

## 4 Scripting in Analytics Designer

### 4.1 Why Scripting?

You might be wondering why you would want to script and what advantage it could possibly be.

Most modern analytics tools avoid scripting to simplify the designer's tasks. Users may find it easier to use at first, but they quickly find themselves limited to the scenarios built into the tool.

Scripting allows you to go beyond present narratives, to respond to user interaction in a custom way, to change data result sets, and to dynamically alter layout. Scripting frees your creativity.

### 4.2 Scripting Language Overview

The analytics designer scripting language is a **limited subset** of JavaScript. It's extended with a logical type system at design time enforcing type safety. Being a true JavaScript subset allows executing it in browser natively. All scripts are run and validated against strict mode. Some more advanced JavaScript features are hidden. Scripts are either tied to events or global script objects.

#### 4.2.1 Type System

The logical type system runs on top of plain JavaScript. It enforces strict types to enable more powerful tooling. The behavior at runtime doesn't change as it's still plain JavaScript.

#### 4.2.2 Tooling – Code Completion and Value Help

The analytics designer scripting framework exposes analytics data and metadata during script creation and editing. This enables

- Code completion in the traditional sense like completing local or global Identifiers
- Semantic code completion by suggesting member functions or similar
- Value help in the form of context-aware value proposals like measures of a data source for function parameters

For example, when calling a script API method on an SAP BW data source, the code completion can propose measures as code completion options or values to specify a filter.

#### 4.2.3 Events

Scripts always run in response to something happening in the application. Application events are your hook. There are several types of events in analytic applications. Some happen in the application itself and some happen on individual widgets.

##### 4.2.3.1 Application Events

The application has two events: one that fires when the app starts, and another that's triggered in certain embedded scenarios.

- **onInitialization**: This event runs once when the application is instantiated by a user. It's where you script anything that you want to be done during startup. Like most events, it has no input parameters.

- **onPostMessageRecieved**: If your application is embedded in an iFrame, your analytic application can communicate bidirectionally with the host web app using JavaScript `PostMessage` (see also <https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage>) calls. It allows the host application to pass information into the analytic application. This event is called whenever the host application makes a post message call into the analytic application.

Designers have access to this information and to the event's two input parameters:

- **origin**: It's the domain of the host application. The contents of an iFrame don't need to be in the same origin as the host app, even when same origin policies are in effect. It can be convenient but be careful about clickjacking attacks and malicious iFrame hosts. For the sake of security, we recommend checking this parameter to ensure that the iFrame host is what you expect.
- **message**: It's the standard message parameter of the JavaScript `PostMessage` passed into SAP Analytics Cloud. It doesn't follow any format and could be almost anything. It's encoded using the structured clone algorithm and there are a few documented restrictions in what can and can't be encoded.

#### 4.2.3.2 Individual Widget Events

Most widgets have an event that's fired when the widget is clicked by a user. However, some widgets have no events, such as text labels. Data-bound widgets generally have an event that's fired when the result set of the data source changes.

Most events have no input parameters, like `onSelect` and `onResultChanged`.

#### 4.2.4 Global Script Objects

Global script objects act as containers. They allow you to maintain and organize script functions that aren't tied to any event and are invoked directly. You can maintain libraries of re-usable functions. These library scripts are called functions.

#### 4.2.5 Accessing Objects

You can access every object in the *Outline* such as widgets, script variables, or script objects by its name when you're working on a script.

#### 4.2.6 Script Variables

By referencing Script Variabl in Calculated Measure, users can easily build a what-if simulation with query results.

For example, you, as an analytic application developer, can bind a calculated measure which references one script variable (`ScriptVariable_Rate`) to a chart.

