PastaLean

Transpiling and *Verifying* Python in Lean 4 — without LLMs

Anirudh Gupta · IISc Bengaluru
Petros Markopoulos · UC San Diego
Swarnava Chakraborty · IISc Bengaluru
Siddhartha Gadgil · Professor · IISc Bengaluru

Summer School: Lean for Programming · IISc Bengaluru · 9 July 2026
Organized by **IISc Bengaluru** × **Emergence AI**

GitHub PyAstLean

PastaLean

Overview

PastaLean

Overview

- **Why** verify Python at all — and why in Lean

PastaLean

Overview

- **Why** verify Python at all — and why in Lean
- **How** PastaLean turns Python into Lean, node by node

PastaLean

Overview

- **Why** verify Python at all — and why in Lean
- **How** PastaLean turns Python into Lean, node by node
- **What** we support: control flow, classes, libraries…

Overview

- **Why** verify Python at all — and why in Lean
- **How** PastaLean turns Python into Lean, node by node
- **What** we support: control flow, classes, libraries…
- **Proving** the generated code correct

PastaLean

What are we cooking?

PastaLean

What are we cooking?

Pasta

PastaLean

What are we cooking?

Pasta

+

PastaLean

What are we cooking?

Pasta

+

Python

PastaLean

What are we cooking?

Pasta

+

Python

+

PastaLean

What are we cooking?

Pasta

+

Python

+

Lean 4

PastaLean

What are we cooking?

Pasta

+

Python

+

Lean 4

=

PastaLean

What are we cooking?

Pasta

+

Python

+

Lean 4

=

What are we cooking?

Pasta + Python + Lean 4 =

P·*ast*·aLean → PastaLean

PastaLean

What are we cooking?

Pasta

+ python Python + Lean 4 = PAPTa∀

P·*ast*·aLean → PastaLean

How to eat(use) PastaLean?

PastaLean

What are we cooking?


Pasta

+ python Python + Lean 4 = PAsTa∀

P·*ast*·aLean → PastaLean

How to eat(use) PastaLean?

- Take ordinary, *logical* Python — the kind data scientists and practical logical code's LLMs actually write.

PastaLean

What are we cooking?

Pasta

+ python Python + Lean 4 = PAsTa∀

P·*ast*·aLean → PastaLean

How to eat(use) PastaLean?

- Take ordinary, *logical* Python — the kind data scientists and practical logical code's LLMs actually write.
- Preprocess Python code for Type annotations, obtain it's Abstract Syntax Tree(AST) and give Lean it's JSON format.

PastaLean

What are we cooking?

Pasta

+ python Python + Lean 4 = PAsTa∀

P·*ast*·aLean → PastaLean

How to eat(use) PastaLean?

- Take ordinary, *logical* Python — the kind data scientists and practical logical code's LLMs actually write.
- Preprocess Python code for Type annotations, obtain it's Abstract Syntax Tree(AST) and give Lean it's JSON format.
- *Transpile* it into Lean 4 — deterministically, with no LLM, so the meaning never drifts.

PastaLean

What are we cooking?


Pasta

+ 

Python

+ 
Lean 4

= PA  Ta∀

P·*ast*·aLean → PastaLean

How to eat(use) PastaLean?

- Take ordinary, *logical* Python — the kind data scientists and practical logical code's LLMs actually write.
- Preprocess Python code for Type annotations, obtain it's Abstract Syntax Tree(AST) and give Lean it's JSON format.
- *Transpile* it into Lean 4 — deterministically, with no LLM, so the meaning never drifts.
- *Prove* it correct — and let Lean's proof automation do the heavy lifting with a little help by LLM's.

PastaLean

What are we cooking?

Pasta + Python + Lean 4 = PastaLean

P·*ast*·aLean → PastaLean

How to eat(use) PastaLean?

- Take ordinary, *logical* Python — the kind data scientists and practical logical code's LLMs actually write.
- Preprocess Python code for Type annotations, obtain it's Abstract Syntax Tree(AST) and give Lean it's JSON format.
- *Transpile* it into Lean 4 — deterministically, with no LLM, so the meaning never drifts.
- *Prove* it correct — and let Lean's proof automation do the heavy lifting with a little help by LLM's.

The Goal: write in Python, get the guarantees of a proof assistant.

What is the Inspiration for this dish?

PastaLean

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.
- But here's the uncomfortable question:

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.
- But here's the uncomfortable question:

Who checks that the generated code is actually *correct*?

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.
- But here's the uncomfortable question:

Who checks that the generated code is actually *correct*?

- *Tests only catch the bugs you thought to write — never the ones you didn't.*

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.
- But here's the uncomfortable question:

Who checks that the generated code is actually *correct*?

- *Tests only catch the bugs you thought to write — never the ones you didn't.*
- To *prove* correctness (not merely test it), we reach for **formal methods**: Hoare logic, model checking, multi-modal verification…

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.
- But here's the uncomfortable question:

Who checks that the generated code is actually *correct*?

- *Tests only catch the bugs you thought to write — never the ones you didn't.*
- To *prove* correctness (not merely test it), we reach for **formal methods**: Hoare logic, model checking, multi-modal verification…
- **Proof assistants** make that practical — and there are several: **Lean 4**, Rocq (Coq), Isabelle, Dafny, …

What is the Inspiration for this dish?

- Two things dominate how code gets written today: **Python** (the language) and **LLMs** (increasingly, the author).
- A huge and growing share of the Python running in the world was drafted by an LLM.
- But here's the uncomfortable question:

Who checks that the generated code is actually *correct*?

- *Tests only catch the bugs you thought to write — never the ones you didn't.*
- To *prove* correctness (not merely test it), we reach for **formal methods**: Hoare logic, model checking, multi-modal verification…
- **Proof assistants** make that practical — and there are several: **Lean 4**, Rocq (Coq), Isabelle, Dafny, …
- We use **Lean 4**: a proof assistant that is *also* a fast, real programming language.

Why Lean?

Lean 4 is a proof assistant *and* a general-purpose language (it compiles to C — genuinely fast).

1.9M+
lines in Mathlib, its math library

282k+
theorems · 134k definitions

772
contributors · 1 tiny trusted kernel

- **Powerful automation** — tactics like `simp`, `grind`, `aesop`, etc. search proofs and close goals for you, often in one line.
- **Endlessly customizable** — custom syntax, macros, and tactics let you extend the language itself — exactly how PastaLean is built.
- **Trusted where it matters** — used by Fields medallist **Terence Tao**, backed by the **Lean FRO**, applied to industrial verification.
- A small, auditable **kernel** re-checks every proof — nothing is taken on faith.

See Leo de Moura's blog **Why Lean?**.

Why convert Python into Lean?

PastaLean

Why convert Python into Lean?

- Blindly trusting an LLM's Python means shipping logic *no one has proved* — silent wrong answers, bad edge cases, security holes.

Why convert Python into Lean?

- Blindly trusting an LLM's Python means shipping logic *no one has proved* — silent wrong answers, bad edge cases, security holes.

***PastaLean* gives that code a home where the compiler is the reviewer:**

Why convert Python into Lean?

- Blindly trusting an LLM's Python means shipping logic *no one has proved* — silent wrong answers, bad edge cases, security holes.

***PastaLean* gives that code a home where the compiler is the reviewer:**

- if it type-checks against our runtime, the **translation** is faithful;

Why convert Python into Lean?

- Blindly trusting an LLM's Python means shipping logic *no one has proved* — silent wrong answers, bad edge cases, security holes.

***PastaLean* gives that code a home where the compiler is the reviewer:**

- if it type-checks against our runtime, the **translation** is faithful;
- if the proof closes, the **logic** is correct.

Why convert Python into Lean?

- Blindly trusting an LLM's Python means shipping logic *no one has proved* — silent wrong answers, bad edge cases, security holes.

***PastaLean* gives that code a home where the compiler is the reviewer:**

- if it type-checks against our runtime, the **translation** is faithful;
- if the proof closes, the **logic** is correct.

Write it in Python. Trust it like Lean.

Why convert Python into Lean?

- Blindly trusting an LLM's Python means shipping logic *no one has proved* — silent wrong answers, bad edge cases, security holes.

***PastaLean* gives that code a home where the compiler is the reviewer:**

- if it type-checks against our runtime, the **translation** is faithful;
- if the proof closes, the **logic** is correct.

Write it in Python. Trust it like Lean.

And because pure Python becomes bare Lean *terms*, we can state and prove theorems about it.

Why transpile — and not just ask an LLM?

Why transpile — and not just ask an LLM?

The obvious shortcut: *"LLM, translate this Python to Lean."* We deliberately do *not* do that for the core translation. LLMs earn their keep elsewhere — proposing *contracts* — never as the translator.

7.1

PastaLean

Why transpile — and not just ask an LLM?

The obvious shortcut: *"LLM, translate this Python to Lean."* We deliberately do *not* do that for the core translation. LLMs earn their keep elsewhere — proposing *contracts* — never as the translator.

Prompt an LLM to translate	**PastaLean (AST transpiler)**
Hallucinates; silently changes semantics	Deterministic — same node, same Lean, always
Not reproducible, not auditable	One runtime pins *what each operation means*
Breaks on long / scaled programs	Compositional: per-statement, streamed
No ground truth that it is correct	Output is *checked* by the Lean elaborator

PastaLean

Why transpile — and not just ask an LLM?

The obvious shortcut: *"LLM, translate this Python to Lean."* We deliberately do *not* do that for the core translation. LLMs earn their keep elsewhere — proposing *contracts* — never as the translator.

Prompt an LLM to translate	**PastaLean (AST transpiler)**
Hallucinates; silently changes semantics	Deterministic — same node, same Lean, always
Not reproducible, not auditable	One runtime pins *what each operation means*
Breaks on long / scaled programs	Compositional: per-statement, streamed
No ground truth that it is correct	Output is *checked* by the Lean elaborator

Other Possible methods maybe?

7.3

PastaLean

Why transpile — and not just ask an LLM?

The obvious shortcut: *"LLM, translate this Python to Lean."* We deliberately do *not* do that for the core translation. LLMs earn their keep elsewhere — proposing *contracts* — never as the translator.

Prompt an LLM to translate	**PastaLean (AST transpiler)**
Hallucinates; silently changes semantics	Deterministic — same node, same Lean, always
Not reproducible, not auditable	One runtime pins *what each operation means*
Breaks on long / scaled programs	Compositional: per-statement, streamed
No ground truth that it is correct	Output is *checked* by the Lean elaborator

Other Possible methods maybe?

- Other Methods like using FFI is possible, however theorem proving would not be possible.
- Converting Lean to Python is possible, however legacy python checking would not be possible and there are a lot more python programmers.

7.4

PastaLean

How does PastaLean taste code?

