
# Software management and HPC computing

marco milanesio, paola goatin, regis duvigneau

11-12 / 3 / 2024

part I: Scientific Software and Development Life Cycle

UNIVERSITÉ CÔTE D'AZUR

UNIVERSITÉ CÔTE D'AZUR
MAISON DE LA MODÉLISATION, DE LA SIMULATION ET DES INTERACTIONS

Inria

# Course Overview

- 11 - 3 - 2024
  - Scientific software
  - Design and development
  - Data structures
  - Versioning
  - Open science

- 12 - 3 - 2024
  - Parallelisation and HPC
  - Documentation

# Course Overview

- Lectures + Live coding
- 5 practical sessions
  - data structures
  - versioning
  - testing
  - parallelisation
  - HPC cluster

`https://github.com/UCA-MSI/formation_datahyking`

# Contacts

marco.milanesio@univ-cotedazur.fr

paola.goatin@inria.fr

regis.duvigneau@inria.fr

# Agenda

- 11 - 3 - 2024
  - **Scientific software**
  - Design and development
  - Data structures
  - Versioning
  - Open science

- 12 - 3 - 2024
  - Parallelisation and HPC
  - Documentation

# Scientific Software

# Examples of Scientific Software

- Third party libraries  and tools
- Developed algorithms
- Testing suites
- Pipelines
- Statistics and reporting
- Data preprocessing
- …

More generally:

- Third party
- Active development

# Definitions

- Data acquisition and management
- Numerical analysis
- Visualisation
- Reproducibility
- Open source

Benefits:

- Increased efficiency
- Improved accuracy
- Greater collaboration

# Importance

- Enhance efficiency and productivity
  - automation
  - data management and organisation
- Improve accuracy and reliability
  - high precision
  - standardised protocols and methods
- Facilitate collaboration
  - sharing, transparency
  - open science
- Drive innovation
  - test hypothesis
  - data analysis

# Challenges

- Domain specific and complexity
- Funding and sustainability
- Reproducibility and OSS Dilemmas
- UI / UX
- Integration and interoperability
- Testing and validation
- Education and training

# Agenda

- 11 - 3 - 2024
  - Scientific software
  - **Design and development**
  - Data structures
  - Versioning
  - Open science

