



Functional Program Design

Functional Programming

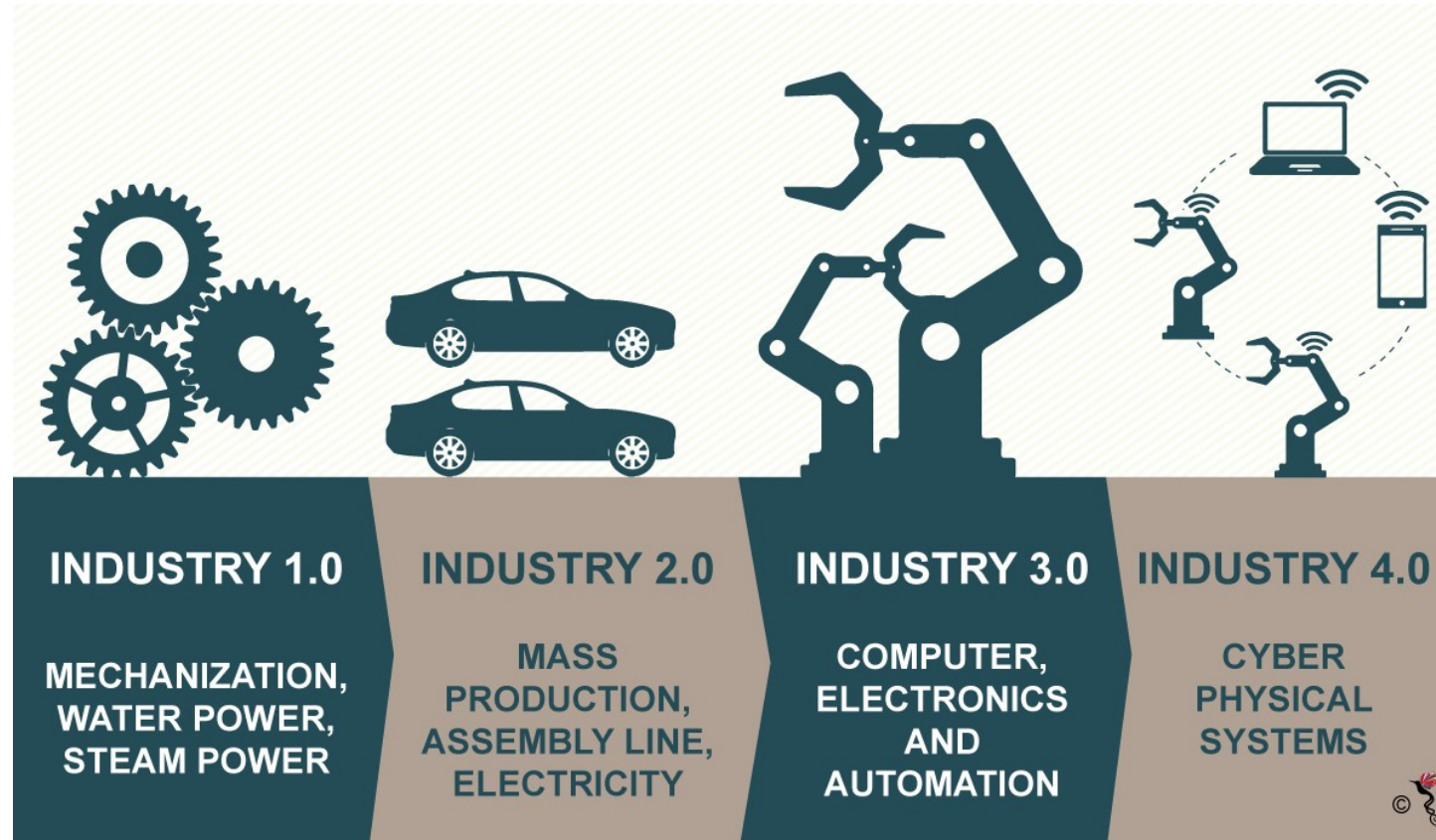
- Dates back to Church's Lambda calculus in the 1930s
- Procedural and OO code update the state of the running program
 - Functional programming avoids the idea of state by avoiding changing data
- Functional code is based on the idea of a mathematical function
 - e.g. Square function: $f(x) = x * x$
 - Functions do not change the input data but transform it to a new value
 - In pure functional programming languages, variables are immutable, they bind to newly created values instead of modifying existing data in memory
 - A functional program maps an input set of data to a new output set of data
- Complex computations are done by functional composition
 - e.g. $f(g(x))$ produces an output where $f()$ takes as input the result of applying $g()$ to an input x
 - Algorithms are expressed as a series of functions representing the steps of the algorithms
 - Each step of the algorithm is implemented as a function
 - A program can be represented as a series of function calls

