

Educator pack

This pack turns the core 16 lessons of the [Learn Pyfun](#) track into a ready-made functional-programming unit you can slot into an existing Python course. It is organized as five class sessions with demo scripts, exercise assignments, and an instructor answer key. You can lift it wholesale or adapt any part of it.

For a department that circulates paper, the whole pack (this overview, the five sessions and the answer keys) is also a single [printable PDF](#). Its links still open the lessons and playground exercises.

Who this is for

Instructors of an intro or intermediate Python course who want to add a functional-programming unit without leaving the Python ecosystem. The sessions assume you are comfortable teaching Python and want to introduce ideas like immutability, algebraic data types, exhaustive matching, and effects using a language that compiles to readable Python your students already recognize.

What students need beforehand

Students should already know Python variables, functions, lists, and dictionaries. No functional programming background is assumed. The unit introduces every functional idea from scratch and keeps returning to the Python each construct compiles to, so students always have a familiar anchor.

Time budget

The unit runs as five sessions of about 90 minutes of class time each, plus homework between sessions. Each session doc includes a rough minute budget for the demo, discussion, and in-class exercises.

If your schedule favors shorter sessions, split the unit into six. The natural split point is inside Session 5: run lessons 12 and 13 as one session and lessons 14 through 16 as a sixth, with the capstone (lesson 16) as the finale. That split is noted again in the Session 5 doc.

Beyond the core unit, the course has a **Going further** set (recursion, async, active patterns, building a computation expression, string/number details, opaque types, and keyword arguments at the Python boundary). It works well as stretch material for students who finish early, as self-study after the unit, or as the seed of an extra session if your course has room.

Session	Title	Lessons
1	Functional basics in Python's clothing	1 to 2
2	Making illegal states unrepresentable	3 to 5
3	Data modeling	6 to 8
4	The Pyfun workflow	9 to 11
5	Reaching the Python ecosystem	12 to 16

Session docs:

- [Session 1: Functional basics in Python's clothing](#)
- [Session 2: Making illegal states unrepresentable](#)
- [Session 3: Data modeling](#)
- [Session 4: The Pyfun workflow](#)
- [Session 5: Reaching the Python ecosystem](#)
- [Answer keys](#)

Running the material with zero setup

The whole unit runs in a browser. Every lesson and every demo step links to the [playground](#), where the real compiler checks code as you type and a Run button executes it. Nothing needs to be installed for students to follow along or to do the exercises.

If you have lab machines and want students working locally, two options add a native workflow:

```
pip install pyfun-lang
pyfun run lesson.pyfun
```

Or a notebook workflow with the Jupyter kernel:

```
pip install "pyfun-lang[jupyter]"  
python -m pyfun_kernel.install
```

Every lesson is also published as a notebook, one `.ipynb` per lesson in [docs/src/notebooks/](#), ready to hand out through a course platform. The notebooks are generated from the lesson pages, so they always match what the site says, and each solution stays folded away until a student opens it.

Only lesson 15 (modules across files) needs the installed compiler, because it uses a folder of source files. Every other lesson runs in the playground as it is.

Licensing

This teaching material is licensed under [Creative Commons Attribution 4.0 International \(CC BY 4.0\)](#). You are free to adapt, remix, and redistribute it, including for classroom and commercial use, as long as you give attribution. Institutions should feel free to lift it wholesale into their own courses.

The Pyfun code samples themselves remain under the Apache 2.0 license that covers the rest of the repository.

Session 1: Functional basics in Python's clothing

Objectives

By the end of this session, students can:

- Bind values with `let` and explain why a binding is immutable by default.
- Read the type the compiler infers for a value without writing any annotations.
- Define a function with inline parameters and call it without parentheses or commas.
- Explain currying by showing that a partial call returns another function.
- Read a pipeline `a |> f |> g` and predict the equivalent nested Python call.

Prerequisites

Python variables and functions. No functional programming background. This is the first session of the unit.

