Creative Coding 6

David Lynch

Slide Deck by Ruán Murgatroyd

Last Class

- Tuples
- Objects
- Classes
- Class Methods
- Instantiating Classes
- Simulating Collisions

Today's Class

- *Declarative* vs *Imperative* programming
- Introduction to *Recursion*
- Solving problems recursively
- Looking at lists recursively
- Implementing recursive art in *Processing*

Imperative Programming

Imperative Programming

- Imperative Programming is the paradigm of how computers actually function.
- It follows Alan Turing's *Turing Machine* model of computation.

- **Turing Machine:** An abstract machine which is given an infinite tape with squares. The machine can read from and write to the squares on the tape using a finite set of instructions.

- Most languages are imperative, Python, C, Java, etc.

Declarative Programming

- Declarative programming is an alternate model of programming which aligns more to evaluating expressions instead of manipulating symbols when following a set of instructions.

- **Lambda Calculus:** A computational model expressed using pure anonymous functions. No state is manipulated, instead, an expression is given and evaluated.
- Lambda calculus is *‘Turing Complete’* which means that any program that can be done on a *Turing Machine* can be done in lambda calculus.
- Declarative languages include Prolog, LISP, Haskell, etc.

Comparing Imperative Vs Declarative

Imperative	Declarative
Mutable State	Immutable State
Side effects from function	‘Pure’ functions
Executing instructions	Evaluating statements

Pure Functions

Pure Functions

- Pure functions are functions which don't affect or be influenced by any aspect of the program's state.
- This means that when you call a function, it will only generate some data based on it's arguments regardless of what the rest of the program is doing.
- This means that a pure function will *always* give the same result given the same arguments.

- **NOTE:** I/O operations like `print()`, `input()` and *Processing*'s drawing functions will make your functions impure.

Rules of Functional Programming

- Once you define a value, you cannot change it.
- Functions must be pure.
- Looping using iterators may not be used.

Recursive Programming

Recursive Programming

- Recursion is when we define behaviours in terms of themselves, meaning functions will call themselves.
- Recursive functions allow us to define problems in their simplest form and then solve the larger iterations of the problem.
- Recursion is useful as it allows us to solve problems by accurately describing them in terms of itself.

E.g. 1: Imperative Solution

X = 0 i = 0

printNums.py

```
def printNums(x):
    for i in range(1, x):
        print(i)
```

Terminal

>

E.g. 1: Imperative Solution

x = 4 i = 0

printNums.py

```
def printNums(x):
    for i in range(1, x):
        print(i)
```

Terminal

>

E.g. 1: Imperative Solution

x = 4 i = 1

printNums.py

```
def printNums(x):
    for i in range(1, x):
        print(i)
```

Terminal

>

E.g. 1: Imperative Solution

x = 4 i = 1

printNums.py

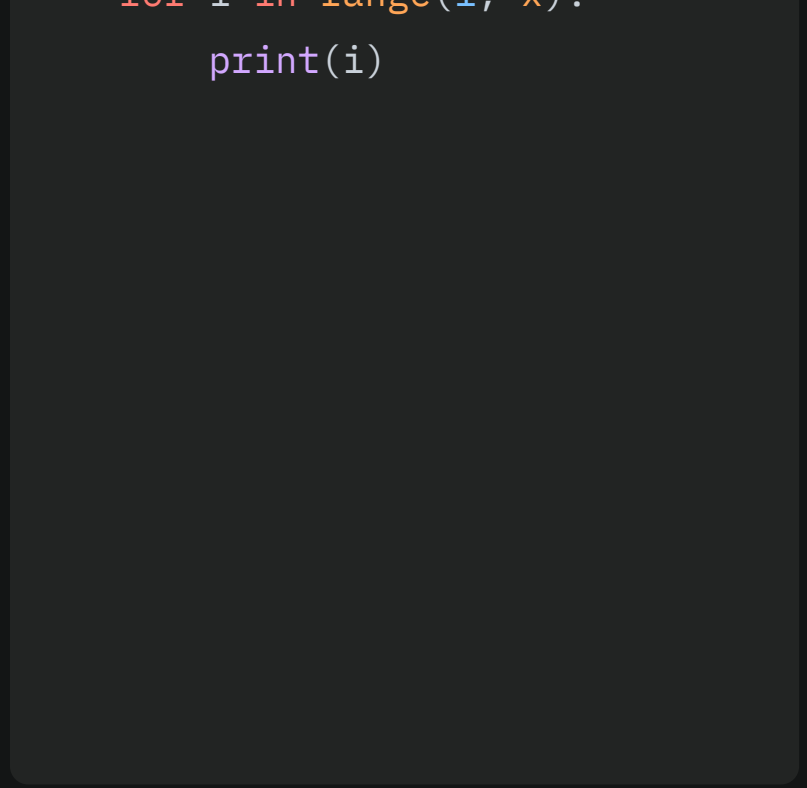
```
def printNums(x):
    for i in range(1, x):
        print(i)
```

