

From Python to Lean: a quick start

Functional Programming in Lean, Chapter 1

Le Chen

Auburn University

Department of Mathematics and Statistics

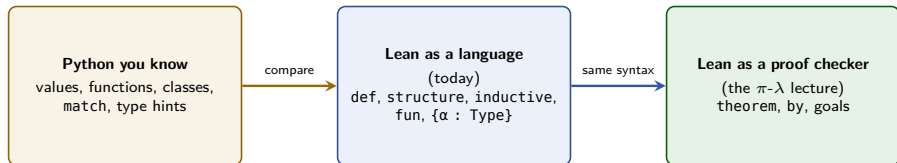
Math 7800: Probability Theory

Auburn University, Fall 2026

Following *Functional Programming in Lean* by David Thrane Christiansen

(lean-lang.org/functional_programming_in_lean)

Why a programmer's view helps a mathematician



Why start with programs?

- ▶ Every Mathlib file is written in the same language as a program: `def`, `structure`, implicit arguments, dot notation. Reading Mathlib means reading code.
- ▶ Lean's error messages are type errors. Learning to read them on small programs is cheaper than on measure theory.
- ▶ Lean computes: `#eval` runs your definitions, so you can test an idea before you try to prove anything.

Today's map: FPIL Chapter 1, section by section

FPIL section	Lean	Python you know
1.1 Evaluating expressions	<code>#eval</code> , <code>f x</code>	the REPL, <code>f(x)</code>
1.2 Types	<code>Nat</code> , <code>Int</code> , <code>#check</code>	<code>int</code> , <code>type()</code> , hints
1.3 Functions and definitions	<code>def</code> , <code>:=</code> , currying	<code>def</code> , <code>partial</code>
1.4 Structures	<code>structure</code> , <code>with</code>	<code>@dataclass</code> , <code>replace</code>
1.5 Datatypes and patterns	<code>inductive</code> , <code>match</code>	<code>Enum</code> , unions, <code>match</code>
1.6 Polymorphism	<code>{α : Type}</code> , <code>List</code> , <code>Option</code>	<code>TypeVar</code> , <code>list</code> , <code>None</code>
1.7 Strings and slices	<code>String</code> , <code>Char</code>	<code>str</code>
1.8 Conveniences	<code>fun</code> , <code>·</code> , <code> ></code> , namespaces, <code>s!</code>	<code>lambda</code> , comprehensions, modules, f-strings
1.9 Summary	then: from programs to proofs	

Companion files (run them yourself): `student/Math7800Lean/FPIL/Ch1.lean` and `student/python/fpil_ch1.py`.

#eval is your REPL

Python a script, run with python3

```
print(1 + 2 * 5)           # → 11
print((1 + 2) * 5)         # → 15
print("Hello, " + "Lean!") # → Hello, Lean!
print(max(13, 14))         # → 14
print("".join(["great ", "oak ", "tree"]))
# → great oak tree
```

Lean a file; results appear in the Infoview

```
#eval 1 + 2 * 5    -- → 11
#eval (1 + 2) * 5  -- → 15
#eval String.append "Hello, " "Lean!"
-- → "Hello, Lean!"
#eval max 13 14    -- → 14
#eval String.append "great "
  (String.append "oak " "tree")
-- → "great oak tree"
```

Python instinct vs Lean

- ▶ No prompt to type into: write #eval e in the file and put the cursor on it. The editor is the REPL.
- ▶ Function application is a space: max 13 14, not max(13, 14). Parentheses only group: String.append "great " (String.append ...).
- ▶ #eval shows a string with its quotes, like the Python REPL's repr; print drops them.

Only expressions: `if` always has a value

Python an expression and a statement

```
print("it is " + ("yes" if 1 > 2 else "no"))  
# ⇒ it is no  
# The if statement has no value; it runs code.  
if 1 > 2:  
    answer = "yes"  
else:  
    answer = "no"  
print(answer) # ⇒ no
```

Lean only the expression

```
#eval String.append "it is "  
  (if 1 > 2 then "yes" else "no")  
-- ⇒ "it is no"  
#eval if 3 == 3 then 5 else 7 -- ⇒ 5
```

```
String.append "it is " (if 1 > 2 then "yes" else "no")  
⇒ String.append "it is " (if false then "yes" else "no")  
⇒ String.append "it is " "no"  
⇒ "it is no"
```

Evaluation = substitution (FPIL 1.1).

Python instinct vs Lean

Python has an `if` statement (runs code, no value) and a conditional expression (`a if c else b`). Lean has only `if c then a else b`, an expression, so the `else` is not optional. Chapter 1 never reassigns a variable: once a name has a value, it keeps it. `do` blocks offer `for` and `let mut` as sugar over the same pure functions; FPIL covers them later (Monads, Monad Transformers).

