Discrete Structures!

CMPSC 102

Programming Constructs

ALLEGHENY COLLEGE

Key Questions & Learning Objectives

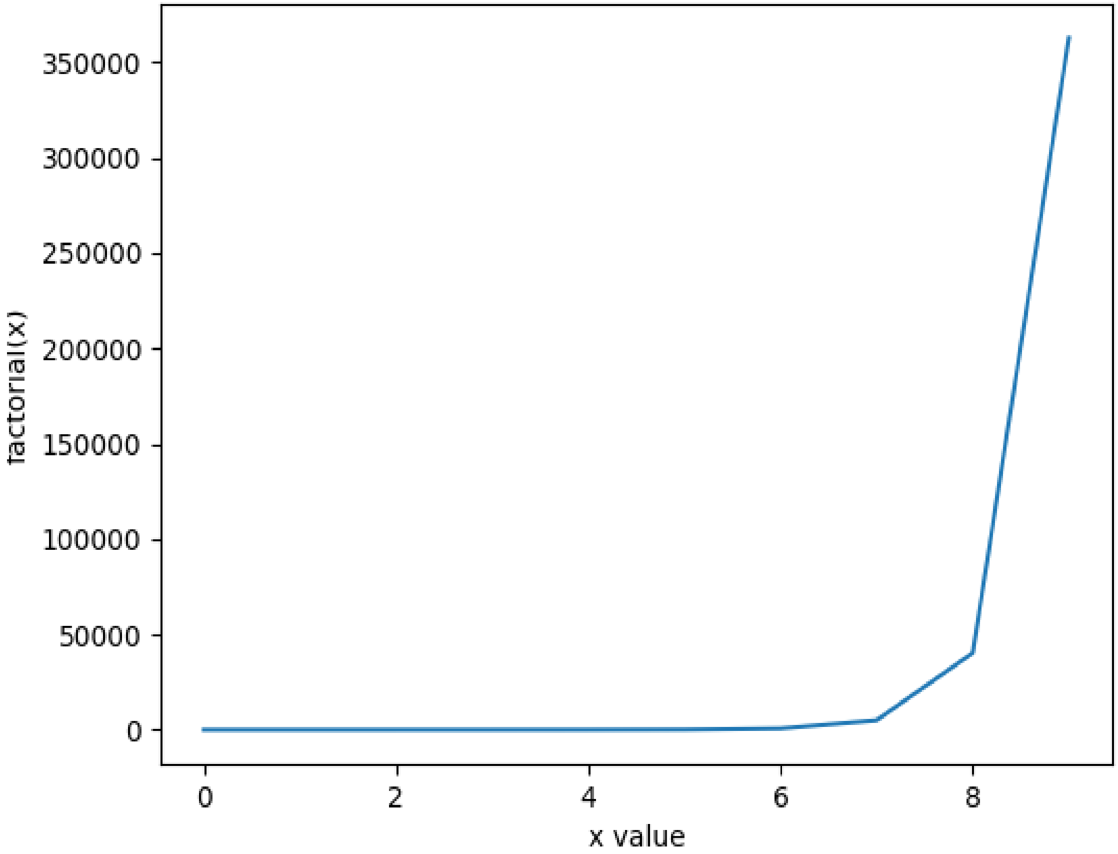
- How do I use non-recursive functions, recursive functions, and lambda expressions to perform mathematical operations such as computing the absolute value of a number and the means of a sequence of numbers?

- To remember and understand some discrete mathematics and Python programming concepts, setting the stage for exploring of discrete structures.