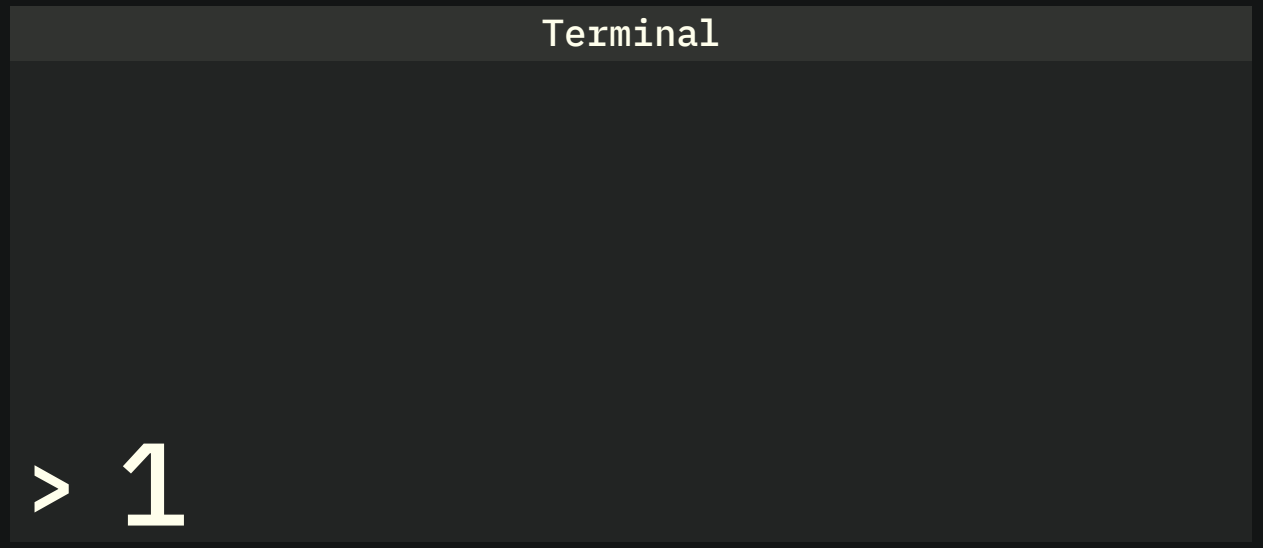
Terminal

```
> 1
```

E.g. 1: Imperative Solution

x = 4 i = 2





E.g. 1: Imperative Solution

x = 4 i = 2

printNums.py

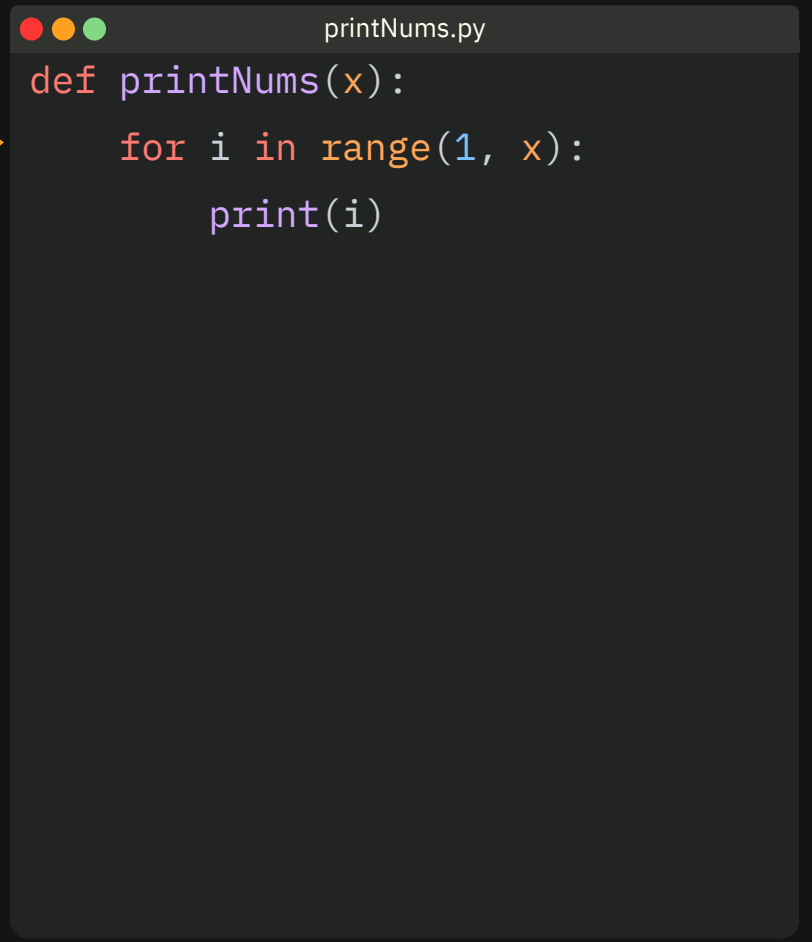
```
def printNums(x):
    for i in range(1, x):
        print(i)
```

