I've just sold myself to the gods of click-baiting by making an "x things you didn't know about y" post. But hey, at least there no subtitle that says "number three will blow your mind!"

Jokes aside the items in this list are concepts that I usually see beginners struggling with. At the same time, learning these concepts will vastyly improve your testing game. I know it did with mine.

1. Everything is a DOM node

This is usually the first misconception that beginners have when they start approaching Testing Library. It is especially true for those developers like me that came from Enzyme.

When you work with Testing Library you are dealing with DOM nodes. This is made clear by the first Guiding Principle:

> If it relates to rendering components, then it should deal with DOM nodes rather than component instances, and it should not encourage dealing with component instances.

If you want to check for yourself, it's as simple as this:

```
import React from "react";
import { render, screen } from "@testing-library/react";

test("everything is a node", () => {
  const Foo = () => <div>Hello</div>;
  render(<Foo />);
  expect(screen.getByText("Hello")).toBeInstanceOf(Node);
});
```

Once you realize that you're dealing with DOM nodes you can start taking advantage of all the DOM APIs like `querySelector` or `closest`:

```
import React from "react";
import { render, screen } from "@testing-library/react";

test("the button has type of reset", () => {
  const ResetButton = () => (
    <button type="reset">
      <div>Reset</div>
    </button>
  );
  render(<ResetButton />);
  const node = screen.getByText("Reset");
```

```
});
```

2. debug's optional parameter

Since we now know that we're dealing with a DOM structure, it would be helpful to be able to "see" it. This is what debug is meant for:

```
const { debug } = render(<MyComponent />);
debug();
```

Sometimes thought debug's output can be very long and difficult to navigate. In those cases, you might want to isolate a subtree of your whole structure. You can do this easily by passing a node to debug:

```
const { debug } = render(<MyComponent />);
const button = screen.getByText("Click me").closest();
debug(button);
```

3. Restrict your queries with `within`

Imagine you're testing a component that renders this structure:

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Fruit</th>
```

```
      <td>1</td>
      <td>Apples</td>
    </tr>
    <tr>
      <td>2</td>
      <td>Oranges</td>
    </tr>
    <tr>
      <td>3</td>
      <td>Apples</td>
    </tr>
  </tbody>
</table>
```

You want to test that each ID gets its correct value. You can't use `getByText('Apples')` because there are two nodes with that value. Even if that wasn't the case you have no guarantee that the text is in the correct row.

What you want to do is to run `getByText` only inside the row you're considering at the moment. This is exactly what `within` is for:

```
import React from "react";
import { render, screen, within } from "@testing-library/react"; //
import "jest-dom/extend-expect";

test("the values are in the table", () => {
  const MyTable = ({ values }) => (
    <table>
      <thead>
        <tr>
          <th>ID</th>
          <th>Fruits</th>
        </tr>
```

```
            <td>{id}</td>
            <td>{fruit}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
  const values = [
    ["1", "Apples"],
    ["2", "Oranges"],
    ["3", "Apples"],
  ];
  render(<MyTable values={values} />);

  values.forEach(([id, fruit]) => {
    const row = screen.getByText(id).closest("tr");
    // highlight-start
    const utils = within(row);
    expect(utils.getByText(id)).toBeInTheDocument();
    expect(utils.getByText(fruit)).toBeInTheDocument();
    // highlight-end
  });
});
```

4. Queries accept functions too

You have probably seen an error like this one:

```
Unable to find an element with the text: Hello world.
This could be because the text is broken up by multiple elements.
In this case, you can provide a function for your text
matcher to make your matcher more flexible.
```

```
<div>Hello <span>world</span></div>
```

The solution is contained inside the error message: "[...] you can provide a function for your text matcher [...]".

What's that all about? It turns out matchers accept strings, regular expressions or functions.

The function gets called for each node you're rendering. It receives two arguments: the node's content and the node itself. All you have to do is to return `true` or `false` depending on if the node is the one you want.

An example will clarify it:

```
import { render, screen, within } from "@testing-library/react";
import "jest-dom/extend-expect";

test("pass functions to matchers", () => {
  const Hello = () => (
    <div>
      Hello <span>world</span>
    </div>
  );
  render(<Hello />);

  // These won't match
  // getByText("Hello world");
  // getByText(/Hello world/);

  screen.getByText((content, node) => {
    const hasText = (node) => node.textContent === "Hello world";
    const nodeHasText = hasText(node);
    const childrenDontHaveText = Array.from(node.children).every(
      (child) => !hasText(child)
```

```
});
```

We're ignoring the `content` argument because in this case, it will either be "Hello", "world" or an empty string.

What we are checking instead is that the current node has the right `textContent`. `hasText` is a little helper function to do that. I declared it to keep things clean.

That's not all though. Our `div` is not the only node with the text we're looking for. For example, `body` in this case has the same text. To avoid returning more nodes than needed we are making sure that none of the children has the same text as its parent. In this way we're making sure that the node we're returning is the smallest—in other words the one closes to the bottom of our DOM tree.

5. You can simulate browsers events with `user-event`

Ok, this one is a shameless plug since I'm the author of `user-event`. Still, people—myself included—find it useful. Maybe you will too.

All `user-event` tries to do is to simulate the events a real user would do while interacting with your application. What does it mean? Imagine you have an `input` field, and in your tests, you want to enter some text in it. You would probably do something like this:

```
fireEvent.change(input, { target: { value: "Hello world" } });
```

events (`blur`, `focus`, `mouseEnter`, `keyDown`, `keyUp`...). `user-event` simulates all those events for you:

```
import userEvent from "@testing-library/user-event";

userEvent.type(input, "Hello world");
```

Check out the README to see the available APIs and feel free to propose new ones too.

:q!<Enter>