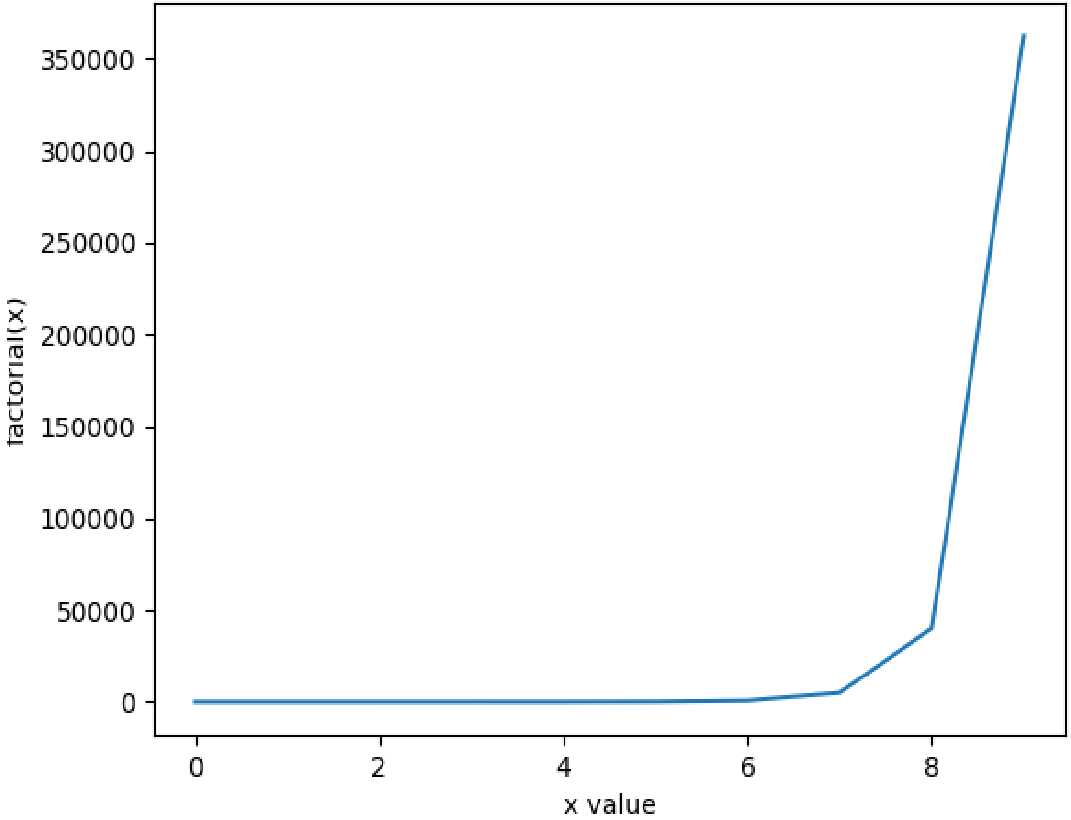
Python Programming Retrospective

- Python code is designed to be **intuitive**
- Key components of Python programming include:
    - Function and their definitions
    - Input parameters for functions
    - The code block that completes the function's work
    - Return statements
    - Invocations of functions (calls to functions)
    - Collecting the returned values (function outputs).
- Investigate the ways to make the above commands possible with definitions and call using Python.

Factorials - values get quickly get big



Factorials - values get quickly get big



x	$fac(x)$
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800

Plotting factorials – use Jupyter for this code!

```
import matplotlib.pyplot as plt
import math

# get factorial data
x_list = [i for i in range(10)]
factorials_list = [math.factorial(x) for x in x_list]
print("x,factorial(x)")
# formatting data
for i in range(len(x_list)):
        xvalue_int = x_list[i]
        fvalue_int = factorials_list[i]

# prepare plot
print(f"x values :{x_list}")
print(f"factorial(x) : {factorials_list}")
plt.plot(x_list, factorials_list)
plt.xlabel('x value')
plt.ylabel('factorial(x)')
plt.show()
```

Factorials

Factorials: one definition

$$N! = \pi_{i=1}^{N} i = 1 * 2 *..* (N-1) * N$$

Factorials: another definition

$$N! = \frac{(N+1)!}{(N+1)} = \frac{(N+1) * N!}{(N+1)}$$

Factorials are applied to integers

Factorials

Factorials

$$N! = N * (N-1) * (N-2) * \cdots * (2) * (1)$$

$$5! = 5 * 4 * 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$3! = 3 * 2 * 1$$

$$2! = 2 * 1$$

$$1! = 1$$

$0! = 1$ (Special case by convention)

Factorials defined

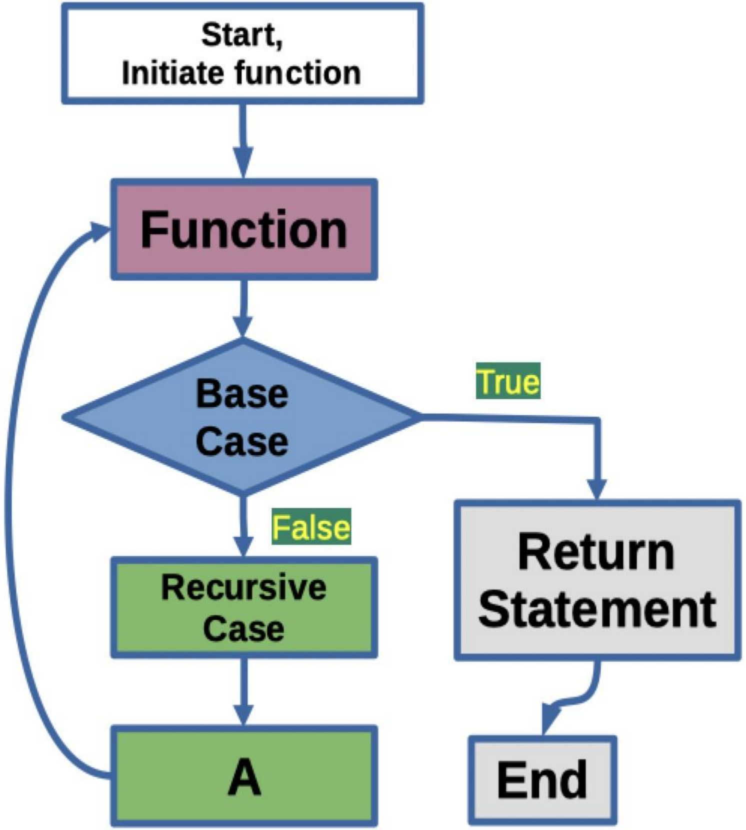
$$N! = [(N-1)! + (N-2)!] * (N-1)$$

$$7! = (6! + 5!) * 6$$

$$6! = (5! + 4!) * 5$$

$$5! = (4! + 3!) * 4$$

Creating Solutions



Calculating Factorials by Recursion

```
def factorial(number: int):
        if number == 1:
                return 1
        return number * factorial(number - 1)

num = 5
print("The factorial of " + str(num)
        + " is " + str(factorial(num)))
```

