Hacettepe University

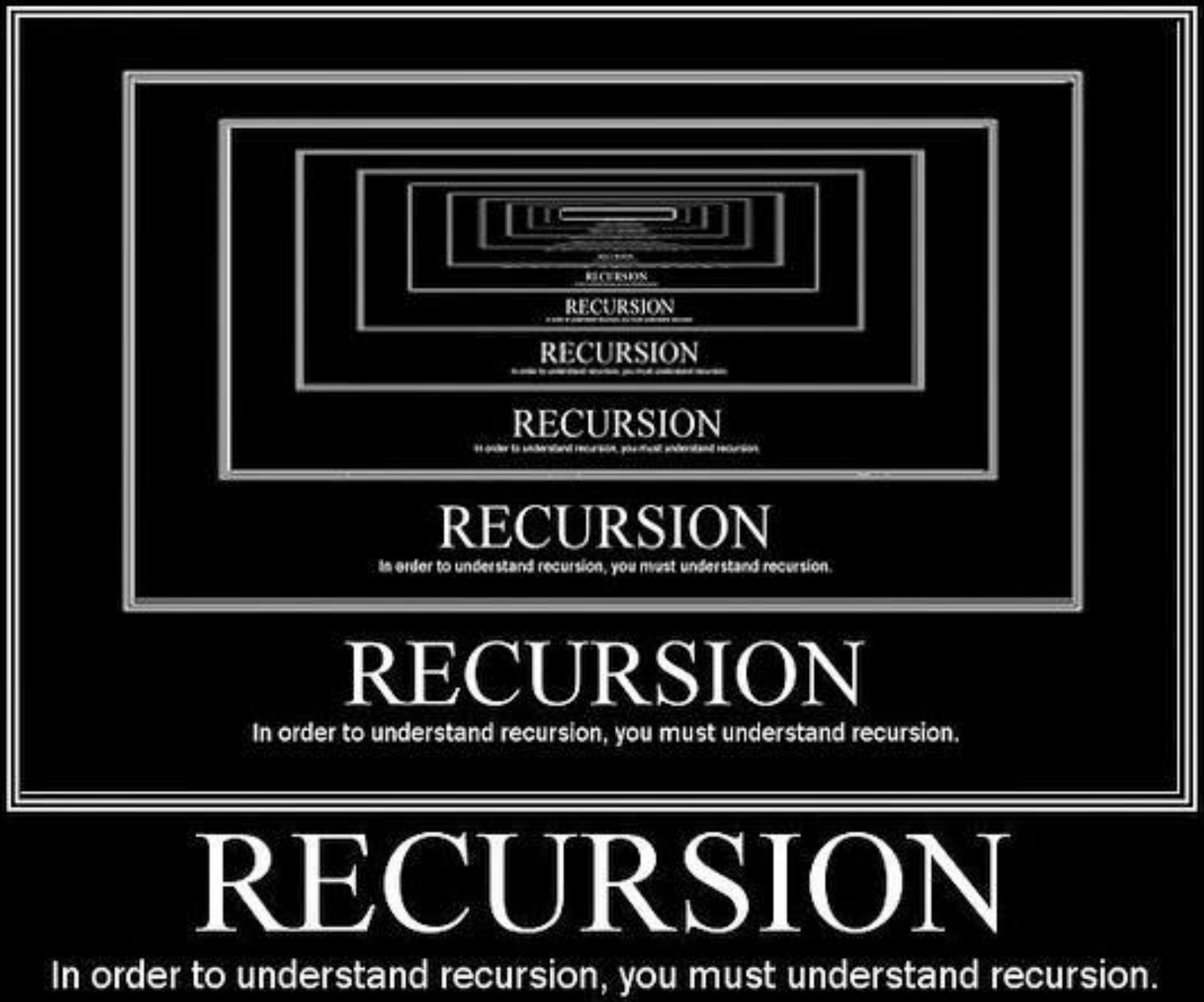
Computer Engineering Department

Programming in python

BBM103 Introduction to Programming Lab 1

Week 8

Fall 2018


RECURSION
In order to understand recursion, you must understand recursion.

WHAT IS RECURSION?