A function waiting for its argument (FPIL 1.1.1, messages)

Python a partially applied function prints as an object

```
prefix_it_is = partial(operator.add, "it is ")
print(prefix_it_is)
# => functools.partial(<built-in function add>, 'it is ')
print(prefix_it_is("no")) # => it is no
```

Lean #eval refuses to print a function

```
#eval String.append "it is "
-- => error
```

```
error: could not synthesize a `Repr` or `ToString` instance for
↳ type
  String → String
```

Lean but #check shows its type, and it can be applied

```
#check String.append "it is "
-- => "it is ".append : String → String
#eval (String.append "it is ") "no"
-- => "it is no"
```

Types are checked before anything runs

Python hints are not enforced at run time

```
def add1(n: int) -> int:
    return n + 1

try:
    add1("7")          # the hint is not enforced
except TypeError as e:
    print(e)
# => can only concatenate str (not "int") to str
print(type(1 - 2))    # => <class 'int'>
```

Lean (e : T) annotates; #check asks

```
#eval (1 + 2 : Nat)    -- => 3
#check (1 - 2 : Int)   -- => 1 - 2 : Int
#check "hello"         -- => "hello" : String
```

```
#check String.append ["hello", " "] "world"
```

```
error: Application type mismatch: The argument
["hello", " "]
has type
  List String
but is expected to have type
  String
in the application
  String.append ["hello", " "]
```

Python instinct vs Lean

type(x) runs x first; #check e only computes the type. In Python the bad call fails when it runs (a checker such as mypy would catch it earlier, if you run one). In Lean the file does not compile.

The Nat trap: $1 - 2 = 0$

Python int is Lean's Int

```
print(1 - 2)           # → -1
print(max(1 - 2, 0))   # → 0
print(2 ** 100)
# → 1267650600228229401496703205376
```

Lean Nat subtraction stops at 0

```
#eval (1 - 2 : Nat)    -- → 0
#eval (1 - 2 : Int)    -- → -1
#eval (2 - 5 + 5 : Nat) -- → 5
#eval (2 + 5 - 5 : Nat) -- → 2
#eval 2 ^ 100
-- → 1267650600228229401496703205376
```

Python instinct vs Lean

A literal like 1 is a Nat unless you say otherwise. On Nat, $a - b$ is $\max(a - b, 0)$, so $(2 - 5) + 5 = 5 \neq 2$: the algebra you expect breaks. Like Python's int, Nat and Int are arbitrary precision: 2^{100} is exact.

Seen before

The same truncation as $\mathbb{R}_{\geq 0}$ in the π - λ lecture, where $\mu(A^c) = \mu(\Omega) - \mu(A)$ needs $\mu(A) \neq \infty$.

/ on Nat is floor division

Python // floors, / gives a float

```
print(7 // 2)    # => 3
print(7 % 2)     # => 1
print(-7 // 2)   # => -4
print(7 // -2)   # => -4
print(7 % -2)    # => -1
print(7 / 2)     # => 3.5
try:
    7 // 0
except ZeroDivisionError as e:
    print(e)
# => integer division or modulo by zero
```

Lean / follows the type

```
#eval 7 / 2           -- => 3
#eval 7 % 2           -- => 1
#eval 7 / 0           -- => 0
#eval (-7 / 2 : Int)  -- => -4
#eval (7 / -2 : Int)  -- => -3
#eval (7 % -2 : Int)  -- => 1
#eval Int.fdiv 7 (-2) -- => -4
#eval 7.0 / 2.0        -- => 3.500000
#eval (7 : Float) / 2  -- => 3.500000
```

Python instinct vs Lean

- ▶ $7 / 2$ is 3 on Nat: there, Lean's $/$ is Python's $//$. For 3.5, use Float.
- ▶ On Int, $/$ is `Int.ediv` (Euclidean: the remainder $\%$ is never negative). It agrees with Python only for a positive divisor: $7 / -2$ is -3 (Python -4), $7 \% -2$ is 1 (Python -1). Python's $//$ is `Int.fdiv`.
- ▶ $7 / 0 = 0$ is a convention (Python raises `ZeroDivisionError`), so $/$ needs no proof argument. Crashing functions exist too (panic!; head!, on the Option slide).

def for everything; := defines

Python = for values, def for functions

```
hello = "Hello"
lean: str = "Lean"
print(hello + " " + lean)    # → Hello Lean

def maximum(n: int, k: int) -> int:
    return k if n < k else n

print(add1(7))               # → 8
print(maximum(5 + 8, 2 * 7)) # → 14
```

Lean one keyword, def

```
def hello := "Hello"
def lean : String := "Lean"
#eval String.append hello (String.append " " lean)
-- → "Hello Lean"

def add1 (n : Nat) : Nat := n + 1
def maximum (n : Nat) (k : Nat) : Nat :=
  if n < k then k else n
#eval add1 7 -- → 8
#eval maximum (5 + 8) (2 * 7) -- → 14
```

Lean = is a statement, == a test

```
#check 3 = 3 -- → 3 = 3 : Prop
#check 3 == 3 -- → 3 == 3 : Bool
```

Python instinct vs Lean

Python's = assigns and can be repeated. Lean's := defines once; a name can be used only after its definition.