- The recursive *factorial* function calls itself!
- How does this function ever stop executing?
- What are the benefits to using recursive functions?

Calculating Factorials by Recursion

```
def factorial(number: int):
        if number == 1:
                return 1
        return number * factorial(number - 1)

num = 5
print("The factorial of " + str(num) +
        " is " + str(factorial(num)))
```

- Where is the base case?
- Where is the recursive case?
- How could this code work without these two functions?

Recursive Factorial Function - To consider

- As an equation: $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$
  - What are the **parts** of a recursive function in Python?
  - Defined by **cases** using conditional logic (*a case to go, and one to force a stop*)
  - A mathematical function defined to **call itself**
  - A recursive call that makes progress to a **base case**
  - A **base case** that **stops** the **recursive function calls**
  - Repeatedly perform an operation through (*self*) function calls
  - What would happen if you input a **negative number**?
- How could you write this function with **iteration**?

A Solution Using Basic Conditions - No numbers less than zero

```
def factorial(number: int):
        if number == 1:
                return 1
        if number < 0: #Catch negative numbers
                print("cannot compute")
        if number > 1:
                return number * factorial(number - 1)

num = -5
print("The factorial of " + str(num)
+ " is " + str(factorial(num)))
```

A Solution Using While - No numbers less than zero

```
def factorial(number: int):
        while number > 0:
                if number == 1:
                        return 1
                if number > 1:
                        return number * factorial(number - 1)
        print("cannot compute")
```

What Can YOU Do With Higher-Order Functions

You can pass a **function** as an **argument** to a **function**!

Why Do We Care About Higher-Order Functions!?

- Supports general-purpose function creation
- Allows executable functions as function input
- Supports both code reuse and modularity

Higher-Order Functions - library declaration and square()

Functions that allow another function as a parameter

```
from typing import Callable

# define a function that can square a number;
# use print statements for the purposes of debugging so that the behavior of this
# function and the next function are made clear

def square(number: int) -> int:
        print(f"Called square({number})")
        print(f" returning {number*number}")
        return number * number

print(square(5))
```

Higher-Order Functions - Call_twice() with execution code

```
# define a higher-order function that can accept a function
# as input and a number as input and then call the provided
# function with the provided input; again, use print
# statements for the purposes of debugging so that the
# behavior of this function is made clear

def call_twice(f: Callable[[int], int], number: int) -> int:
        print(f"Calling twice {f} with number {number}")
        return f(f(number))

# execution
num = 5

# give function and function's parameter
result = call_twice(square, num)

print("Calling the square twice with "
+ str(num) + " is " + str(result))
```

Higher-Order Functions

```
def square(number: int):
        print(f"Called square({number})")
        print(f" returning {number*number}")
        return number * number
```

- The behavior of **higher-order** functions in Python:
- square() is a function computes number*number and returns value.

Higher-Order Functions

```
def call_twice(f, number: int):
        print(f"Calling twice {f} with number {number}")
        return f(f(number))
```

- call_twice() is a function that calls a function f twice
- First, call_twice() calls f with number
- Then, call_twice() calls f with f(number)
- Finally, call_twice() returns result of f(f(number))
- Can you predict the output of the call_twice() function?
- How would you test the call_twice() function? Can you express it differently?

Higher-Order Functions

Calling twice <function square at 0x104c30940> with number 5
Called square(5)
 returning 25
Called square(25)
 returning 625
Calling the square twice with 5 is 625

Lambda Expressions - Also known as, “anonymous functions”

```
def call_twice(f, number: int):
        print(f"Calling twice {f} with number {number}")
        return f(f(number))
```

```
square = lambda x: x*x
number = 5
result = call_twice(square, number)
print("Calling square lambda twice " +
        "with " + str(number) +
        " is " + str(result))
```

- Functions are values in the Python programming language
- square is an expression that has a function as its value

Lambda Expressions

```python
def call_twice(f, number: int):
    print(f"Calling twice {f} with number {number}")
    return f(f(number))
```

```python
square = lambda x: x*x
number = 5
result = call_twice(square, number)
print("Calling square lambda twice " +
      "with "  + str(number) +
      " is " + str(result))
```

```
Calling twice <function <lambda> at 0x37500c8> with number 5
Calling square lambda twice with 5 is 625
```

- Lambda functions are known as anonymous functions and add simplicity in programming
- Useful for small function input to other functions