- 12 - 3 - 2024
  - Parallelisation and HPC
  - Documentation

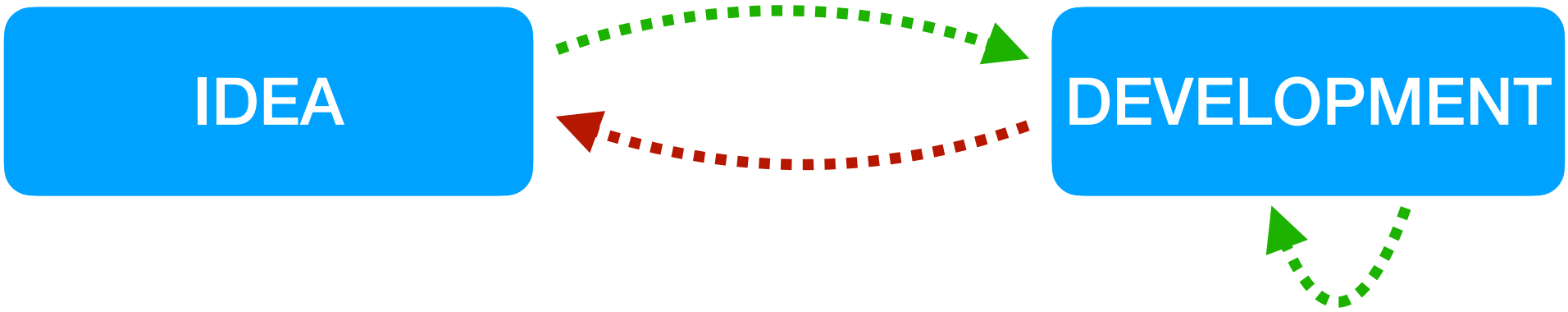
# Design and development

# Software development life-cycle

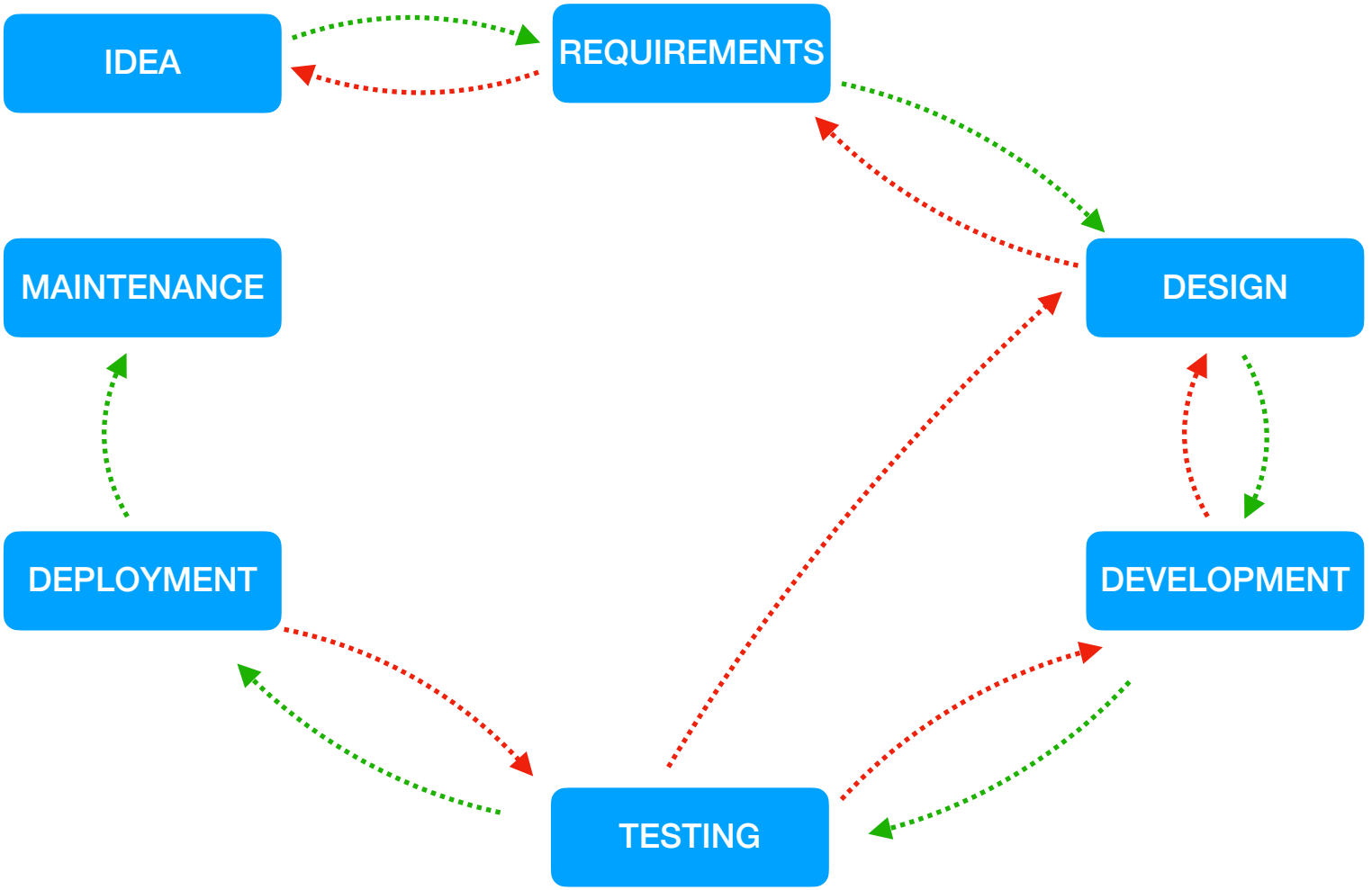
Goals of a SDLC:

- Provide reproducible steps and backlogs for all decisions that are made during a (CS) research project.
- Find inconsistencies ASAP
- Evaluate the cascading impact of earlier decisions
- Set up a 1-to-1 (hopefully) mapping between design and development
- Three types:
  - The usual
  - The optimal
  - The sensible