Every function takes one argument (currying)

Python uncurried by default; currying by hand

```
Uncurried = Callable[[int, int], int]
Curried = Callable[[int], Callable[[int], int]]
f: Uncurried = maximum
g: Curried = lambda n: lambda k: maximum(n, k)
print(g(3)(1), g(3)(10))      # → 3 10
print(partial(maximum, 3)(10)) # → 10
try:
    maximum(3)
except TypeError as e:
    print(e)
# → maximum() missing 1 required
#    positional argument: 'k'
```

Lean $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ means $\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat})$

```
#check add1      -- → add1 (n : Nat) : Nat
#check (add1)    -- → add1 : Nat → Nat
#check (maximum)
-- → maximum : Nat → Nat → Nat
#check maximum 3 -- → maximum 3 : Nat → Nat
#eval (maximum 3) 1 -- → 3
#eval (maximum 3) 10 -- → 10
```

```
#check spaceBetween "Hello "
-- → spaceBetween "Hello " : String → String
```

How to read it

In Lean `maximum 3` is a function: give it `k` and it returns `maximum 3 k`. In Python `maximum(3)` is a `TypeError`; partial or nested lambdas build the function by hand. So $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ is `Callable[[int], Callable[[int], int]]`, not `Callable[[int, int], int]`. `#check add1` shows the signature; `#check (add1)` shows the type.

Types are values too (FPIL 1.3.2)

Python a type alias; classes are objects

```
# 3.12+: type Str = str
Str: TypeAlias = str
a_str: Str = "This is a string."
print(type(str)) # => <class 'type'>
```

Lean a type is an expression of type Type

```
def Str : Type := String
def aStr : Str := "This is a string."
#check (Str) -- → Str : Type
#check Nat -- → Nat : Type
def NaturalNumber : Type := Nat
```

def vs abbrev

A def of a type is not unfolded when Lean looks for how to read the numeral 38. An abbrev is always unfolded ("reducible"). **Why it matters for proofs:** what simp, instance search and rfl can see through depends on this. simp and instance search unfold an abbrev, not a def; rfl unfolds both.

```
def thirtyEight : NaturalNumber := 38 -- → error
```

error: failed to synthesize instance of type class
OfNat NaturalNumber 38

```
def thirtyEight : NaturalNumber := (38 : Nat)
abbrev N : Type := Nat
def thirtyNine : N := 39
```

structure $\approx$ @dataclass(frozen=True)

Python a frozen dataclass

```
@dataclass(frozen=True)
class Point:
    x: float
    y: float

def add_points(p1: Point, p2: Point) -> Point:
    return Point(p1.x + p2.x, p1.y + p2.y)
origin = Point(0.0, 0.0)
print(origin) # => Point(x=0.0, y=0.0)
a, b = Point(1.5, 32), Point(-8, 0.2)
print(add_points(a, b))
# => Point(x=-6.5, y=32.2)
```

Lean a new type with named fields

```
structure Point where
  x : Float
  y : Float
def origin : Point := { x := 0.0, y := 0.0 }
#eval origin
-- => { x := 0.000000, y := 0.000000 }
#eval origin.x -- => 0.000000
def addPoints (p1 : Point) (p2 : Point) : Point :=
  { x := p1.x + p2.x, y := p1.y + p2.y }
#eval addPoints { x := 1.5, y := 32 }
               { x := -8, y := 0.2 }
-- => { x := -6.500000, y := 32.200000 }
```

Python instinct vs Lean

Braces with `:=` build a value: `{ x := 0.0, y := 0.0 }`; Lean must know the expected type (here from `: Point`). Fields are read with dot notation, as in Python. Float prints six decimals.