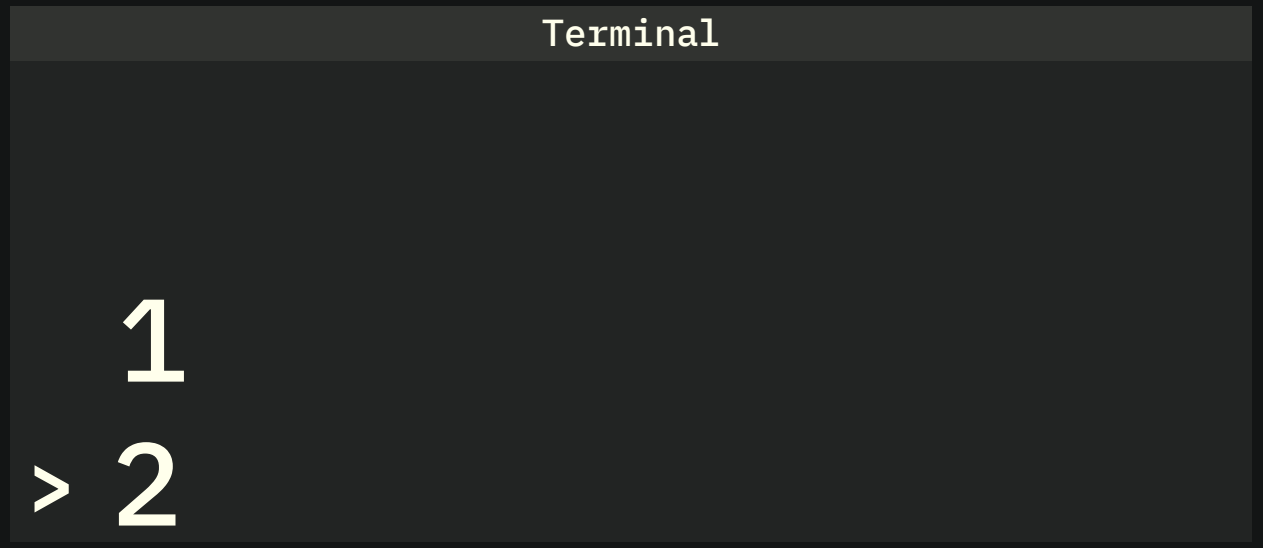
Terminal

```
1
> 2
```

E.g. 1: Imperative Solution

x = 4 i = 3





E.g. 1: Imperative Solution

x = 4 i = 3

printNums.py

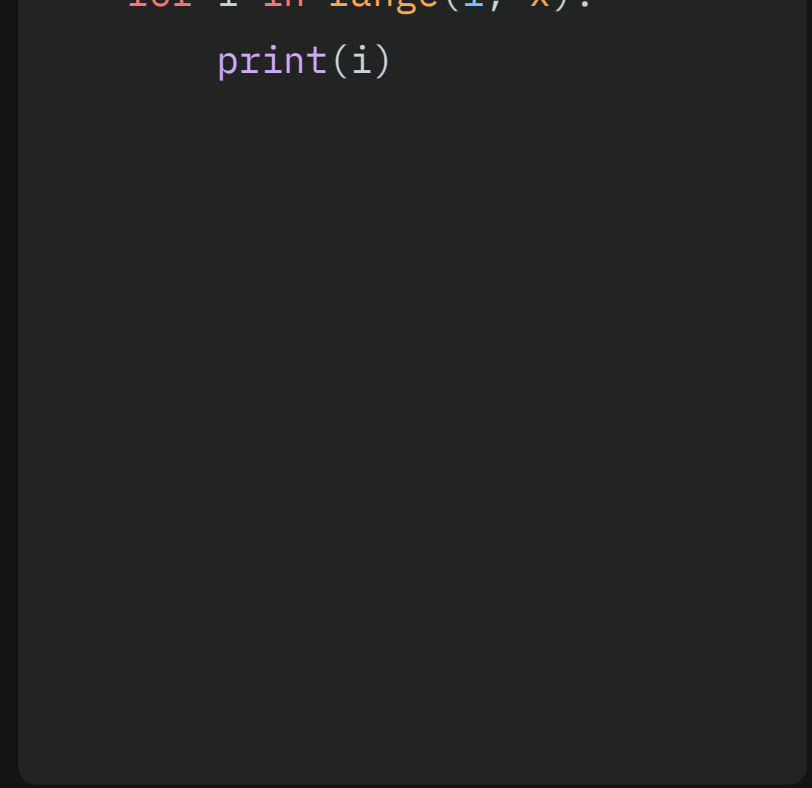
```
def printNums(x):
    for i in range(1, x):
        print(i)
```