PastaLean

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

PastaLean

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```
def sum_to_n(n: int) → int:
    Requires(n ≥ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```
def sum_to_n(n: int) → int:
    Requires(n ⩾ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```python
def sum_to_n(n: int) → int:
    Requires(n ⩾ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

```
FunctionDef(name='sum_to_n',
  args=[arg('n', Name('int'))],
  body=[
    Expr(Call(Requires,
      Compare(Name'n', GtE, 0))),
    AnnAssign('total', Name'int', 0),
    For('i', Call(range, …), body=[
      Expr(Call(Invariant, …)),
      Assign('total', BinOp(…, Add, …))]),
    Expr(Call(Ensures, …)),
    Return(Name'total')])
```

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```python
def sum_to_n(n: int) → int:
    Requires(n ⩾ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

```
FunctionDef(name='sum_to_n',
  args=[arg('n', Name('int'))],
  body=[
    Expr(Call(Requires,
      Compare(Name'n', GtE, 0))),
    AnnAssign('total', Name'int', 0),
    For('i', Call(range, …), body=[
      Expr(Call(Invariant, …)),
      Assign('total', BinOp(…, Add, …))]),
    Expr(Call(Ensures, …)),
    Return(Name'total')])
```

→ *transpile*

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```python
def sum_to_n(n: int) → int:
    Requires(n ⩾ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

```
FunctionDef(name='sum_to_n',
  args=[arg('n', Name('int'))],
  body=[
    Expr(Call(Requires,
      Compare(Name'n', GtE, 0))),
    AnnAssign('total', Name'int', 0),
    For('i', Call(range, …), body=[
      Expr(Call(Invariant, …)),
      Assign('total', BinOp(…, Add, …))]),
    Expr(Call(Ensures, …)),
    Return(Name'total')])
```

→ *transpile*

```
def sum_to_n := fun (n : Int) ↦
  (do
    let mut total := (0 : Int)
    for i in (PastaLean.pyRange (n +ₚ (1 : Int)))do
      let _ := Libraries.passta.pyPassInvariant ((2 : Int) *
      total := total +ₚ i
    let _ := Libraries.passta.pyPassEnsures ((2 : Int) *ₚ to
    return total : Id _)
```

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```python
def sum_to_n(n: int) -> int:
    Requires(n ≥ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

```
FunctionDef(name='sum_to_n',
  args=[arg('n', Name('int'))],
  body=[
    Expr(Call(Requires,
      Compare(Name'n', GtE, 0))),
    AnnAssign('total', Name'int', 0),
    For('i', Call(range, …), body=[
      Expr(Call(Invariant, …)),
      Assign('total', BinOp(…, Add, …))]),
    Expr(Call(Ensures, …)),
    Return(Name'total')])
```

→ *transpile*

```
def sum_to_n := fun (n : Int) ↦
  (do
    let mut total := (0 : Int)
    for i in (PastaLean.pyRange (n +ₚ (1 : Int)))do
      let _ := Libraries.passta.pyPassInvariant ((2 : Int) *
      total := total +ₚ i
    let _ := Libraries.passta.pyPassEnsures ((2 : Int) *ₚ to
    return total : Id _)
```

↓ *prove — automatically*

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```python
def sum_to_n(n: int) -> int:
    Requires(n ≥ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

```
FunctionDef(name='sum_to_n',
  args=[arg('n', Name('int'))],
  body=[
    Expr(Call(Requires,
      Compare(Name'n', GtE, 0))),
    AnnAssign('total', Name'int', 0),
    For('i', Call(range, …), body=[
      Expr(Call(Invariant, …)),
      Assign('total', BinOp(…, Add, …))]),
    Expr(Call(Ensures, …)),
    Return(Name'total')])
```

→ *transpile*

```
def sum_to_n := fun (n : Int) ↦
  (do
    let mut total := (0 : Int)
    for i in (PastaLean.pyRange (n +ₚ (1 : Int)))do
      let _ := Libraries.passta.pyPassInvariant ((2 : Int) *
      total := total +ₚ i
    let _ := Libraries.passta.pyPassEnsures ((2 : Int) *ₚ to
    return total : Id _)
```

↓ *prove — automatically*

```
theorem sum_to_n_spec : ⦃⌜n ≥ (0 : Int)⌝⦄ sum_to_n n ⦃⇓_ ⇒ ⌜
  by
  mvcgen [sum_to_n, PastaLean.pyRange_forIn, PastaLean.pyRa
  · ⇓⟨cur, total⟩ ⇒
    ⌜let i := (cur.prefix.length : Int);
      (2 : Int) *ₚ total = i *ₚ (i -ₚ (1 : Int))⌝
  simp_all [taste_ingr]; grind +locals +suggestions
```

How does PastaLean taste code?

It reads Python's **AST** and emits Lean you can *prove*:

```python
def sum_to_n(n: int) -> int:
    Requires(n ≥ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total == i*(i - 1
        total = total + i
    Ensures(2*total == n*(n + 1))
    return total
```

↓ *ast-parse*

```
FunctionDef(name='sum_to_n',
  args=[arg('n', Name('int'))],
  body=[
    Expr(Call(Requires,
      Compare(Name'n', GtE, 0))),
    AnnAssign('total', Name'int', 0),
    For('i', Call(range, …), body=[
      Expr(Call(Invariant, …)),
      Assign('total', BinOp(…, Add, …))]),
    Expr(Call(Ensures, …)),
    Return(Name'total')])
```

→ *transpile*

```
def sum_to_n := fun (n : Int) ↦
  (do
    let mut total := (0 : Int)
    for i in (PastaLean.pyRange (n +ₚ (1 : Int)))do
      let _ := Libraries.passta.pyPassInvariant ((2 : Int) *
      total := total +ₚ i
    let _ := Libraries.passta.pyPassEnsures ((2 : Int) *ₚ to
    return total : Id _)
```

↓ *prove — automatically*

```
theorem sum_to_n_spec : ⦃⌜n ≥ (0 : Int)⌝⦄ sum_to_n n ⦃⇓_ ⇒ ⌜
  by
  mvcgen [sum_to_n, PastaLean.pyRange_forIn, PastaLean.pyRa
  · ⇓⟨cur, total⟩ ⇒
    ⌜let i := (cur.prefix.length : Int);
      (2 : Int) *ₚ total = i *ₚ (i -ₚ (1 : Int))⌝
  simp_all [taste_ingr]; grind +locals +suggestions
```

```
#eval sum_to_n 10
55
```

PastaLean

Two Functions for each code

PastaLean

Two Functions for each code

PastaLean converts each function in Python into 2 functions each with it's own purpose:

PastaLean

Two Functions for each code

PastaLean converts each function in Python into 2 functions each with it's own purpose:

- **Provable** - This function is meant to be used for property proving of the function.
- **Computable** - This function is meant to be computable with `#eval`. These are marked with `'rn` in function name, which stands for `run`.

PastaLean

Two Functions for each code

PastaLean converts each function in Python into 2 functions each with it's own purpose:

- **Provable** - This function is meant to be used for property proving of the function.
- **Computable** - This function is meant to be computable with `#eval`. These are marked with `'rn` in function name, which stands for `run`.

Difference	Provable	Computable
Python `float`	Converted to $\mathbb{Q}$ or $\mathbb{R}$(if transcendental functions are used).	Lean's inbuilt `Float` is always used.
Provable	Yes	Sometimes depending on function, not always.
Computable	Sometimes. If $\mathbb{R}$ is used then never.	Yes
API differences(`print`, `exceptional-handling`)	Special non-IO Monad based functions are used to support proving	IO Monad functions are used for real behaviour

Examples

One Python function forks into two Lean definitions:

Examples

One Python function forks into two Lean definitions:

```python
def euclidean_distance(p1, p2):
    if len(p1) ≠ len(p2):
        raise ValueError("Points must have the same number of dimensions")

    # Using zip, list comprehension, and math.pow
    sq_diffs = [math.pow(a - b, 2) for a, b in zip(p1, p2)]
    return math.sqrt(sum(sq_diffs))
```

Examples

One Python function forks into two Lean definitions:

```python
def euclidean_distance(p1, p2):
    if len(p1) ≠ len(p2):
        raise ValueError("Points must have the same number of dimensions")

    # Using zip, list comprehension, and math.pow
    sq_diffs = [math.pow(a - b, 2) for a, b in zip(p1, p2)]
    return math.sqrt(sum(sq_diffs))
```

↙ *provable*

Examples

One Python function forks into two Lean definitions:

```python
def euclidean_distance(p1, p2):
    if len(p1) ≠ len(p2):
        raise ValueError("Points must have the same number of dimensions")

    # Using zip, list comprehension, and math.pow
    sq_diffs = [math.pow(a - b, 2) for a, b in zip(p1, p2)]
    return math.sqrt(sum(sq_diffs))
```

↙ *provable*

```
noncomputable def euclidean_distance := fun (p1 : List Int) ↦ fun
  ((do
      if h_1 : PastaLean.pyLen p1 ≠ PastaLean.pyLen p2 then
        throw
            (PastaLean.PyException.Raise "ValueError"
              (ToString.toString "Points must have the same numbe
      else
        let _ := ()
      -- Using zip, list comprehension, and math.pow
      let mut sq_diffs :=
        (PastaLean.pyIter (PastaLean.pyZip p1 p2)).map fun _pair_
          let a := Prod.fst _pair_1;
          let b := Prod.snd _pair_1;
          Libraries.math.pyMathPowExact (a -ₚ b) (2 : Int)
      let __py_ret_1 := Libraries.math.pyMathSqrtR (PastaLean.pyS
      return __py_ret_1) :
    PastaLean.PyExceptId _)
```

Examples

One Python function forks into two Lean definitions:

```python
def euclidean_distance(p1, p2):
    if len(p1) ≠ len(p2):
        raise ValueError("Points must have the same number of dimensions")

    # Using zip, list comprehension, and math.pow
    sq_diffs = [math.pow(a - b, 2) for a, b in zip(p1, p2)]
    return math.sqrt(sum(sq_diffs))
```

↙ *provable* ↘ *computable*

```
noncomputable def euclidean_distance := fun (p1 : List Int) ↦ fun
  ((do
      if h_1 : PastaLean.pyLen p1 ≠ PastaLean.pyLen p2 then
        throw
            (PastaLean.PyException.Raise "ValueError"
              (ToString.toString "Points must have the same numbe
      else
        let _ := ()
      -- Using zip, list comprehension, and math.pow
      let mut sq_diffs :=
        (PastaLean.pyIter (PastaLean.pyZip p1 p2)).map fun _pair_
          let a := Prod.fst _pair_1;
          let b := Prod.snd _pair_1;
          Libraries.math.pyMathPowExact (a -ₚ b) (2 : Int)
      let __py_ret_1 := Libraries.math.pyMathSqrtR (PastaLean.pyS
      return __py_ret_1) :
    PastaLean.PyExceptId _)
```