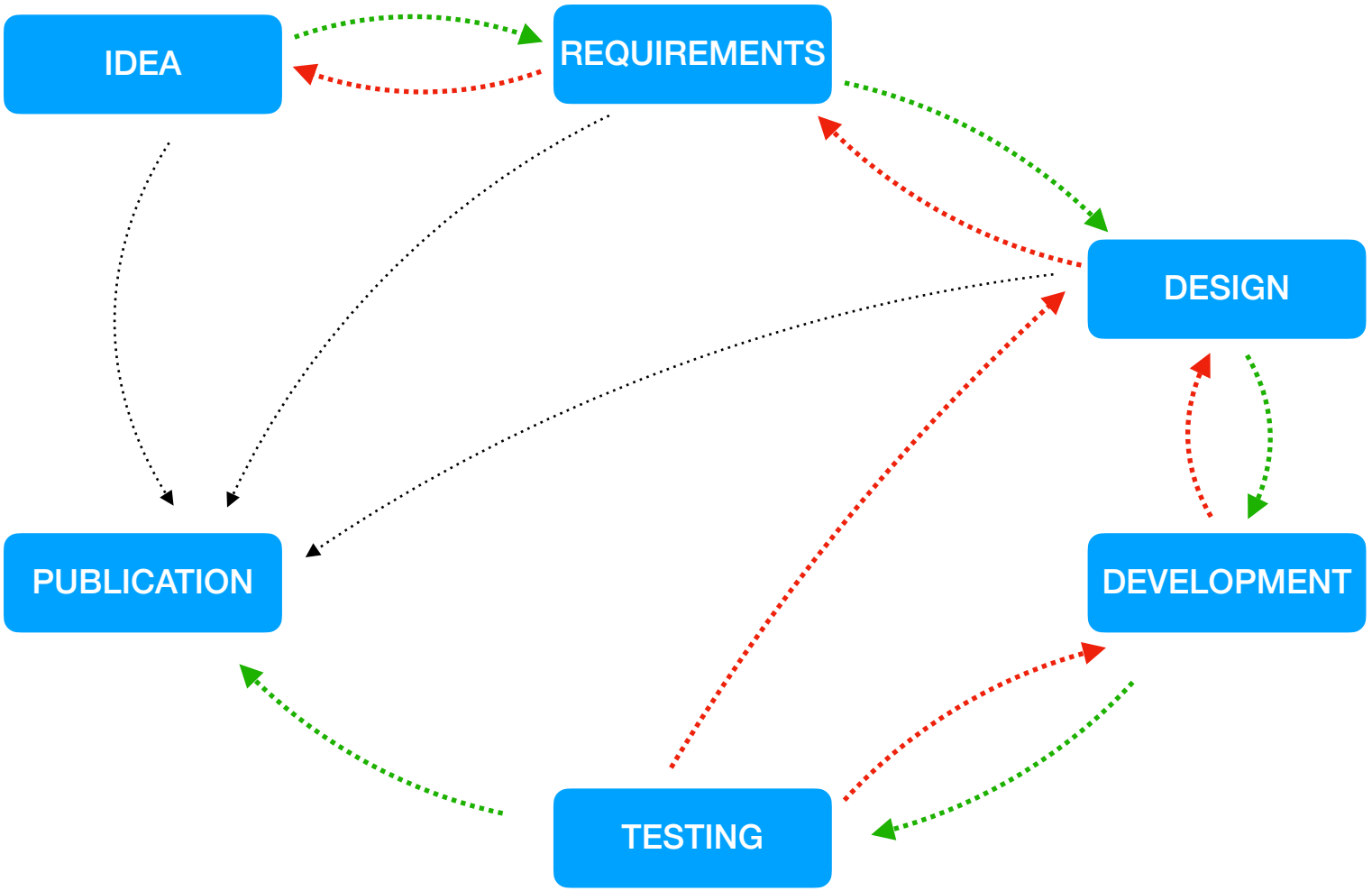
# The usual



# The optimal

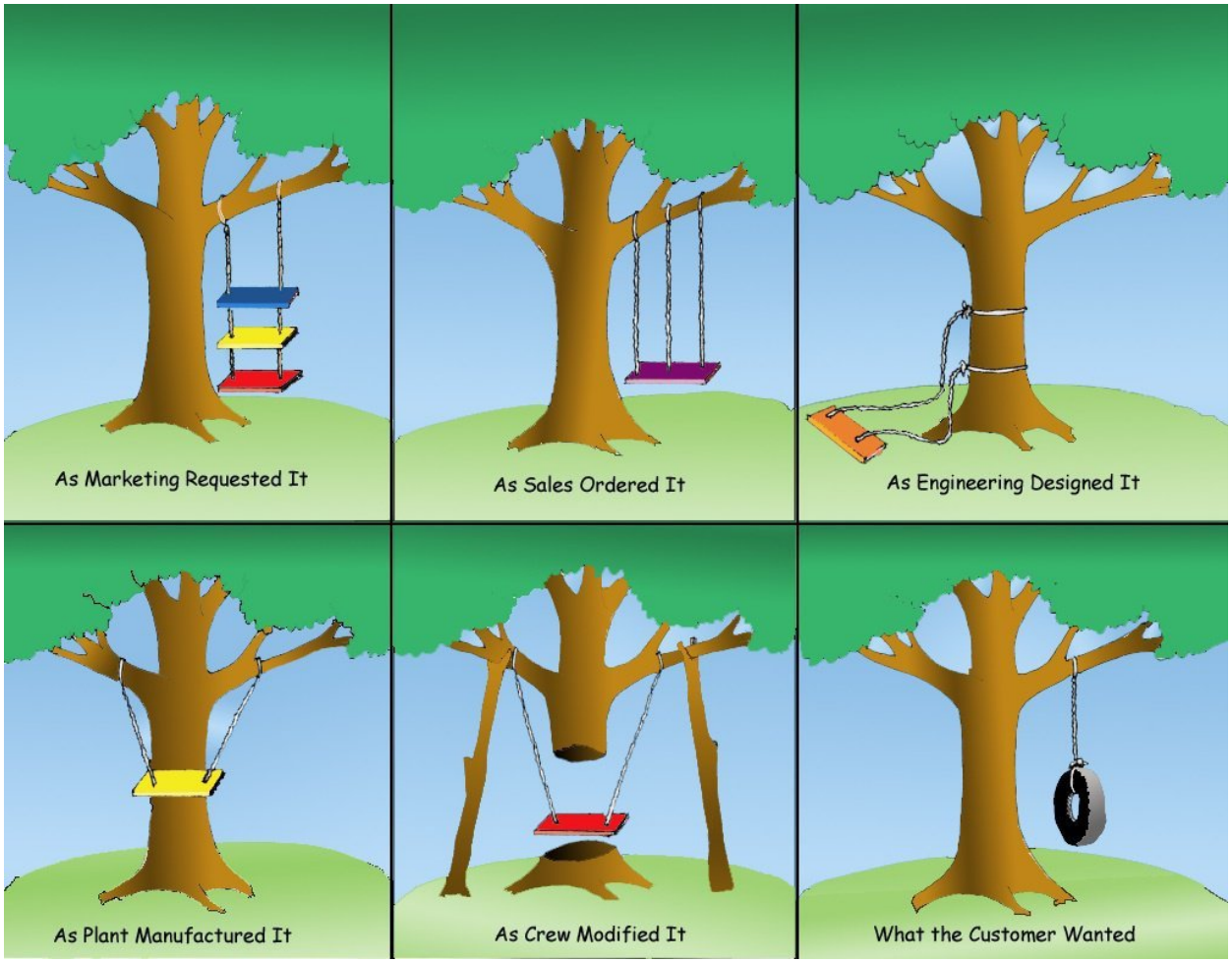

IDEA
REQUIREMENTS
MAINTENANCE
DESIGN
DEPLOYMENT
DEVELOPMENT
TESTING

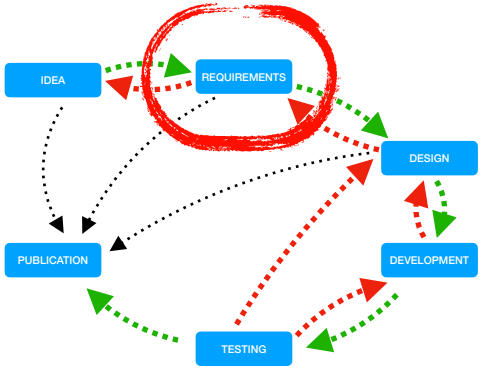
# The sensible



# Requirements

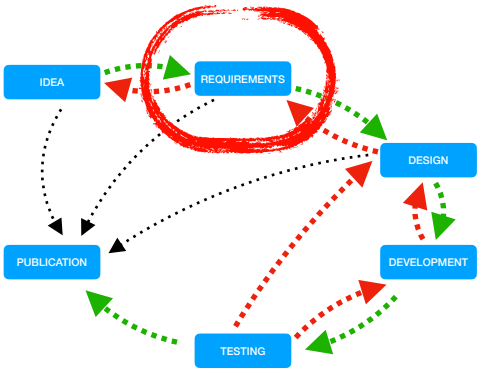
- Elicitation
- Goals
  - what to do
  - how to do it
- Why
  - use cases



# Requirements

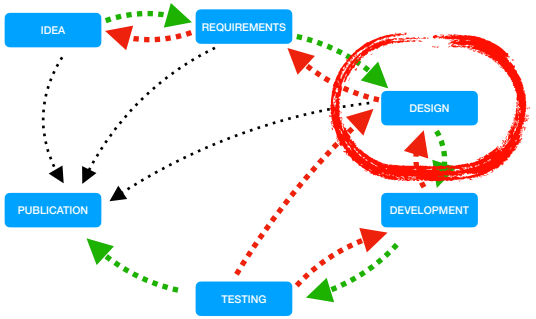
- "Customer" specification requirements
  - "*I want a tool to invert a matrix*"
- Software architecture requirements
  - "*I want a backend to serve my inversions*"
