanisms for garbage collection and reference counting to manage memory usage efficiently.

The Stack

★ The stack is a **last-in-first-out (LIFO)** structure used for storing function call information and local variables, automatically managed by the system, with **data being deleted as soon as its scope ends**.

Summary

The Heap

★ The heap is a **larger pool of memory** used for objects that outlive a single function call, such as global variables and objects created within functions, which require **manual management and garbage collection to avoid memory leaks**.

Stack vs Heap Allocation

★ Stack **allocation is quick and automatically managed**, suitable for temporary or local variables, whereas heap allocation is necessary for objects whose lifetime is longer or not easily determined, **managed by the Python memory allocator**.

Summary

Recursion and Stack Overflow

★ Recursive function **calls each add a new frame to the stack**, which can lead to a stack overflow if the recursion is too deep or improperly managed; this is mitigated by **limiting recursion depth or using alternative algorithms**.

Iterative vs Recursive Functions

★ **Iterative functions are often preferred over recursive ones for their efficiency and lower memory footprint**, particularly in languages like Python that lack tail-call optimization, thus preventing stack overflow in large computations.

Further Learning

- Real Python - Memory Management in Python
- GeeksforGeeks - Memory Management in Python
- Stack Abuse - Basics of Memory Management in Python
- Scout APM Blog - Python Memory Management: The Essential Guide

In Python, where are the references to heap-allocated objects stored?

A. In the Heap

B. In the Stack

C. In the CPU Registers

D. On the Disk

What is a primary characteristic of memory allocation in the heap compared to the stack?

A. Faster access times

B. Memory is automatically managed

C. Slower allocation and deallocation of memory

D. Only supports primitive data types

Questions and Answers

Questions around Title