Examples

One Python function forks into two Lean definitions:

```python
def euclidean_distance(p1, p2):
    if len(p1) ≠ len(p2):
        raise ValueError("Points must have the same number of dimensions")

    # Using zip, list comprehension, and math.pow
    sq_diffs = [math.pow(a - b, 2) for a, b in zip(p1, p2)]
    return math.sqrt(sum(sq_diffs))
```

↙ *provable*

```
noncomputable def euclidean_distance := fun (p1 : List Int) ↦ fun
  ((do
      if h_1 : PastaLean.pyLen p1 ≠ PastaLean.pyLen p2 then
        throw
            (PastaLean.PyException.Raise "ValueError"
              (ToString.toString "Points must have the same numbe
      else
        let _ := ()
      -- Using zip, list comprehension, and math.pow
      let mut sq_diffs :=
        (PastaLean.pyIter (PastaLean.pyZip p1 p2)).map fun _pair_
          let a := Prod.fst _pair_1;
          let b := Prod.snd _pair_1;
          Libraries.math.pyMathPowExact (a -ₚ b) (2 : Int)
      let __py_ret_1 := Libraries.math.pyMathSqrtR (PastaLean.pyS
      return __py_ret_1) :
    PastaLean.PyExceptId _)
```

↘ *computable*

```
def euclidean_distance'rn : List Int → List Int → PastaLean.PyExc
  fun (p2 : List Int) ↦ do
  if h_1 : PastaLean.pyLen p1 ≠ PastaLean.pyLen p2 then
    throw
        (PastaLean.PyException.Raise "ValueError" (ToString.toStr
  else
    let _ := ()
  -- Using zip, list comprehension, and math.pow
  let mut sq_diffs :=
    (PastaLean.pyIter (PastaLean.pyZip p1 p2)).map fun _pair_1 ⇒
      let a := Prod.fst _pair_1;
      let b := Prod.snd _pair_1;
      Libraries.math.pyMathPow (a -ₚ b) (2 : Int)
  let __py_ret_1 := Libraries.math.pyMathSqrt (PastaLean.pySum sq
  return __py_ret_1
```

PastaLean

How much Python is convertable?

PastaLean

How much Python is convertable?

Most everyday Python is convertable into Lean4. Extending new Python syntax is made very trivial with the help of our underlying design architecture.

PastaLean

How much Python is convertable?

Most everyday Python is convertable into Lean4. Extending new Python syntax is made very trivial with the help of our underlying design architecture.

Area	Python constructs handled
Control flow	`if / elif / else`, `for`, `while`, `break`, `continue`, `match`
Functions	`def`, `lambda`, `return`, default args, nested defs
Data structures	`list`, `dict`, `set`, `tuple`, slicing, `x[i]`, `x[i] = v`
Comprehensions	list / dict / set / generator comprehensions
Exceptions	`try` / `except` / `finally`, `raise`, `assert`
OOP	`class`, methods, `self`, fields, dunder methods, inheritance

PastaLean

How much Python is convertable?

Most everyday Python is convertable into Lean4. Extending new Python syntax is made very trivial with the help of our underlying design architecture.

Area	Python constructs handled
Control flow	`if / elif / else`, `for`, `while`, `break`, `continue`, `match`
Functions	`def`, `lambda`, `return`, default args, nested defs
Data structures	`list`, `dict`, `set`, `tuple`, slicing, `x[i]`, `x[i] = v`
Comprehensions	list / dict / set / generator comprehensions
Exceptions	`try` / `except` / `finally`, `raise`, `assert`
OOP	`class`, methods, `self`, fields, dunder methods, inheritance

Control flow

Branches & recursion

Loops & mutation → an `Id.run do` block:

Control flow

Branches & recursion

```
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Loops & mutation → an `Id.run do` block:

Control flow

Branches & recursion

```python
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

→

Loops & mutation → an `Id.run do` block:

Control flow

Branches & recursion

```python
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```
partial def fibonacci : Int → Int := fun (n : Int) ↦
  if decide (n ≤ (0 : Int)) then (0 : Int)
  else if n == (1 : Int) then (1 : Int)
  else fibonacci (n -ₚ (1 : Int)) +ₚ fibonacci (n -ₚ (2 : Int))
```

Loops & mutation → an `Id.run do` block:

Control flow

Branches & recursion

```python
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

→

```
partial def fibonacci : Int → Int := fun (n : Int) ↦
  if decide (n ≤ (0 : Int)) then (0 : Int)
  else if n == (1 : Int) then (1 : Int)
  else fibonacci (n -ₚ (1 : Int)) +ₚ fibonacci (n -ₚ (2 : Int))
```

Loops & mutation → an `Id.run do` block:

```python
def classify(nums: list[int]):
    total = 0
    for n in nums:
        if n % 2 == 0:
            total += n
        else:
            total -= n
    return total
```

Control flow

Branches & recursion

```python
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

→

```
partial def fibonacci : Int → Int := fun (n : Int) ↦
  if decide (n ≤ (0 : Int)) then (0 : Int)
  else if n == (1 : Int) then (1 : Int)
  else fibonacci (n -ₚ (1 : Int)) +ₚ fibonacci (n -ₚ (2 : Int))
```

Loops & mutation → an `Id.run do` block:

```python
def classify(nums: list[int]):
    total = 0
    for n in nums:
        if n % 2 == 0:
            total += n
        else:
            total -= n
    return total
```

→

Control flow

Branches & recursion

```python
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

→

```
partial def fibonacci : Int → Int := fun (n : Int) ↦
  if decide (n ≤ (0 : Int)) then (0 : Int)
  else if n == (1 : Int) then (1 : Int)
  else fibonacci (n -ₚ (1 : Int)) +ₚ fibonacci (n -ₚ (2 : Int))
```

Loops & mutation → an `Id.run do` block:

```python
def classify(nums: list[int]):
    total = 0
    for n in nums:
        if n % 2 == 0:
            total += n
        else:
            total -= n
    return total
```

→

```
def classify := fun (nums : List Int) ↦
  Id.run
    (do
      let mut total := (0 : Int)
      for n in (PastaLean.pyIter nums)do
        if h_1 : n %ₚ (2 : Int) = (0 : Int) then
          total := total +ₚ n
        else
          total := total -ₚ n
      return total)
```

Control flow

Branches & recursion

```python
def fibonacci(n: int):
    if n ⩽ 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

→

```
partial def fibonacci : Int → Int := fun (n : Int) ↦
  if decide (n ≤ (0 : Int)) then (0 : Int)
  else if n == (1 : Int) then (1 : Int)
  else fibonacci (n -ₚ (1 : Int)) +ₚ fibonacci (n -ₚ (2 : Int))
```

Loops & mutation → an `Id.run do` block:

```python
def classify(nums: list[int]):
    total = 0
    for n in nums:
        if n % 2 == 0:
            total += n
        else:
            total -= n
    return total
```

→

```
def classify := fun (nums : List Int) ↦
  Id.run
    (do
      let mut total := (0 : Int)
      for n in (PastaLean.pyIter nums)do
        if h_1 : n %ₚ (2 : Int) = (0 : Int) then
          total := total +ₚ n
        else
          total := total -ₚ n
      return total)
```

Statements outside functions(in open) are also well supported.

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

List & string ops

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
             if x % 2 == 0}
```

List & string ops

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
             if x % 2 == 0}
```

List & string ops

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```python
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
            if x % 2 == 0}
```

→

```
def squares :=
  (List.filter (fun x ⇒ x %ₚ (2 : Int) = (0 : Int))
    (PastaLean.pyRange (10 : Int))).map fun x ⇒ x *ₚ x

def cubes :=
  Std.HashMap.ofList
    ((PastaLean.pyRange (5 : Int)).map fun n ⇒ (n, n ^ₚ (3 : Int)))

def evens :=
  PastaLean.pySetFromList
    ((List.filter (fun x ⇒ x %ₚ (2 : Int) = (0 : Int))
      (PastaLean.pyRange (10 : Int))).map fun x ⇒ x)
```

List & string ops

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```python
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
             if x % 2 == 0}
```

→

```
def squares :=
  (List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
    (PastaLean.pyRange (10 : Int))).map fun x ⇒ x *ₚ x

def cubes :=
  Std.HashMap.ofList
    ((PastaLean.pyRange (5 : Int)).map fun n ⇒ (n, n ^ₚ (3 : Int)))

def evens :=
  PastaLean.pySetFromList
    ((List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
      (PastaLean.pyRange (10 : Int))).map fun x ⇒ x)
```

List & string ops

```python
nums = [3, 1, 2, 4]
head = nums[0]          # index
tail = nums[1:]         # slice
size = len(nums)

s = "hello world"
words = s.split(" ")
loud = s.upper()
```

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```python
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
             if x % 2 == 0}
```

→

```
def squares :=
  (List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
    (PastaLean.pyRange (10 : Int))).map fun x ⇒ x *ₚ x

def cubes :=
  Std.HashMap.ofList
    ((PastaLean.pyRange (5 : Int)).map fun n ⇒ (n, n ^ₚ (3 : Int)))

def evens :=
  PastaLean.pySetFromList
    ((List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
      (PastaLean.pyRange (10 : Int))).map fun x ⇒ x)
```

List & string ops

```python
nums = [3, 1, 2, 4]
head = nums[0]          # index
tail = nums[1:]         # slice
size = len(nums)

s = "hello world"
words = s.split(" ")
loud = s.upper()
```

→

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```python
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
            if x % 2 == 0}
```

→

```
def squares :=
  (List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
    (PastaLean.pyRange (10 : Int))).map fun x ⇒ x *ₚ x

def cubes :=
  Std.HashMap.ofList
    ((PastaLean.pyRange (5 : Int)).map fun n ⇒ (n, n ^ₚ (3 : Int)))

def evens :=
  PastaLean.pySetFromList
    ((List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
      (PastaLean.pyRange (10 : Int))).map fun x ⇒ x)
```

List & string ops

```python
nums = [3, 1, 2, 4]
head = nums[0]          # index
tail = nums[1:]         # slice
size = len(nums)

s = "hello world"
words = s.split(" ")
loud = s.upper()
```

→

```
def nums := [(3 : Int), (1 : Int), (2 : Int), (4 : Int)]
def head := nums[(0 : Int)]
def tail := PastaLean.pySlice nums (some (1 : Int)) none none
def size := PastaLean.pyLen nums

def s := "hello world"
def words := PastaLean.pyStringSplit s " "
def loud := PastaLean.pyStringUpper s
```

Data structures & comprehensions

`list` · `dict` · `set` · `tuple`, slicing, indexing, container methods:

Comprehensions → `map` / `filter`

```python
squares = [x*x for x in range(10)
               if x % 2 == 0]

cubes = {n: n**3 for n in range(5)}

evens = {x for x in range(10)
            if x % 2 == 0}
```

→

```
def squares :=
  (List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
    (PastaLean.pyRange (10 : Int))).map fun x ⇒ x *ₚ x

def cubes :=
  Std.HashMap.ofList
    ((PastaLean.pyRange (5 : Int)).map fun n ⇒ (n, n ^ₚ (3 : Int)))

def evens :=
  PastaLean.pySetFromList
    ((List.filter (fun x ⇒ x %ₚ (2 : Int) == (0 : Int))
      (PastaLean.pyRange (10 : Int))).map fun x ⇒ x)
```

List & string ops

```python
nums = [3, 1, 2, 4]
head = nums[0]          # index
tail = nums[1:]         # slice
size = len(nums)

s = "hello world"
words = s.split(" ")
loud = s.upper()
```

→

```
def nums := [(3 : Int), (1 : Int), (2 : Int), (4 : Int)]
def head := nums[(0 : Int)]
def tail := PastaLean.pySlice nums (some (1 : Int)) none none
def size := PastaLean.pyLen nums

def s := "hello world"
def words := PastaLean.pyStringSplit s " "
def loud := PastaLean.pyStringUpper s
```

`x[i]` → `pyGetItem`, slicing → `pySlice`, `iter` → `pyIter`

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

```
def divide_add(a, b, c):
    try:
        result = a / b
        while result < c:
            result += 1
        return result
    except ZeroDivisionError:
        return "Division by zero error"
    finally:
        print("PastaLean handles exceptions gracefully.")
```

14.1

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

```python
def divide_add(a, b, c):
    try:
        result = a / b
        while result < c:
            result += 1
        return result
    except ZeroDivisionError:
        return "Division by zero error"
    finally:
        print("PastaLean handles exceptions gracefully.")
```

↙ *provable*

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

```python
def divide_add(a, b, c):
    try:
        result = a / b
        while result < c:
            result += 1
        return result
    except ZeroDivisionError:
        return "Division by zero error"
    finally:
        print("PastaLean handles exceptions gracefully.")
```

↙ *provable*

```
def divide_add := fun a ↦ fun b ↦ fun c ↦
  ((do
      try
        let mut result := a /ₚ b
        while (result < c) do
          result := result +ₚ (1 : Int)
        return result
      catch caught ⇒
        if (caught).OfKind == "ZeroDivisionError" then
          let _ ← pyPrintNoop [pyPrintArg "Division by zero error"]
          let __py_ret_1 := -(1 : Int)
          return __py_ret_1
        else
          throw caught
      finally
        do
          let _ ← pyPrintNoop [pyPrintArg "PastaLean handles except
    PastaLean.PyExceptId _)
```

14.3

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

```python
def divide_add(a, b, c):
    try:
        result = a / b
        while result < c:
            result += 1
        return result
    except ZeroDivisionError:
        return "Division by zero error"
    finally:
        print("PastaLean handles exceptions gracefully.")
```

↙ *provable* ↘ *computable*

```
def divide_add := fun a ↦ fun b ↦ fun c ↦
  ((do
      try
        let mut result := a /ₚ b
        while (result < c) do
          result := result +ₚ (1 : Int)
        return result
      catch caught ⇒
        if (caught).OfKind == "ZeroDivisionError" then
          let _ ← pyPrintNoop [pyPrintArg "Division by zero error"]
          let __py_ret_1 := -(1 : Int)
          return __py_ret_1
        else
          throw caught
      finally
        do
          let _ ← pyPrintNoop [pyPrintArg "PastaLean handles except
    PastaLean.PyExceptId _)
```

14.4

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

```python
def divide_add(a, b, c):
    try:
        result = a / b
        while result < c:
            result += 1
        return result
    except ZeroDivisionError:
        return "Division by zero error"
    finally:
        print("PastaLean handles exceptions gracefully.")
```

↙ *provable* ↘ *computable*

```
def divide_add := fun a ↦ fun b ↦ fun c ↦
  ((do
      try
        let mut result := a /ₚ b
        while (result < c) do
          result := result +ₚ (1 : Int)
        return result
      catch caught ⇒
        if (caught).OfKind == "ZeroDivisionError" then
          let _ ← pyPrintNoop [pyPrintArg "Division by zero error"]
          let __py_ret_1 := -(1 : Int)
          return __py_ret_1
        else
          throw caught
      finally
        do
          let _ ← pyPrintNoop [pyPrintArg "PastaLean handles except
    PastaLean.PyExceptId _)
```

```
def divide_add'rn := fun a ↦ fun b ↦ fun c ↦
  ((do
      try
        let mut result := a /ₚ b
        while (result < c) do
          result := result +ₚ (1 : Int)
        return result
      catch caught ⇒
        if (caught).OfKind == "ZeroDivisionError" then
          let _ ← pyPrintIO [pyPrintArg "Division by zero error"]
          let __py_ret_1 := -(1 : Int)
          return __py_ret_1
        else
          throw caught
      finally
        do
          let _ ← pyPrintIO [pyPrintArg "PastaLean handles exceptic
    PastaLean.PyExcept _)
```

14.5

Exception handling

`try` / `except` / `finally`, `raise`, typed `except ValueError as e` — one function, two twins:

```python
def divide_add(a, b, c):
    try:
        result = a / b
        while result < c:
            result += 1
        return result
    except ZeroDivisionError:
        return "Division by zero error"
    finally:
        print("PastaLean handles exceptions gracefully.")
```

↙ *provable* ↘ *computable*

```
def divide_add := fun a ↦ fun b ↦ fun c ↦
  ((do
      try
        let mut result := a /ₚ b
        while (result < c) do
          result := result +ₚ (1 : Int)
        return result
      catch caught ⇒
        if (caught).OfKind == "ZeroDivisionError" then
          let _ ← pyPrintNoop [pyPrintArg "Division by zero error"]
          let __py_ret_1 := -(1 : Int)
          return __py_ret_1
        else
          throw caught
      finally
        do
          let _ ← pyPrintNoop [pyPrintArg "PastaLean handles except
    PastaLean.PyExceptId _)
```

```
def divide_add'rn := fun a ↦ fun b ↦ fun c ↦
  ((do
      try
        let mut result := a /ₚ b
        while (result < c) do
          result := result +ₚ (1 : Int)
        return result
      catch caught ⇒
        if (caught).OfKind == "ZeroDivisionError" then
          let _ ← pyPrintIO [pyPrintArg "Division by zero error"]
          let __py_ret_1 := -(1 : Int)
          return __py_ret_1
        else
          throw caught
      finally
        do
          let _ ← pyPrintIO [pyPrintArg "PastaLean handles exceptic
    PastaLean.PyExcept _)
```

`PyExceptId` (pure, provable) vs `PyExcept` = `ExceptT PyException IO` (real `IO`).

14.6

Classes / OOP

A Python `class` becomes a Lean `structure` plus namespaced method definitions. `self` is the structure value, and mutation is value-semantics (`C.new` builds one):

```python
class Point:
    def __init__(self, x,
        self.x = x
        self.y = y

    def norm_sq(self):
        return self.x*self
             + self.y*self

obj = Point(1,2)
norm = obj.norm_sq()
```

```
structure Point where
  x : Int
  y : Int
  deriving Inhabited, Repr, BEq

def Point.new := fun x ↦ fun y ↦ ({ x := x, y := y } : Point)
def Point.norm_sq := fun (self : Point) ↦ self.x *ₚ self.x +ₚ self.y *ₚ self.y

def obj := Point.new (1 : Int) (2 : Int)
def norm := Point.norm_sq obj
```

PastaLean

Classes / OOP

A Python `class` becomes a Lean `structure` plus namespaced method definitions. `self` is the structure value, and mutation is value-semantics (`C.new` builds one):

```python
class Point:
    def __init__(self, x,
        self.x = x
        self.y = y

    def norm_sq(self):
        return self.x*self
             + self.y*self

obj = Point(1,2)
norm = obj.norm_sq()
```

```
structure Point where
  x : Int
  y : Int
  deriving Inhabited, Repr, BEq

def Point.new := fun x ↦ fun y ↦ ({ x := x, y := y } : Point)
def Point.norm_sq := fun (self : Point) ↦ self.x *ₚ self.x +ₚ self.y *ₚ self.y

def obj := Point.new (1 : Int) (2 : Int)
def norm := Point.norm_sq obj
```

```
#check obj
```

```
obj : Point
```

```
#eval norm
```

```
5
```

PastaLean Libraries

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

A slice of `numpy`:

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

A slice of `numpy`:

```
open Libraries.numpy in
example : String → Lean.Name := fun member =
  match member with
  | "array"    ⇒ ``pyNumpyArray
  | "dot"      ⇒ ``pyNumpyDot
  | "mean"     ⇒ ``pyNumpyMean
  | "std"      ⇒ ``pyNumpyStd
  | "linspace" ⇒ ``pyNumpyLinspace
  | "norm"     ⇒ ``pyNumpyNorm
  | _          ⇒ .anonymous
```

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

A slice of `numpy`:

```
open Libraries.numpy in
example : String → Lean.Name := fun member =
  match member with
  | "array"    ⇒ ``pyNumpyArray
  | "dot"      ⇒ ``pyNumpyDot
  | "mean"     ⇒ ``pyNumpyMean
  | "std"      ⇒ ``pyNumpyStd
  | "linspace" ⇒ ``pyNumpyLinspace
  | "norm"     ⇒ ``pyNumpyNorm
  | _          ⇒ .anonymous
```

→ *transpile*

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

A slice of `numpy`:

```
open Libraries.numpy in
example : String → Lean.Name := fun member =
  match member with
  | "array"    ⇒ ``pyNumpyArray
  | "dot"      ⇒ ``pyNumpyDot
  | "mean"     ⇒ ``pyNumpyMean
  | "std"      ⇒ ``pyNumpyStd
  | "linspace" ⇒ ``pyNumpyLinspace
  | "norm"     ⇒ ``pyNumpyNorm
  | _          ⇒ .anonymous
```

→ *transpile*

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

A slice of `numpy`:

```
open Libraries.numpy in
example : String → Lean.Name := fun member =
  match member with
  | "array"    ⇒ ``pyNumpyArray
  | "dot"      ⇒ ``pyNumpyDot
  | "mean"     ⇒ ``pyNumpyMean
  | "std"      ⇒ ``pyNumpyStd
  | "linspace" ⇒ ``pyNumpyLinspace
  | "norm"     ⇒ ``pyNumpyNorm
  | _          ⇒ .anonymous
```

→ *transpile*

where `pyNumpyNorm` and other functions are as simple as:

PastaLean Libraries

Implementing Library and integrating it in PastaLean is very trivial.

