**Programmation Fonctionnelle : des concepts aux applications web (NFP119)**

# Paradigme fonctionnel

**Tristan Crolard**

Laboratoire CEDRIC
Equipe « Systèmes Sûrs »

**tristan.crolard@cnam.fr**

**cedric.cnam.fr/sys/crolard**

# Imperative vs Functional programming

- imperative programming usually seems simpler to most learners
- useful to implement an algorithm, but not to define a specification
- functional programming provides more declarative alternatives:
  - `list` and `dict` comprehension
  - immutable data structures
  - recursive functions
- more **advanced features** are also available:
  - higher order functions
  - lambda expressions (anonymous functions)
  - `map`, `filter`, `reduce`, often used with lambda expressions

# Pure functional programming – requirements

Pure functions are effect-free functions that contains:

- no mutation of (mutable) datatypes such as `list` and `dict`
  can be enforced by using instead `Sequence` and `Mapping`
- no assignement of variables (only initializations are allowed)
  can be enforced by adding a `Final` type hint
- no composite statement except `if-else` (and `match`).
  - no block (a block statement is a group of statements)
    only local declarations are allowed
  - no loop (no `for` loop, no `while` loop)
    only list comprehension and recursive functions are allowed

**Note.** By fulfilling these requirements, you can ensure that your code is purely functional, and that its mathematical meaning is obvious. Pure functions can be used in specifications (models and properties), and thus also for testing.

# Pure functions – definitions

Pure functions can be defined only:

- explicitely
  - by just returning an expression (list comprehension is allowed)
- by case analysis
  - using `if-else` or `match`
- by recursion (and also by case analysis)
  - on primitive datatypes (integer, lists)
  - on user-defined recursive datatypes (tree-like data structures)

**Note.** You should ensure that recursive functions always terminate on valid inputs.