Why Functional Programming

- Functional programming languages have been around since the 1950s
 - They didn't go mainstream for decades since there was no large scale need that only functional programming could do efficiently
 - For example, OO went mainstream because it could solve the emerging problem of networked applications in the 1990s that structured programming could not
 - Similarly, functional programming has gone mainstream recently because it solves a newly emerging problem: processing large amounts of streaming big data

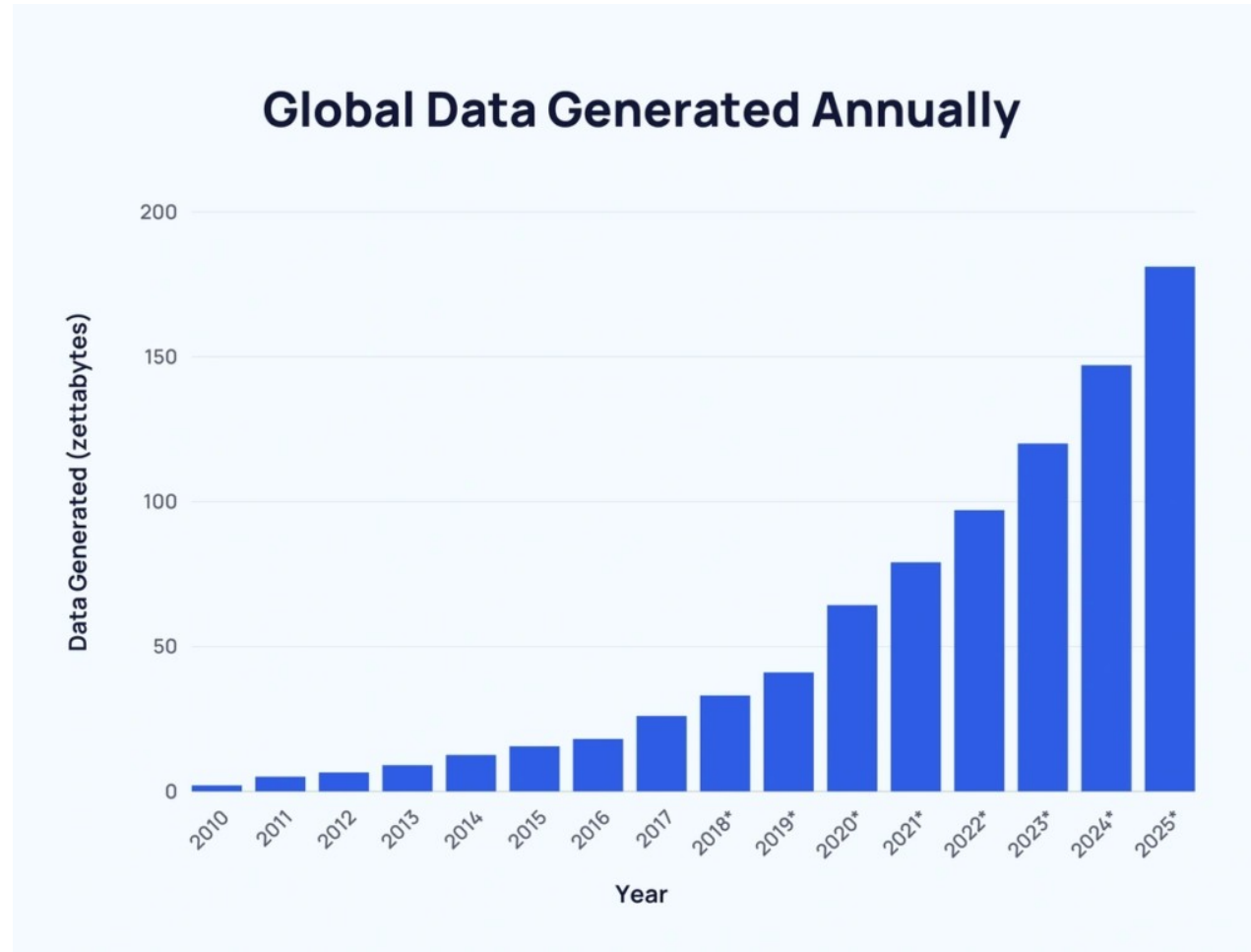
The Fourth Industrial Revolution

- Considered to have started with the Internet of Things
 - Connecting physical devices to the Internet in addition to people
 - Sensors, cameras, scanners – anything that captures and collects data