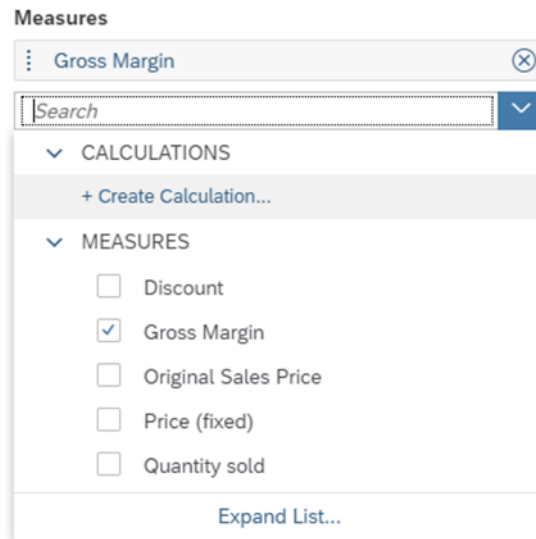


Figure 22: Create Calculation

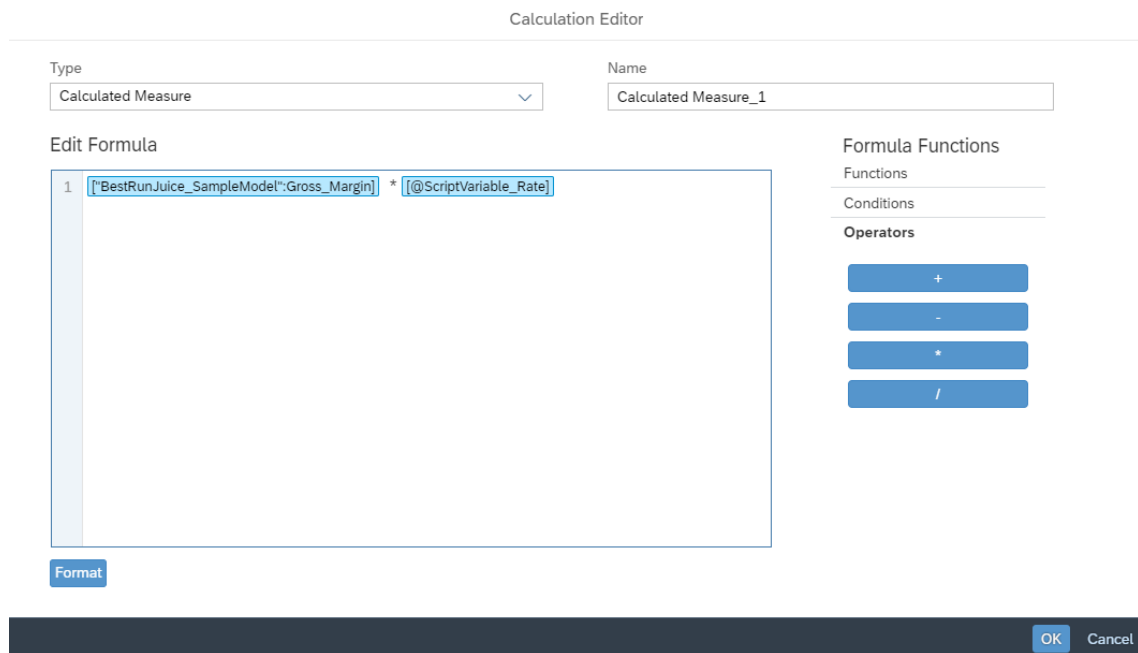


Figure 23: Reference Script Variable

### 4.3 Script Editor

The script editor is a tool within analytics designer to specify the actions taking place when an event is triggered by an application user. By adding a script to a widget, you can influence the behavior of this widget and thus enable user interaction, also referred to as events, at runtime. A script typically consists of several statements. A statement is a programmatic instruction within a script. The execution of a statement is typically triggered by user interaction with the widget.

### 4.3.1 Creating and Editing Event-Based Scripts

Scripts are presented in the *Outline*, at the left-hand side of the analytics designer editor environment.

Find them by hovering over the widget name in the *Outline*, or as a menu entry in the quick action menu of each widget. The *fx* icon indicates the event. By clicking on it, the script editor opens the selected function.

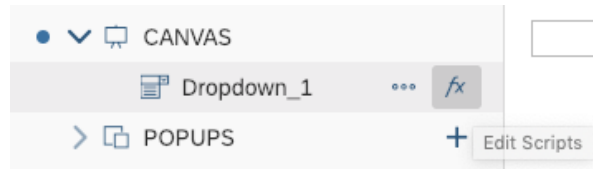


Figure 24: Edit Scripts

If a widget has multiple available events, you're presented with a choice in the hover menu.

