A *validator* is a process that checks whether a UTXO transaction may spend funds. We can define a validator by writing a function of type

$$Data \to Data \to Data \to ()$$

The first parameter is the *datum*. The datum will contain state information about the smart contract. The second parameter is the *redeemer*. The redeemer can represent the action a wallet wishes to perform. The third and final parameter is the *script context*. The context holds information about the current transaction. If the transaction is valid then the value () is returned, otherwise the validator will raise an exception. Below is arguably the simplest possible validator one can write. It simply ignores its arguments and returns ().

`Validator0.hs`∋:

```
{-# INLINABLE always #-}
always :: Data → Data → Data → ()
always _ _ _ = ()
```

0.1

Let us now look at a validator that always fails.

`Validator0.hs`∋:

```
{-# INLINABLE never #-}
never :: Data → Data → Data → ()
never _ _ _ = traceError ”Never!”
```

0.2

Of course, in reality our validators will need to inspect their arguments in order to decide whether to succeed or not. The type Data defined in the module PlutusTx and has the following constructors:

- *B*, which takes a bytestring;
- *I*, which takes an integer;
- *List*, which takes a list of **Data** values;

- *Map*, which takes a map of key-value pairs, where both the key and the value are **Data** values;
- *Constr*, which takes an integer and a list of **Data** values.

Let us write a validator that will succeed if its datum argument is the integer 0.

Validator0.hs∋:

```
{-# INLINABLE zero #-}
zero :: Data → Data → Data → ()
zero datum _ _
  | datum == I 0 = ()
  | otherwise = traceError ”datum is not the integer 0!”
```

0.3

While we could write our validators in the manner described above, it is more convenient to develop our validators as functions of the type

$$Datum \to Redeemer \to \textbf{ScriptContext} \to \textbf{Bool}$$

where *Datum* and *Redeemer* are programmer-defined types while **ScriptContext** is, found in the Plutus) module, is a record with two fields:

- *scriptContextTxInfo* :: **TxInfo**, which is the validator's view of the pending transaction;
- *scriptContextPurpose* :: **ScriptPurpose**, which is the purpose of the currently running script, be it minting, spending, rewarding or certification.

We will look at *scriptContextTxInfo* in more detail later, but for now let us use *scriptContextPurpose* to write a validator that will accept spending scripts only.

⟨**Simple validators**⟩∋:

```
{-# INLINABLE spend #-}
spend :: () → () → ScriptContext → Bool
spend _ _ ctx  = case scriptContextPurpose ctx of
  Spending _ → True
  otherwise  → False
```

0.4

What might a redeemer look like? Remember, a redeemer can represent the action that wallet wishes to perform. The only constraint on the type of the redeemer is that it is an instance of the IsData class. A number of basic types already have IsData instances, but if one does not exist for your type, it is not too hard to obtain one. We will look at that a later, for now let us use a type which we know already has an instance.

⟨**Simple validators**⟩∋:

```
{-# INLINABLE feed #-}
feed :: () → ByteString → ScriptContext → Bool
feed () bytes ctx  = bytes == ”FEED”
```

0.5

For the sake of completion, let's look at a validator that uses the datum to check that a script is executes after a certain time. We make use of the **Slot** type which resides in the Plutus.V1.Ledger.Slot module.

⟨**Simple validators**⟩∋:

```
{-# INLINABLE vested #-}
vested :: Slot → () → ScriptContext → Bool
spend t () ctx  = traceIfFalse ”Too early” checkDeadline
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    checkDeadline :: Bool
    from t ‘contains‘ txInfoValidRange info
```

0.6

A validator is what is known as "on-chain" code, so we must compile the validator before deploying it. Suppose we have written a validator function *f*. If *f* is a low-level validator, where the datum, the redeemer and the script context are all of type **Data**, then we may compile it as follows:

```
$$(PlutusTx.compile [|| f ||]
```

However, suppose f has type **Slot** → **ByteString** → **ScriptContext** → **Bool**. Then we need to do a bit more work:

⟨Simple validators⟩∋:

```
 data Typed
 instance Scripts.ScriptType Typed where
  type instance DatumType Typed = Slot
  type instance RedeemerType Typed = ByteString

inst :: Scripts.ScriptInstance Typed
inst = Scripts.validator @Typed
  ££(PlutusTx.compile [|| f ||])
  ££(PlutusTx.compile [|| wrap ||])
  where
    wrap = Scripts.wrapValidator @Slot @ByteString

validator :: Validator
validator = Scripts.validatorScript inst
```