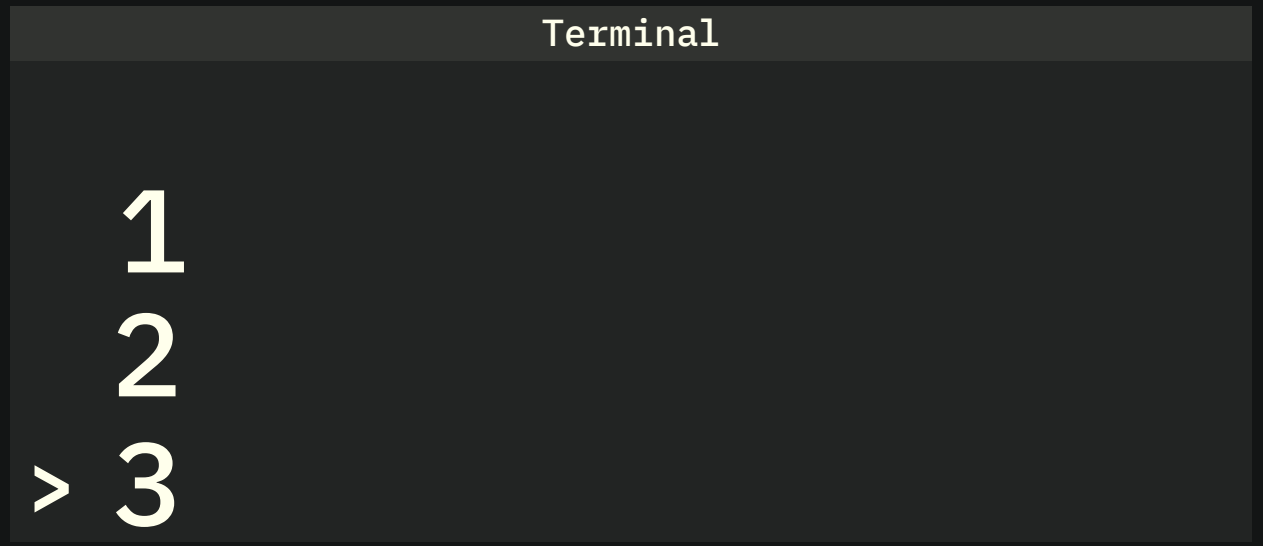
Terminal

```
> 1
  2
  3
```

E.g. 1: Imperative Solution

$x = 4$ $i = 4$





E.g. 1: Imperative Solution

$x = 4$ $i = 4$

printNums.py

```
def printNums(x):
    for i in range(1, x):
        print(i)
```

Terminal

```
1
2
3
> 4
```

E.g. 1: Imperative Solution

x = 4 i = 4

Terminal

```
1
2
3
> 4
```

printNums.py

```
def printNums(x):
    for i in range(1, x):
        print(i)
```