- **Goal**: simplify the problem by solving the same problem for smaller input
  - Solve problems by **divide(decrease)-and-conquer**
- Function calls itself (but not infinitely!)
  - One or more base cases

ITERATION vs. RECURSION

- An **ITERATIVE** function is one that loops to repeat some part of the code.
- A **RECURSIVE** function is one that calls itself again to repeat the code.

Multiplication Example: ITERATIVE Solution

`a*b` is equal to “add **`a`** to itself **`b`** times”

$$a*b = \underbrace{a + a + a + a + \ldots + a}_{\text{b times}}$$

```python
def multiply_iterative(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

Iteration

Multiplication Example: RECURSIVE Solution

$$a*b = \underbrace{a + \underbrace{a + a + a + \ldots + a}_{\text{b-1 times}}}_{\text{b times}} = \boxed{a + a*(b-1)}$$

```
def mult_recursive(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recursive(a, b-1)
```

Base case

Recursive Step

Factorial Example: ITERATIVE Solution

$$n! = n*(n-1)*(n-2)*(n-3)* \ldots * 1$$

```
def factorial_iterative(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

Iteration

Factorial Example: RECURSIVE Solution

`n! = n*(n-1)*(n-2)*(n-3)* … * 1`

- **Base Case:** if n = 1 → 1! = 1
- **Recursive step:** n! = n * (n-1)!

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

Base case

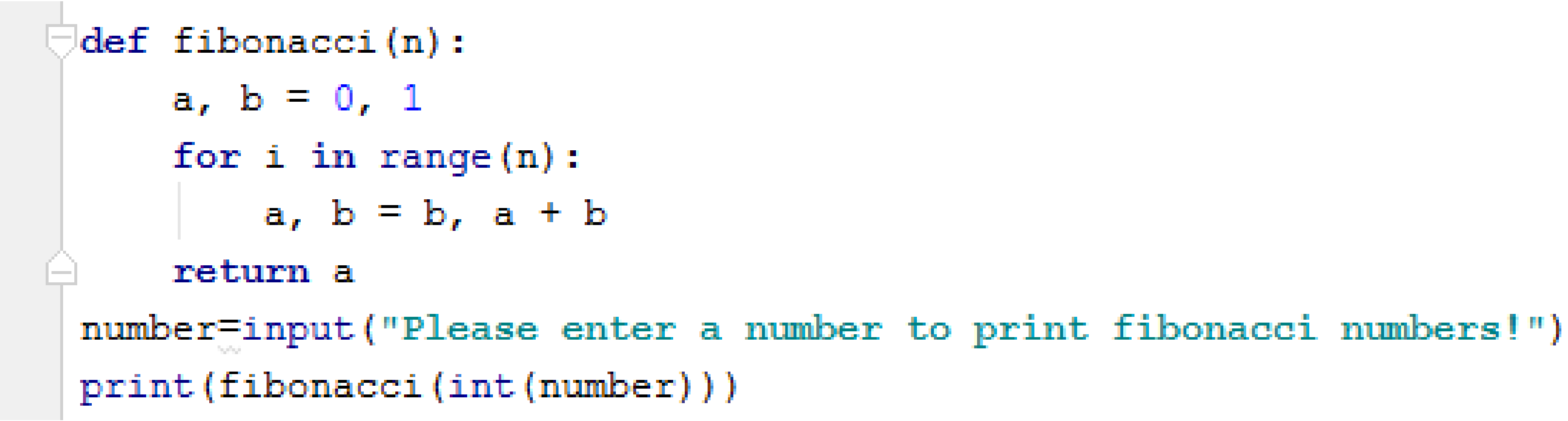
Recursive Step

ITERATION vs. RECURSION

- recursion may be simpler, more intuitive, and also efficient and natural for a programmer.
- BUT! Recursion may not be efficient from the computer's point of view.
  - Ex. Computing n^{th} Fibonacci number recursively takes $O(2^n)$ steps!