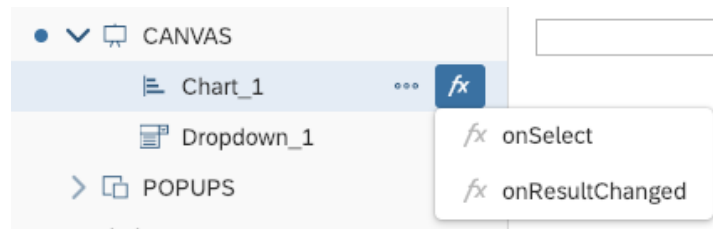


Figure 25: Multiple Events

If there is an event with an attached script, you can see the *fx* icon in the *Outline*. If there are no attached script, there is no visible icon. In the following figure, the *onSelect* event of *Dropdown\_1* has a script, but there are no scripts attached to *Chart\_1*.

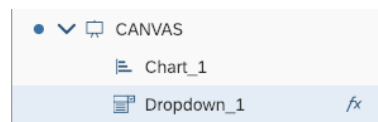


Figure 26: Script for Dropdown

If a widget has multiple events and at least one has a script attached, then the *fx* icon will be displayed.

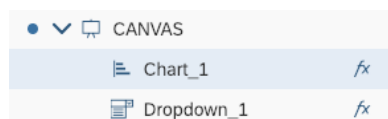


Figure 27: Script for Chart

The hover menu will show which of the events have attached scripts.

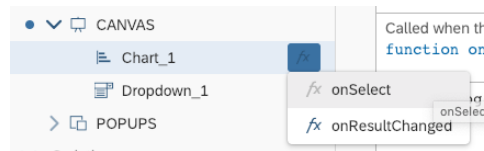


Figure 28: Hover Menu

### 4.3.2 Creating and Editing Functions in Global Script Objects

Functions are found under the global script objects portion of the *Outline*. Before you can add functions, you'll need to add your first script object. Do this by clicking the plus sign, next to the *Script Objects* header.



Figure 29: Add Script Object

Within a script object, you can add several functions, by invoking *Add Script Function* in the context menu. Keep in mind that the script object container is an organizational aid for you.

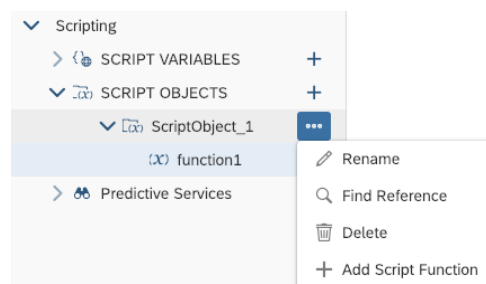


Figure 30: Add Script Function

Individual functions are nested within global script objects. For example, in the figure below you see the `function1` nested within a script object called `ScriptObject_1`.

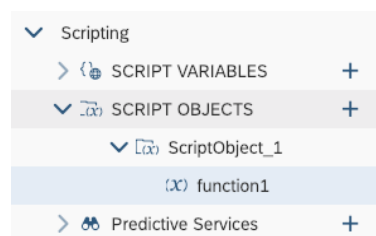


Figure 31: Script Object Function

Like Canvas widgets, the scripts attached to a function are created by clicking the `fx` icon in the hover menu of that function. Functions that have and don't have scripts are visible in the *Outline*, just as with widgets.

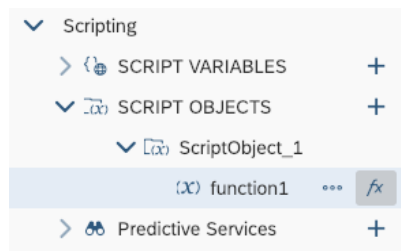


Figure 32: Script of Script Object Function

Once you've a script attached to a function, you can call it whenever you like, from any other script. You can access script objects by name. Within script objects you can access individual functions. If you wanted to invoke the `function1` script within `ScriptObject_1`, you would call it like this:

```
ScriptObject_1.function1();
```

### 4.3.3 Script Editor Layout

Once an open script is in the editor, it shows up as a tab along the top of the Canvas. You can open several script editor tabs at the same time.