Updating a structure makes a new one

Python `dataclasses.replace`

```
four_and_three = Point(4.3, 3.4)
print(replace(four_and_three, x=0.0))
# → Point(x=0.0, y=3.4)
print(four_and_three)
# → Point(x=4.3, y=3.4)
try:
    four_and_three.x = 0.0 # not allowed
except AttributeError as e:
    print(type(e).__name__)
# → FrozenInstanceError
```

Lean `{ p with x := 0 }`

```
def zeroX (p : Point) : Point := { p with x := 0 }
def fourAndThree : Point := { x := 4.3, y := 3.4 }
#eval zeroX fourAndThree
-- → { x := 0.000000, y := 3.400000 }
#eval fourAndThree
-- → { x := 4.300000, y := 3.400000 }
```

Python instinct vs Lean

“Set x to 0” in Lean always means: build a fresh `Point` with the new `x` and the old `y`. There is no `p.x = 0` at all. Python’s frozen dataclass plus `replace` is the same discipline, by choice.

Behind the scenes: constructor, accessors, dot notation

Python methods; an int passes as a float

```
print(str.upper("abc"), "abc".upper())  
# => ABC ABC  
  
def modify_both(f: Callable[[float], float],  
               p: Point) -> Point:  
    return Point(f(p.x), f(p.y))  
  
print(modify_both(math.floor, four_and_three))  
# => Point(x=4, y=3)
```

Lean Point.mk, Point.x, t.f

```
#check Point.mk 1.5 2.8  
-- => { x := 1.5, y := 2.8 } : Point  
#check (Point.mk)  
-- => Point.mk : Float → Float → Point  
#check (Point.x) -- => Point.x : Point → Float  
#eval "one string".append " and another"  
-- => "one string and another"  
def Point.modifyBoth (f : Float → Float)  
  (p : Point) : Point :=  
  { x := f p.x, y := f p.y }  
#eval fourAndThree.modifyBoth Float.floor  
-- => { x := 4.000000, y := 3.000000 }
```

Differences from Python classes

- ▶ A Lean constructor only stores data: no `__init__` code, no validation.
- ▶ `t.f` args calls `T.f` with `t` as its first argument of type `T`, wherever it sits. So `Point.modifyBoth` works with dot notation although `p` comes second.
- ▶ `math.floor` returns an `int`, and even `mypy` accepts an `int` where a `float` is hinted (numeric promotion), so the Python point holds `ints`. Lean's `Float.floor` returns a `Float`, so the Lean point holds `Floats`.

inductive: a type made of constructors

Python a union of frozen dataclasses

```
print(True + True) # → 2

@dataclass(frozen=True)
class Zero:
    pass
@dataclass(frozen=True)
class Succ:
    n: "Zero | Succ"

four = Succ(Succ(Succ(Succ(Zero()))))
print(four)
# → Succ(n=Succ(n=Succ(n=Succ(n=Zero()))))
```

Python no data in any constructor: an Enum

```
class Firewood(Enum):
    BIRCH = auto()
    PINE = auto()
    BEECH = auto()

print(list(Firewood))
# → [<Firewood.BIRCH: 1>, <Firewood.PINE: 2>,
#    <Firewood.BEECH: 3>]
```

Lean the book's definitions of Bool and Nat

```
inductive Bool where
| false : Bool
| true  : Bool

inductive Nat where
| zero : Nat
| succ (n : Nat) : Nat
```

```
#eval Nat.succ (Nat.succ Nat.zero) -- → 2
```

```
#eval true + true -- → error
```

```
error: failed to synthesize instance of type class
HAdd Bool Bool ?m.4
```

Python instinct vs Lean

Python's bool is a kind of int (`True + True == 2`); Lean's Bool is its own type with two constructors and no arithmetic. Nat is zero or succ n: 4 is succ (succ (succ (succ zero))).

match: Lean checks every case is covered

Python match (3.10+) on dataclasses

```
def pred(n: Zero | Succ) -> Zero | Succ:
    match n:
        case Zero():
            return Zero()
        case Succ(k):
            return k

print(pred(Succ(Succ(Zero()))))
# → Succ(n=Zero())
print(pred(Zero())) # → Zero()
```

```
def is_zero(n: Zero | Succ) -> bool:
    match n:
        case Zero():
            return True

print(is_zero(Succ(Zero()))) # → None
```

Lean match on constructors

```
def isZero (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ _ => false
def pred (n : Nat) : Nat :=
  match n with
  | Nat.zero => Nat.zero
  | Nat.succ k => k
#eval isZero 5 -- → false
#eval pred 5 -- → 4
#eval pred 0 -- → 0
```

```
def isZero (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
```

error: Missing cases:
(Nat.succ _)

Why it matters for proofs

An exhaustive match is the same idea as cases and induction: one goal per constructor. In the π - λ proof, induction `t`, `ht` using `...` with `opens | empty => ...`, `| basic ...`, `| compl ...`, `| iUnion ...`, and Lean insists that every case is handled.

