Introduction to Recursion and Object-Oriented Programming

Sven Simikin
Computer Science Teaching Associate
Room S744 - ss17116@nyu.edu

Overview

25 minutes + 25 minutes

Agenda

- Iterative Programming
- Introduction to Recursion
- Object-Oriented Programming

Takeaways

- Limits of iterative programming
- How does recursion work
- What is Object-Oriented Programming

Iterative Programming

Part 1

Recap Summary

Control Structures - if, elif, and else

```
if some_condition:
   print("some_condition is True")
else:
   print("some_condition is False")
```

- Executes code based on conditions
- Directs program's execution path
- Enhances clarity and modularity

For- and While-Loops

```
while some_condition:
   print("some_condition is True")
   break
print("we are out of the loop")
```

- Repeats code multiple times
    - *for-loops* run a finite number of times
    - *while-loops* run based on a condition
- Processes elements in sequences
- Processes elements one-by-one

NYU SHANGHAI 上海纽约大学

Complexity with Nested Loops

Task: Create the sum of all elements in a *4-Dimensional* array

4D-Array

```
# Array of Shape:
# [
#     [
#         [[1, 2], [3, 4]],
#         [[5, 6], [7, 8]]
#     ],[
#         [[ 9, 10], [11, 12]],
#         [[13, 14], [15, 16]]
#     ]
# ]
```

Sum of Four Dimensional Array

So many loops!

```
sum = 0

for i in range(len(arr)):
    for j in range(len(arr[i])):
        for k in range(len(arr[i][j])):
            for l in range(len(arr[i][j][k])):
                sum += arr[i][j][k][l]


print("Sum of all elements:", sum)
```

NYU SHANGHAI 上海纽约大学

Code Readability

Task: Add some *conditions* which make your code hard to read

What!?

4D-Array

```
# Array of Shape:
# [
#     [
#         [[1, 2], [3, 4]],
#         [[5, 6], [7, 8]]
#     ],[
#         [[ 9, 10], [11, 12]],
#         [[13, 14], [15, 16]]
#     ]
# ]
```

Sum of Four Dimensional Array

What level is this?

```
sum = 0

for i in range(len(arr)):
    for j in range(len(arr[i])):
        for k in range(len(arr[i][j])):
            for l in range(len(arr[i][j][k])):
                sum += arr[i][j][k][l]
        if j >= 3:
            break
print("Sum of all elements:", sum)
```

NYU 上海 SHANGHAI 纽约大学

Limitations of Iterative Programming

Complexity with Nested Loops

- Solutions become hard to maintain
- Not suitable for complex data structures

Code Readability

- Readability of the code decreases significantly
- Making it harder to understand and extend

Memory Management

- We won't discuss memory management at this point

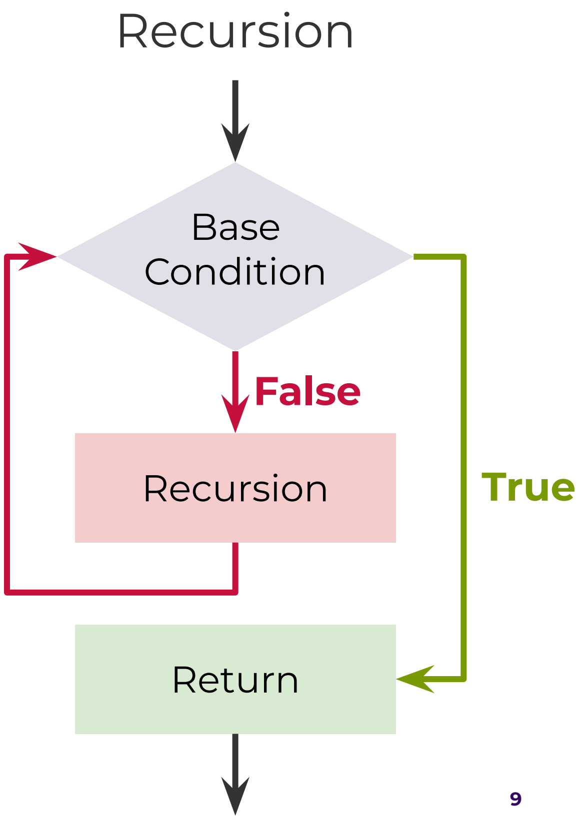
Introduction to Recursion

Part 2

Recursion Definition

“Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem.”