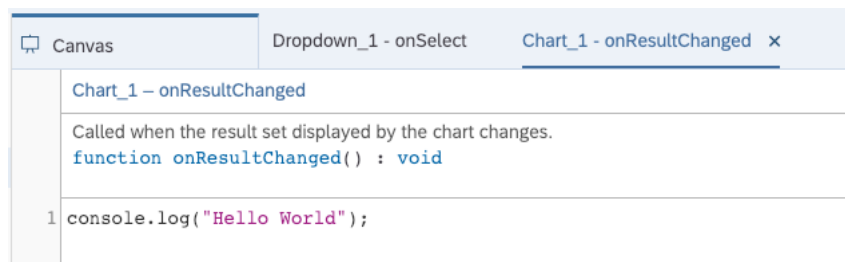


Figure 33: Script Editor

The script editor has three areas:

1. the widget and event
2. the documentation
3. the main body of the script itself



Figure 34: Areas of Script Editor

Write script in the main body using the inbuilt help features like code completion and value help.

### 4.3.4 Keyboard Shortcuts

The script editor provides several keyboard shortcuts, which let you, for example, undo or redo your editing operations.

Find a list of keyboard shortcuts in the help page “*Using Keyboard Shortcuts in the Script Editor*”: <https://help.sap.com/doc/00f68c2e08b941f081002fd3691d86a7/release/en-US/68dfa2fd057c4d13ad2772825e83b491.html>.

### 4.3.5 Info Panel: Errors and Reference List

All errors are listed in the *Errors* tab of the *Info* panel. Search for errors and filter out only warnings or errors. Double-click an error to open the script in a new tab and jump directly to the error location in the script.

Find all places where a widget or a scripting object is used with the *Find References* feature. You can find it in the context menu per object in the *Outline*. The result is displayed in the *Reference* list tab of the *Info* panel.

### 4.3.6 Renaming Widgets, Script Variables, and Script Functions

While creating an analytic application in analytics designer you can change the name of an analytics designer widget, component, script variable, script object, script object function, and script object function arguments. Analytics designer then applies the new name to all relevant places, for example in analytics designer scripts.

You can change the name of a widget, gadget, script variable, script object, or script object function by selecting it in the *Outline*, clicking the *More* button, selecting *Rename*, and entering a new name.

You can change the name of a widget or gadget by selecting it in the *Outline*, then entering in the *Styling* panel a new name in the *Name* input field.

You can change the name of a script variable, script object, or script object function by selecting it in the *Outline*, entering in the *Styling* panel a new name in the *Name* input field, then clicking button *Done*.

You can change the name of a script object function argument by selecting the script object function in the *Outline*, clicking the *Edit* button of the function argument in the *Styling* panel, entering a new name in the *Name* input field, then clicking button *Done*.

## 4.4 Scripting Language Features

### 4.4.1 Typing

Normal JavaScript is weakly typed and dynamically typed. Weak typing means that the script writer can implicitly coerce variables to act like different types. For example, you could have an integer value and treat it as if it were a string. Dynamic typing means that the runtime will try to guess the type from the context at that moment and the user can even change the type after the variable is already in use. For example, you could change the value of the beforementioned integer to another type of object at will: “Dear integer, you’re now a duck”.

Analytics designer forbids both. Once you've a duck, it remains a duck and you can't recycle variable names as new types. If you want something else, you'll need another variable. It's also strongly typed, meaning that if you want to use an integer as a string, you'll have to explicitly cast it. Both are a consequence of enabling the rich code completion capabilities in the editing environment.

The analytics designer scripting language is still JavaScript. You can write perfectly valid JavaScript while treating the language as if it was strongly and statically typed.

### 4.4.2 No Automatic Type Casting

A consequence of strong typing is that you can't expect automatic conversions. The following is valid JavaScript:

```
var nth = 1;
console.log("Hello World, " + nth);
```

In analytics designer, you'll see an error in the script editor, informing you that auto-type conversion isn't possible, and the script will be disabled at runtime, until fixed. Instead, you should explicitly cast `nth` to a string.

```
var nth = 1;
console.log("Hello World, " + nth.toString());
```

### 4.4.3 Accessing Objects

Every object (widget or global script object) is a global object with the same name as in the *Outline*. Suppose you've a chart in your application, named `Chart_1` and want to check and see if it's visible. You can access `Chart_1` as a global variable and then access its functions, in this case to see if it's currently visible.

```
var isVisible = Chart_1.isVisible();
```