- All you do is write the Lean function for your Python implementation, use our builtin typeclasses support if needed. All Libraries are independently implementable.
- Now just map the member name to a Lean runtime function. This will automatically integrate with our code generation pipeline.
- `numpy`, `pandas`, `scipy`, `math`, `typing` ship today; anything unsupported degrades to a flagged `pyUnsupported` so the file still compiles.

A slice of `numpy`:

```
open Libraries.numpy in
example : String → Lean.Name := fun member =
  match member with
  | "array"    ⇒ ``pyNumpyArray
  | "dot"      ⇒ ``pyNumpyDot
  | "mean"     ⇒ ``pyNumpyMean
  | "std"      ⇒ ``pyNumpyStd
  | "linspace" ⇒ ``pyNumpyLinspace
  | "norm"     ⇒ ``pyNumpyNorm
  | _          ⇒ .anonymous
```

→ *transpile*

where `pyNumpyNorm` and other functions are as simple as:

```
def pyNumpyNorm {α} [PyNumpyScalar α] (xs : List α) : Float :=
  let ys := xs.map toFloat
  Float.sqrt (ys.foldl (fun acc x ⇒ acc + x * x) 0.0)
```

PastaLean

Showcases

Real programs — transpiled, type-checked, and (for the pure parts) *proved*. Full code on *GitHub*. Press ↓.

Showcase	Python	→ Lean	Checks in	Exercises
`eg1`	39 loc	148 loc	5.2 s	exceptions · `zip` · comprehensions · `math`
`eg2`	36 loc	97 loc	5.1 s	`numpy` mean · matmul · `try` / `except`
`orbital`	186 loc	386 loc	7.8 s	vector algebra · conserved quantities · *proved*
`cnn`	141 loc	355 loc	5.4 s	a from-scratch CNN class · forward + backprop
`nn`	51 loc	134 loc	5.2 s	a neural-net forward pass
`linalg`	107 loc	208 loc	5.9 s	2×2 matrix algebra · all *proved*

~560 lines of Python → ~1,330 lines of *verified* Lean — each type-checked and verified within[17]

PastaLean

Showcases

Real programs — transpiled, type-checked, and (for the pure parts) *proved*. Full code on *GitHub*. Press ↓.

Showcase	Python	→ Lean	Checks in	Exercises
`eg1`	39 loc	148 loc	5.2 s	exceptions · `zip` · comprehensions · `math`
`eg2`	36 loc	97 loc	5.1 s	`numpy` mean · matmul · `try` / `except`
`orbital`	186 loc	386 loc	7.8 s	vector algebra · conserved quantities · *proved*
`cnn`	141 loc	355 loc	5.4 s	a from-scratch CNN class · forward + backprop
`nn`	51 loc	134 loc	5.2 s	a neural-net forward pass
`linalg`	107 loc	208 loc	5.9 s	2×2 matrix algebra · all *proved*

~560 lines of Python → ~1,330 lines of *verified* Lean — each type-checked and verified within 17.1

Orbital Physics · conserved quantities

Orbital Physics · conserved quantities

```
def cross_x(ax: float, ay: float, az: float, bx: float, by:
    """x-component of a x b."""
    return ay * bz - az * by

def norm_sq(ax, ay, az):
    return ax*ax + ay*ay + az*az

def kinetic(m, vx, vy, vz):
    return 0.5 * m * norm_sq(vx, vy, vz)

def momentum(m, vx, vy, vz):
    return m * norm_sq(vx, vy, vz)

def spring_energy(k: float, dx: float, dy: float, dz: float
    """Hooke potential energy (1/2) k |d|^2 stored in a spr
    return 0.5 * k * norm_sq(dx, dy, dz)

def momentum_conserved(m1: float, v1: float, m2: float, v2:
    """Equal and opposite impulses +j / -j conserve total l
    assert (m1 * v1 + j) + (m2 * v2 - j) == m1 * v1 + m2 *

def spring_force_is_central(k: float, dx: float, dy: float,
    """The Hooke force F = -k d is central (anti-parallel t
    no torque about the spring axis: d x F = 0 (x-component
    assert cross_x(dx, dy, dz, -k * dx, -k * dy, -k * dz)
```

Orbital Physics · conserved quantities

```
def cross_x(ax: float, ay: float, az: float, bx: float, by:
    """x-component of a x b."""
    return ay * bz - az * by

def norm_sq(ax, ay, az):
    return ax*ax + ay*ay + az*az

def kinetic(m, vx, vy, vz):
    return 0.5 * m * norm_sq(vx, vy, vz)

def momentum(m, vx, vy, vz):
    return m * norm_sq(vx, vy, vz)

def spring_energy(k: float, dx: float, dy: float, dz: float
    """Hooke potential energy (1/2) k |d|^2 stored in a spr
    return 0.5 * k * norm_sq(dx, dy, dz)

def momentum_conserved(m1: float, v1: float, m2: float, v2:
    """Equal and opposite impulses +j / -j conserve total l
    assert (m1 * v1 + j) + (m2 * v2 - j) == m1 * v1 + m2 *

def spring_force_is_central(k: float, dx: float, dy: float,
    """The Hooke force F = -k d is central (anti-parallel t
    no torque about the spring axis: d x F = 0 (x-component
```

Orbital Physics · conserved quantities

```python
def cross_x(ax: float, ay: float, az: float, bx: float, by:
    """x-component of a x b."""
    return ay * bz - az * by

def norm_sq(ax, ay, az):
    return ax*ax + ay*ay + az*az

def kinetic(m, vx, vy, vz):
    return 0.5 * m * norm_sq(vx, vy, vz)

def momentum(m, vx, vy, vz):
    return m * norm_sq(vx, vy, vz)

def spring_energy(k: float, dx: float, dy: float, dz: float
    """Hooke potential energy (1/2) k |d|^2 stored in a spr
    return 0.5 * k * norm_sq(dx, dy, dz)

def momentum_conserved(m1: float, v1: float, m2: float, v2:
    """Equal and opposite impulses +j / -j conserve total l
    assert (m1 * v1 + j) + (m2 * v2 - j) == m1 * v1 + m2 *

def spring_force_is_central(k: float, dx: float, dy: float,
    """The Hooke force F = -k d is central (anti-parallel t
    no torque about the spring axis: d x F = 0 (x-component
```

```
def cross_x := fun (ax : Rat) ↦ fun (ay : Rat) ↦ fun (az : Rat) ↦ fun (bx : Rat) ↦ fun
  ay *ₚ bz -ₚ az *ₚ cy

def norm_sq := fun (ax : Rat) ↦ fun (ay : Rat) ↦ fun (az : Rat) ↦ ax *ₚ ax +ₚ ay *ₚ ay +

def kinetic := fun (m : Rat) ↦ fun (vx : Rat) ↦ fun (vy : Rat) ↦ fun (vz : Rat) ↦ (0.5

def momentum := fun (m : Rat) ↦ fun (vx : Rat) ↦ fun (vy : Rat) ↦ fun (vz : Rat) ↦ m *ₚ

def spring_energy := fun (k : Rat) ↦ fun (dx : Rat) ↦ fun (dy : Rat) ↦ fun (dz : Rat) ↦
  /-
  Hooke potential energy (1/2) k |d|^2 stored in a spring stretched by displacement d.
  -/
  (0.5 : Rat) *ₚ k *ₚ norm_sq dx dy dz

@[taste_ingr]
theorem momentum_conserved :
    ∀ (m1 : Rat),
      ∀ (v1 : Rat), ∀ (m2 : Rat), ∀ (v2 : Rat), ∀ (j : Rat), m1 *ₚ v1 +ₚ j +ₚ (m2 *ₚ v2
  by intros; simp_all (config := { zetaDelta := true }) [taste_ingr]

@[taste_ingr]
theorem spring_force_is_central :
    ∀ (k : Rat),
      ∀ (dx : Rat), ∀ (dy : Rat), ∀ (dz : Rat), cross_x dx dy dz (-k *ₚ dx) (-k *ₚ dy) (
  by intros; simp_all (config := { zetaDelta := true }) [taste_ingr]; grind +locals
```

cnn · a CNN, transpiled

cnn · a CNN, transpiled

```
class CNN:
    def conv(self, img):
        # valid 2x2 conv: 8x8 → 7x7
        out = []
        for i in range(7):
            row = []
            for j in range(7):
                s = 0.0
                for a in range(2):
                    for b in range(
                        s += img[i+
                            * self.k
                row.append(s)
            out.append(row)
        return out
```

cnn · a CNN, transpiled

```
class CNN:
    def conv(self, img):
        # valid 2x2 conv: 8x8 → 7x7
        out = []
        for i in range(7):
            row = []
            for j in range(7):
                s = 0.0
                for a in range(2):
                    for b in range(
                        s += img[i+
                            * self.k
                row.append(s)
            out.append(row)
        return out
```

cnn · a CNN, transpiled

```python
class CNN:
    def conv(self, img):
        # valid 2x2 conv: 8x8 → 7x7
        out = []
        for i in range(7):
            row = []
            for j in range(7):
                s = 0.0
                for a in range(2):
                    for b in range(
                        s += img[i+
                            * self.k
                row.append(s)
            out.append(row)
        return out
```

```
structure CNN where
  kernel : List (List Real)
  dense_w : List Real
  dense_b : Real
  deriving Inhabited

noncomputable def CNN.conv := fun (self : CNN) ↦ fun (img : List (List Rat)) ↦
  Id.run
    (do
      let mut out := []
      for i in (PastaLean.pyRange (7 : Int))do
        let mut row := []
        for j in (PastaLean.pyRange (7 : Int))do
          let mut s := (0.0 : Real)
          for a in (PastaLean.pyRange (2 : Int))do
            for b in (PastaLean.pyRange (2 : Int))do
              s := s +ₚ img[i +ₚ a][j +ₚ b] *ₚ self.kernel[a][b]
          row := PastaLean.pyAppend row s
        out := PastaLean.pyAppend out row
      return out)
```

numpy + exceptions

numpy + exceptions

```
def process_data(data, weights):
    try:
        m = np.mean(data)
        print(f"Mean: {m}")
        centered = np.subtract(data,
                               [[m, m], [m, m]])
        return np.matmul(centered, weights)
    except ValueError as e:
        print(f"Failed: {e}")
        return np.zeros((2, 2))
```

numpy + exceptions

```
def process_data(data, weights):
    try:
        m = np.mean(data)
        print(f"Mean: {m}")
        centered = np.subtract(data,
                               [[m, m], [m, m]])
        return np.matmul(centered, weights)
    except ValueError as e:
        print(f"Failed: {e}")
        return np.zeros((2, 2))
```

numpy + exceptions

```python
def process_data(data, weights):
    try:
        m = np.mean(data)
        print(f"Mean: {m}")
        centered = np.subtract(data,
                               [[m, m], [m, m]])
        return np.matmul(centered, weights)
    except ValueError as e:
        print(f"Failed: {e}")
        return np.zeros((2, 2))
```

```
def process_data := fun (data : List (List Rat)) ↦ fun (weights : List (List Rat)) ↦
  ((do
      try
        let mut m := Libraries.numpy.pyNumpyMean data
        let _ ← pyPrintNoop [pyPrintArg s! "Mean: {m}"]
        let mut centered := Libraries.numpy.pyNumpySubtract data [[m, m], [m, m]]
        let mut result := Libraries.numpy.pyNumpyMatmul centered weights
        return result
      catch caught ⇒
        if (caught).OfKind == "ValueError" then
          let e := caught
          let _ ← pyPrintNoop [pyPrintArg s! "Failed: {e}"]
          let __py_ret_1 := Libraries.numpy.pyNumpyZeros ((2 : Int), (2 : Int))
          return __py_ret_1
        else
          throw caught) :
    PastaLean.PyExceptId _)
```

Design & architecture

The choices that make a *faithful*, *provable* translation possible — press ↓.