Recursion instead of loops, and it must stop

Python fails at run time, after 1000 calls

```
def even(n: int) -> bool:
    return True if n == 0 else not even(n - 1)

print(even(10)) # → True
try:
    even(-1)     # never reaches 0
except RecursionError as e:
    print(e)
# → maximum recursion depth exceeded
```

Lean structural recursion; checked when defined

```
def even (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (even k)
#eval even 10 -- → true
```

```
def evenLoops (n : Nat) : Bool :=
  match n with
  | Nat.zero => true
  | Nat.succ k => not (evenLoops n)
```

error: fail to show termination for
evenLoops
with errors
failed to infer structural recursion:
Not considering parameter n of evenLoops:
it is unchanged in the recursive calls
no parameters suitable for structural recursion

well-founded recursion cannot be used, `evenLoops` does not
↪ take any (non-fixed) arguments

Polymorphism: types as arguments

Python TypeVar and Generic

```
@dataclass(frozen=True)
class PPoint(Generic[T]):
    x: T
    y: T

def replace_x(point: PPoint[T],
              new_x: T) -> PPoint[T]:
    return replace(point, x=new_x)

nat_origin: PPoint[int] = PPoint(0, 0)
print(replace_x(nat_origin, 5))
# => PPoint(x=5, y=0)
```

Lean ($\alpha : \text{Type}$) is an ordinary argument

```
structure PPoint ( $\alpha : \text{Type}$ ) where
  x :  $\alpha$ 
  y :  $\alpha$ 
def natOrigin : PPoint Nat :=
  { x := Nat.zero, y := Nat.zero }
```

```
def replaceX ( $\alpha : \text{Type}$ ) (point : PPoint  $\alpha$ )
  (newX :  $\alpha$ ) : PPoint  $\alpha$  :=
  { point with x := newX }
#check (replaceX)
-- => replaceX : ( $\alpha : \text{Type}$ ) -> PPoint  $\alpha$  ->  $\alpha$  -> PPoint  $\alpha$ 
#eval replaceX Nat natOrigin 5
-- => { x := 5, y := 0 }
```

Lean implicit: $\{\alpha : \text{Type}\}$ is filled in by Lean, like Python's T

```
def replaceX { $\alpha : \text{Type}$ } (point : PPoint  $\alpha$ )
  (newX :  $\alpha$ ) : PPoint  $\alpha$  :=
  { point with x := newX }
#eval replaceX natOrigin 5
-- => { x := 5, y := 0 }
```

Lists: an inductive type, used without loops

Python list, len, a comprehension

```
primes_under_10: list[int] = [2, 3, 5, 7]
def length(xs: list[T]) -> int:
    match xs:
        case []:
            return 0
        case [_ , *ys]:
            return 1 + length(ys)

print(length(primes_under_10)) # → 4
print(len(primes_under_10))   # → 4
print([p * 2 for p in primes_under_10])
# → [4, 6, 10, 14]
```

Lean [] is nil, x :: xs is cons

```
def primesUnder10 : List Nat := [2, 3, 5, 7]
-- (core) inductive List (α : Type) where
--   | nil : List α
--   | cons : α → List α → List α
def length {α : Type} (xs : List α) : Nat :=
  match xs with
  | [] => 0
  | _ :: ys => Nat.succ (length ys)
#eval length primesUnder10 -- → 4
#eval primesUnder10.length -- → 4
#eval primesUnder10.map (fun p => p * 2)
-- → [4, 6, 10, 14]
```

Python instinct vs Lean

In Python you would write `len` and a comprehension, not recursion (slicing copies, and the recursion limit bites). In Lean, the list functions of the library (`length`, `map`, `filter`, `foldl`) replace the `for` loop; each is a short recursion like `length`.

Option instead of None

Python T | None: nothing forces the check

```
def head(xs: list[T]) -> T | None:
    return xs[0] if xs else None
print(head(primes_under_10)) # => 2
print(head([]))              # => None
try:
    head([]) + 1
except TypeError as e:
    print(e)
# => unsupported operand type(s) for +:
# 'NoneType' and 'int'
print(Optional[Optional[int]] == Optional[int])
# => True
```

Lean Option α is none or some x

```
#eval primesUnder10.head?      -- => some 2
#eval ([] : List Int).head?    -- => none
#eval ([] : List Nat).headD 0  -- => 0
```

```
#eval primesUnder10.head? + 1
-- => error
```

```
error: failed to synthesize instance of type class
HAdd (Option Nat) Nat ?m.1
```

```
def headPlusOne (xs : List Nat) : Nat :=
  match xs.head? with
  | none => 0
  | some x => x + 1
#eval headPlusOne primesUnder10 -- => 3
```

The book's naming: head? returns an Option, head! crashes (panics) on [], headD takes a default.