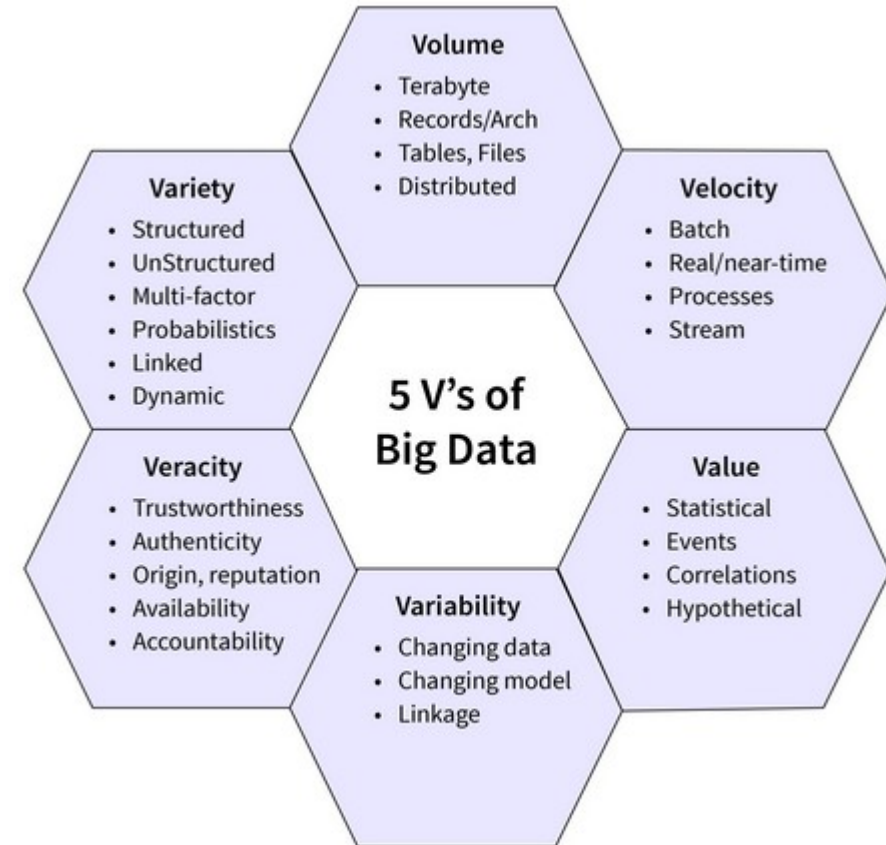
Big Data

- The volume of data generated to be processed increased exponentially



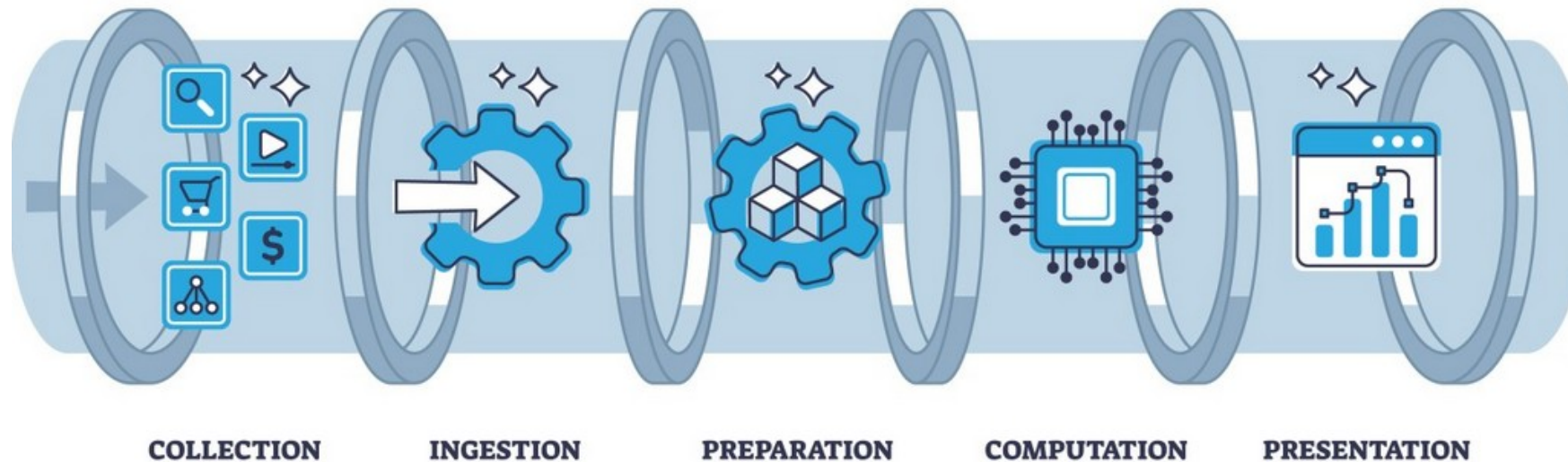
Big Data

- Generally referred to as big data
- Characterized by the V's
- The problem is processing this data at scale
 - High volumes and velocity is streamed into our systems
 - It has to be processed, often in real time
 - For example, to filter out suspect or duplicate data
 - OO and structured programming can't do this at scale
 - But functional programming can