Idea: We are solving a problem through repeated calls of the same function.



Simple Example

Question: How to solve a Jigsaw puzzle?

Intuitive Algorithm - Solve the Puzzle

- Is the puzzle complete? If so, stop
- Find a puzzle piece and place it
- *Solve the Puzzle*

Recursive Functions

Each function call changes the parameter until the base-condition is meet

Recursive functions consist of

- a recursive-step through repeated function calls with new parameters
- one or many conditions to terminate the recursion and return a solution

Recursive Code

```
def some_function(n):
    if n == 0:
        return          # base-case

    print("Hello, world!")
    some_function(n - 1)  # recursive-step


def main():
    some_function(5)

main()
```

Some Function

Iterative Workflow

1. Invoke main
2. Invoke some_function with parameter **5**

Recursive Workflow

3. Invoke some_function with parameter **4**
4. Invoke some_function with parameter **3**
5. Invoke some_function with parameter **2**
6. Invoke some_function with parameter **1**
7. Invoke some_function with parameter **0**

Recursive Code

```
def some_function(n):
    if n == 0:
        return

    print("Hello, world!")
    some_function(n - 1)


def main():
    some_function(5)

main()
```

call itself

return

Interchangeability

Recursive Code

```
def some_function(n):
   if n == 0:
       return

   print("Hello, world!")
   some_function(n - 1)


def main():
   some_function(5)

main()
```

Iterative Code

```
def some_function(n):
   while n > 0:
      print("Hello, world!")
      n = n -1




def main():
   some_function(5)

main()
```

Why Recursion?

Every recursive implementation can be turned into a iterative solution

- Input to a recursive problem is always a smaller version of the same problem
- The length of code for recursive solutions compared to iterative solutions significantly decreases for complex problems

Power Function

Task: Write a recursive function that calculates ***x*** to the power of ***n***.

Basic Understanding

We need to analyse how *power* works

- $x^0 = 1$
- $x^n = x * x^{n-1}$

x and **n** will be positive numbers

Implementation

```
def power(x, n):

    if n == 0:
        return 1

    return x * power(x, n - 1)
```

Power Function

Call the function: `print(power(5, 3))`

Call Stack

```
4 call: power(5, 0)

3 call: power(5, 1)

2 call: power(5, 2)

1 call: print(power(5, 3))
```

```
# first call: power(5, 3)
```

```
# second call: power(5, 2)
```

```
# third call: power(5, 1)
```

```
# fourth call: power(5, 0)
def power(x, n):

    if n == 0:
        return 1

    return x * power(x, n - 1)
```

NYU 上海 SHAI 纽约

Power Function

Concept: Each previous call waits for the next call to finish

Call Stack

```
4 call: power(5, 0) # returns 1

3 call: power(5, 1)

2 call: power(5, 2)

1 call: print(power(5, 3))
```

This call will return 1

```
# first call: power(5, 3)
```

```
# second call: power(5, 2)
```

```
# third call: power(5, 1)
```

```
# fourth call: power(5, 0)
def power(x, n):

    if n == 0:
        return 1

    return x * power(x, n - 1)
```

Power Function

Concept: Each previous call waits for the next call to finish

Call Stack

```
4 call: power(5, 0) # returns 1

3 call: power(5, 1) # returns 5

2 call: power(5, 2)

1 call: print(power(5, 3))
```

```
# first call: power(5, 3)
```

```
# second call: power(5, 2)
```

```
# third call: power(5, 1)
def power(x, n):

    if n == 0:
        return 1

    return x * power(x, n - 1)
```

Equals 1 from previous call

This entire statement returns 5 * 1

Power Function

Concept: Each previous call waits for the next call to finish

Call Stack

```
4 call: power(5, 0) # returns 1

3 call: power(5, 1) # returns 5

2 call: power(5, 2) # returns 25

1 call: print(power(5, 3))
```

```
# first call: power(5, 3)
```

```
# second call: power(5, 2)
def power(x, n):

    if n == 0:
        return 1

    return x * power(x, n - 1)
```

