# Understanding Errors

## The anatomy of an error

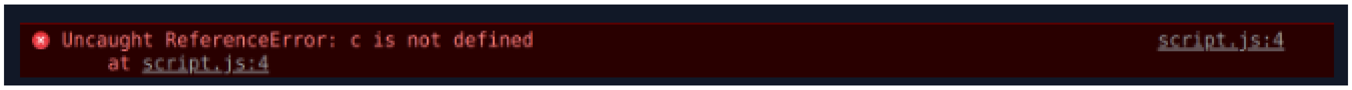
An error is a type of object built into the JS language, consisting of a name/type and a message. Errors contain crucial information that can assist you in locating the code responsible for the error, determining why you have this error, and resolving the error.

Let's say you have the following code:

```
const a = "Hello";
const b = "World";

console.log(c);
```

The code will run, but it will generate an error. This is called "throwing" an error. The first part of an error displays the type of error. This provides the first clue as to what you're dealing with. In this example, we have a `ReferenceError`.



```
Uncaught ReferenceError: c is not defined                                    script.js:4
    at script.js:4
```

A `ReferenceError` is thrown when one refers to a variable that is not declared and/or initialized within the current scope. This has occurred because `c is not defined`.

Different errors of this type have different messages based on what is causing the `ReferenceError`. For example, another message you may run into is `ReferenceError: can't access lexical declaration 'X' before initialization`.

This points to a completely different reason than our original `Reference Error` above. Understanding both the error type and the error message is crucial to comprehending why you are receiving the error.

The next part of an error gives us the name of the file in which you can find the error (here `script.js`), and also the line number.

This allows you to easily navigate to the problematic line in your code. The file/line appears as a link that navigates to the exact line of code and the reset of your script in the Sources tab of the Developer Tools.

Sometimes the browser also displays the column (or character) in the error line. In this example, this occurs `at script.js:4:13`.

Another important part of an error is the `stack trace`. This helps you understand when the error was thrown in your application, and what functions were called that led up to the error. For example:

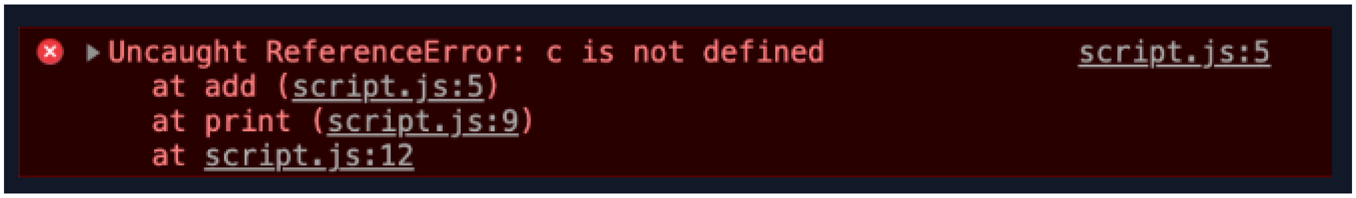
```
const a = 5;
const b = 10;

function add() {
  return c;
}

function print() {
  add();
}

print();
```

The `print()` should call on `add()`, which returns `c`, which currently has not been declared. The error is:



```
▸Uncaught ReferenceError: c is not defined                script.js:5
    at add (script.js:5)
    at print (script.js:9)
    at script.js:12
```

The stack trace tells us that:

1. `c is not defined` in the scope of `add()`, which is declared on line 5.
2. `add()` was called by `print()`, which was declared on line 9.
3. `print()` itself was called on line 12.

Thus the stack trace lets you trace the evolution of an error back to its origin, which here is the declaration of `add()`.