Streaming Data

- This is often modeled as a data pipeline
 - Processing has to be done at each stage on a constant stream of data objects



Functional Data Pipelines

- Generally these applications are designed as
 - A chain of functions called a stream
 - Passes each item through the stream function by function
 - Each function does one transformation or operation on a data item
 - Then passes it on to the next function in the stream
- The functional programming program architecture makes this straightforward
 - The next few slides cover the main aspects of functional programming

Functions as First-Class Citizens

Functions are treated like any other value. They can be:

- Assigned to variables
- Passed as arguments
- Returned as results from other functions

Enables higher-order functions that can take other functions and parameters and/or return other functions as return values

```
def square(x): return x * x
nums = [1, 2, 3]
squares = list(map(square, nums)) # passing function as argument
```

Pure Functions

- A function is pure if:
 - Its output depends only on its inputs
 - It causes no side effects (doesn't change global state, I/O, etc.)
- Pure functions are referentially transparent
 - Calling the same function with the same arguments always yields the same result

Immutability

- Data is not modified after it's created
 - Instead, new data structures are returned as the result of being processed by a function
- Advantages
 - Avoids side effects that may happen when multiple functions use the same data as input
 - Avoids the classic race condition that where multiple processes are accessing the same data value at the same time
 - Makes concurrency easier to implement

Higher-Order Functions

- Functions that operate on other functions
 - Other functions can be passed as arguments or be the return value of a higher order function
- Allows for abstractions for iteration, transformation, event handling, etc.
 - Examples: map, filter, reduce, custom combinators.

```
def apply_twice(f, x):  
    return f(f(x))
```

Lambda Functions

- Allows passing or using a function without having to name it
 - In functional programming, the body of a function is data that can be stored in a variable
 - The name of a function is the variable that the function body is assigned to
 - We can use the function body on its own using some form of lambda notation
 - Called a function literal
- Allows for more declarative programming
 - Instead of writing loops, FP uses expressions and function composition.
 - The example below returns an array of the squares of the even integers in the input array

```
nums = [1, 2, 3, 4]
evens_squared = list(map(lambda x: x*x, filter(lambda n: n % 2 == 0, nums)))
```

Recursion over Iteration

- Functional programming often uses recursion instead of imperative loops
 - Imperative loops use some form of mutable counter that is modified on each iteration
 - Recursion eliminates mutable loop counters
 - Recursion expresses computation in terms of base and recursive cases

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


Expressions

- Emphasis on expressions, not statements
 - Almost everything is an expression that returns a value
 - Allows for creating new functionality by chaining functions together via function composition
 - $h(x) = f(g(x))$
 - Creates a new function $h(x)$ by applying g to x , then applying f to the output of $g(x)$

Streams

- Streams are pipelines that start with a source
- Java generators create the stream from:
 - Collections / Arrays
 - Eg. `Stream<Integer> s = Arrays.asList(1,2,3).stream();`
 - Functions / Iterators
 - Eg. `Stream<Integer> s = Stream.iterate(0, n -> n + 1);` // infinite sequence
 - Files / I/O
 - Eg. `Stream<String> lines = Files.lines(Path.of("data.txt"));`

Stream Generator (Creating Streams)

- In Python, the equivalent concept is iterators and generators:

```
def naturals():  
    n = 0  
    while True:  
        yield n  
        n += 1  
  
stream = naturals()  # generator as a stream
```

Intermediate Methods (Transformations)

- Intermediate methods transform a stream into another stream
 - They are lazy: they don't process elements until a terminal operation is applied
- Common ones:
 - map: transform each element
 - filter: select elements by a predicate
 - distinct: remove duplicates
 - sorted: sort elements
 - limit, skip: slice streams
 - FlatMap: flatten nested structures

java

```
Stream.of(1,2,3,4,5)  
    .filter(n -> n % 2 == 0)    // keep even  
    .map(n -> n * n)           // square
```