E.g. 2: Recursive Solution

X = 0 i = 0

Terminal

```
>
```

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

E.g. 2: Recursive Solution

x = 3 i = 0

recursivePrintNums.py

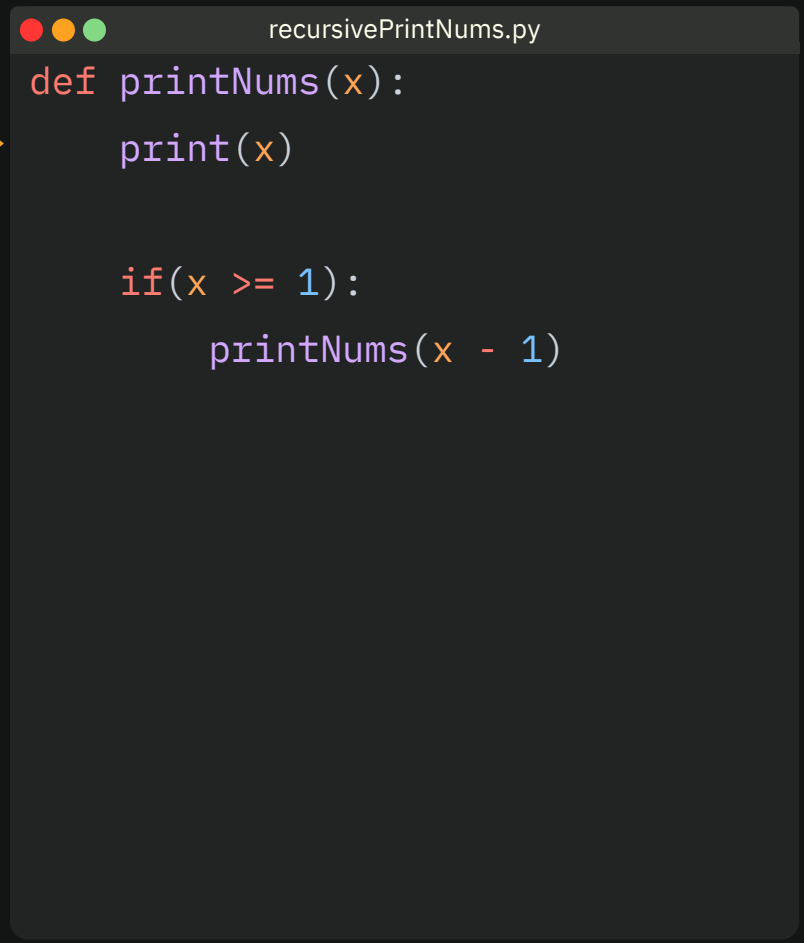
```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

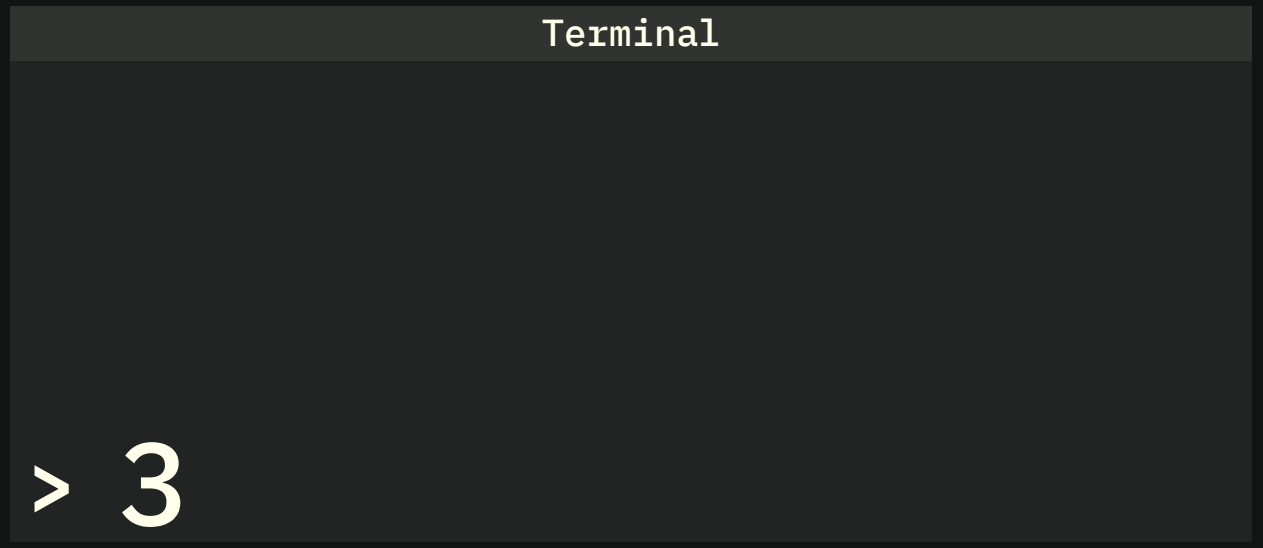
Terminal

>

E.g. 2: Recursive Solution

x = 3 i = 0





E.g. 2: Recursive Solution

$x = 2 \quad i = 0$

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Terminal

```
> 3
```

E.g. 2: Recursive Solution

$x = 2 \quad i = 0$

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Terminal