Demo script

The thread for this session is that Pyfun is just Python underneath. Keep the playground's Python output panel visible the whole time so the class sees the emitted Python change as you type.

1. Open the Session 1 worked example in the playground: [functions and currying](#). Press Run first so the class sees `5`, `11`, `12`. Ask them to predict what each line does.
2. Point at the Python panel. Show that `let add a b = a + b` became a plain `def add(a, b)`, and that `print (add 2 3)` became `print(add(2, 3))`. Make the point that there is no runtime library: the output is ordinary Python.
3. Highlight `let inc = add 1`. This is the currying moment. Show that it compiled to `inc = functools.partial(add, 1)`. Explain that a call missing an argument hands back a

function rather than raising, which is what lets `inc` exist.

4. Highlight `let result = 5 |> inc |> double`. Show that it compiled to `double(inc(5))`. Read the pipe left to right (“start with 5, then inc, then double”) and contrast with the inside-out Python nesting. This is the reading-order payoff of the pipe.
5. Break it live. Change `let inc = add 1` to `let inc = add 1 2 3`. The compiler reports too many arguments. Undo, then change `double x = x * 2` to `double x = x * "2"` and show the type error arriving before any Python is produced. Restore the working program.
6. Switch to the Lesson 1 idea of inference. Open the [Lesson 1 exercise](#). Run `check`, show that the hole `?count` reports the type `int` and lists names in scope, and fill it with `oranges`. This previews the hole workflow the class will lean on all unit.
7. To close the “just Python” thread, add a line `let label = "age: " + 36` to the Lesson 1 program and show the diagnostic that `+` is numeric and does not concatenate strings. Contrast with Python, where the same mistake would surface only at runtime.

Assigned exercises

- In class: the [Lesson 1 exercise](#) (fill the hole), done together in step 6 above.
- Homework: the [Lesson 2 exercise](#), finishing the pipeline so the result is 18. Students should read the hole’s reported type before filling it.

Common misconceptions

- “`let` is just assignment, so I can reassign it later.” Correction: a `let` binds a value once and reassigning it is a compile error. Mutation is a separate, explicit feature that arrives in lesson 10.
- “`add 1` is a bug, because `add` takes two arguments.” Correction: a partial call is intentional. It returns a function waiting for the remaining argument. That is currying.
- “The pipe must add overhead, like a stream or an iterator.” Correction: `|>` is compile-time sugar. `5 |> inc |> double` compiles to `double(inc(5))` with no runtime layer.
- “No type annotations means the language is dynamically typed like Python.” Correction: types are inferred and fully checked before any Python is emitted. The absence of annotations is a design choice, not the absence of static typing.
- “Calling `add 2 3` without commas looks like a syntax error.” Correction: function application in Pyfun is by juxtaposition, so `add 2 3` is the call. Parentheses are only for

grouping.

Timing

- 10 min: framing the unit and what Pyfun is (compiles to readable Python).
- 30 min: demo script steps 1 to 5 (functions, currying, pipes, breaking it live).
- 20 min: demo script steps 6 to 7 (inference and the hole workflow).
- 20 min: students start the Lesson 2 exercise in the playground while you circulate.
- 10 min: recap and assign the Lesson 2 exercise as homework.

Answer keys: [answer-keys.md](#)

Session 2: Making illegal states unrepresentable

Objectives

By the end of this session, students can:

- Explain why `None` is a source of runtime bugs in Python and how `Option` moves that possibility into the type.
- Take an `Option` or `Result` apart with `match` and handle both shapes.
- Turn a Python exception into a `Result` value with `try` and read its `errorKind` and `errorMessage`.
- Define their own algebraic data type with several constructors.
- Read a non-exhaustive-match diagnostic, name the case it reports, and add it.

Prerequisites

Session 1 (values, functions, inference). Python's own `match / case` helps but is not required.

Demo script

The centerpiece is the exhaustiveness error naming the exact case the code forgot. Build up to it with `Option` and `Result`, then land it on a hand-written ADT.