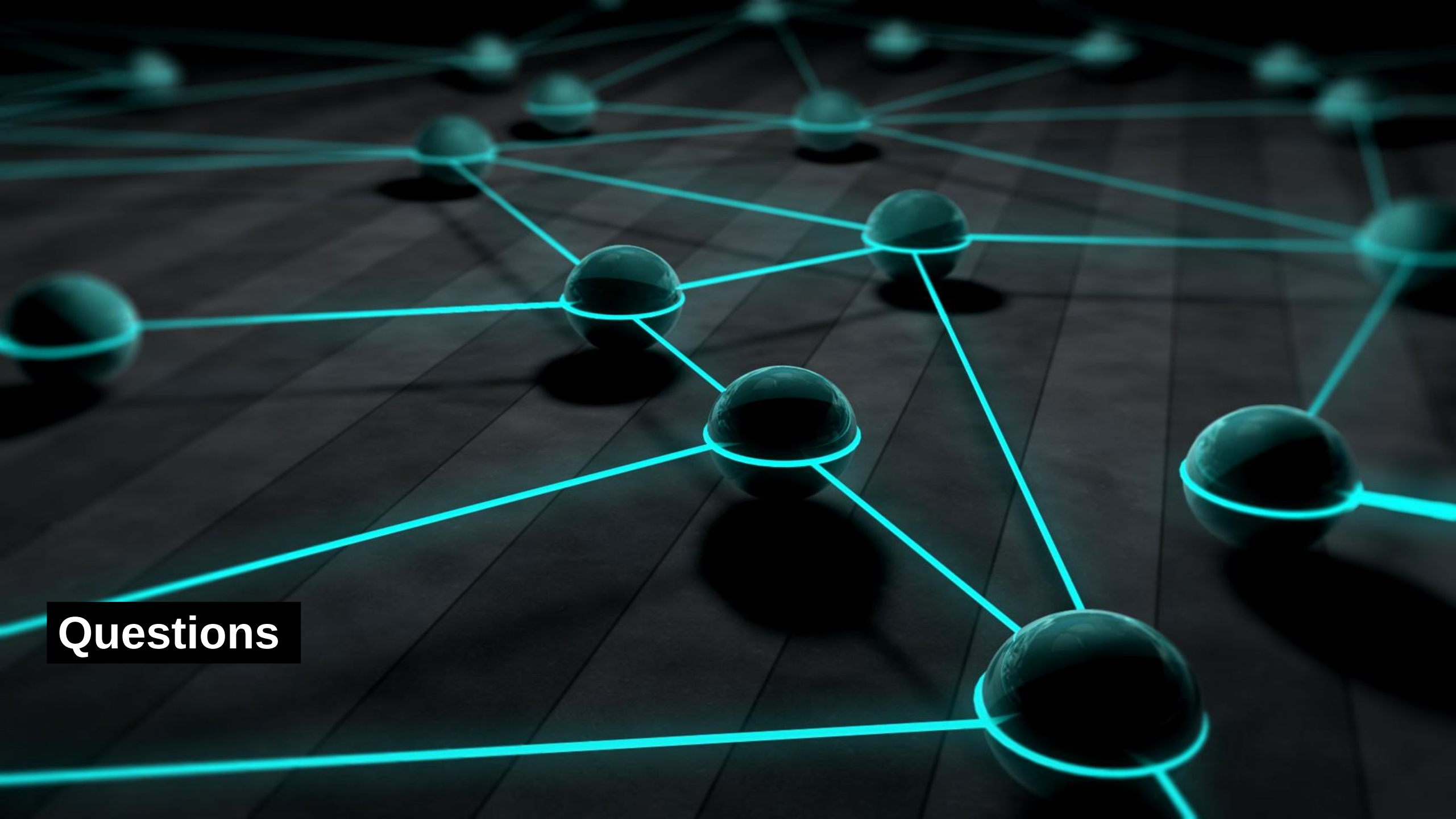
python

```
nums = [1,2,3,4,5]  
result = map(lambda n: n*n, filter(lambda n: n%2==0, nums))
```

Terminal Methods (Consumption)

- Terminal methods end the pipeline and produce a result (value, collection, side effect)
- Common ones:
 - ForEach: apply action to each element
 - collect: gather into a list, set, map, etc.
 - reduce: aggregate into a single result (sum, product, etc.)
 - count, min, max: statistical operations
 - anyMatch, allMatch, noneMatch: boolean checks

```
java
int sum = Stream.of(1,2,3,4,5)
                .filter(n -> n % 2 == 0)
                .map(n -> n * n)
                .reduce(0, Integer::sum); // 20
```



Questions