```
> 3
```

E.g. 2: Recursive Solution

x = 2 i = 0 →

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Terminal

```
> 2
  3
```

E.g. 2: Recursive Solution

$x = 1 \quad i = 0$

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Terminal

```
3
> 2
```

E.g. 2: Recursive Solution

$x = 1 \quad i = 0$

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Terminal

```
  3
> 2
```

E.g. 2: Recursive Solution

x = 1 i = 0

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Terminal

```
  3
  2
> 1
```

E.g. 2: Recursive Solution

X = 0 i = 0

Terminal

```
  3
  2
> 1
```

recursivePrintNums.py

```python
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

E.g. 2: Recursive Solution

X = 0 i = 0

recursivePrintNums.py

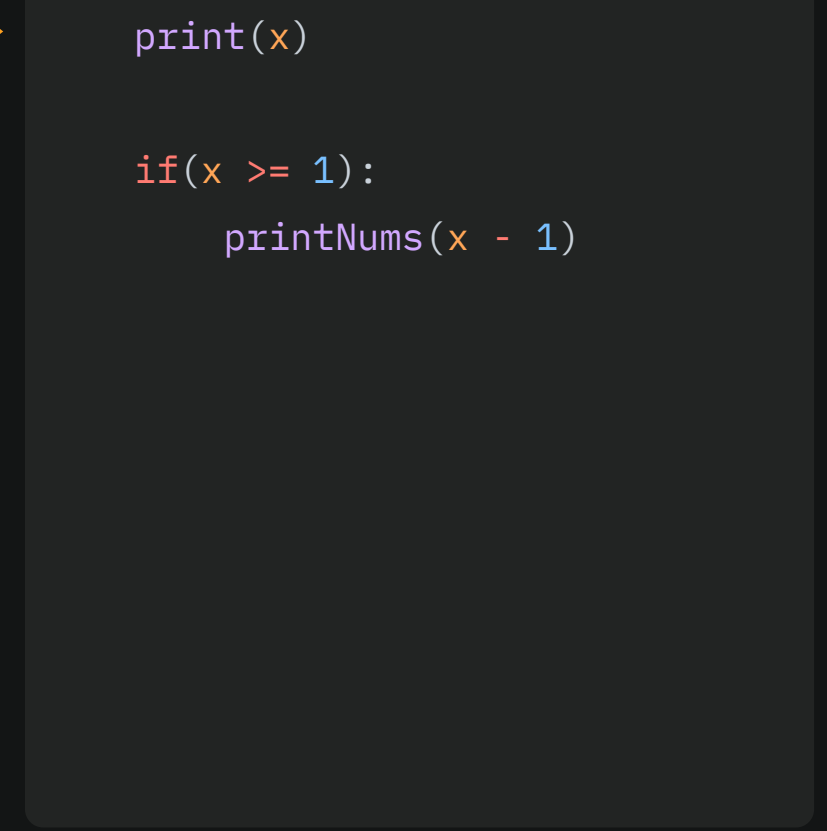
```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

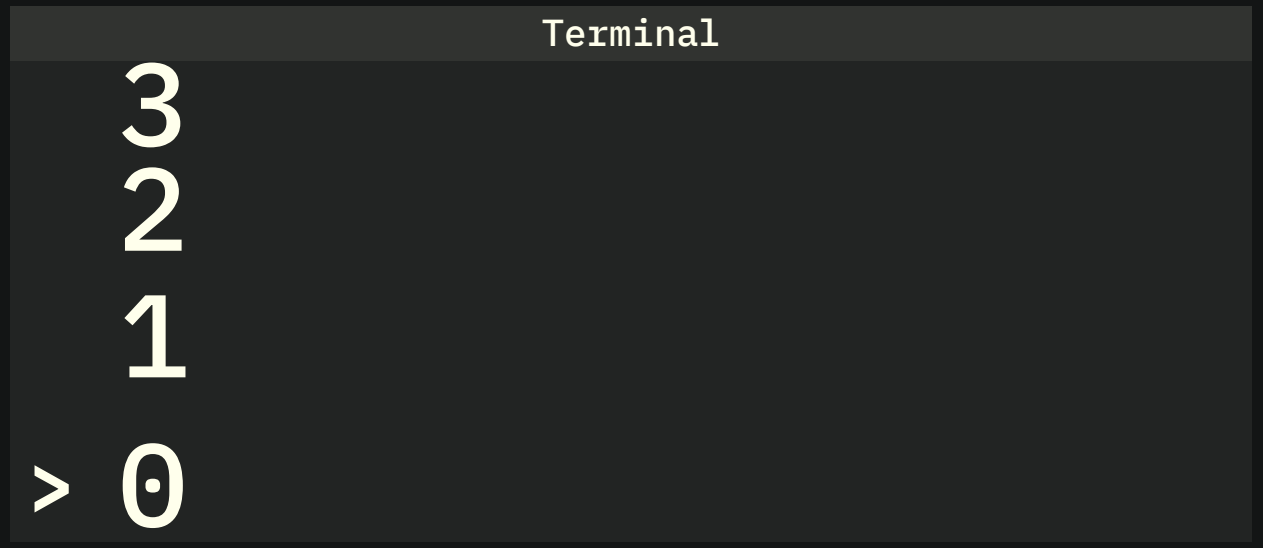
Terminal