1. Open the [Lesson 3 exercise](#). Run `check`. The compiler reports `non-exhaustive match: None is not matched`. Ask the class what value is missing before revealing it, then add `case None: "no number"` and Run.
2. Make the point that in Python `int("nope")` raises, and a caller has to remember to guard it. Here the `None` case is not a convention to remember, it is a case the compiler refuses to let you skip.

3. Open the [Lesson 4 exercise](#). It matches on `parseInt s` directly, so `check` reports a type mismatch: it found `int`, not a `Result`. Wrap the call as `try (parseInt s)` and Run. Show that the caught `ValueError` reaches the `Error` arm as data.
4. Draw the contrast: `Option` models “a value might be absent,” `Result` models “an operation might fail with a reason.” Both force the caller to answer for the unhappy path.
5. Now the centerpiece. Open the [Lesson 5 exercise](#). The `Light` type has three constructors but `action` answers for two. Run `check` and let the diagnostic speak: `non-exhaustive match: Amber is not matched`. The compiler names the exact forgotten state.
6. Add `case Amber: "wait"` and Run to get the three expected lines. Then delete `case Green:` instead and show the diagnostic now names `Green`. The point is that the checker tracks every constructor, not just the first gap.
7. Show the emitted Python for the working `Light` program. Point out the trailing `case _: raise RuntimeError("non-exhaustive match")` guard and explain that because the compiler already proved coverage, that guard can never fire. Exhaustiveness is checked before any Python exists.
8. Optional stretch: add a fourth constructor `| Flashing` to the type without touching `action`. The same diagnostic reappears naming `Flashing`, showing that widening a type reopens every match over it.

Assigned exercises

- In class: the [Lesson 3 exercise](#) and the [Lesson 5 exercise](#), both driven from the demo.
- Homework: the [Lesson 4 exercise](#) (wrap the parse in `try`). Ask students to read `errorKind` and `errorMessage` off the caught value in their answer.

Common misconceptions

- “`Option` is just `None` with extra steps.” Correction: the difference is enforcement. With `Option`, the compiler refuses to compile until the `None` case is handled, so the forgotten check cannot happen.
- “`try` here works like Python’s `try / except` block.” Correction: `try e` is an expression that produces a `Result` value. There is no jump to a handler. The failure is data you match on in the same expression.

- “An ADT is basically a Python `Enum`.” Correction: an `Enum` lists bare names, but ADT constructors can carry data of different types (`Circle float`, `Rect float float`), and matching binds that data.
- “I can add a `case _:` catch-all to silence the exhaustiveness error.” Correction: that works but throws away the guarantee. When you later add a constructor, a real `match` reopens with a diagnostic, while a catch-all silently swallows the new case.
- “The exhaustiveness check happens at runtime, like the `RuntimeError` guard.” Correction: the check runs during type checking, before any Python is emitted. The runtime guard is a backstop that a proven-total match never reaches.

Timing

- 10 min: recap Session 1, then motivate the `None` problem in Python.
- 20 min: demo steps 1 to 4 (`Option` and `Result`).
- 25 min: demo steps 5 to 8 (the exhaustiveness centerpiece on a hand-written ADT).
- 25 min: students work the Lesson 3 and Lesson 5 exercises in the playground.
- 10 min: recap and assign the Lesson 4 exercise as homework.

Answer keys: [answer-keys.md](#)

Session 3: Data modeling

Objectives

By the end of this session, students can:

- Declare a record type with named fields and construct a value with the tagged constructor syntax.
- Copy and update a record and explain why the original is left unchanged.
- Match on a record, binding a subset of fields with the `{ x }` shorthand.
- Work over lists with `List.map`, `List.filter`, and `List.fold`, using lambdas and operator sections.
- Group values by position with tuples and destructure them in a `match` arm.

Prerequisites

Sessions 1 and 2 (immutability, functions, `match`, ADTs). Python dictionaries and dataclasses give useful points of comparison.