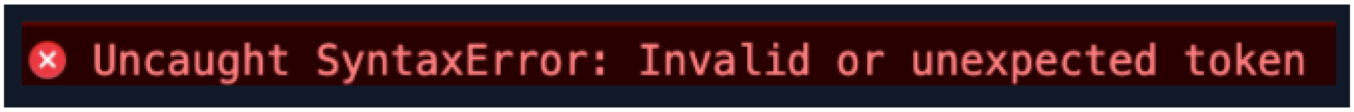
## Common types of errors

### Syntax error

A syntax error occurs when the code you are trying to run is not written correctly, i.e., not in accordance with the grammatical rules of JavaScript. For example:

```
function helloWorld() {
  console.log "Hello World!"
}
```

will throw the following error, because we forgot the parentheses for `console.log()`!



```
Uncaught SyntaxError: Invalid or unexpected token
```

## Reference error

Encountered before; the variable you are trying to reference does not exist (within the current scope) - or it has been spelled incorrectly!

## Type error

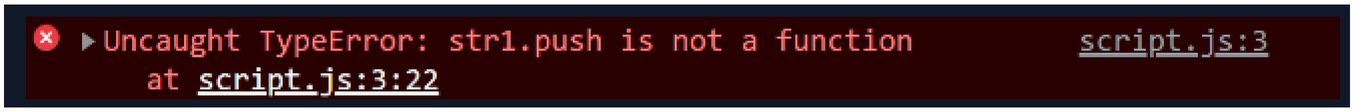
These are thrown for a few different reasons:

A `TypeError` may be thrown when:

- an operand or argument passed to a function is incompatible with the type expected by that operator or function;
- or when attempting to modify a value that cannot be changed;
- or when attempting to use a value in an inappropriate way

Here we try to combine strings to create one message:

```
const str1 = "Hello";
const str2 = "World!";
const message = str1.push(str2);
```



```
▶Uncaught TypeError: str1.push is not a function        script.js:3
    at script.js:3:22
```

Here we get a `TypeError` with a message stating that `str1.push is not a function`. You might know that `.push()` is certainly a function (for example, if you have used it to add items to *arrays* before).

But that's the point - `.push()` is not a String method, it's an Array method. Hence, it is "not a function" that you can find as a String method. If we change `.push()` to `.concat()`, a proper String method, our code runs as intended!

## Tips for resolving errors

1. Understand that the error message is your friend - not your enemy. Error messages tell you *exactly* what is wrong with your code, and which lines to examine to find the source of the error. Without error messages it would be a *nightmare* to debug our code - because it would still not work, we just wouldn't know why!
2. Google the error! Chances are, you can find a fix or explanation on StackOverflow or in the documentation. If nothing else, you will receive more clarity as to why you are receiving this error.
3. Use the debugger! The debugger is great for more involved troubleshooting, and is a critical tool for a developer. You can set breakpoints, view the value of any given variable at any point in your application's execution, step through the code line by line, and more!
4. Make use of the console! `console.log()` is a popular choice for quick debugging. For more involved troubleshooting, using the debugger might be more appropriate, but using `console.log()` is great for getting immediate feedback without needing to step through your functions. There are also other useful methods such as `console.table()`, `console.trace()`, and more!

## Errors vs. warnings

Many people are met with warnings and treat them as errors. Errors will stop the execution of your program or whatever process you may be attempting to run and prevent further action. Warnings, on the other hand, are messages that provide you insight on potential problems that may not necessarily crash your program at runtime, or at all!

While you should address these warnings if possible and as soon as possible, warnings are not as significant as errors and are more likely to be informational. Warnings are typically shown in yellow, while errors are typically shown in red.

## Assignment

### ReferenceError

Catching a Reference Error:

```
try {
  let a = undefinedVariable;
} catch (e) {
  console.log(e instanceof ReferenceError); // true
  console.log(e.message); // "undefinedVariable is not defined"
  console.log(e.name); // "ReferenceError"
  console.log(e.stack); // Stack of the error
}
```

Creating a ReferenceError:

```
try {
  throw new ReferenceError("Hello");
} catch (e) {
  console.log(e instanceof ReferenceError); // true
  console.log(e.message); // "Hello"
  console.log(e.name); // "ReferenceError"
  console.log(e.stack); // Stack of the error
}
```

Other errors are worked with similarly.

## What went wrong? Troubleshooting JavaScript

Errors will only show up when they are run (because JavaScript is interpreted like Python)

Example of a string error:

`SyntaxError: expected expression, got 'string' or SyntaxError: string literal contains an unescaped line break`