Equals 5 from previous call

This entire statement returns 5 * 5

Power Function

Concept: Each previous call waits for the next call to finish

Call Stack

```
4 call: power(5, 0) # returns 1

3 call: power(5, 1) # returns 5

2 call: power(5, 2) # returns 25

1 call: print(power(5, 3)) # returns 125
```

```python
# first call: power(5, 3)
def power(x, n):

    if n == 0:
        return 1

    return x * power(x, n - 1)
```

Equals 25 from previous call

This entire statement returns 5 * 25

1D-Array-Sum Recursive

Task: Create the sum of all elements in a *1-Dimensional* array

1D-Array

```
# Array of Shape:
# [1, 2, 3, 4, 5, 6, 7]
#
# The sum of this array is:
# 1 + 2 + 3 + 4 + 5 + 6 + 7
# The expected result is 28
```

1-D Recursion

```python
def sum_arr(arr):
    if not arr:
        return 0
    else:
        return arr[0] + sum_arr(arr[1:])

print("Sum of all elements:", sum_arr(arr))
```

String of Digits

Task: Convert a string of digits into a comma-separated format

String of Digits

```
# input = "1234567"
#
# The expected result is:
# result = "1,234,567"
```

Approach

- Develop the base case
- Develop the recursive step

Recursion Implementation

```
def format(s):
    if len(s) <= 3:
        return s
    else:
        return format(s[:-3]) + ',' + s[-3:]

print("The result is:", format(input))
```

NYU SHANGHAI 上海纽约大学

nD-Array-Sum Recursive

Task: Create the sum of all elements in a *n-Dimensional* array

Any-Array

```
# Array of Shape:
# [
#     [1, 2, 3],
#     [4, 5, 6],
#     [7, 8, 9]
# ]
```

Any-Dimensional Recursion

```python
def sum_arr(arr):
    if not arr:
        return 0
    if isinstance(arr[0], list):
        return sum_arr(arr[0]) + sum_arr(arr[1:])
    else:
        return arr[0] + sum_arr(arr[1:])

print("Sum of all elements:", sum_arr(arr))
```

Difficult Recursion

There is no template for creating recursive algorithms!

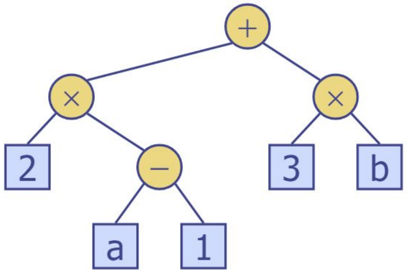
Guidelines for Creating Recursive Solutions

1. There must be a case for all valid inputs
2. There must be a case that makes no recursive call
3. When making recursive calls, the call should be to a simpler instance and make forward progress towards the base-case

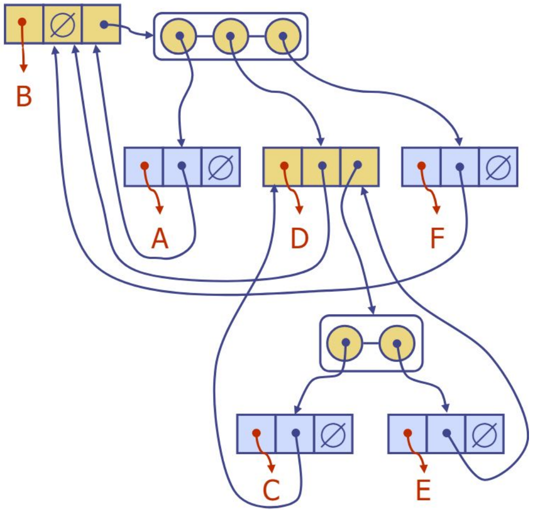
Recursion-Required Solutions

Some problems require recursive algorithms for (*efficient*) processing

Tree Structures



Linked Structures



Other Algorithms/Structures

- Graph-Algorithms
- Traversal-Algorithms
- Optimisation Problems
- Dynamic Data Structures
- Handling Infinite Seq.
- Structures with unknown properties and attributes
- ...

Reinforcement

Requirement: You have to use recursion to solve the below problems.

Flatten Array

Convert a multi-dimensional array into a single-dimensional array, combining all elements into a single list without any nested structure.

Input: A nested array containing integers or other nested arrays (e.g., `[1, [2, [3, 4], 5], 6]`).

