TA Class of Python and Algorithm

TA: Dongchen(Felix) HE

Contact: dc23.he@gmail.com

A Brief Review of Lecture One

1. course requirements
2. recursive formulas
3. python basic variables
4. python functions
5. function stack frames * (optional)
6. python List and built-in function append
7. Tower of Hanoi problem

Recursive Formulas

Introduction

- Recursive formulas define a sequence or function in terms of previous terms or values.
- They involve self-referential relationships.
- Widely used in mathematics and computer science.

Recursive Formula Examples

Fibonacci Sequence

- The Fibonacci sequence is defined recursively as:
    - $F(0) = 0$
    - $F(1) = 1$
    - $F(n) = F(n-1) + F(n-2)$, for $n >= 2$

Factorial Function

- The factorial function is defined recursively as:
    - $0! = 1$
    - $n! = n * (n-1)!$, for $n > 0$

Mathematical Notation

- Recursive formulas can be represented using mathematical notation.
- Example: The Fibonacci sequence
    - $F(0) = 0$
    - $F(1) = 1$
    - $F(n) = F(n-1) + F(n-2)$, for $n >= 2$

Python Code Examples

Fibonacci Sequence (Recursive)

```python
def fibonacci_recursive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Factorial Function (Recursive)

```
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n-1)
```

Python Code Examples (Iterative)

Fibonacci Sequence (Iterative)

```python
def fibonacci_iterative(n):
    if n <= 1:
        return n
    else:
        a, b = 0, 1
        for _ in range(2, n+1):
            a, b = b, a + b
        return b
```

Factorial Function (Iterative)

```python
def factorial_iterative(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

Conclusion

1. Recursive formulas are powerful tools for defining sequences and functions.
2. They offer elegant solutions to various mathematical and computational problems.
3. Python provides both recursive and iterative approaches for implementing them.

Python Basic Variables

Introduction

- Variables are used to store data in computer programs.
- In Python, variables are dynamically typed, meaning they can hold values of different types.

Variable Declaration and Assignment

- In Python, variables are declared and assigned values in a single step.
- Example: Assigning a value to a variable named `x`
    - `x = 5`

Data Types

Numeric Types

- `int`: integers (e.g., `42`, `-10`, `0`)
- `float`: floating-point numbers (e.g., `3.14`, `1.0`, `-2.5`)

String Type

- `str`: sequences of characters (e.g., `"Hello"`, `'World'`, `"42"`)

Boolean Type

- `bool`: representing truth values (`True` or `False`)

Variable Naming

- Variables in Python must follow certain naming conventions.
- Rules:
  - Must start with a letter or underscore.
  - Can contain letters, numbers, and underscores.
  - Case-sensitive (`myVar` and `myvar` are different variables).

Variable Examples

Numeric Variables

```
x = 42
y = 3.14
```

String Variables

```
name = "John Doe"
message = 'Hello, World!'
```

Boolean Variables

```
is_active = True
has_permission = False
```

Variable Reassignment

Variables can be reassigned new values at any time.

Example:

```
x = 5
x = x + 1  # x is now 6
```

Variable Interpolation

Variables can be used within strings using f-strings.

Example:

```
name = "Alice"
age = 25
message = f"My name is {name} and I'm {age} years old."
```

Conclusion

1. Python allows dynamic typing, enabling variables to hold different types of data.
2. Variables are declared and assigned values in a single step.
3. Following naming conventions and reassigning variables helps in program flexibility and readability.

Python Functions

Introduction

- Functions are reusable blocks of code that perform specific tasks.
- They help modularize code, improve readability, and promote code reuse.

Defining Functions

- In Python, functions are defined using the `def` keyword, followed by the function name and parentheses.
- Example:

```python
def greet():
    print("Hello, world!")
```

Function Parameters

- Functions can have parameters, which are placeholders for values passed into the function.
- Parameters are specified within the parentheses after the function name.
  Example:

```python
def greet(name):
    print(f"Hello, {name}!")
```

Function Return Values

- Functions can return values using the return statement.
- The returned value can be assigned to a variable or used directly.

  Example:

```
def add(a, b):
    return a + b
```

Calling Functions

- Functions are called by using their name followed by parentheses.
- Arguments can be passed into the function within the parentheses.

  Example:

```
greet("Alice")
result = add(3, 5)
```

Default Parameter Values

- Functions can have default parameter values.
- If a value is not provided for a parameter, the default value is used.

  Example:

```python
def power(base, exponent=2):
    return base ** exponent

# Calling the function:

result1 = power(3)       # 3^2 = 9
result2 = power(3, 4)    # 3^4 = 81
```

Variable Number of Arguments

- Functions can accept a variable number of arguments using *args or **kwargs.
- *args collects positional arguments into a tuple.
- **kwargs collects keyword arguments into a dictionary.

