

Functions, function expression, arrow function

Function Declaration

To declare a function follow the syntax:

```
function function_name(parameter1, parameter2, ..., parameterN) {  
  // Body of the function  
}
```

The function can have access to outer variable, as well as modifying it.

The outer variable is only used if there is no local one, if there is a same-named variable that is declared inside the function then it shadows the outer one. The outer one is ignored.

JavaScript's function parameter are passed by value, meaning a copy of the argument is copied into the parameter.

Default values

If you call a function without providing the arguments required, then the corresponding value for those parameters becomes `undefined`.

You can provide default values in the function declaration if the argument is not passed for that parameter then it will use the default values. It is also used if the argument is specified but is equal to `undefined`.

```
function foo(from, text="hello world") {  
  console.log(from + ": " + text);  
}
```

Call `foo("Ann")` will result in `"Ann: hello world"`

Call `foo("Ann", undefined)` will also result in `"Ann: hello world"`

Return Value

A function without `return` or without returning a explicit value returns `undefined`, just like how Python returns `None`.

Function expressions

Another way of creating a function is via function expressions. It let you create a new function in the middle of any expression:

```
let foo = function() {  
  console.log("foo!");  
};  
  
foo(); // Print out foo!
```

The function creation occurs in the middle of a assignment expression thus it is a function expression!

Function expression should have a semicolon at the end because it is an assignment statement, which is good practice.

Functions in JavaScript are higher order, meaning that you can pass a function as an argument and return a function as a return value.

Anonymous functions

If you just write a function expression without storing it into a variable, then that is an anonymous function. It is only accessible to the context that it is passed into, and not anywhere else.

```
function run_callback(callback) {  
  callback();  
}  
  
run_callback(function() { console.log("hi im callback!"); });
```

Function declaration vs function expression

A function expression is created when the execution reaches it and is usable only from that moment and onward.

On the other hand, a function declaration can be called earlier than it is defined, and it will still work.

```
foo(); // fooing!

function foo() {
  console.log("fooing!");
}
```

Global function declaration is visible in the entire script, no matter where it is.

```
foo(); // error

let foo = function() {
  console.log("fooing!");
}
```

However, function expression are created when execution reaches them, which means line number 1 wouldn't know what foo is at that point.

Arrow function

A much more concise way of creating a function compared to function expression is via arrow functions.

Here is how one looks in function expression vs arrow function:

```
let func = (arg1, arg2, ..., argN) => expression;
```

This arrow function accepts `n` arguments and then evaluates the `expression` on the right side and then return the result.

```
let func = function(arg1, arg2) {
  return arg1 + arg2;
}

let func = (arg1, arg2) => arg1 + arg2;
```

In this case, both `func` does the same thing, but the arrow function on line 5 is much more concise in that it will evaluate `arg1 + arg2` and then returns it without you having to specify a return.

One argument

If the arrow function you are writing only have one argument then you can skip the parentheses around the parameter. But having it makes it much more readable.

```
let double = n => n * 2;
```

No argument

If the function takes no argument, the empty parentheses must be present.

```
let foo = () => console.log("fooing!");
```

Function expression & arrow function

You can use them the same way, for example, to create anonymous callback functions:

```
function do_callback(callback) {  
  callback();  
}  
  
do_callback(() => { console.log("doing callback") }); // Using arrow function  
  
do_callback(function() { console.log("doing callback") }); // using function expression
```

Multi-line arrow functions

If your arrow function's logic is longer than one line, then you can use multi-line arrow functions:

```
let sum = (a, b) => {  
  let result = a + b;  
  return result;  
}
```

This is still a arrow function but you can do multiple lines now. However, by adding brackets to denote multi-line arrow function you now must provide an explicit return statement. Otherwise, the arrow function will just return `undefined` just like a regular function.

Unless you don't need the multi-line arrow function to return a value, then you wouldn't need the `return` statement.

Revision #2

Created 18 December 2022 19:32:25 by Tamarine

Updated 18 December 2022 22:27:30 by Tamarine