Where does Coq define booleans and numbers?

In the standard library.

What's a "type" in Coq?

A set of data values.

Make a named assertion that `~true` is `false`, then prove it.

```
Example test_negation:
   (negb true) = false.
Proof. simpl. reflexivity. Qed.
```

Name three ways to check that a function works.

- Use `Eval simpl in (expr)` on a test case and observe the result.
- Use `Example`/`Theorem`/whatever to record expected result, and then prove it. Coq will only accept your proof if it's correct.
- "extract" the function definition to OCaml, Scheme, or Haskell.

How might you create "unit tests"?

Create `Example`s that are nothing but an equation with the expected value on one side and a term built from function applications on the other.

Apply negation to the boolean `true`.

```
Eval simpl in (negb true).
```

How do you fill in a hole in a `Definition`/`Fixpoint'? In an `Example`?

`admit` fills in holes in a `Definitions`/`Fixpoint'.
`Admitted` fills in holes in proofs.

How does Coq write the type of a boolean conjunction function?

```
bool->bool->bool
```

What does the `Check` command do?

It causes Coq to print the type of an expression.

How will we use the module system?

If you put declarations between `Module X` and `End X` then after `End` the definitions are referred to as `X.foo`.

What is an enumerated type?

It's a type (a set of data values) whose members are fully enumerated in in the type's definition.

When we use `Inductive` to define a type, we should see it as what?

A set of *expressions*, inductively defined. The definition tells us exactly how members of the type can be constructed, and excludes all other expressions.

What "magic" does Coq provide for natural numbers?

The ability to use numerals instead of tediously constructing numbers with the `O` and `S` constructors.

What is the fundamental difference between a data constructor and functions?

Book: Functions come with *computation rules*. Data constructors have no behavior attached.
Me: The application of constructors are values **as written**. The application of functions are **never** values as written, they are terms which must be evaluated.

Name some keywords that can introduce a function.

- `Definition`
- `Fixpoint` in case of recursion

What kind of recursion does Coq allow?

Structural (or *primitive*) recursion. That means recursive calls must be on strictly smaller values, guaranteeing termination.

What notational convenience does Coq provide for multiple parameters of the same type?

The following are equivalent:

```
(n m : nat)
(n: nat) (m: nat)
```

How does one match on *multiple* expressions?

A comma is placed between them in the scrutinee and between the two sides of each matching pattern.

What is an underscore in the context of `match` expressions?

It is a *wildcard pattern*, matching any expression without giving that expression a name.

How is "language support" introduced for some definitions?

Name two kinds of language support available.

With `Notation` constructions which also define associativity and precedence.

- Numerals
- Operators
- Collections syntax

How can one choose between multiple notation interpretations for an expression.

expression%notation_scope

Which tactic is like `simpl` "on steroids"?

compute

The `reflexivity` tactic implicitly does what?

What's the difference between the simplification of `simpl` and that of `reflexivity`?

Simplifies both sides before testing (including by using `simpl`).

Among other things, `reflexivity` may expand definitions. `simpl` never will.

What does the `intros` tactic do?

For a conditional it introduces the antecedent as an assumption. For a universally quantified statement it introduces an arbitrary element of the domain and discharges the quantifier.

What is the syntax of `intros`?

The keyword `intros` followed by a space-delimited list of names for the assumptions. These may be names of variables already in context, or they may be ones you're *introducing*. The names are interpreted in the order the relevant expressions appear in the current context.

Describe the `rewrite` tactic.

What does rewriting *left-to-write* mean?

It rewrites the current goal using the provided rule and in the provided direction.

For example:

```
rewrite -> H
```

Left-to-write means rewriting the terms in the subgoal that match the left-hand-side of the rule being used.

How are are propositions with multiple hypothesis written?

```
Hyp1 -> ... -> HypN -> Conclusion
```

Why doesn't simplification be used to prove all theorems?

Unknown values may appear as arguments to functions, preventing simplification.

Give the syntax of the `destruct` tactic.

```
destruct var as [pattern].
```

`as [pattern]` is optional.

The pattern consists of names for the data of the possible data constructors of `var` separated by `|`.
For a nullary constructor just put the pipe.

Why don't we say `destruct b as [true | false].`?

Remember, the `as` pattern in a `destruct`/`induction` is for the **data** associated with a constructor. Nullary constructors (*values*) have none.
So you would write either of these two:

```
destruct b as [|].
destruct b.
```

`destruct` proves a theorem about an enumerated type for each possible ...

... constructor used to create that type.

Don't confused `case` with ...

... our hack `Case` and `SCase`.

What is the syntax of the `induction` tactic?

Just the same as the `destruct` tactic.

How can you create sub-theorems without creating a new top-level name?

Use the `assert` tactic.

```
assert (H: whatever).
  Case "Proof of assertion". whatever.
```

What is a common non-stylistic reason for using `assert`?

Coq is choosing the wrong instance of a pattern to rewrite when you use the `rewrite` tactic.
In this case you can prove as a sub-theorem exactly the rewrite you want, and then use `rewrite` in terms of this sub-theorem.