Characters, strings, and slices

Python str; no character type

```
print("Hello, " + "world")
# → Hello, world
print("Hello" + "!")      # → Hello!
print(type("L"))          # → <class 'str'>
print(len("λx"))          # → 2
print(len("λx".encode())) # → 3
```

```
print("small tortoiseshell"[6:-5])
# → tortoise
print("red admiral".rstrip("l"))
# → red admira
s = "red admiral".rstrip(ascii_letters)
print(repr(s)) # → 'red '
```

Lean String, Char, String.Slice

```
#eval "Hello, " ++ "world"
-- → "Hello, world"
#eval "Hello".push '!' -- → "Hello!"
#check 'L'             -- → 'L' : Char
#eval "λx".length      -- → 2
#eval "λx".utf8ByteSize -- → 3
```

```
#eval (("small tortoiseshell".drop 6).dropEnd 5).copy
-- → "tortoise"
#eval "red admiral".dropEndWhile 'l'
-- → red admira
#eval ("red admiral".dropEndWhile Char.isAlpha).copy
-- → "red "
```

Same and different

Both count Unicode code points ("λx" has 2, stored in 3 UTF-8 bytes) and both strings are immutable. 'L' is a Char in Lean, a one-letter str in Python. drop, dropEnd, dropWhile return slices that share the original string; .copy makes a String. A slice prints without quotes (red admira).

Pattern-matching definitions and local let

Lean FPIL 1.8.1–1.8.3: no argument names, no match keyword

```
def length : List  $\alpha$  → Nat --  $\alpha$ : automatic implicit
| [] => 0
| _ :: ys => Nat.succ (length ys)
def unzip : List ( $\alpha \times \beta$ ) → List  $\alpha \times$  List  $\beta$ 
| [] => ([], [])
| (x, y) :: xys =>
  let (xs, ys) := unzip xys -- local definition
  (x :: xs, y :: ys)
#eval unzip [(1, "a"), (2, "b")]
--  $\Rightarrow$  ([1, 2], ["a", "b"])
```

Python tuple unpacking

```
pairs = [(1, "a"), (2, "b")]
xs, ys = zip(*pairs) # tuple unpacking
print(xs, ys)       #  $\Rightarrow$  (1, 2) ('a', 'b')
```

What changed

- ▶ Automatic implicits: α is mentioned, never declared; Lean adds $\{\alpha : \text{Type}\}$.
- ▶ The cases follow `:` and the type `List  $\alpha \rightarrow$  Nat`, with no `:= match xs with`.
- ▶ `let (xs, ys) := unzip xys` names the recursive result once, like Python's `xs, ys = ...`; it cannot be reassigned.

fun is lambda; pipelines replace loops

Python lambda, comprehension, loop

```
print((lambda x: x + x)(5)) # → 10
odd_squares = [x ** 2 for x in [1, 2, 3, 4, 5]
               if x % 2 == 1]
print(odd_squares) # → [1, 9, 25]
total = 0
for k in range(11):
    total += k # rebinding in a loop
print(total, sum(range(11))) # → 55 55
print(reduce(operator.add, range(11), 0))
# → 55
```

Lean fun, ·, |>

```
#check fun x => x + 1
-- → fun x => x + 1 : Nat → Nat
#check fun (x : Int) => x + 1
-- → fun x => x + 1 : Int → Int
#eval (fun x => x + x) 5 -- → 10
#eval (· * 2) 5 -- → 10
#eval [1, 2, 3, 4, 5]
|>.filter (· % 2 == 1)
|>.map (· ^ 2)
-- → [1, 9, 25]
#eval (List.range 11).foldl (· + ·) 0 -- → 55
#eval 5 |> add1 |> add1 -- → 7
```

Python instinct vs Lean

fun x => x + 1 is lambda x: x + 1; (· * 2) is shorter still (each · is the next argument). Outside do blocks there is no total += k: fold instead. e |>.f means e.f, and x |> g means g x; read the pipeline top to bottom, like a comprehension.

Namespaces, dot notation, s! interpolation

Python modules; built-ins are closed

```
try:
    int.double = lambda self: self + self
except TypeError as e:
    print(e)
# ⇒ cannot set 'double' attribute of
#   immutable type 'int'
# a module is a namespace
print(math.sqrt(16.0)) # ⇒ 4.0
print(f"three fives is {3 * 5}")
# ⇒ three fives is 15
```

Lean Nat.double, namespace, open

```
def Nat.double (x : Nat) : Nat := x + x
#eval (4 : Nat).double -- ⇒ 8
namespace NewNamespace
def triple (x : Nat) : Nat := 3 * x
end NewNamespace
#check NewNamespace.triple
-- ⇒ NewNamespace.triple (x : Nat) : Nat
open NewNamespace in
#eval triple 4 -- ⇒ 12
#eval s!"three fives is {NewNamespace.triple 5}"
-- ⇒ "three fives is 15"
```

Same and different

A namespace is a family of names, like a module: `List.map` and `Array.map` do not clash. Unlike Python, anyone can add a name to `Nat`, and then dot notation finds it: `(4 : Nat).double`. `s!"... {e}"` is an f-string.

