Hemos vistos que los operadores aritméticos +, -, *, / como, además de las funciones div/2 y rem/2 para la división entera y el resto.

También vimos ++ y — para manipular listas:

```
iex> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex> [1, 2, 3] -- [2]
[1, 3]
```

La concatenación de cadenas se realiza con <>:

```
iex> "foo" <> "bar"
"foobar"
```

Aun que también es muy común el uso de:

```
iex> [foo, bar] = ["foo", "bar"]
["foo", "bar"]
iex> "#{foo}#{bar}"
"foobar"
```

Hay tres operadores booleanos: or, and y not. Estos operadores son estrictos en el sentido de que esperan algo que se evalúe como booleano (verdadero o falso) como primer argumento:

```
iex> true and true
true
iex> false or is_atom(:example)
true
```

Proporcionar un valor no booleano generará una excepción:

```
iex> 1 and true
** (BadBooleanError) expected a boolean on left-side of "and", got: 1
```

Or y and son operadores de cortocircuito. Solo ejecutan el lado derecho si el lado izquierdo no es suficiente para determinar el resultado:

```
iex> false and raise("This error will never be raised")
false
iex> true or raise("This error will never be raised")
true
```

Además de estos operadores booleanos, Elixir también proporciona ||, && y! que aceptan argumentos de cualquier tipo. Para estos operadores, todos los valores, excepto falso y nulo, se evaluarán como verdaderos:

```
# or
iex> 1 || true
1
iex> false || 11
11

# and
iex> nil && 13
nil
iex> true && 17
17

# !
iex> !true
false
iex> !1
false
iex> !nil
true
```

Como regla general, use y, o no y cuando esté esperando booleanos. Si alguno de los argumentos no es booleano, use &&, || y!

También podemos usar ==,! =, ===,! ==, <=,> =, como operadores de comparación:

```
iex> 1 == 1
true
iex> 1 != 2
```

```
true
iex> 1 < 2
true
```

La diferencia entre == y === es que este último es más estricto:

```
iex> 1 == 1.0
true
iex> 1 === 1.0
false
```

También podemos comparar dos tipos de datos diferentes:

```
iex> 1 < :atom
true
```

La razón por la que podemos comparar diferentes tipos de datos es el pragmatismo. Los algoritmos de clasificación no necesitan preocuparse por los diferentes tipos de datos para ordenar. El orden de clasificación general se define a continuación:

```
number < atom < reference < function < port < pid < tuple < map < list < bitstring
```

En realidad no necesita memorizar este pedido; es suficiente saber que existe este orden.