```
  3
  2
> 1
```

E.g. 2: Recursive Solution

X = 0 i = 0





E.g. 2: Recursive Solution

X = 0 i = 0

Terminal

```
  3
  2
  1
> 0
```

recursivePrintNums.py

```
def printNums(x):
    print(x)

    if(x >= 1):
        printNums(x - 1)
```

Lists and Recursive Structures

Lists and Recursive Structures

- We can describe lists recursively by using the idea of a list having a ‘head’ and a ‘tail’.

- **Head:** First element of a list

- **Tail:** Rest of the list

Lists and Recursive Structures

- Thinking of lists like this allows us to iterate over them recursively.

Headsntails.py

```
head(x) = 1
tail(x) = [2, 3, 4, 5]
```

Recursive vs Imperative Code

printList.py

```
def printList(xS):
    for x in xS:
        print(x)
```

recursivePrint.py

```
def printList(xs):
  print(head(xs))
  if tail(xs) != []:
    printList(tail(xs))
```

Cost of Recursion

- If you remember our visualised *Call Stack*, every time we call a function, Python pushes it into memory.
- In recursion it is important to remember how much memory your function uses as each recursive call adds another function to the call stack.

Problem Qs

Problem Qs

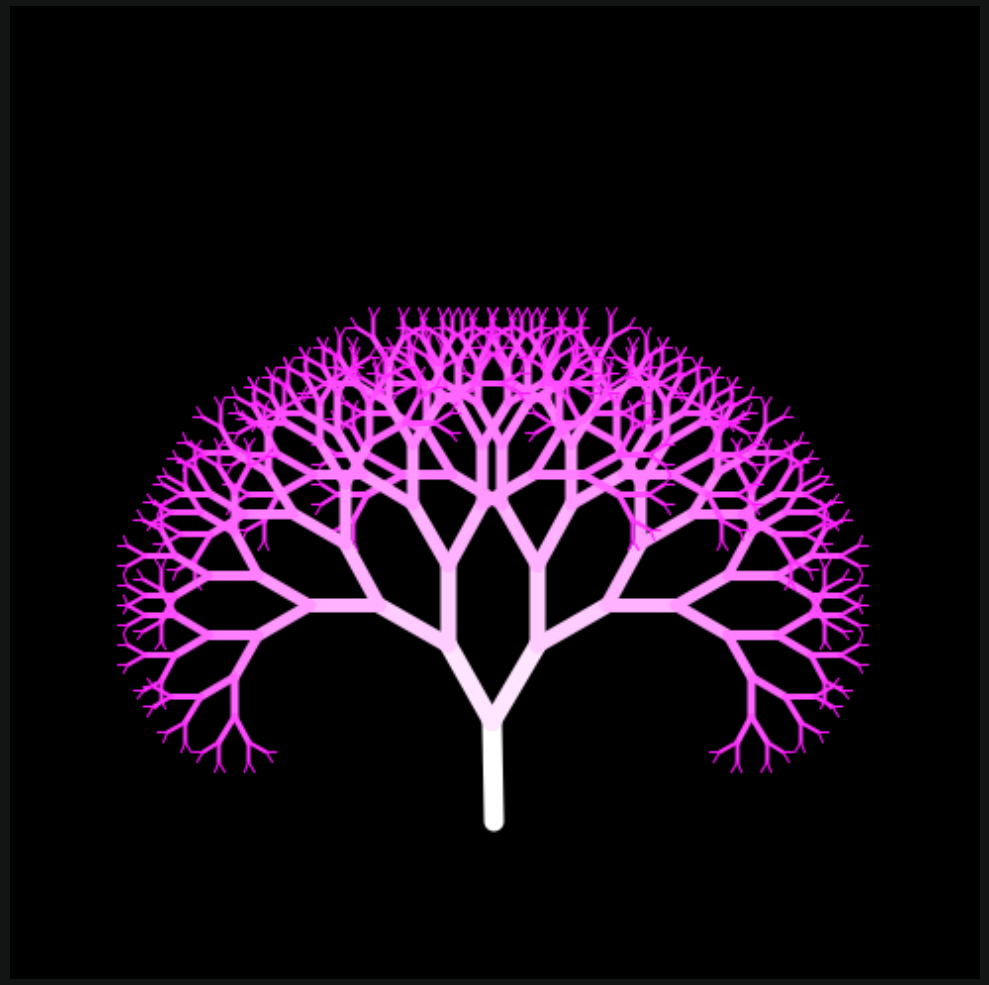
- **Download:** https://github.com/cs-soc-tudublin/introToRecursion

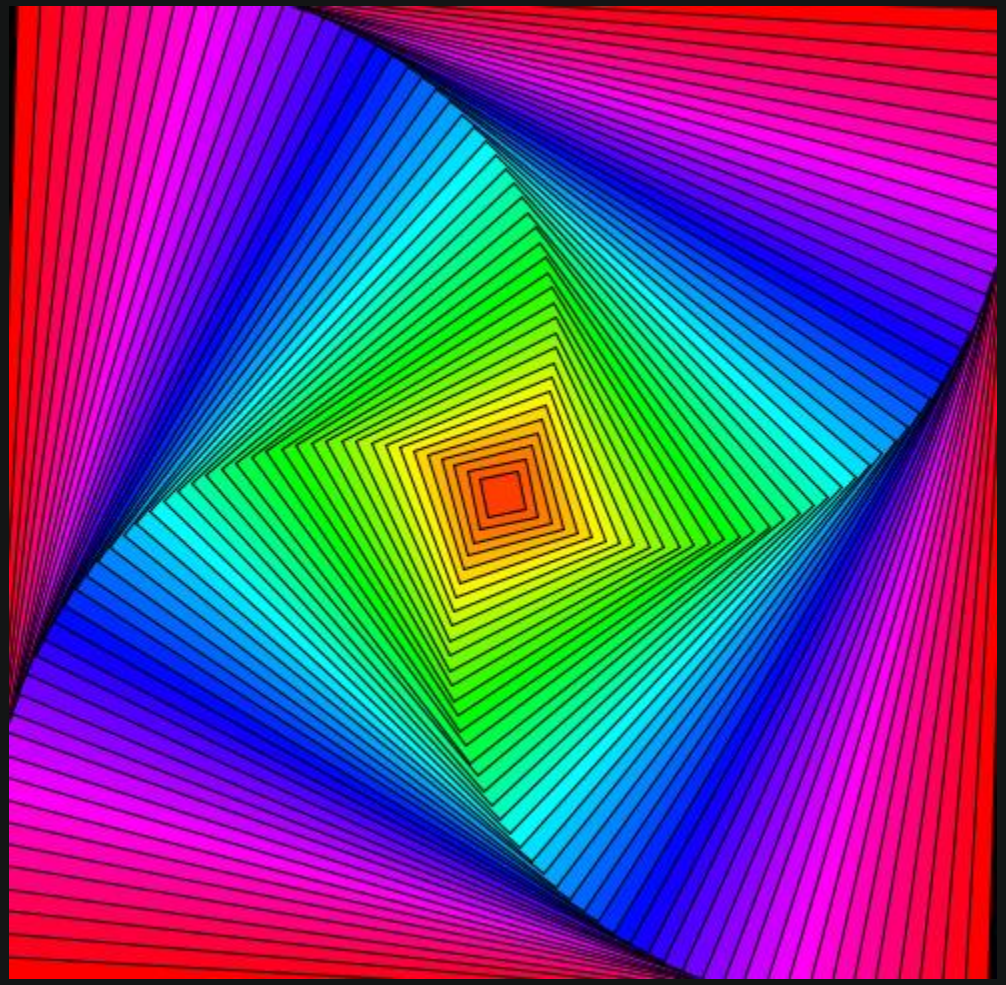
1. Write a function which returns the factorial of a given number.
2. Find the amount of elements in a list.
3. Find the sum of all elements in a list of integers.
4. Write a function to return a given Fibonacci number.
5. Write a function which takes a list and an integer *n* as an argument. Return the first *n* elements of the list.
6. Write a function which drops *n* elements from a list. Return the list without the first *n* elements.

Use Recursion for each question.

Processing Images

Processing Images





The End

Thanks for coming!

Notes uploaded on the Discord.

See you next week!