In the standard library.

A set of data values.

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

- 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.

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.

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

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

`Admitted` fills in holes in proofs.

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

It causes Coq to print the type of an expression.

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

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

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

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

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.

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

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

The following are equivalent:

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

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

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

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

- Numerals
- Operators
- Collections syntax

```
expression%notation_scope
```

- nat
- Z
- type_scope

- `simpl`
- `compute` - results in possibly larger terms

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

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

`reflexivity` ends the current goal so it doesn't matter if the resulting term is horribly large and unwieldly.

- Example
- Lemma
- Fact
- Remark

The list of current assumptions that can be used in proving the goal.

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

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.

- There may be conditions to the proposition being proved. Without first using `intros` to discharge the quantifiers you can't use `intros` to introduce the hypotheses into context.
- Without eliminating quantifiers you cannot use (by `rewrite`) theorems of the universal quantification form, since they operate on free variables.

It rewrites the current goal using the provided rule (in context or previously defined) and in the provided direction.

For example:

```
rewrite -> H
```

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

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

Unknown values may appear as arguments to functions, preventing simplification.
For example, given an arbitrary `n:nat` we can't simplify since we don't know which constructor applies.

```
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.

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.
```

... constructor used to create that type.

They add a string to the context of the current subgoal and is discharged when the current subgoal is proved.

... `case`.

Just the same as the `destruct` tactic.

Inductive Hypothesis for `n'`

- S n' + m = S (n' + m)
- S (n + m) = n + (S m)

Use the `assert` tactic.

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

After the proof is done `H` will be added to context.

You want to use the `rewrite` tactic on an instance of the pattern that is not outermost.

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.

`replace (t) with (u)`
replaces (all copies of) expression `t` in the goal with expression `u` and generates `t=u` as an additional subgoal.

It's often used when `rewrite` acts on the wrong part of the goal.