Demo script

The thread is choosing the right shape for data: a record for one thing with named parts, a collection for many things, a tuple for a short positional pairing. Keep the Python panel visible so students see records become frozen dataclasses.

1. Open the [Lesson 6 exercise](#). Run `check` and read the hole note: the hole has type `int`. Fill it with `100` and Run. Draw attention to the last line: `opened.balance` still prints `0`.
2. Explain the copy-and-update line `{ opened with balance = 100 }`. It builds a fresh `Account` and leaves `opened` alone, which is why the original balance is unchanged. Connect this back to the immutability from Session 1.
3. Show the Python panel: `Account` compiled to a `@dataclass(frozen=True, slots=True)`, and the update compiled to a new `Account(...)` call. The immutability is

literal in the output.

4. Break it live: change `Account { name = "Ada", balance = 0 }` to drop the `balance` field. The compiler reports the missing field. Restore it. This shows records require all their fields.
5. Move to collections. Open the [Lesson 7 exercise](#). Run `check` and show the two hole notes. The first hole wants `int -> int -> int`. Fill it with the operator section `(+)`, fill the second with `0`, and Run.
6. Make the totality point: there is no `ns[10]` bracket indexing in Pyfun. `List.get 10 ns` returns an `Option`, and `Option.withDefault` supplies the fallback, so an out-of-range lookup cannot crash. This is the same lesson from Session 2 applied to collections.
7. Move to tuples. Open the [Lesson 8 exercise](#). Show that `List.zip names scores` produced a list of pairs and that the single `case (name, score)` arm deconstructs each pair. Fill the hole with `f"{name}: {score}"` and Run.
8. Close by contrasting the three shapes with one line each: a record when parts have names, a list when there are many of the same thing, a tuple for a short fixed pairing like a name and a score.

Assigned exercises

- In class: the [Lesson 6 exercise](#) and the [Lesson 7 exercise](#), both driven from the demo.
- Homework: the [Lesson 8 exercise](#) (build one line per pair, then join). Ask students to note how the tuple pattern binds both parts at once.

Common misconceptions

- “A record is a dictionary, so I can add keys later.” Correction: a record’s fields are fixed by its type. It compiles to a frozen dataclass, and every field must be present when you build it.
- “`{ opened with balance = 100 }` mutates `opened`.” Correction: copy-and-update returns a new record. The original is untouched, which is why `opened.balance` still reads `0`.
- “`List.get 10 ns` will raise `IndexError` like Python indexing.” Correction: there is no bracket indexing. `List.get` returns an `Option`, so an out-of-range index is `None`, a value you handle.

- “`(+)` and `(*) 2` are odd syntax I have to memorize.” Correction: an operator in parentheses is just that operator as a function, and supplying one argument (`(*) 2`) is the same partial application from Session 1.
- “A one-element tuple `(x)` groups one value into a tuple.” Correction: `(x)` is just `x` in parentheses. There is no one-element tuple. A real tuple has two or more elements, and `()` is unit.

Timing

- 10 min: recap Session 2, then motivate choosing a data shape.
- 20 min: demo steps 1 to 4 (records and copy-and-update).
- 20 min: demo steps 5 to 6 (collections and totality).
- 15 min: demo steps 7 to 8 (tuples and the summary).
- 20 min: students work the Lesson 6 and Lesson 7 exercises in the playground.
- 5 min: recap and assign the Lesson 8 exercise as homework.

Answer keys: [answer-keys.md](#)

Session 4: The Pyfun workflow

Objectives

By the end of this session, students can:

- Use a typed hole to sketch a program and let the compiler report the type and fitting names at each gap.
- Fill a pipeline inward from the types the holes report, without guessing function signatures.
- Declare a deliberate local accumulator with `let mut` and reassign it with `<-`.
- Explain why immutability is the default and mutation is opt-in and visible.
- Read an inferred effect, assert purity with `let pure`, and fix a failed purity assertion by moving the effect to the call site.