Example: Fibonacci Numbers

The Fibonacci numbers are the numbers of the following sequence of integer values:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
The Fibonacci numbers are defined by:
$F_n = F_{n-1} + F_{n-2}$
with $F_0 = 0$ and $F_1 = 1$

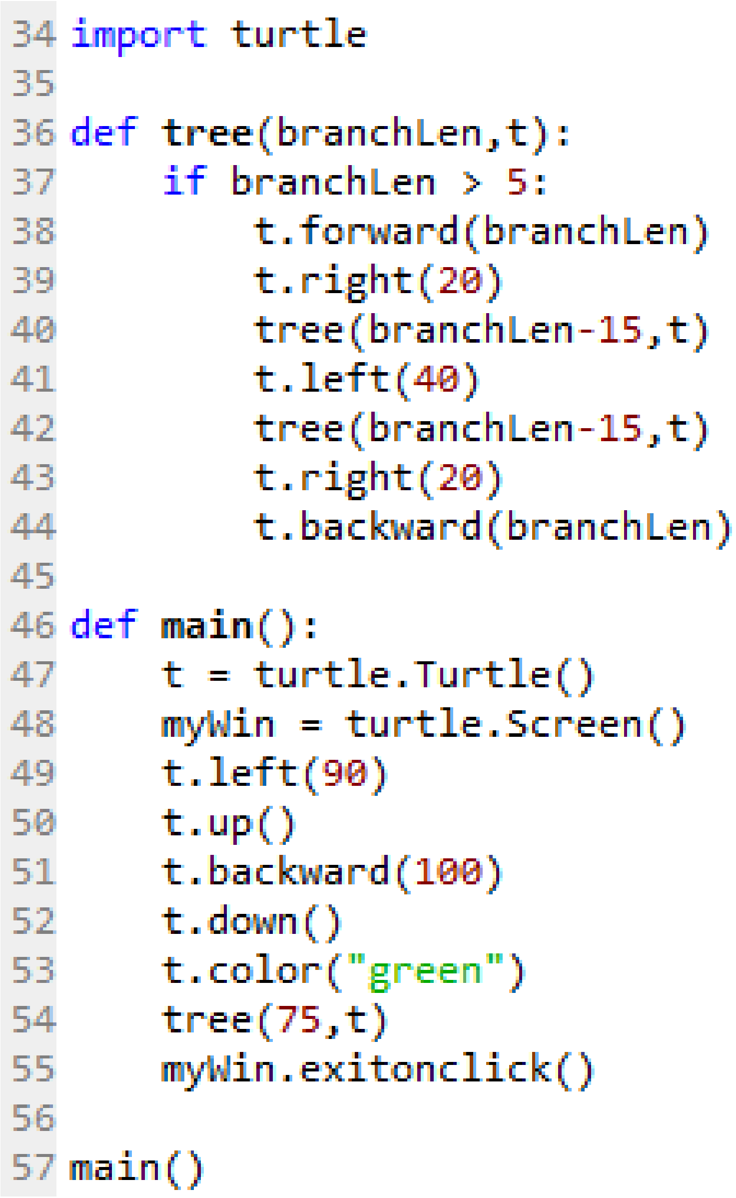


```python
def fibonacci(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
number=input("Please enter a number to print fibonacci numbers!")
print(fibonacci(int(number)))
```

Output:

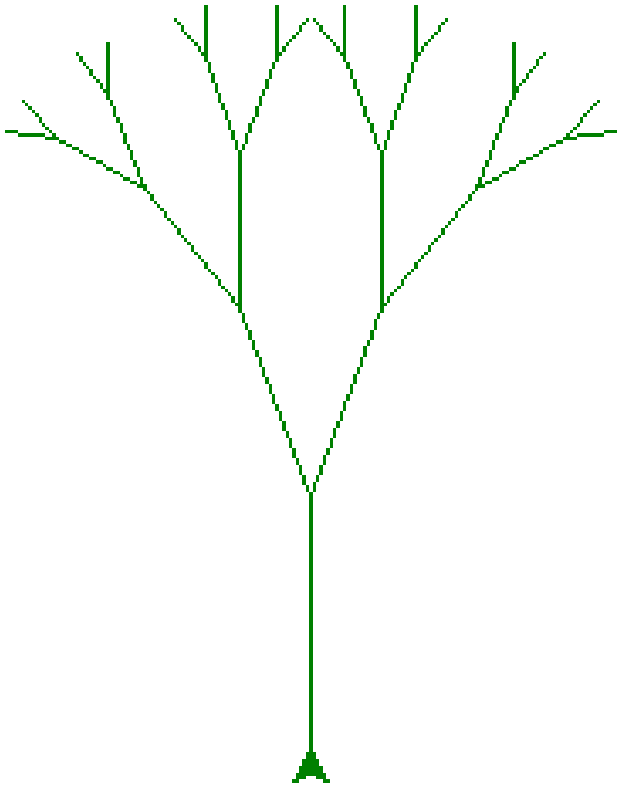
```
Please enter a number to print fibonacci numbers!4
3
```