Summary: Python instinct vs Lean

Python instinct	Lean
type at the prompt	#eval in the file; the editor is the REPL
f(x, y)	f x y; f x alone is a function
hints are optional	every expression has a type, checked first
1 - 2 == -1, 7 / 2 == 3.5	on Nat: 1 - 2 = 0, 7 / 2 = 3, 7 / 0 = 0
x = ... again	a name keeps its value; let and { p with ... }
for loop	recursion, map, filter, foldl
@dataclass, Enum	structure, inductive
match may fall through	match must cover every case
recursion depth error	termination checked when you define it
None anywhere	Option α ; none must be handled
TypeVar	{ α : Type}, filled in by Lean
type(x)	#check x, without running it

Note: do blocks offer for and let mut as sugar over the same pure functions; FPIL covers them later (Monads, Monad Transformers).

From programs to proofs

Lean a program and a theorem about it

```
#eval add1 7 -- → 8
theorem add1_pos (n : Nat) : 0 < add1 n :=
  Nat.succ_pos n
#check (add1_pos)
-- → add1_pos : ∀ (n : Nat), 0 < add1 n
#check add1_pos 7
-- → add1_pos 7 : 0 < add1 7
theorem add1_seven : add1 7 = 8 := rfl
```

a type

a value of the type

def

a function of (n : Nat)

↔ a statement

↔ a proof

↔ theorem

↔ “for all n”

Looser analogy: #eval shows a value, the goal window shows what is left to prove.

Back to the π - λ lecture

The key line is #check add1_pos 7, which prints add1_pos 7 : 0 < add1 7: the value is a proof, and its type is the statement it proves. add1_pos is literally a function from n to a proof. theorem ... (hP : IsPiSystem P) ... : $\mu = \nu$:= by ... has the same shape as def f (n : Nat) : Nat := ...: arguments with types, then : and the result type (the statement), then := and the body (the proof).

Try it (answers in the backup slides)

1. Functions (FPIL 1.3)

Define `joinStringsWith : String → String → String → String` that puts its first argument between the other two, so `joinStringsWith ", " "one" "and another"` is `"one, and another"`. What is the type of `joinStringsWith " : " ?` Ask `#check`.

2. Structures (FPIL 1.4)

Define a structure `RectangularPrism` with `Float` fields `height`, `width`, `depth`, and `volume : RectangularPrism → Float`. Python warm-up: the same as a frozen dataclass.

3. Polymorphism (FPIL 1.6)

Write a function that finds the last entry of a list. It should return an `Option`. Which cases must your match cover?

References

- ▶ David Thrane Christiansen, *Functional Programming in Lean*, Chapter 1 “Getting to Know Lean”, sections 1.1–1.9. lean-lang.org/functional_programming_in_lean
- ▶ *Theorem Proving in Lean 4* (the logic side), and *Mathematics in Lean* (Mathlib). lean-lang.org/documentation
- ▶ Python documentation: dataclasses, typing, structural pattern matching (PEP 634, tutorial PEP 636). docs.python.org/3
- ▶ Our π - λ lecture: [talk/Talk-Lean-Math7800.pdf](#) in the course repository.

Every code panel in this deck is cut from `student/Math7800Lean/FPIL/Ch1.lean` (Lean v4.34.0, core only) or `student/python/fpil_ch1.py` (run on CPython 3.12). Every output shown is the output of that run.

Backup: Prod, Sum, Unit (FPIL 1.6.3)

```
def fives : String × Int := ("five", 5)
#eval fives.fst          -- → "five"
def PetName : Type := String ⊕ String
def animals : List PetName :=
  [Sum.inl "Spot", Sum.inr "Tiger", Sum.inl "Fifi",
   Sum.inl "Rex", Sum.inr "Floof"]
def howManyDogs (pets : List PetName) : Nat :=
  match pets with
  | [] => 0
  | Sum.inl _ :: morePets => howManyDogs morePets + 1
  | Sum.inr _ :: morePets => howManyDogs morePets
#eval howManyDogs animals -- → 3
```

Python analogues

`String × Int` is a pair, like `tuple[str, int]`. `String ⊕ String` is a tagged choice (`Sum.inl`, `Sum.inr`); Python's `str | str` would collapse to `str`, so you would need two small dataclasses. `Unit`, written `()`, plays the role of a `None` return value.