Prerequisites

Sessions 1 through 3 (values, functions, `match`, records, collections). Students have already seen holes in passing when the earlier exercises reported a hole type.

Demo script

The thread is the day-to-day loop of writing Pyfun: sketch with holes, let types guide the fill, reach for mutation only when it is genuinely clearest, and let the compiler track effects for you.

1. Open the [Lesson 9 exercise](#). Run `check`. Two holes report at once. Read the `?render` note aloud: it has type `int -> 'a` and suggests `String.fromInt` among the fits. This is type-driven development in one screen.
2. Fill `?missing` with `0` and `?render` with `String.fromInt`, then Run to get `10` and `0`. Make the point that you never looked up a signature: each hole reported exactly what belonged there.

3. Emphasize that a hole blocks compilation by design. Nothing runs until every blank is filled, so a half-written program cannot accidentally execute. Contrast with a Python `pass` or `TODO`, which runs and fails later.
4. Move to mutation. Open the [Lesson 10 exercise](#). It reassigns `acc` with `<-` but declared it with a plain `let`. Run `check` and read the diagnostic: it says `acc` is immutable and points to `let mut`. Add `mut` and Run to get `120`.
5. Show the emitted Python: the block became the plain statement sequence you would write by hand, `total = ...` reassigned in place. The `<-` arrow is what makes mutation visible in the source, and it is scoped to the block. Note there is no `for` or `while`, so `let mut` is for a genuine accumulator, not for iterating a collection.
6. Move to effects. Open the [Lesson 11 exercise](#). `greet` is declared `pure` but prints inside its body. Run `check` and read the diagnostic: `greet is declared pure but performs io`, pointing at the `print` line.
7. Fix it the right way. Make `greet` return the string (`let pure greet name = f"Hello, {name}"`) and move the `print` to the call site (`print (greet "ada")`). Run. Make the point: the fix is to separate the calculating part, which stays provably pure, from the part that talks to the world, which now says so.
8. To reinforce inference, add a second function that calls `greet` and prints, and show that its `io` effect is inferred without any annotation, while a `let pure` on it would be rejected. Effect tracking propagates outward on its own.

Assigned exercises

- In class: the [Lesson 9 exercise](#) and the [Lesson 10 exercise](#), both driven from the demo.
- Homework: the [Lesson 11 exercise](#) (make `greet` pure by returning the string and printing at the call site).

Common misconceptions

- “A hole is like `pass`, so the program still runs with a blank.” Correction: a hole blocks compilation. Nothing executes until every hole is filled, which is what makes it safe to sketch.
- “I have to know a function’s signature before I can use it.” Correction: the hole reports the type it expects and lists in-scope names that fit. You fill inward from what the compiler tells you.

- “`let mut` means I should reach for it whenever I loop.” Correction: there is no `for` or `while`. Iterating a collection is `List.fold` and friends. `let mut` is for a local accumulator built from a few explicit steps.
- “`acc <- x` is the same as `acc = x`, just different syntax.” Correction: `<-` reassigns an existing `mut` binding and is visible on purpose. A plain `let` cannot be reassigned at all.
- “Marking something `pure` makes it pure.” Correction: `let pure` is an assertion the compiler checks. If the body performs an effect, it is a compile error. You cannot declare away an effect.

Timing

- 10 min: recap Session 3, then frame the write-with-holes workflow.
- 20 min: demo steps 1 to 3 (typed holes and type-driven development).
- 20 min: demo steps 4 to 5 (deliberate mutation).
- 20 min: demo steps 6 to 8 (inferred effects and `let pure`).
- 15 min: students work the Lesson 9 and Lesson 10 exercises in the playground.
- 5 min: recap and assign the Lesson 11 exercise as homework.

