RECURSION

Fundamentals

- **Recursion** is a process where a function calls itself once or multiple times to solve a problem.
- Any function that calls itself is **recursive**.
- Recursion that involves a method directly calling itself is called **direct recursion**.
- **Indirect recursion** occurs when a method invokes another method, eventually resulting in the original method being invoked again.
- When a recursive function fails to stop recursion, **infinite recursion** occurs.
- A recursive algorithm must:
    - Have a base case – A **base case** is the condition that allows the algorithm to stop recurring.
    - Change its state and move toward the base case – A **change of state** means that some data that the algorithm is using is modified.
    - Call itself, recursively.

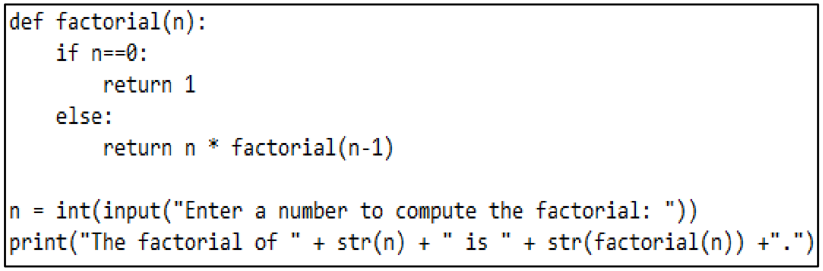
Recursion vs. Iteration

- **Iteration** is a process of repeating a set of instructions. This is also known as "**looping**."

Recursion	**Iteration**
It terminates when a base case is reached.	It terminates when a condition is proven to be false.
Each recursive call requires extra memory space.	Each iteration does not require extra memory space.
An infinite recursion may cause the program to run out of memory and may result in stack overflow.	An infinite loop could loop forever since there is no extra memory being created.
Solutions to some problems are easier to formulate recursively.	Iterative solutions to a problem may not always be as obvious as a recursive solution.

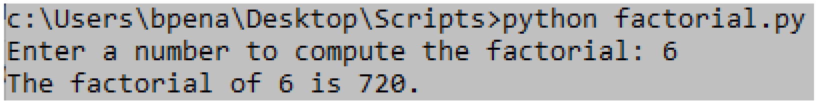
Types of Recursion

- **Linear recursion** – The function calls itself **once** each time it is invoked. Ex. finding the factorial



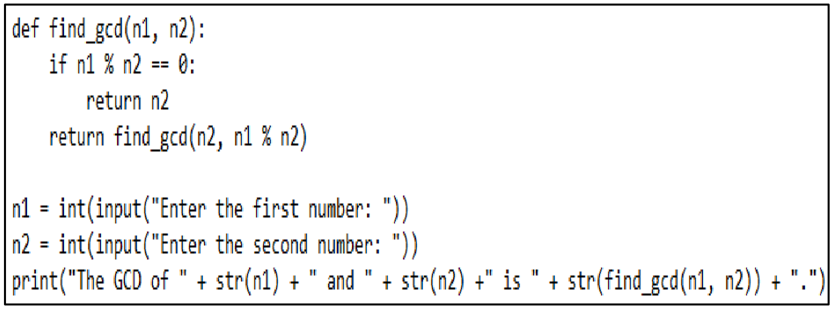
```
def factorial(n):
    if n==0:
        return 1
    else:
        return n * factorial(n-1)

n = int(input("Enter a number to compute the factorial: "))
print("The factorial of " + str(n) + " is " + str(factorial(n)) +".")
```

Output:



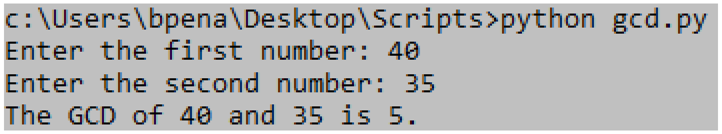
```
c:\Users\bpena\Desktop\Scripts>python factorial.py
Enter a number to compute the factorial: 6
The factorial of 6 is 720.
```

- **Tail recursion** – The function makes a recursive call as its very **last** operation. Ex. finding the greatest common divisor of two (2) non-zero integers



```
def find_gcd(n1, n2):
    if n1 % n2 == 0:
        return n2
    return find_gcd(n2, n1 % n2)

n1 = int(input("Enter the first number: "))
n2 = int(input("Enter the second number: "))
print("The GCD of " + str(n1) + " and " + str(n2) +" is " + str(find_gcd(n1, n2)) + ".")
```

Output:

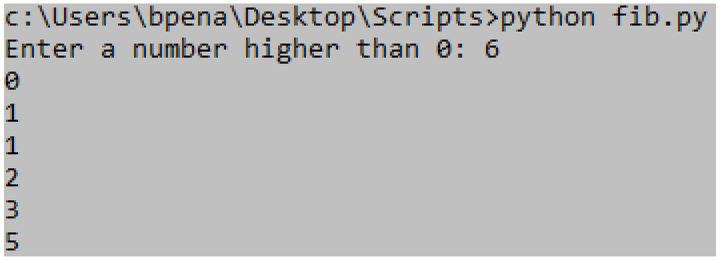


```
c:\Users\bpena\Desktop\Scripts>python gcd.py
Enter the first number: 40
Enter the second number: 35
The GCD of 40 and 35 is 5.
```

- **Binary recursion** – The function calls itself **twice** in the run of the function. Ex. Fibonacci series

```
def fib(num):
    if num <= 1:
        return num
    return fib(num - 1) + fib(num - 2)

num = int(input("Enter a number higher than 0: "))
for i in range(num):
    print(fib(i))
```

Output:

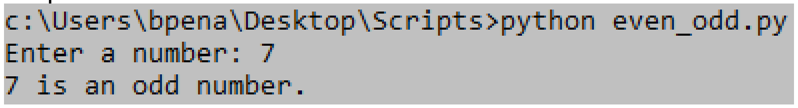


```
c:\Users\bpena\Desktop\Scripts>python fib.py
Enter a number higher than 0: 6
0
1
1
2
3
5
```

- **Mutual recursion** – The function works in a **pair** or a group. Ex. determining whether an integer is even or odd

```
def is_even(num):
    if num == 0:
        return True
    else:
        return is_odd(num - 1)

def is_odd(num):
    if num == 0:
        return False
    else:
        return is_even(num - 1)

num = int(input("Enter a number: "))
if is_even(num):
    print(str(num) + " is an even number.")
else:
    print(str(num) + " is an odd number.")
```

Output:



```
c:\Users\bpena\Desktop\Scripts>python even_odd.py
Enter a number: 7
7 is an odd number.
```

References:

Karumanchi, N. (2017). *Data structures and algorithms made easy*. Hyderabad: CareerMonk Publications.

Runestone Academy (n.d.). *Citing sources.* Retrieved from https://interactivepython.org/runestone/static/pythonds/index.html