PastaLean

Design & architecture

The choices that make a *faithful*, *provable* translation possible — press ↓.

- **Problem → What we did → Why**, one architectural decision at a time.

Design & architecture

The choices that make a *faithful*, *provable* translation possible — press ↓.

- **Problem → What we did → Why**, one architectural decision at a time.
- Pure architecture — the elaborator and instance resolution do the heavy lifting.

1 · The logic subset

Not all of Python — only the part that *has* a static meaning.

1 · The logic subset

Not all of Python — only the part that *has* a static meaning.

- **Problem** — Python is dynamic: `eval`, `yield`, monkey-patching — no single static meaning.

1 · The logic subset

Not all of Python — only the part that *has* a static meaning.

- **Problem** — Python is dynamic: `eval`, `yield`, monkey-patching — no single static meaning.
- **We do** — target the typed "logic" subset; anything else degrades to a flagged `pyUnsupported`.

1 · The logic subset

Not all of Python — only the part that *has* a static meaning.

- **Problem** — Python is dynamic: `eval`, `yield`, monkey-patching — no single static meaning.
- **We do** — target the typed "logic" subset; anything else degrades to a flagged `pyUnsupported`.
- **Why** — a faithful, *provable* Lean model only exists for the static part. Refuse loudly, never miscompile.

1 · The logic subset

Not all of Python — only the part that *has* a static meaning.

- **Problem** — Python is dynamic: `eval`, `yield`, monkey-patching — no single static meaning.
- **We do** — target the typed "logic" subset; anything else degrades to a flagged `pyUnsupported`.
- **Why** — a faithful, *provable* Lean model only exists for the static part. Refuse loudly, never miscompile.

```
# in scope — a static, provable core
xs: list[int] = [1, 2, 3]
total = sum(x * x for x in xs)

# out of scope → `pyUnsupported`, flagged
async def fetch(): await conn.read()   # async
result = eval(user_input)              # reflection
def gen(): yield 1                     # laziness
```

2 · One name, many types

`len(x)` is *one* Lean call — the type system picks the implementation.

2 · One name, many types

`len(x)` is *one* Lean call — the type system picks the implementation.

- **Problem** — `len`, `x[i]`, `x in y`, `x+y` work on many types; codegen often can't *see* the type yet.

2 · One name, many types

`len(x)` is *one* Lean call — the type system picks the implementation.

- **Problem** — `len`, `x[i]`, `x in y`, `x+y` work on many types; codegen often can't *see* the type yet.
- **We do** — emit one stable name; Lean's *instance resolution* (after elaboration) selects the impl.

2 · One name, many types

`len(x)` is *one* Lean call — the type system picks the implementation.

- **Problem** — `len`, `x[i]`, `x in y`, `x+y` work on many types; codegen often can't *see* the type yet.
- **We do** — emit one stable name; Lean's *instance resolution* (after elaboration) selects the impl.
- **Why** — codegen stays type-blind; a new container = one new instance, *zero* codegen change.

2 · One name, many types

`len(x)` is *one* Lean call — the type system picks the implementation.

- **Problem** — `len`, `x[i]`, `x in y`, `x+y` work on many types; codegen often can't *see* the type yet.
- **We do** — emit one stable name; Lean's *instance resolution* (after elaboration) selects the impl.
- **Why** — codegen stays type-blind; a new container = one new instance, *zero* codegen change.

```
-- one Lean name; instance resolution picks the impl per type:
#check @pyLen

@pyLen : {α : Type} → [PyLen α] → α → ℤ
example : Int := pyLen [1, 2, 3]                    -- List
example : Int := pyLen "hello"                      -- String
example : Int := pyLen (Std.HashMap.ofList [(1, 2)])  -- dict
```

3 · Operators via `outParam`

`1 + 2.0 : Float` falls straight out of instance resolution.

3 · Operators via `outParam`

`1 + 2.0 : Float` falls straight out of instance resolution.

- **Problem** — Python mixes numeric types (`int+float` , `int+Rat`); Lean's `HAdd` has no widening instances.

3 · Operators via `outParam`

`1 + 2.0 : Float` falls straight out of instance resolution.

- **Problem** — Python mixes numeric types (`int+float` , `int+Rat`); Lean's `HAdd` has no widening instances.
- **We do** — custom `+ₚ -ₚ *ₚ` over `PyHAdd α β γ` , the *result* `γ` an `outParam` .

3 · Operators via `outParam`

`1 + 2.0 : Float` falls straight out of instance resolution.

- **Problem** — Python mixes numeric types (`int+float` , `int+Rat`); Lean's `HAdd` has no widening instances.
- **We do** — custom `+ₚ -ₚ *ₚ` over `PyHAdd α β γ` , the *result* `γ` an `outParam` .
- **Why** — the result type *reduces* to a concrete type, so a later `pyLen` /print/index can resolve. (`Rat` defaults stay off — subtle, load-bearing.)

3 · Operators via `outParam`

`1 + 2.0 : Float` falls straight out of instance resolution.

- **Problem** — Python mixes numeric types (`int+float` , `int+Rat`); Lean's `HAdd` has no widening instances.
- **We do** — custom `+ₚ -ₚ *ₚ` over `PyHAdd α β γ` , the *result* `γ` an `outParam` .
- **Why** — the result type *reduces* to a concrete type, so a later `pyLen` /print/index can resolve. (`Rat` defaults stay off — subtle, load-bearing.)

```
#check @PyHAdd

-- the result type is COMPUTED from the operands (the outParam γ):
PyHAdd : Type → Type → outParam Type → Type
example : Int    := (1 : Int) +ₚ (2 : Int)
example : Float  := (1 : Int) +ₚ (2.0 : Float)
example : String := "a" +ₚ "b"
```

4 · Classes are values

`self.x = 5` — on an *immutable* Lean structure.

4 · Classes are values

`self.x = 5` — on an *immutable* Lean structure.

- **Problem** — Python objects mutate in place; Lean structures are immutable values.

4 · Classes are values

`self.x = 5` — on an *immutable* Lean structure.

- **Problem** — Python objects mutate in place; Lean structures are immutable values.
- **We do** — mutation = *rebuild-and-return*; the receiver is reassigned at the call site.

4 · Classes are values

`self.x = 5` — on an *immutable* Lean structure.

- **Problem** — Python objects mutate in place; Lean structures are immutable values.
- **We do** — mutation = *rebuild-and-return*; the receiver is reassigned at the call site.
- **Why** — classes stay *pure & provable* (no heap / `IORef` monad). Cost: no aliasing — surfaced as an error, never a silent bug.

4 · Classes are values

`self.x = 5` — on an *immutable* Lean structure.

- **Problem** — Python objects mutate in place; Lean structures are immutable values.
- **We do** — mutation = *rebuild-and-return*; the receiver is reassigned at the call site.
- **Why** — classes stay *pure & provable* (no heap / `IORef` monad). Cost: no aliasing — surfaced as an error, never a silent bug.

```
-- `self.n = self.n + 1` ↦ record update, rebuilt & returned:
structure Counter where
  n : Int
  deriving Inhabited, Repr, BEq

def Counter.new : Int → Counter := fun (start : Int) ↦
  ({ n := start } : Counter)

def Counter.bump := fun (self : Counter) ↦
  Id.run
    (do
      let mut self := self
      self := { self with n := self.n +ₚ (1 : Int) }
      return self)
```

5 · Numbers: exact by default

Python `float` is lossy — a proof can't trust `0.1 + 0.2`.

5 · Numbers: exact by default

Python `float` is lossy — a proof can't trust `0.1 + 0.2`.

- **Problem** — Python `float` is IEEE-lossy; `0.1 + 0.2 ≠ 0.3`, so a proof can't rely on it.

5 · Numbers: exact by default

Python `float` is lossy — a proof can't trust `0.1 + 0.2`.

- **Problem** — Python `float` is IEEE-lossy; `0.1 + 0.2 ≠ 0.3`, so a proof can't rely on it.
- **We do** — `float` defaults to exact `ℚ` (provable); `--approx` → Lean `Float`; transcendentals → noncomputable `ℝ`.

5 · Numbers: exact by default

Python `float` is lossy — a proof can't trust `0.1 + 0.2`.

- **Problem** — Python `float` is IEEE-lossy; `0.1 + 0.2 ≠ 0.3`, so a proof can't rely on it.
- **We do** — `float` defaults to exact `ℚ` (provable); `--approx` → Lean `Float`; transcendentals → noncomputable `ℝ`.
- **Why** — exact rationals are decidable and *provable*; you opt into `Float` only when raw speed beats proof.

5 · Numbers: exact by default

Python `float` is lossy — a proof can't trust `0.1 + 0.2`.

- **Problem** — Python `float` is IEEE-lossy; `0.1 + 0.2 ≠ 0.3`, so a proof can't rely on it.
- **We do** — `float` defaults to exact `ℚ` (provable); `--approx` → Lean `Float`; transcendentals → noncomputable `ℝ`.
- **Why** — exact rationals are decidable and *provable*; you opt into `Float` only when raw speed beats proof.

```
-- Python floats default to EXACT rationals (ℚ):
example : (1 / 10 + 2 / 10 : Rat) = 3 / 10 := by
  native_decide

-- --approx picks Lean Float; transcendentals → ℝ:
#check (Float.sqrt 2.0)
```

```
Float.sqrt 2.0 : Float
```

```
#check (Real.sqrt 2)
```

```
√2 : ℝ
```

6 · Built to extend

New syntax, new library, new container — each is a *local, additive* change.