Example: Visualizing Recursion

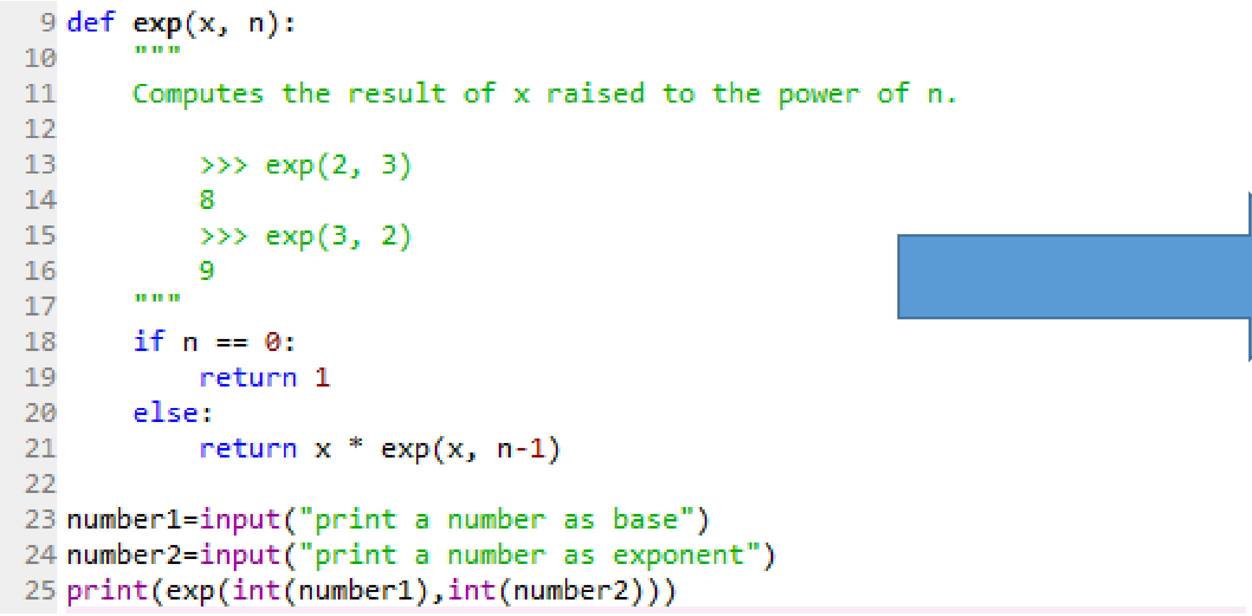


```
import turtle

def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()

main()
```

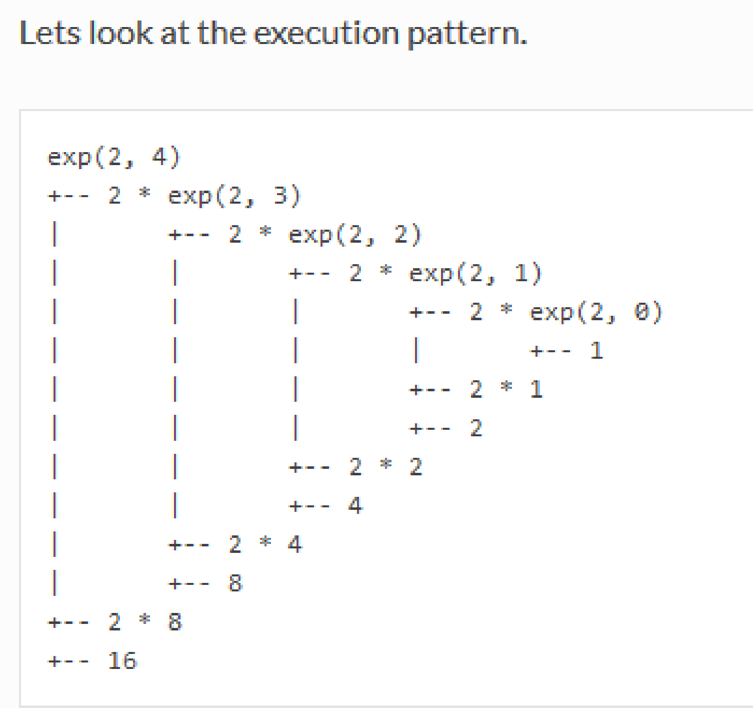
Output:



Example: Computing Exponent



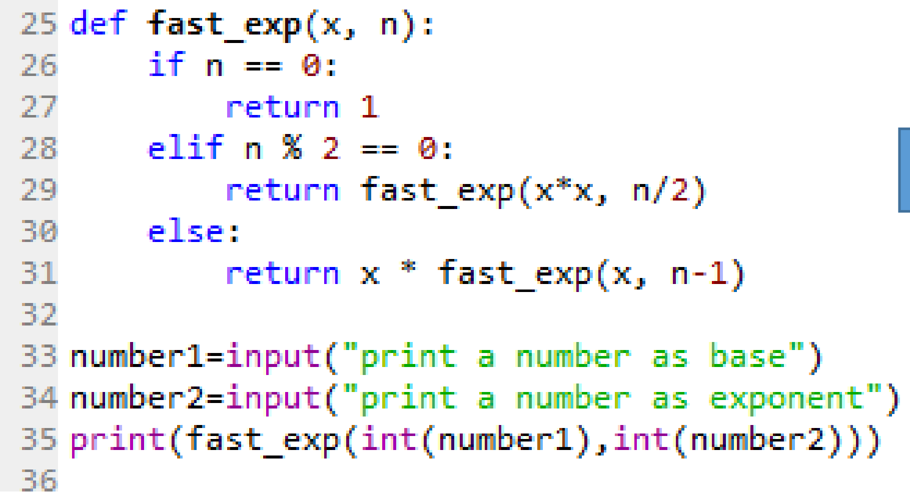
```
def exp(x, n):
    """
    Computes the result of x raised to the power of n.

        >>> exp(2, 3)
        8
        >>> exp(3, 2)
        9
    """
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)

number1=input("print a number as base")
number2=input("print a number as exponent")
print(exp(int(number1),int(number2)))
```



Lets look at the execution pattern.

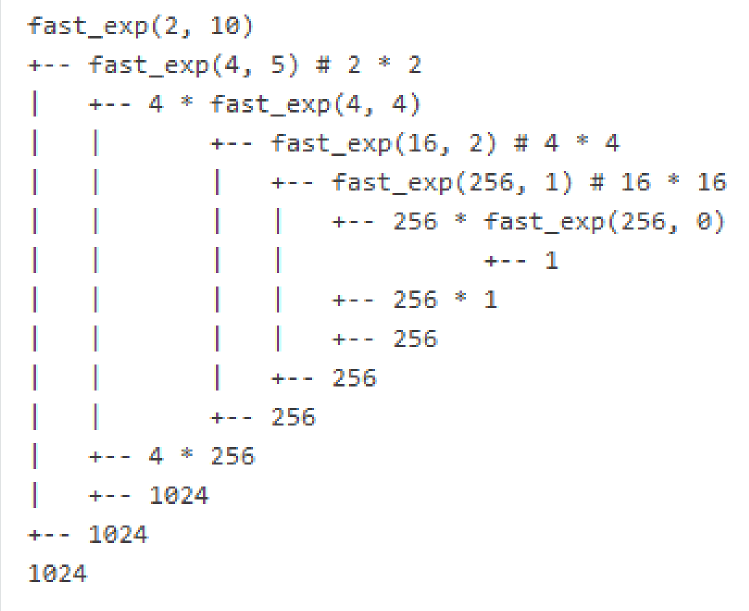
```
exp(2, 4)
+-- 2 * exp(2, 3)
|       +-- 2 * exp(2, 2)
|       |       +-- 2 * exp(2, 1)
|       |       |       +-- 2 * exp(2, 0)
|       |       |       |       +-- 1
|       |       |       +-- 2 * 1
|       |       |       +-- 2
|       |       +-- 2 * 2
|       |       +-- 4
|       +-- 2 * 4
|       +-- 8
+-- 2 * 8
+-- 16
```

We can compute exponent in fewer steps if we use successive squaring.



