Data-driven Unit Tests are an easy and efficient way to run a test with a range of input data. That can be useful in situations like

* testing against a broader range of values
* testing for null and undefined values
* testing against edge cases, e.g., for dates (beginning of the month, end of the month, …)
* or to tweak your unit test stats - because each input row counts as a separate test. Your manager will be impressed. 😉

**Simple sample**

Let’s define a simple test to check if a number is even:

**it('should be even', () => {**

**expect(2 % 2).toBe(0);**

**});**

✓ is even

To check a series of numbers, we can use Jest’s it.each (or test.each, since it is just an alias of test). We pass the input data as an array, each item is then tested individually (n).

**it.each([2,4,6])('should be even', (n) => {**

**expect(n % 2).toBe(0);**

**});**

✓ should be even

✓ should be even

✓ should be even

We can use [several predefined](https://jestjs.io/docs/en/api#testeachtable-name-fn-timeout) tokens to improve the name of the test, which makes it easier to locate a failing test:

it.each([2,4,5,6])('%i should be even', (n) => {

expect(n % 2).toBe(0);

});

✓ 2 should be even

✓ 4 should be even

✕ 5 should be even

✓ 6 should be even

**Multiple parameters**

You can pass multiple parameters to your test as well. In the next sample, we pass the expected value to the test as well:

**it.each([[2,4], [6,36], [8,64]])('can square %i to %s', (n, expected) => {**

**expect(n\*n).toBe(expected);**

**});**

✓ can square 2 to 4

✓ can square 6 to 36

✓ can square 8 to 64

The input table is defined as an array of arrays. Each inner array is one row of the input table.

I like tests to be well readable, understandable and clear in their intention. The focus should be on the actual test and not on the implementation. Jest provides a syntax that supports exactly that:

it.each`

n | expected

**${1} | ${1}**

**${2} | ${4}**

**${4} | ${16}**

**${12} | ${144}**

**`('can square $n to $expected', ({ n, expected }) => {**

**expect(n\*n).toBe(expected);**

**});**

✓ can square 1 to 1

✓ can square 2 to 4

✓ can square 4 to 16

✓ can square 12 to 144

The input table now looks like an actual table. It is defined as a template string, in which the first row contains the variables names separated by a pipe |. Each subsequent row is one combination of input data to run the test against, defined as a template literal expression.  
You can reference the variable names inside the test name with $variable.

**Run multiple tests**

.each is also available for describe-Blocks, which makes it easy to run multiple tests or even a whole test suite against a set of input data.

**describe.each`**

**x | y**

**${10} | ${2}**

**${9} | ${3}**

**${8} | ${4}**

**`('With x=$x and y=$y', ({x, y}) => {**

**it('x should be larger then y', () => {**

**expect(x).toBeGreaterThan(y);**

**});**

**it('should x should by dividable by y without rest', () => {**

**expect(x % y).toBe(0);**

**});**

**});**

With x=10 and y=2

✓ x should be larger then y

✓ should x should by dividable by y without rest

With x=9 and y=3

✓ x should be larger then y (1ms)

✓ should x should by dividable by y without rest

With x=8 and y=4

✓ x should be larger then y

✓ should x should by dividable by y without rest (1ms)

You can even combine describe.each and it.each, but keep an eye on comprehensibility.