  Example:

```python
def sum_values(*args):
    return sum(args)

# Calling the function:
result = sum_values(1, 2, 3, 4, 5)   # 1 + 2 + 3 + 4 + 5 = 15
```

Conclusion

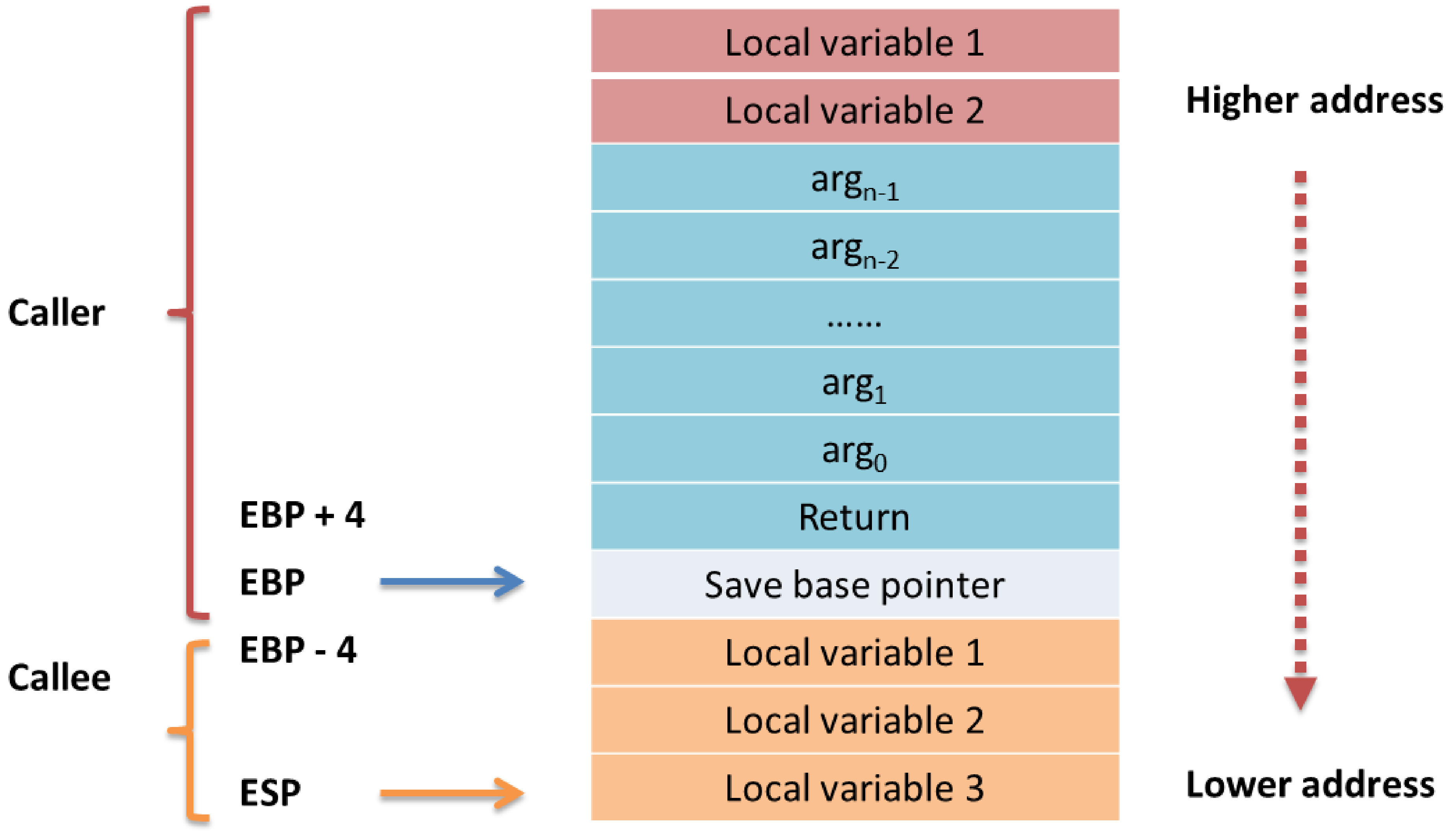
- Functions provide a way to organize and reuse code in Python.
- They can have parameters and return values.
- Default parameter values and variable number of arguments increase flexibility.

Function Stack Frames * (optional)

Introduction

- Function stack frames are used to manage function calls and their local variables.
- Each function call creates a new stack frame that contains information specific to that call.