Answer keys: [answer-keys.md](#)

Session 5: Reaching the Python ecosystem

Objectives

By the end of this session, students can:

- Import a real Python callable with `extern` and give it a Pyfun type, using `extern pure` where a call is genuinely side-effect free.
- Decode untrusted JSON into a typed record or a structured error with the `Decode` module.
- Rewrite a staircase of `Result` matches as a flat `result { }` computation expression using `let!` and `return`.
- Tag a number with a unit of measure and let the compiler check dimensions, then explain that units erase at lowering.
- Group definitions into a module and qualify calls as `Module.member`.
- Assemble a small end-to-end pipeline that decodes, folds, and computes a unit-checked result.

Prerequisites

Sessions 1 through 4. The capstone leans on records (Session 3), `Result` and `match` (Session 2), and folds (Session 3), so this session is best after the whole unit.

Six-session variant

If you are running the unit as six sessions, split here. Session 5 becomes lessons 12 and 13 (`extern` and decoding, then computation expressions), and Session 6 becomes lessons 14 through 16 (units, modules, and the capstone), with the capstone as the finale. The demo script below is ordered so the split falls cleanly after step 4.

Demo script

The thread is the boundary versus the engine: Pyfun earns its keep where the outside world is untyped and can fail, and it stays out of the way of fast native code.

1. Open the [Lesson 12 exercise](#). Run `check` and read the `?titleDec` note: it has type `Decoder string` and suggests `Decode.string`. Fill both holes (`Decode.string`, `Decode.int`) and Run. The well-formed object becomes a typed `Book`, and the object missing `pages` becomes `failed (KeyError): 'pages'`.
2. Make the boundary-versus-engine point: the JSON is the untyped, failable edge, and decoding turns it into a `Book` or a structured error before the rest of the program sees it. Contrast with wrapping something like `numpy`, where Pyfun adds little because the speed lives in native code.
3. Open the [Lesson 13 exercise](#). Before filling it, show the staircase form from the top of lesson 13 (two nested `Result` matches) and point out that every `Error` branch does the same thing. Then fill the two holes with `Option.toResult "not a number" (String.toInt a)` and its `b` twin and Run.
4. Make the point that `let!` unwraps an `Ok`, `return` wraps the final value, and the short-circuit on the first `Error` is automatic. The `result { }` block is the staircase written once. (If you are splitting into six sessions, this is the break point.)
5. Open the [Lesson 14 exercise](#). It adds a distance to a time. Run `check` and read the mismatch: `expected float<m>, found float<s>`. Change the `+` to `/` and Run to get `8.0`. Show the Python panel: the units are gone, the output is plain `distance / elapsed`. Units are checked, then erased.
6. Open the [Lesson 15 exercise](#). Wrap the two functions in `module Temp =` and qualify the calls as `Temp.cToF` and `Temp.fToC`. Run. Note that this in-file module runs in the playground, while one-module-per-file projects need the installed compiler.
7. Finish with the capstone. Open the [Lesson 16 exercise](#). Run `check` and let the three hole notes name each type: `Decoder float`, `float<m>`, `float<'a>`. Fill them (`Decode.float`, `1.0<m>`, `time`) and Run to get the one-line report. Point at the join where JSON meets units: a decoded number is a plain `float`, and multiplying by `1.0<m>` lifts it into metres so the fold carries a `float<m>`.
8. Close the unit by naming what the capstone touched: types, matching, records, collections, decoding, computation expressions, units, and modules. Every piece the language showcases, in one small program that fails safely on bad input.

Assigned exercises

- In class: the [Lesson 12 exercise](#) and the [Lesson 13 exercise](#), driven from the demo.
- Homework: the [Lesson 14 exercise](#) and the [Lesson 15 exercise](#).
- Capstone: the [Lesson 16 exercise](#), fill all three holes. Suitable as an in-class finale or a take-home assessment. As a stretch, ask students to feed the decoder a malformed JSON string and confirm the `Error` branch reports rather than crashes.