- Software design requirements
  - "*I want it as an independent module*"
- Functional requirements
  - "*Check for invertibility*"
  - "*Given a matrix, return a matrix*"
- User interface requirements
  - "*I want a web application*"



# Design

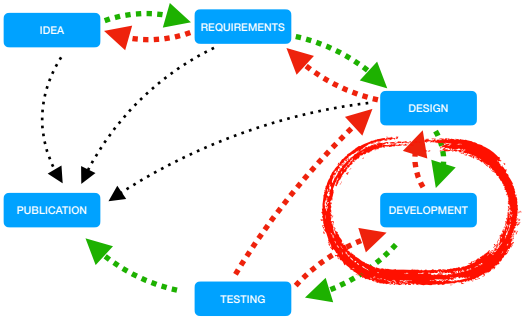
- Architecture
  - How and where
  - Tiers
- Functionalities
- Interactions
- Dependencies
  - Flows
  - Third party
- Language
- Paradigm
  - Object Oriented / FP / …

Notice the order…



# Development

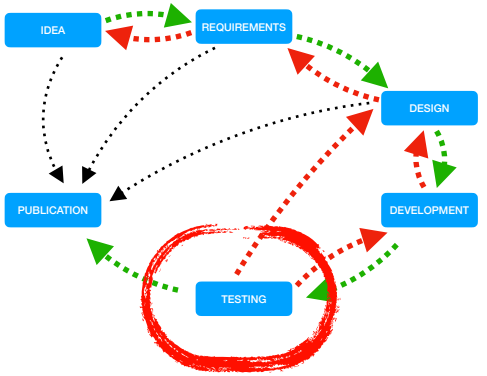
- Exploration
- Identify functionalities
- Modules
- (Classes)
- Packages
- Dependencies
- Unit tests
- Early documentation



# Testing

- Requirements analysis
- Planning
- Test case development
- Environment setup
- Test execution
- Test cycle closure

- Unit testing
- API testing
- Integration testing
- System testing
- Setup (installing/uninstall)
- Agile testing



# Best practices

- Modularity
- Maintainability
- "Clean" code
- Version control
- Testing and validation
- Documentation

# Paradigms

- Imperative
  - Describe **what** to do and **how** to do it
    - procedural (*procedures calling each others)*
    - object oriented (*state and behaviour + messages)*