6 · Built to extend

New syntax, new library, new container — each is a *local, additive* change.

- **Problem** — normally, adding syntax or a library means touching the whole pipeline.

6 · Built to extend

New syntax, new library, new container — each is a *local, additive* change.

- **Problem** — normally, adding syntax or a library means touching the whole pipeline.
- **We do** — four open extension points: a `@[pygen]` registry, a name→name table, per-library `Mapping.lean`, `outParam` protocol instances.

6 · Built to extend

New syntax, new library, new container — each is a *local, additive* change.

- **Problem** — normally, adding syntax or a library means touching the whole pipeline.
- **We do** — four open extension points: a `@[pygen]` registry, a name→name table, per-library `Mapping.lean`, `outParam` protocol instances.
- **Why** — codegen, IR, and backend never change; the elaborator + instance resolution absorb the variation — new features stay cheap.

6 · Built to extend

New syntax, new library, new container — each is a *local, additive* change.

- **Problem** — normally, adding syntax or a library means touching the whole pipeline.
- **We do** — four open extension points: a `@[pygen]` registry, a name→name table, per-library `Mapping.lean`, `outParam` protocol instances.
- **Why** — codegen, IR, and backend never change; the elaborator + instance resolution absorb the variation — new features stay cheap.

```
-- every extension point produces a real runtime name:
#check @pyLen                          -- a container protocol
  @pyLen : {α : Type} → [PyLen α] → α → ℤ
#check @pyAppend                       -- a list method
  @pyAppend : {α : Type u_1} → List α → α → List α
#check @Libraries.numpy.pyNumpyMean    -- a library member
  @pyNumpyMean : {α : Type} → [PyNumpyScalar α] → List (List α) → Float
```

Limitations — where the line is

Honest about what PastaLean does *not* do:

Limitations — where the line is

Honest about what PastaLean does *not* do:

- **Not all of Python** — only the static "logic" subset; no `async`, `yield`, `eval`, or monkey-patching.

Limitations — where the line is

Honest about what PastaLean does *not* do:

- **Not all of Python** — only the static "logic" subset; no `async`, `yield`, `eval`, or monkey-patching.
- **One return type per function** — returning `int` on one path and `str` on another has no single Lean type.

Limitations — where the line is

Honest about what PastaLean does *not* do:

- **Not all of Python** — only the static "logic" subset; no `async`, `yield`, `eval`, or monkey-patching.
- **One return type per function** — returning `int` on one path and `str` on another has no single Lean type.
- **Proofs aren't free** — `taste?` / `mvcgen` are *best-effort* automation; hard goals still need a manual proof.

Limitations — where the line is

Honest about what PastaLean does *not* do:

- **Not all of Python** — only the static "logic" subset; no `async`, `yield`, `eval`, or monkey-patching.
- **One return type per function** — returning `int` on one path and `str` on another has no single Lean type.
- **Proofs aren't free** — `taste?` / `mvcgen` are *best-effort* automation; hard goals still need a manual proof.
- **Libraries are hand-written** — a new library's functions are implemented + mapped once, then reused everywhere.

Limitations — where the line is

Honest about what PastaLean does *not* do:

- **Not all of Python** — only the static "logic" subset; no `async`, `yield`, `eval`, or monkey-patching.
- **One return type per function** — returning `int` on one path and `str` on another has no single Lean type.
- **Proofs aren't free** — `taste?` / `mvcgen` are *best-effort* automation; hard goals still need a manual proof.
- **Libraries are hand-written** — a new library's functions are implemented + mapped once, then reused everywhere.
- **Unsupported → `pyUnsupported`** — anything out of scope degrades to a flagged placeholder, so the rest of the file *still compiles*.

Limitations — where the line is

Honest about what PastaLean does *not* do:

- **Not all of Python** — only the static "logic" subset; no `async`, `yield`, `eval`, or monkey-patching.
- **One return type per function** — returning `int` on one path and `str` on another has no single Lean type.
- **Proofs aren't free** — `taste?` / `mvcgen` are *best-effort* automation; hard goals still need a manual proof.
- **Libraries are hand-written** — a new library's functions are implemented + mapped once, then reused everywhere.
- **Unsupported → `pyUnsupported`** — anything out of scope degrades to a flagged placeholder, so the rest of the file *still compiles*.

When we can't translate faithfully, we refuse *loudly* — never miscompile.

PastaLean

Proving we cooked correctly

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

PastaLean

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ **Non-Monadic**

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ Non-Monadic

```
def transform_and_cube := fun a ↦ fun b ↦
  let c := a +ₚ b
  let d := a -ₚ b
  let e := c *ₚ d
  e ^ₚ (3 : Int)
```

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ Non-Monadic

```
def transform_and_cube := fun a ↦ fun b ↦
  let c := a +ₚ b
  let d := a -ₚ b
  let e := c *ₚ d
  e ^ₚ (3 : Int)
```

- **Proves** — equalities, identities
- **Tactics** — `taste?` → `ring` · `grind` · `nlinarith`
- **When** — pure terms, no effects
- Best-Effort use of `taste?`

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ Non-Monadic **↘ Monadic verification**

```
def transform_and_cube := fun a ↦ fun b ↦
  let c := a +ₚ b
  let d := a -ₚ b
  let e := c *ₚ d
  e ^ₚ (3 : Int)
```

- **Proves** — equalities, identities
- **Tactics** — `taste?` → `ring` · `grind` · `nlinarith`
- **When** — pure terms, no effects
- Best-Effort use of `taste?`

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ Non-Monadic

```
def transform_and_cube := fun a ↦ fun b ↦
  let c := a +ₚ b
  let d := a -ₚ b
  let e := c *ₚ d
  e ^ₚ (3 : Int)
```

↘ Monadic verification

```
def total (xs : List Int) : Int :=
  Id.run do
    let mut s := 0
    for x in xs do
      s := s +ₚ x
    return s
```

- **Proves** — equalities, identities
- **Tactics** — `taste?` → `ring` · `grind` · `nlinarith`
- **When** — pure terms, no effects
- Best-Effort use of `taste?`

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ Non-Monadic

```
def transform_and_cube := fun a ↦ fun b ↦
  let c := a +ₚ b
  let d := a -ₚ b
  let e := c *ₚ d
  e ^ₚ (3 : Int)
```

- **Proves** — equalities, identities
- **Tactics** — `taste?` → `ring` · `grind` · `nlinarith`
- **When** — pure terms, no effects
- Best-Effort use of `taste?`

↘ Monadic verification

```
def total (xs : List Int) : Int :=
  Id.run do
    let mut s := 0
    for x in xs do
      s := s +ₚ x
    return s
```

- **Proves** — Hoare triples `⦃P⦄ prog ⦃Q⦄`
- **Tactics** — `mvcgen` · `taste?`
- **When** — loops · mutation · `raise` / `IO`
- Uses LLM's for contracts, `mvcgen` is feed Loop invariants and subgoals are proven(best-effort) with `taste?`

PastaLean

Proving we cooked correctly

Is it correct? The proof path forks on one thing — a bare *term* or a `do` block:

↙ Non-Monadic

```
def transform_and_cube := fun a ↦ fun b ↦
  let c := a +ₚ b
  let d := a -ₚ b
  let e := c *ₚ d
  e ^ₚ (3 : Int)
```

- **Proves** — equalities, identities
- **Tactics** — `taste?` → `ring` · `grind` · `nlinarith`
- **When** — pure terms, no effects
- Best-Effort use of `taste?`

↘ Monadic verification

```
def total (xs : List Int) : Int :=
  Id.run do
    let mut s := 0
    for x in xs do
      s := s +ₚ x
    return s
```

- **Proves** — Hoare triples `⦃P⦄ prog ⦃Q⦄`
- **Tactics** — `mvcgen` · `taste?`
- **When** — loops · mutation · `raise` / `IO`
- Uses LLM's for contracts, `mvcgen` is feed Loop invariants and subgoals are proven(best-effort) with `taste?`

A preference to Non-monadic pure functional code is given while converting. However it will be lowered to Monadic code with `do`, when necessary.

Why lower into a monad at all?

PastaLean

Why lower into a monad at all?

Pure functions are total and side-effect free, so they cannot directly express state updates, early exits, or external effects like IO. Lowering into a monad gives these behaviors a precise semantic model while keeping the code composable and type-safe.

Why lower into a monad at all?

Pure functions are total and side-effect free, so they cannot directly express state updates, early exits, or external effects like IO. Lowering into a monad gives these behaviors a precise semantic model while keeping the code composable and type-safe.

- `print` / `input` → `IO`

Why lower into a monad at all?

Pure functions are total and side-effect free, so they cannot directly express state updates, early exits, or external effects like IO. Lowering into a monad gives these behaviors a precise semantic model while keeping the code composable and type-safe.

- `print` / `input` → **`IO`**
- `raise` / `try` / `except` → **`PyExcept`** = `ExceptT PyException IO`

Why lower into a monad at all?

Pure functions are total and side-effect free, so they cannot directly express state updates, early exits, or external effects like IO. Lowering into a monad gives these behaviors a precise semantic model while keeping the code composable and type-safe.

- `print` / `input` → **`IO`**
- `raise` / `try` / `except` → **`PyExcept`** = `ExceptT PyException IO`
- loops / mutation → **`Id.run do`** (pure, but still a `do` block)

Why lower into a monad at all?

Pure functions are total and side-effect free, so they cannot directly express state updates, early exits, or external effects like IO. Lowering into a monad gives these behaviors a precise semantic model while keeping the code composable and type-safe.

- `print` / `input` → **`IO`**
- `raise` / `try` / `except` → **`PyExcept`** = `ExceptT PyException IO`
- loops / mutation → **`Id.run do`** (pure, but still a `do` block)

Tradeoff : Extensibility *vs* Harder to Prove

The one place an LLM helps: contracts

The one place an LLM helps: contracts

How do we even say what we want to prove?

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.
- The **LLM proposes** the contracts; `mvcgen` + `taste?` discharge the *proof*.

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.
- The **LLM proposes** the contracts; `mvcgen` + `taste?` discharge the *proof*.

Before · Python + contracts

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.
- The **LLM proposes** the contracts; `mvcgen` + `taste?` discharge the *proof*.

Before · Python + contracts

```
def factorial(n: int) → int:
    Requires(n ⩾ 0)
    Ensures(Result() ⩾ 1)
    result, i = 1, 1
    while i ⩽ n:
        Invariant(i ⩾ 1)
        Invariant(n - i + 1 ⩾ 0)
        Invariant(result ⩾ 1)
        Decreases(n - i + 1)
        result = result * i
        i = i + 1
    return result
```

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.
- The **LLM proposes** the contracts; `mvcgen` + `taste?` discharge the *proof*.