Function Call and Stack Frames



Stack Frame Components

- Each stack frame consists of the following components:

1. **Function Arguments**: Values passed to the function.
2. **Local Variables**: Variables defined within the function.
3. **Return Address**: Address to return to after the function call.
4. **Caller's Stack Frame**: Stack frame of the calling function.

Example: Recursive Factorial Function

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
```

Conclusion

- Function stack frames are used to manage function calls and local variables.
- Each function call creates a new stack frame with its own components.
- Understanding stack frames helps in tracing program execution and debugging.

Python Lists and the `append()` Function

Introduction

- Lists are versatile data structures in Python that can hold multiple values.
- The `append()` function is used to add elements to a list.

List Creation

- Lists are created using square brackets `[]` or the `list()` function.
- Example:

```
my_list = [1, 2, 3, 4, 5]
```

Accessing List Elements

- List elements can be accessed using index notation.
  Indexing starts from 0 for the first element.
- Example:

```
first_element = my_list[0]
second_element = my_list[1]
```

Modifying List Elements

- List elements can be modified by assigning new values.
- Example:

```
my_list[0] = 10
```

List Length

- The length of a list can be determined using the len() function.
- Example:

```
length = len(my_list)
```

List Append Operation

- The append() function is used to add elements to the end of a list.
- Example:

```
my_list.append(6)
```

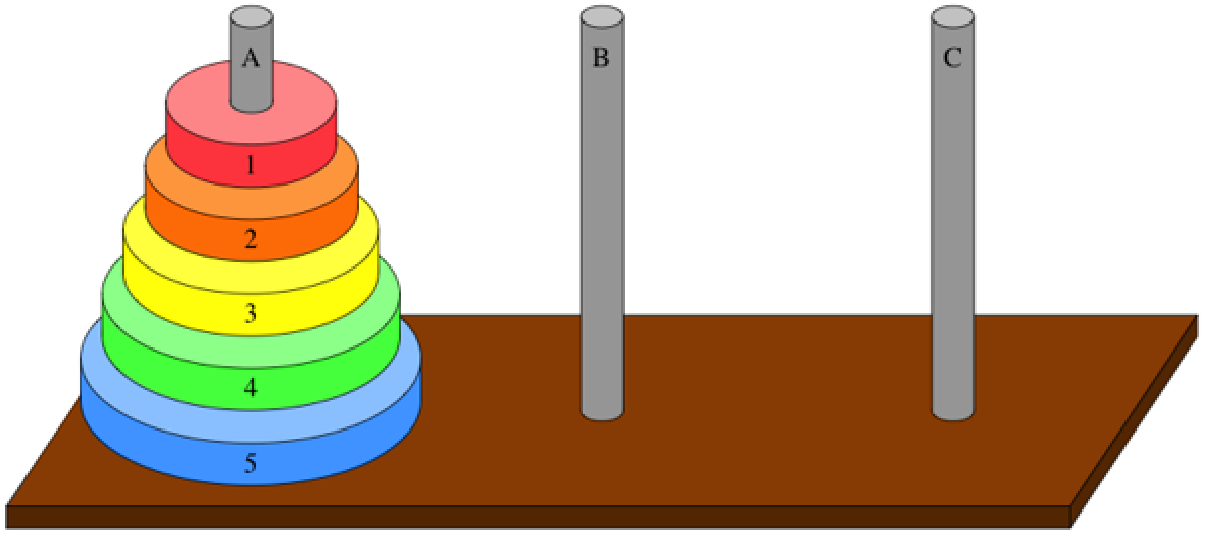
List Append Operation

- The append() function can be used to add multiple elements.
- Example:

```
my_list.append(7)
my_list.append(8)
```

Conclusion

- Lists are powerful data structures for storing and manipulating multiple values in Python.
- The append() function is a convenient way to add elements to the end of a list.



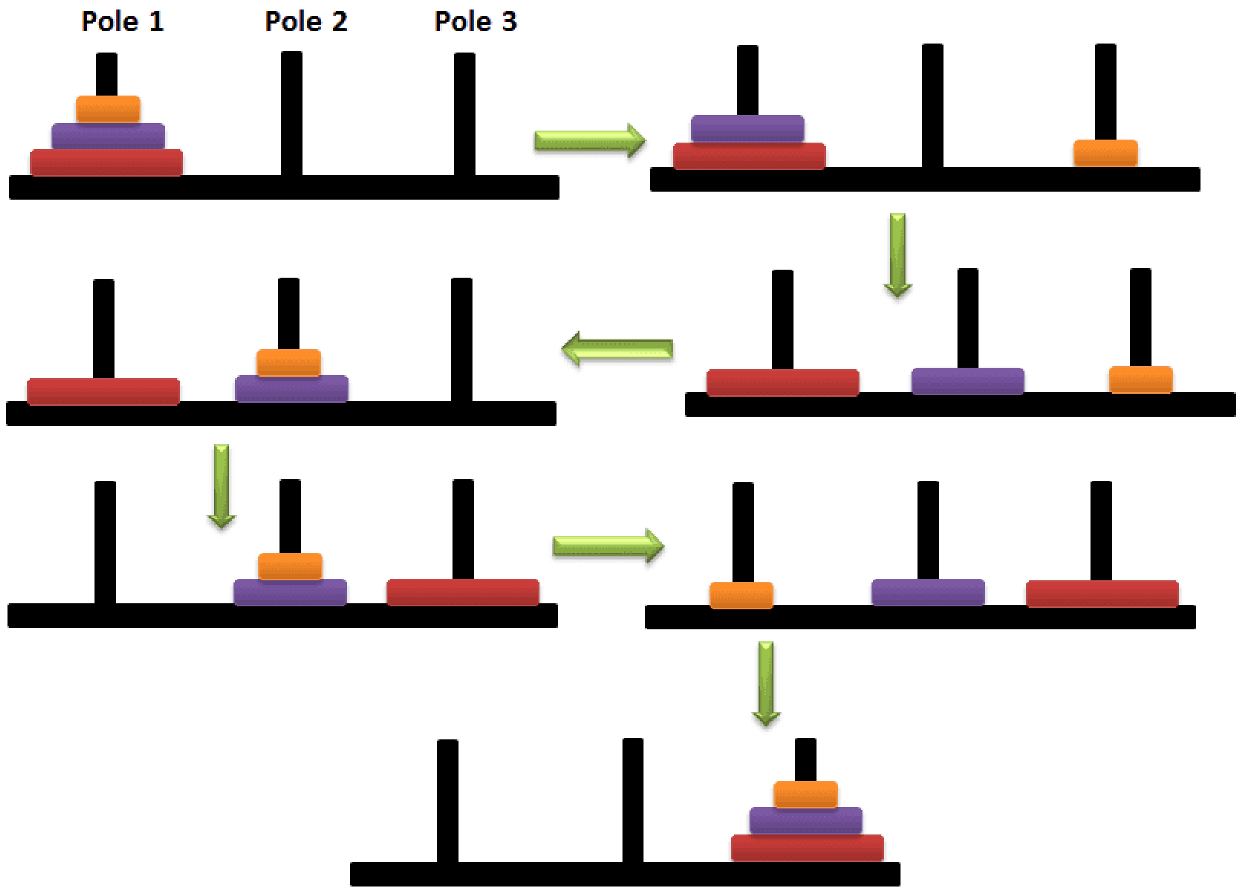
Tower of Hanoi Problem

Introduction

- The Tower of Hanoi is a classic mathematical puzzle.
- It consists of three pegs and a set of disks of different sizes.
- The goal is to move all the disks from one peg to another, following specific rules.

Rules of the Tower of Hanoi

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
3. A larger disk cannot be placed on top of a smaller disk.



Example

- Given:
    - Three pegs: A, B, and C.
    - Initially, all disks are on peg A, in decreasing order of size.

Recursive Solution

- Recursive algorithm to solve the Tower of Hanoi problem:

```python
def tower_of_hanoi(n, source, target, auxiliary):
    if n > 0:
        tower_of_hanoi(n - 1, source, auxiliary, target)
        print(f"Move disk from {source} to {target}")
        tower_of_hanoi(n - 1, auxiliary, target, source)
```

```python
tower_of_hanoi(3,'tower1','tower3','tower2')
```

```
Move disk from tower1 to tower3
Move disk from tower1 to tower2
Move disk from tower3 to tower2
Move disk from tower1 to tower3
Move disk from tower2 to tower1
Move disk from tower2 to tower3
Move disk from tower1 to tower3
```

Recursive Solution

- The algorithm moves n disks from the source peg to the target peg using the auxiliary peg.
- It recursively solves smaller instances of the problem by moving n-1 disks to the auxiliary peg.
- Then it moves the largest disk from the source peg to the target peg.
- Finally, it moves the n-1 disks from the auxiliary peg to the target peg.

Complexity Analysis

- The time complexity of the Tower of Hanoi algorithm is O(2^n).
- As the number of disks increases, the number of moves grows exponentially.

Conclusion

- The Tower of Hanoi problem is a classic puzzle that requires careful thinking and recursive strategies to solve.
- Recursive algorithms can be elegant and intuitive for solving complex problems.
- The Tower of Hanoi problem demonstrates the power and efficiency of recursion.

Thank You