- Declarative
  - Describe the desired **result**
    - functional (*as a sequence of functions evaluations)*
    - logic (*as an answer to a question about rules and facts)*
    - reactive (*as the cascading effect of a data streams)*

# An example (overkill)

- Problem: Extract the odd elements from a list of integers into another list.
- 4 solutions:
  - Imperative (in C)
  - OO (in Python)
  - FP (in Haskell)
  - Logic (in Prolog)

# Solutions

```c
#include <stdio.h>

void extract_odd_c(int *list, int length, int *output) {
  int j = 0;
  for (int i = 0; i < length; i++) {
    if (list[i] % 2 != 0) {
      output[j++] = list[i];
    }
  }
}
```

```python
class ListFilter:
  def __init__(self, list):
    self.list = list

  def extract_odd(self):
    return [item for item in self.list if item % 2 != 0]
```

```haskell
extractOdd :: [Int] -> [Int]
extractOdd xs = [x | x <- xs, x `mod` 2 == 1]
```

```prolog
extract_odd([], []).
extract_odd([H | T], Odd) :-
  extract_odd(T, OddTemp),
  (   H mod 2 =:= 1 -> Odd = [H | OddTemp] ; Odd = OddTemp ).
```

# Paradigms

- Procedural
  - the most straightforward one
  - "enter the car, turn the key, press the pedal, move 1 km"
- OO
  - objects live with a state that has to be maintained
  - messages
  - "instantiate a car object, update its status by invoking the "turn_key" method. Call the "press_pedal" method until the object coordinates are shifted of 1 km.
- FP
  - Just describe the result
  - "move 1 km by applying the function "press_pedal" on the result of the evaluation of the function "turn_key" on "car"

# OO vs FP

- Modular design
- Encapsulation
- Polymorphism
- Inheritance
- Abstraction

- Over engineering
- Performance overhead
- Learning curve

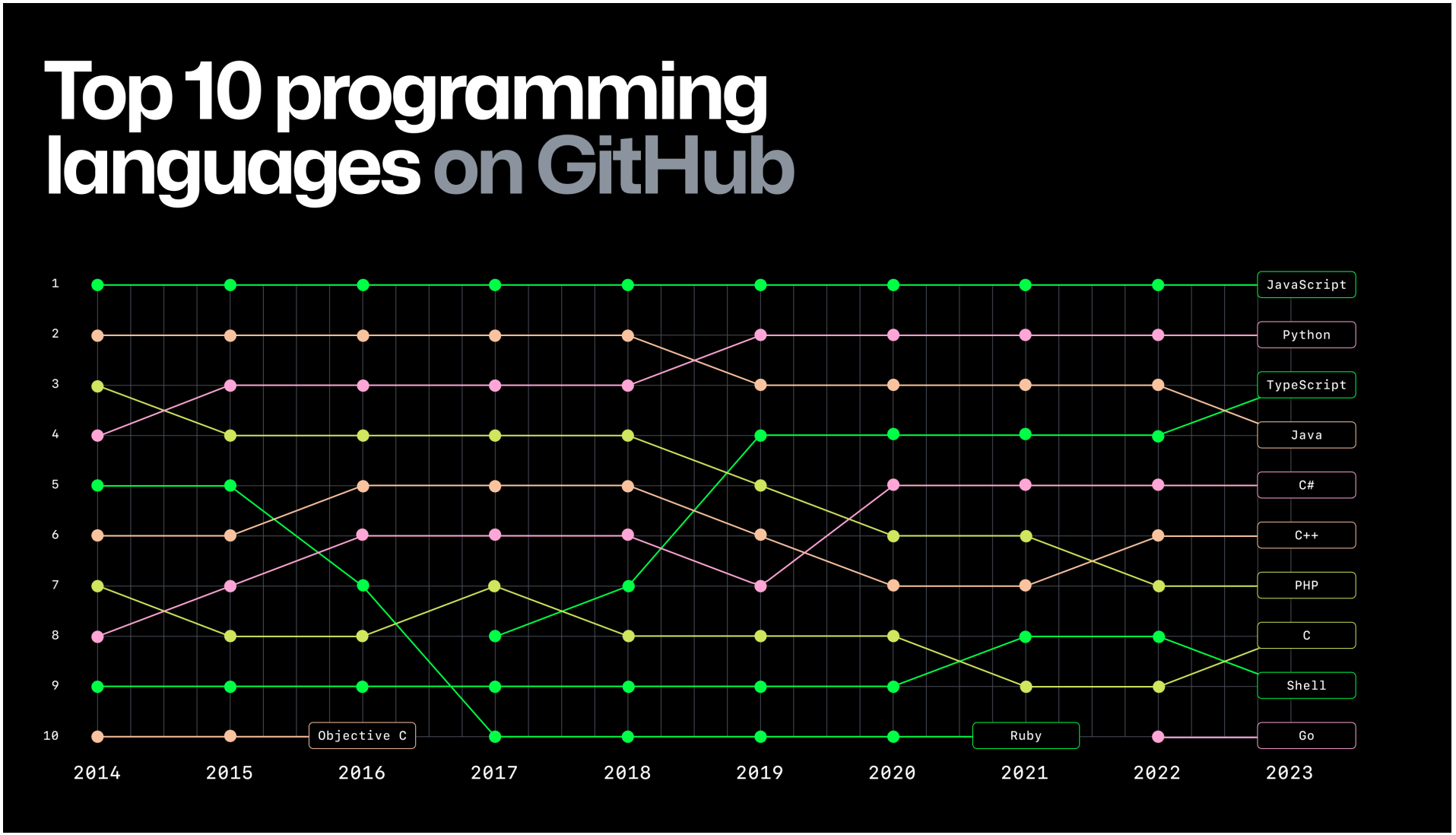
- Immutability
- Declarative style
- Pure functions
- Composability
- Lazy evaluation