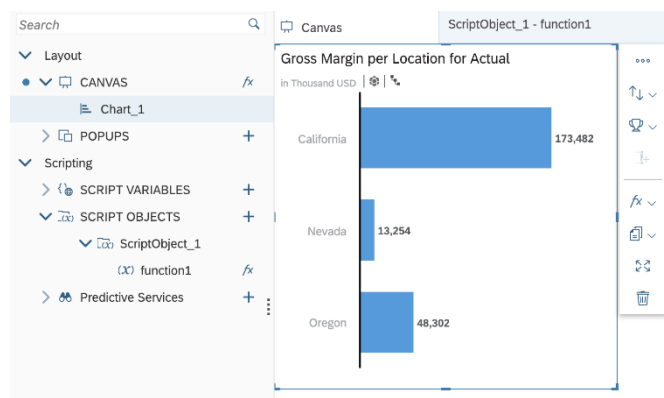


Figure 35: Accessing Objects

### 4.4.4 Finding Widgets with Fuzzy Matching

The application author can type in the complete name of a widget or just some first letters. By typing `CTRL+Spacebar`, the system either

- Completes the code automatically if there is only one valid option
- Displays a value help list from which you can select an option

Fuzzy matching helps you finding the result even if you've made a typo or the written letters are in the middle of the function. Fuzzy matching is applied automatically for the standard code completion (for example, "cose" → "console").

The script validation runs automatically in the background and shows errors and warnings indicated with red and orange underlying and a red or orange marker before the line number.

#### 4.4.5 External Libraries

There is no provision in analytics designer for importing external JavaScript libraries. However, you can use much of the functionality of the standard JavaScript built-in objects such as:

- Math
- Date
- Number
- Array
- Functions on String

The available set of standard JavaScript built-in objects and their functionality is listed in the SAP Analytics Cloud, analytics designer API Reference.

#### 4.4.6 Debugging with console.log()

Scripts are stored as minified variables and aren't directly debuggable in the browser console. Write messages directly to the browser's JavaScript console to aid in troubleshooting. A global variable called console and has a log() function that accepts a string.

```
var nth = 1;
console.log("Hello World, " + nth.toString());
```

This would print "Hello World, 1" to the JavaScript console of the browser. Complex objects can be printed.

#### 4.4.7 Loops

Two types of JavaScript **loops** are possible in analytics designer, `for` and `while` loops. Other types, such as `foreach` iterators, aren't supported.

##### 4.4.7.1 for Loop

`for` loops are standard JavaScript `for` loops, with one caveat. You must explicitly declare the `for` iterator. This is valid JavaScript, but it isn't accepted in the script editor:

```
for (i = 0; i < 3; i++) {
  console.log("Hello for, " + nth.toString());
}
```

Instead, explicitly declare `i`. The example below is valid:

```
for (var i = 0; i < 3; i++) {
  console.log("Hello for, " + nth.toString());
}
```

#### 4.4.7.2 while Loop

Analytics designer fully supports `while` loops:

```
var nth = 1;
while (nth < 3) {
    console.log("Hello while, " + nth.toString());
    nth++;
}
```

#### 4.4.7.3 for in Loop

An additional type of loop is the `for in` iterator. Suppose you had a JavaScript object: you can iterate over the properties with the `for in` loop. Data selections are JavaScript objects and can be iterated over:

```
var selection = {
    "Color": "red",
    "Location": "GER"
};
for (var propKey in selection) {
    var propValue = selection[propKey];
    ...
};
```

### 4.4.8 Double and Triple Equals Operators

Plain JavaScript has two kinds of “equals” comparison operators, `==` (double equals) and `===` (triple equals). The main difference between these is that double equals has automatic type casting while triple equals doesn’t. With triple equals, both the value and type must be the same for the result to be `true`. The triple equals is known as the strict equality comparison operator (see [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality\\_comparisons\\_and\\_sameness](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness)).

SAP Analytics Cloud, analytics designer has no automatic type casting. It supports

- Triple equals
- Double equals only if both sides have the same static type

The examples below show the difference between double and triple equals operators. In both cases, there is a variable `aNumber`, with an integer value and we are comparing it to the string `"1"`.

In the double equals case, `aNumber` is cast to string and compared. The result is `true`, and the `if` block is entered. In the triple equals case, `aNumber` isn’t cast to string and the comparison is `false`, because the values are of a different type.