Common misconceptions

- “`extern` means the Python call is safe now.” Correction: `extern` gives the call a type, but the boundary is effectful by default and can still raise. Wrap a call that can fail in `try`, or decode the data, to turn failure into a value.
- “Decoding JSON is just `json.loads`.” Correction: `json.loads` hands back an untyped shape. `Decode.decodeString` produces your record or a structured error, so the rest of the program never meets an untyped value.
- “`result { }` is a special block that runs differently at runtime.” Correction: it desugars to the same `Result` short-circuit you would write by hand with nested matches. `let!` and `return` are sugar for bind and wrap.
- “Units of measure add runtime overhead to every number.” Correction: units exist only during type checking and erase at lowering. The emitted Python is plain numbers.
- “A module changes how the code runs or performs.” Correction: an in-file module is purely organizational namespacing. Members call each other by bare name inside, and you qualify them as `Module.member` from outside.

Timing

- 10 min: recap the unit so far, then frame boundary versus engine.
- 20 min: demo steps 1 to 2 (extern and JSON decoding).
- 15 min: demo steps 3 to 4 (computation expressions).
- 15 min: demo steps 5 to 6 (units and modules).
- 20 min: demo steps 7 to 8 (the capstone) or students start it themselves.
- 10 min: unit wrap-up and pointers to the full course and playground.

Answer keys: [answer-keys.md](#)

Answer keys

This page is for instructors. It collects the solution and expected output for every exercise in the unit, one section per session. Each solution is the verified answer from the lesson's own collapsed solution block, so it compiles and runs as shown. The linked lesson page holds the full explanation of each answer.

Session 1

Lesson 1: Values and inference

[Lesson page](#)

```
let apples = 4
let oranges = 3
let fruit = apples + oranges
print (f"total fruit: {fruit}")
```

Expected output:

```
total fruit: 7
```

Lesson 2: Functions, currying, and pipes

[Lesson page](#)

```
let double x = x + x
let triple x = x + x + x

let result = 3 |> double |> triple
print result
```

Expected output:

```
18
```

Session 2

Lesson 3: The None problem: Option

[Lesson page](#)

```
let label s =  
  match String.toInt s:  
    case Some n: f"number: {n}"  
    case None: "no number"  
  
print (label "7")  
print (label "nope")
```

Expected output:

```
number: 7  
no number
```

Lesson 4: Errors as values: Result

[Lesson page](#)

```
extern parseInt: string -> int = int  
  
let describe s =  
  match try (parseInt s):  
    case Ok n: f"ok: {n}"  
    case Error e: f"bad: {e.errorKind}"  
  
print (describe "88")  
print (describe "bad")
```

Expected output:

```
ok: 88  
bad: ValueError
```

Lesson 5: Your own types: ADTs and exhaustive match

[Lesson page](#)

```
type Light =  
  | Red  
  | Amber  
  | Green  
  
let action l =  
  match l:  
    case Red: "stop"  
    case Amber: "wait"  
    case Green: "go"  
  
print (action Red)  
print (action Amber)  
print (action Green)
```

Expected output:

```
stop  
wait  
go
```

Session 3

Lesson 6: Records

[Lesson page](#)

```
type Account = { name: string, balance: int }  
  
let opened = Account { name = "Ada", balance = 0 }  
let funded = { opened with balance = 100 }  
  
print funded.name  
print funded.balance  
print opened.balance
```

Expected output:

```
Ada  
100  
0
```

Lesson 7: Lists, sets, and maps

[Lesson page](#)

```
let ns = [4, 8, 15, 16, 23]  
  
let total = List.fold (+) 0 ns  
let atTen = Option.withDefault 0 (List.get 10 ns)  
  
print total  
print atTen
```

Expected output:

```
66  
0
```

Lesson 8: Tuples and destructuring

[Lesson page](#)

```
let names = ["ada", "alan"]  
let scores = [10, 9]  
  
let line pair =  
  match pair:  
    case (name, score): f"{name}: {score}"  
  
let lines = List.zip names scores |> List.map line  
  
lines |> String.join ", " |> print
```