Before · Python + contracts

```
def factorial(n: int) → int:
    Requires(n ⩾ 0)
    Ensures(Result() ⩾ 1)
    result, i = 1, 1
    while i ⩽ n:
        Invariant(i ⩾ 1)
        Invariant(n - i + 1 ⩾ 0)
        Invariant(result ⩾ 1)
        Decreases(n - i + 1)
        result = result * i
        i = i + 1
    return result
```

→ *contracts*

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.
- The **LLM proposes** the contracts; `mvcgen` + `taste?` discharge the *proof*.

Before · Python + contracts

```python
def factorial(n: int) → int:
    Requires(n ⩾ 0)
    Ensures(Result() ⩾ 1)
    result, i = 1, 1
    while i ⩽ n:
        Invariant(i ⩾ 1)
        Invariant(n - i + 1 ⩾ 0)
        Invariant(result ⩾ 1)
        Decreases(n - i + 1)
        result = result * i
        i = i + 1
    return result
```

→ *contracts*

After · transpiled Lean

The one place an LLM helps: contracts

How do we even say what we want to prove?

- Annotate Python with *contracts* — `Requires`, `Ensures`, `Invariant`, `Decreases`, or a plain `assert`.
- PastaLean extracts them as **preconditions**, **postconditions**, and per-loop obligations.
- The **LLM proposes** the contracts; `mvcgen` + `taste?` discharge the *proof*.

Before · Python + contracts

```
def factorial(n: int) → int:
    Requires(n ⩾ 0)
    Ensures(Result() ⩾ 1)
    result, i = 1, 1
    while i ⩽ n:
        Invariant(i ⩾ 1)
        Invariant(n - i + 1 ⩾ 0)
        Invariant(result ⩾ 1)
        Decreases(n - i + 1)
        result = result * i
        i = i + 1
    return result
```

→ *contracts*

After · transpiled Lean

```
def factorial := fun (n : Int) ↦
  (do
    let mut result := (1 : Int)
    for i in (PastaLean.pyRange (n +ₚ (1 : Int)) (1 : Int))do
      let _ := Libraries.passta.pyPassInvariant (decide (i ≥ (1 : I
      let _ := Libraries.passta.pyPassInvariant (decide (n -ₚ i +ₚ
      let _ := Libraries.passta.pyPassInvariant (decide (result ≥ (
      let _ := Libraries.passta.pyPassDecreases (n -ₚ i +ₚ (1 : Int
      result := result *ₚ i
    return result : Id _)
```

PastaLean

Best Effort Automatic Proving with `taste?`

PastaLean

Best Effort Automatic Proving with `taste?`

A Python `assert` on a pure function becomes `theorem … := by taste?` — the *Pastafolio* engine searches (`simp` · `grind` · `ring` · `nlinarith` · `fun_induction`) and *splices the found proof back*:

PastaLean

Best Effort Automatic Proving with `taste?`

A Python `assert` on a pure function becomes `theorem … := by taste?` — the *Pastafolio* engine searches (`simp` · `grind` · `ring` · `nlinarith` · `fun_induction`) and *splices the found proof back*:

```python
# showcase/linalg/matrix_model.py
def det(a, b, c, d):
    return a*d - b*c

# scaling a 2×2 by k scales det by k²
assert det(k*a, k*b, k*c, k*d) \
    == k**2 * det(a, b, c, d)
```

Best Effort Automatic Proving with `taste?`

A Python `assert` on a pure function becomes `theorem … := by taste?` — the *Pastafolio* engine searches (`simp` · `grind` · `ring` · `nlinarith` · `fun_induction`) and *splices the found proof back*:

```python
# showcase/linalg/matrix_model.py
def det(a, b, c, d):
    return a*d - b*c

# scaling a 2×2 by k scales det by k²
assert det(k*a, k*b, k*c, k*d) \
    == k**2 * det(a, b, c, d)
```

→ *prove*

PastaLean

Best Effort Automatic Proving with `taste?`

A Python `assert` on a pure function becomes `theorem … := by taste?` — the *Pastafolio* engine searches (`simp` · `grind` · `ring` · `nlinarith` · `fun_induction`) and *splices the found proof back*:

```python
# showcase/linalg/matrix_model.py
def det(a, b, c, d):
    return a*d - b*c

# scaling a 2×2 by k scales det by k²
assert det(k*a, k*b, k*c, k*d) \
    == k**2 * det(a, b, c, d)
```

→ *prove*

```
def det (a b c d : Rat) : Rat :=
  a * d - b * c

example (a b c d k : Rat) :
    det (k*a) (k*b) (k*c) (k*d)
      = k ^ 2 * det a b c d := by
  taste? -- TryThis: grind +locals +suggestions
```

PastaLean

Best Effort Automatic Proving with `taste?`

A Python `assert` on a pure function becomes `theorem … := by taste?` — the *Pastafolio* engine searches (`simp` · `grind` · `ring` · `nlinarith` · `fun_induction`) and *splices the found proof back*:

```python
# showcase/linalg/matrix_model.py
def det(a, b, c, d):
    return a*d - b*c

# scaling a 2×2 by k scales det by k²
assert det(k*a, k*b, k*c, k*d) \
    == k**2 * det(a, b, c, d)
```

→ *prove*

```
def det (a b c d : Rat) : Rat :=
  a * d - b * c

example (a b c d k : Rat) :
    det (k*a) (k*b) (k*c) (k*d)
      = k ^ 2 * det a b c d := by
  taste? -- TryThis: grind +locals +suggestions
```

The found tactic is *spliced back* over `taste?` — the committed proof is concrete.

Best Effort Automatic Proving with `taste?`

A Python `assert` on a pure function becomes `theorem … := by taste?` — the *Pastafolio* engine searches (`simp` · `grind` · `ring` · `nlinarith` · `fun_induction`) and *splices the found proof back*:

```python
# showcase/linalg/matrix_model.py
def det(a, b, c, d):
    return a*d - b*c

# scaling a 2×2 by k scales det by k²
assert det(k*a, k*b, k*c, k*d) \
    == k**2 * det(a, b, c, d)
```

→ *prove*

```
def det (a b c d : Rat) : Rat :=
  a * d - b * c

example (a b c d k : Rat) :
    det (k*a) (k*b) (k*c) (k*d)
      = k ^ 2 * det a b c d := by
  taste? -- TryThis: grind +locals +suggestions
```

The found tactic is *spliced back* over `taste?` — the committed proof is concrete.

```
theorem factorial_spec : ⦃⌜n ≥ (0 : Int)⌝⦄ factorial n ⦃⇓result ⇒ ⌜result ≥ (1 : Int)⌝⦄ :=
  by
  mvcgen [factorial, PastaLean.pyRange_forIn, PastaLean.pyRange_forIn_start] invariants
  · ⇓⟨cur, result⟩ ⇒
    ⌜let i := (cur.prefix.length : Int);
      (i ≥ (1 : Int) ∧ n -ₚ i +ₚ (1 : Int) ≥ (0 : Int)) ∧ result ≥ (1 : Int)⌝
  simp_all (config := { zetaDelta := true }) [taste_ingr];
  sorry; sorry; omega
```

Proving monadic defs: `mvcgen`

`do`-block functions use `Std.Do`'s `mvcgen` with Hoare triples `⦃P⦄ prog ⦃Q⦄`; Python contracts ride along as `Invariant` / `Ensures`:

PastaLean

Proving monadic defs: `mvcgen`

`do`-block functions use `Std.Do`'s `mvcgen` with Hoare triples `⦃P⦄ prog ⦃Q⦄`; Python contracts ride along as `Invariant` / `Ensures`:

```
def sum_upto_n(n: int) → int
    Requires(n ⩾ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total ==
        total += i
    Ensures(2*total == n*(n +
    return total
```

Proving monadic defs: `mvcgen`

`do`-block functions use `Std.Do`'s `mvcgen` with Hoare triples `⦃P⦄ prog ⦃Q⦄`; Python contracts ride along as `Invariant` / `Ensures`:

```
def sum_upto_n(n: int) → int
    Requires(n ⩾ 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total ==
        total += i
    Ensures(2*total == n*(n +
    return total
```

→ *prove* →

Proving monadic defs: `mvcgen`

`do`-block functions use `Std.Do`'s `mvcgen` with Hoare triples `⦃P⦄ prog ⦃Q⦄`; Python contracts ride along as `Invariant` / `Ensures`:

```python
def sum_upto_n(n: int) -> int
    Requires(n >= 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total ==
        total += i
    Ensures(2*total == n*(n +
    return total
```

→ *prove* →

```
theorem sum_upto_n_spec : ⦃⌜n ≥ (0 : Int)⌝⦄ sum_to_n n ⦃⇓_ ⇒ ⌜True⌝⦄ :=
  by
  mvcgen [sum_to_n, PastaLean.pyRange_forIn, PastaLean.pyRange_forIn_start] invariants
  · ⇓⟨cur, total⟩ ⇒
    ⌜let i := (cur.prefix.length : Int);
      (2 : Int) *ₚ total = i *ₚ (i -ₚ (1 : Int))⌝
  simp_all [taste_ingr]; grind +locals +suggestions
```

Proving monadic defs: `mvcgen`

`do`-block functions use `Std.Do`'s `mvcgen` with Hoare triples `⦃P⦄ prog ⦃Q⦄`; Python contracts ride along as `Invariant` / `Ensures`:

```python
def sum_upto_n(n: int) -> int
    Requires(n >= 0)
    total = 0
    for i in range(n + 1):
        Invariant(2*total ==
        total += i
    Ensures(2*total == n*(n +
    return total
```

→ *prove* →

```
theorem sum_upto_n_spec : ⦃⌜n ≥ (0 : Int)⌝⦄ sum_to_n n ⦃⇓_ ⇒ ⌜True⌝⦄ :=
  by
  mvcgen [sum_to_n, PastaLean.pyRange_forIn, PastaLean.pyRange_forIn_start] invariants
  · ⇓⟨cur, total⟩ ⇒
    ⌜let i := (cur.prefix.length : Int);
      (2 : Int) *ₚ total = i *ₚ (i -ₚ (1 : Int))⌝
  simp_all [taste_ingr]; grind +locals +suggestions
```

`Invariant` / `Ensures` → `pyPassInvariant` / `pyPassEnsures`; `mvcgen` using Hoare Triples, discharges Goals to prove.

Recap

- *Deterministic transpiler*, not an LLM translator — the elaborator is the oracle.
- Broad language surface: control flow, comprehensions, exceptions, *classes*, libraries.
- *Monadic vs non-monadic* picks the proof strategy:
  - Non-monadic pure function → `taste?`
  - Monadic functions in `do` blocks → `mvcgen`
- *LLMs propose contracts*; Lean checks them.

Pasta + Python + Lean = Python you can *prove*.

Thank you

Questions?