Output: A flat array containing all elements in a single dimension (e.g., `[1, 2, 3, 4, 5, 6]`).

Calculate Average

Compute the average of a list of numbers.

Input: A list of numeric values of arbitrary length (e.g., `[10, 20, 30, 40]`).

Output: A single numeric value representing the average of the input list (e.g., 25.0).

Object-Oriented Programming

Part 3

Iterative vs OOP

Iterative Implementation

Main Function

Functions for Vehicle 1

Iterative vs OOP

Iterative Implementation

Main Function

Functions for Vehicle 1

Functions for Vehicle 2

Iterative vs OOP

Iterative Implementation

Main Function

Functions for Vehicle 1 

Functions for Vehicle 2 

Functions for Vehicle 3 

Iterative vs OOP

Iterative Implementation

Main Function

Functions for Vehicle 1

Functions for Vehicle 2

Functions for Vehicle 3

Functions for Vehicle …

Iterative vs OOP

Iterative Implementation

Main Function

Functions for Vehicle 1 

Functions for Vehicle 2 

Functions for Vehicle 3 

Functions for Vehicle ...

OOP Implementation

Blueprint for Vehicle
- properties
- functions (behaviour)

Iterative vs OOP

Iterative Implementation

Main Function

Functions for Vehicle 1 

Functions for Vehicle 2 

Functions for Vehicle 3 

Functions for Vehicle ...

OOP Implementation

Blueprint for Vehicle
- properties
- functions (behaviour)

instance 1

instance 3

instance 2

OOP Definition

"OOP is a programming paradigm that uses objects and their interactions to design applications and computer programs. ***Objects*** *are instances of* ***classes****, which define the* ***properties*** *and* ***behaviors*** *that the objects encapsulate."*

```
<<Class>>
Vehicle
---------------------------
- name: string
- speed: number
- color: string
---------------------------
+ accelerate(self)
+ decelerate(self)
```

Classes

A class is a blueprint for creating objects. Classes defines the structure and behaviors that its objects will have.

Example - Vehicle

The diagram describes a vehicle that has a *name*, *speed*, and *color*, as well as behaviours to *accelerate* and *decelerate*.

```
<<Class>>
 Vehicle
-----------------------
- name: string
- speed: number
- color: string
-----------------------
+ accelerate(self)
+ decelerate(self)
```

Class Implementation

```python
class Vehicle:
    def __init__(self, name, speed, color):
        self.name = name
        self.speed = speed
        self.color = color

    def accelerate(self):
        self.speed += 10
        print(f"{self.name} accelerates to {self.speed} km/h")

    def decelerate(self):
        self.speed -= 10
        print(f"{self.name} decelerates to {self.speed} km/h")
```

This method
will help us later

<<Class>> Vehicle
- name: string - speed: number - color: string
+ accelerate(self) + decelerate(self)

Objects (Instances)

An object in is an instance of a class. Objects are created from their blueprint with concrete values and are able to perform specific actions defined by its methods.

Example - Vehicle

```
red_car = Vehicle("Fast Car",  200, "red")
yellow_car = Vehicle("Slow Car",  80, "yellow")
blue_car = Vehicle("Okay Car",  200, "blue")

red_car.accelerate()
yellow_car.decelerate()
blue_car.accelerate()
```

Accessing Objects

We can access the properties of our object at any time.

```
red_car = Vehicle("Fast Car",  200, "red")
yellow_car = Vehicle("Slow Car",  80, "yellow")
blue_car = Vehicle("Okay Car",  200, "blue")

red_car.accelerate()
yellow_car.decelerate()
blue_car.accelerate()

print(red_car.name) # will print: Fast Car
print(yellow_car.speed) # will print: 70
print(blue_car.speed) # will print: 210
```

Class Initializer

```python
class Vehicle:
    def __init__():
        self.name = "Fast Car"
        self.speed = 200
        self.color = "red"

    def __init__(self, name, speed, color):
        self.name = name
        self.speed = speed
        self.color = color

    def accelerate(self):
        # hidden
    def decelerate(self):
        # hidden
```

Creating Instances of Classes

```python
a_fast_car = Vehicle()
another_fast_car = Vehicle()
more_fast_car = Vehicle()
new_fast_car = Vehicle()

a_fast_car.decelerate() # speed: 190
another_fast_car.accelerate() # speed: 210
more_fast_car.accelerate() # speed: 210
new_fast_car.decelerate() # speed: 190

super_car = Vehicle("Super",  2000, "red")
super_car .accelerate() # speed: 2010
```

Design Principles

Encapsulation

Logically group related information and functions into one unit and defining where that information is accessible.