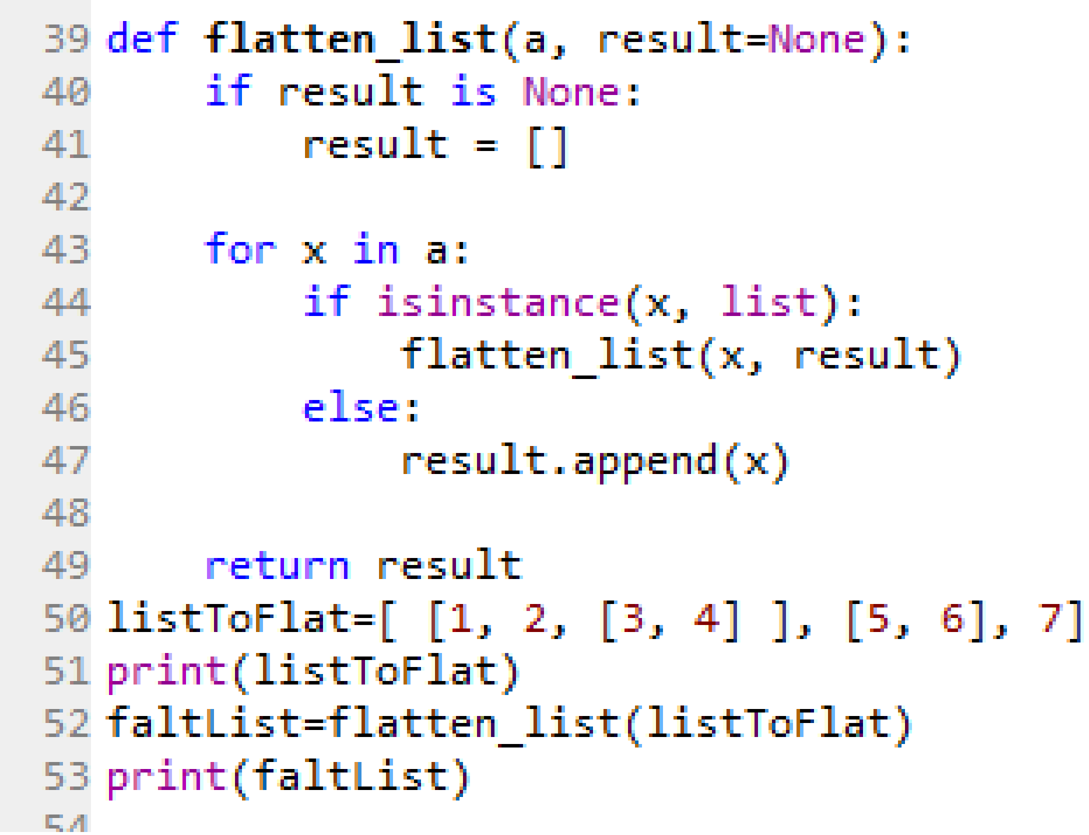
```
def fast_exp(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return fast_exp(x*x, n/2)
    else:
        return x * fast_exp(x, n-1)

number1=input("print a number as base")
number2=input("print a number as exponent")
print(fast_exp(int(number1),int(number2)))
```

Lets look at the execution pattern now.



```
fast_exp(2, 10)
+-- fast_exp(4, 5) # 2 * 2
|   +-- 4 * fast_exp(4, 4)
|   |       +-- fast_exp(16, 2) # 4 * 4
|   |       |   +-- fast_exp(256, 1) # 16 * 16
|   |       |   |   +-- 256 * fast_exp(256, 0)
|   |       |   |             +-- 1
|   |       |   |   +-- 256 * 1
|   |       |   |   +-- 256
|   |       |   +-- 256
|   |       +-- 256
|   +-- 4 * 256
|   +-- 1024
+-- 1024
1024
```

Example: Flatten a List



```python
def flatten_list(a, result=None):
    if result is None:
        result = []

    for x in a:
        if isinstance(x, list):
            flatten_list(x, result)
        else:
            result.append(x)

    return result
listToFlat=[ [1, 2, [3, 4] ], [5, 6], 7]
print(listToFlat)
faltList=flatten_list(listToFlat)
print(faltList)
```

Output:

```
[[1, 2, [3, 4]], [5, 6], 7]
[1, 2, 3, 4, 5, 6, 7]
```