Backup: simultaneous matching and number patterns (FPIL 1.8.5–1.8.6)

```
def sameLength (xs : List  $\alpha$ ) (ys : List  $\beta$ ) : Bool :=
  match xs, ys with
  | [], [] => true
  | _ :: xs', _ :: ys' => sameLength xs' ys'
  | _, _ => false
#eval sameLength [1, 2] ["a", "b"] --  $\rightarrow$  true
def halve : Nat  $\rightarrow$  Nat
  | 0 => 0
  | 1 => 0
  | n + 2 => halve n + 1
#eval halve 7 --  $\rightarrow$  3
```

Why not match (xs, ys) with?

Matching on the pair hides from the termination checker that xs' is smaller than xs ; Lean then rejects the definition. `match xs, ys with` keeps the connection. In a pattern, $n + 2$ means `Nat.succ (Nat.succ n)`; write the literal on the right ($2 + n$ is rejected).

Backup: if let and {...} (FPIL 1.8.9–1.8.10)

Lean if let, and {...} for a structure

```
inductive Inline : Type where
| lineBreak
| string : String → Inline
| emph : Inline → Inline
| strong : Inline → Inline
def Inline.string? (inline : Inline) : Option String :=
  if let Inline.string s := inline then some s else none
#eval (Inline.string "hi").string?
-- → some "hi"
#eval Inline.lineBreak.string? -- → none
#eval ({1, 2} : Point)
-- → { x := 1.000000, y := 2.000000 }
```

Python a dataclass per constructor

```
class LineBreak: pass
@dataclass(frozen=True)
class InlineString:
    s: str
def string_q(i: object) -> str | None:
    match i:
        case InlineString(s):
            return s
        case _:
            return None
print(string_q(InlineString("hi"))) # → hi
print(string_q(LineBreak())) # → None
```

Python analogues

if let Inline.string s := inline then ... else ... is a one-case match with a fallback, like match i: case InlineString(s): ... case _: ... on the right (Inline.string becomes the dataclass InlineString). {1, 2} builds a structure positionally, like Point(1, 2), but only when the expected type is known (type \< and \>).

Backup: two more messages you may meet (FPIL 1.6.4)

Lean no display code for a list of a new type

```
#eval allTools -- → error
```

```
error: could not synthesize a `Repr` or `ToString` instance for
↳ type
  List WoodSplittingTool
```

Lean fix: deriving Repr

```
inductive Firewood where
  | birch | pine | beech
deriving Repr

#eval [Firewood.birch, Firewood.pine, Firewood.beech]
-- → [Firewood.birch, Firewood.pine, Firewood.beech]
```

Lean the empty list has no element type

```
#eval [].head? -- → error
```

```
error: don't know how to synthesize implicit
↳ argument `α`
  @List.head? ?m.3 []
context:
  ⊢ Type ?u.3
```

Fix: `([] : List Int).head?`. The `?m.3` is a *metavariable*: a part of the program Lean could not infer. A second error, for `@List.nil ?m.3`, follows.

Backup: answers to “Try it”

1. joinStrings

```
def joinStrings (sep a b : String) : String :=  
  a ++ sep ++ b  
#eval joinStrings " ", "one" "and another"  
-- => "one, and another"  
#check (joinStrings ": ")  
-- => joinStrings ": " : String → String → String
```

3. the last entry, as an Option

```
def List.last? {α : Type} : List α → Option α  
| [] => none  
| [x] => some x  
| _ :: xs => xs.last?  
#eval [2, 3, 5, 7].last? -- => some 7
```

2. RectangularPrism

```
structure RectangularPrism where  
  height : Float  
  width : Float  
  depth : Float  
def RectangularPrism.volume (r : RectangularPrism) :  
  Float :=  
    r.height * r.width * r.depth  
#eval ({ height := 2, width := 3, depth := 4 }  
       : RectangularPrism).volume  
-- => 24.000000
```

Backup: running the companion files

Lean (core only, about a second)

```
cd student
```

```
lake env lean Math7800Lean/FPIL/Ch1.lean
```

prints one line per `#eval`/`#check`; no errors, no warnings. The deliberate errors sit under `#guard_msgs`, which fails the file if Lean's message ever changes. In VS Code, open the file and read the same results in the Infoview.

Python 3.10 or newer

```
python3 student/python/fpil_ch1.py
```

Check every quoted output

```
python3 talk-fpil/tools/check_outputs.py
```

runs both files and compares each output comment with the real output.