- Performance overhead
- Learning curve
- "Limited libraries"
- Debugging

# Python

- Imperative / OO / FP
- Interactive (REPL)
- Easy
  - Cleaning, counting, organising, …
- Secret weapons:
  - builtins
  - "collections" module
- C-Python, R-Python, Jython, …
- Runs everywhere

# Python




https://github.blog/2023-11-08-the-state-of-open-source-and-ai/

# Flexibility and performance

- Dynamic typing
  - Make data structures out of anything (almost)
- Builtin types are fast for coding
- Explore ideas
- Optimise later
- Abstract away annoying details (memory)
- `import this`

# Agenda

- 11 - 3 - 2024
  - Scientific software
  - Design and development
  - **Data structures**
  - Versioning
  - Open science

- 12 - 3 - 2024
  - Parallelisation and HPC
  - Documentation

# Builtin types

- tuple ()
  - record structure
- list []
  - mutable sequence
- set {}
  - uniqueness
- dict {:}
  - mapping, lookup tables
- collections.Counter
  - histograms

# Iterations & Co.

- Iterations

```
for item in sequence:
    …
```

- Variants

```
for pos, item in enumerate(sequence):
    …
for x, y in zip(sequence1, sequence2):
    …
```

- Reductions

```
sum(sequence)
min(sequence)
max(sequence)
any(sequence)
all(sequence)
```

# [list,set,dict]-comprehension

- List comprehension

```
[ expr for x in iterable if condition ]
```

- Set comprehension

```
{ expr for x in iterable if condition }
```

- Dict comprehension

```
{ k:v for k,v in iterable if condition}
```

# Generators

- Generator expression

```
( expr for x in iterable if condition )
```

- Combined with reduction

```
sum(expr for x in iterable if condition)
```

- This allows you to process HUGE amounts of data **incrementally**
  saving tons of memory!
  – feed loops...

# Superpowers

- Higher order functions

```
map(func, sequence)
filter(func, sequence)
functools.reduce
```

- Anonymous functions

```
filter(lambda x: x % 2 == 0 for x in range(10))
```

- Iterators

```
it = iter(sequence)
it.__next__   # next(it)
```

- Variable arguments functions

```
func(*args, **kwargs)
```

# And more

- numpy
  - numerical computations
- pandas
  - data analysis
- scikit-learn
  - machine learning
- torch
  - deep learning
- …

# Practical Session
# Python