Expected output:

```
ada: 10, alan: 9
```

Session 4

Lesson 9: Typed holes and type-driven development

[Lesson page](#)

```
let scores = Map.add "ada" 10 (Map.add "alan" 9 Map.empty)

let report name =
  Map.tryFind name scores
  |> Option.withDefault 0
  |> String.fromInt

print (report "ada")
print (report "cy")
```

Expected output:

```
10
0
```

Lesson 10: Mutation, on purpose

[Lesson page](#)

```
let balance start =
  let mut acc = start
  acc <- acc + 100
  acc <- acc - 30
  acc <- acc + 50
  acc

print (balance 0)
```

Expected output:

```
120
```

Lesson 11: Effects, inferred

[Lesson page](#)

```
let pure greet name = f"Hello, {name}"  
  
print (greet "ada")
```

Expected output:

```
Hello, ada
```

Session 5

Lesson 12: Talking to Python: extern

[Lesson page](#)

```
type Book = { title: string, pages: int }  
  
let bookDecoder =  
  Decode.map2 (fun title pages -> Book { title = title, pages = pages })  
    (Decode.field "title" Decode.string)  
    (Decode.field "pages" Decode.int)  
  
let describe r =  
  match r:  
  case Ok b: f"{b.title}, {b.pages} pages"  
  case Error e: f"failed ({e.errorKind}): {e.errorMessage}"  
  
let wellFormed = """"{"title": "Dune", "pages": 412}"""  
let missingField = """"{"title": "Dune"}"""  
  
wellFormed |> Decode.decodeString bookDecoder |> describe |> print  
missingField |> Decode.decodeString bookDecoder |> describe |> print
```

Expected output:

```
Dune, 412 pages  
failed (KeyError): 'pages'
```

Lesson 13: Computation expressions

[Lesson page](#)

```
let multiply a b =  
  result {  
    let! x = Option.toResult "not a number" (String.toInt a)  
    let! y = Option.toResult "not a number" (String.toInt b)  
    return x * y  
  }  
  
print (multiply "6" "7")  
print (multiply "6" "nope")
```

Expected output:

```
Ok(42)  
Error('not a number')
```

Lesson 14: Units of measure

[Lesson page](#)

```
measure m  
measure s  
  
let distance = 240.0<m>  
let elapsed = 30.0<s>  
  
# Speed is distance per unit of time.  
let speed = distance / elapsed  
  
print speed
```

Expected output:

```
8.0
```

Lesson 15: Modules and projects

[Lesson page](#)

```

module Temp =
  let cToF c = c * 9 // 5 + 32
  let fToC f = (f - 32) * 5 // 9

  print (Temp.cToF 100)
  print (Temp.fToC 212)

```

Expected output:

```

212
100

```

Lesson 16: Capstone: a typed pipeline

[Lesson page](#)

```

measure m
measure s

type Run = { name: string, distance: float, time: float }

let runDecoder =
  Decode.map3 (fun n d t -> Run { name = n, distance = d, time = t })
    (Decode.field "name" Decode.string)
    (Decode.field "distance" Decode.float)
    (Decode.field "time" Decode.float)

let input = """[
  {"name": "ada", "distance": 100.0, "time": 20.0},
  {"name": "bo", "distance": 140.0, "time": 30.0}
]"""

let report =
  match Decode.decodeString (Decode.list runDecoder) input:
  case Ok runs:
    let names = runs |> List.map (fun r -> r.name) |> String.join ", "
    let dist = runs |> List.fold (fun a r -> a + r.distance * 1.0<m>) 0.0<m>
    let time = runs |> List.fold (fun a r -> a + r.time * 1.0<s>) 0.0<s>
    let avg = dist / time
    f"{names}: average speed {avg} m/s over {List.len runs} runs"
  case Error e: f"could not read input ({e.errorKind})"

print report

```

Expected output:

ada, bo: average speed 4.8 m/s over 2 runs