Example - Student Class

Group information such as *name*, *address*, *grades*, and provide functionalities to calculate the *GPA*.

Abstraction

Abstraction focuses on what an object does instead of how it achieves what it does. Abstraction involves hiding the complex reality while exposing only the necessary parts.

Example - Remote Control Power Button

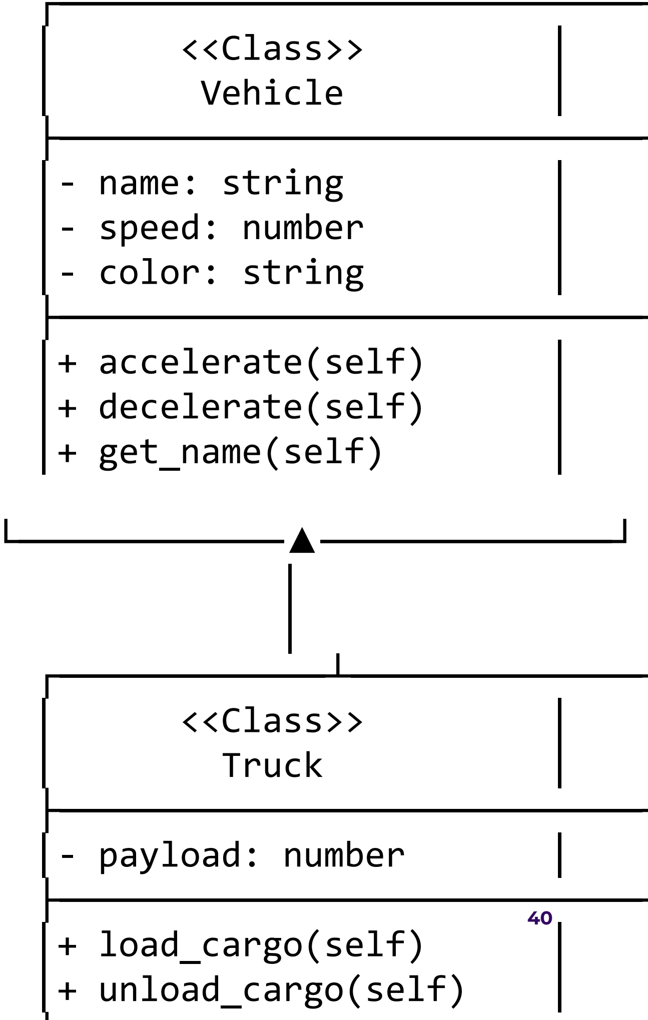
Press the power button to turn the TV on or off without worrying about the electrical mechanisms involved.

Inheritance

Objects can inherit the attributes and behaviour of parent classes. The Vehicle is the parent of the Truck.

If we create a instance of a Truck, the truck will possess all attributes of the vehicle class, be able to accelerate and decelerate, and additionally hold cargo.

```
brown_truck = Truck("Heavy Truck",  70, "brown", 1000)
brown_truck.load_cargo()
brown_truck.accelerate()
```



OOP - Blackmagic!

Object-oriented programming (OOP) is a comprehensive and versatile paradigm that structures software design around data, or objects, rather than functions and logic.

Interfaces - Abstract Classes - Visibility Modifiers - Instance and Class Variables - Primitive and Object Types - Scopes - Enums - Constructors and Destructors - Threading - Exception Handling - Garbage Collection - References - Pointers - Static Variables and Methods - Passing Objects

Takeaways

Part 4

Takeaways

Iterative Programming has limitations

→ Not all problems can be solved (*efficiently)* iteratively

Recursion uses repeated method calls on itself

→ Recursion divides problems into subproblems to form solutions

Object-Oriented Programming introduces Objects

→ Classes are blueprints for objects that have attributes and behaviour

Thank you for your Attention!

Sven Simikin

Computer Science Teaching Associate

Room S744 - ss17116@nyu.edu