This is `true`, and you can see the `if` statement is entered:

```
var aNumber = 1;
if (aNumber == "1") {
    ...
}
```

This is `false`, and you can see the `if` statement is skipped:

```
var aNumber = 1;
if (aNumber === "1") {
```

```
...  
}
```

#### 4.4.9 if and else Statements

The statements `if` and `else` are supported. Remember that there is no automatic type casting and double equals are valid only if both sides have the same static type:

```
if (nth === 1) {  
    console.log("if...");  
} else if (nth < 3) {  
    console.log("else if...");  
} else {  
    console.log("else...");  
}
```

#### 4.4.10 this Keyword

The `this` keyword allows you to ignore the name of the object. It's simply the object that this script is attached to, regardless of what it's called. It doesn't matter and is merely a stylistic choice. With `this`, refer to

- The instance itself within widget scripts or script object functions
- The parent instance explicitly by its variable name, such as `Chart_1`
- The parent instance as `this`

When performing the above console print on one of the events of `Chart_1` itself, use the following variation of the code:

```
var theDataSource = this.getDataSource();  
console.log(theDataSource.getVariables());
```

#### 4.4.11 switch Statements

You can use normal JavaScript `switch` statements:

```
switch (i) {  
    case 0:  
        day = "Zero";  
        break;  
    case 1:  
        day = "One";  
        break;  
    case 2:  
        day = "Two";  
        break;  
}
```

#### 4.4.12 break Statement

You can use `break` to break out of loops and `switch` statements, as seen in the example above.

### 4.4.13 Debugging Analytics Designer Scripts in the Browser

Analytics designer supports debugging analytics designer scripts with the browser's development tools.

Note: Analytics designer supports debugging analytics designer scripts in the Chrome browser only.

Note: Analytics designer transforms the analytics designer scripts before they're run in the browser. Thus, they **won't look exactly** like the script you wrote in the script editor of analytics designer.

Note: To find the analytics designer script in the browser's development tools, the script needs to be run at least once during the current session.

#### Analytics Designer Script Names

Analytics designer script names follow a specific naming convention: All scripts of an application are grouped in a folder `<APPLICATION_NAME>`. Each script is named `<WIDGET_NAME>.<FUNCTION_NAME>.js`.

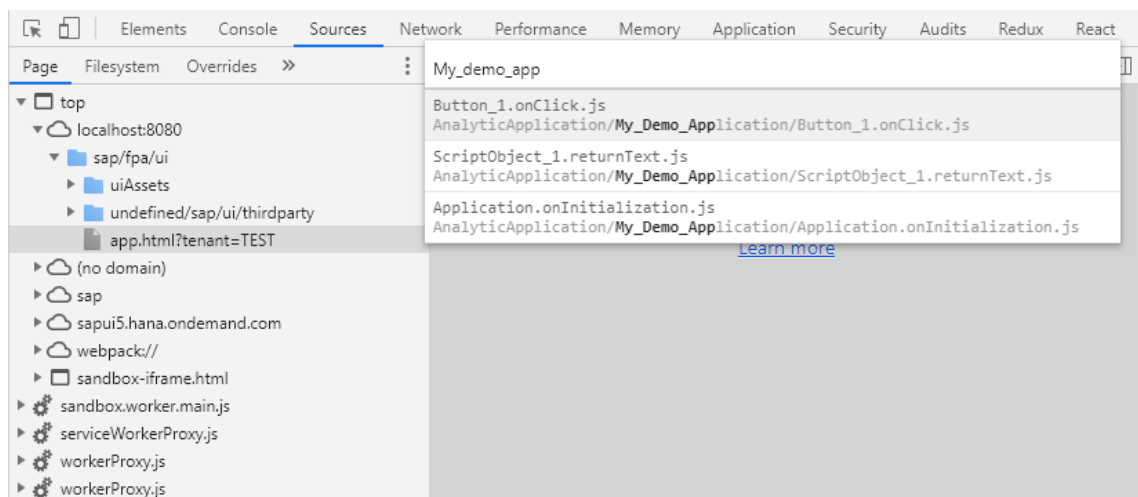
For example: If an application `My Demo Application` contains a button `Button_1` with an `onClick` event script, then the name of the script is `Button_1.onClick.js`, which is in folder `My_Demo_Application`.

Note: Special characters, for example space characters in the application name are replaced by an underscore (`_`), except minus (`-`) and dot (`.`) characters, which remain unchanged.

#### Find an Analytics Designer Script by Name

You can quickly find a script by its name with the following steps:

- Press **F12** to open the browser's development tools.
- Press **CTRL+P**, then start typing a part of the script's name.

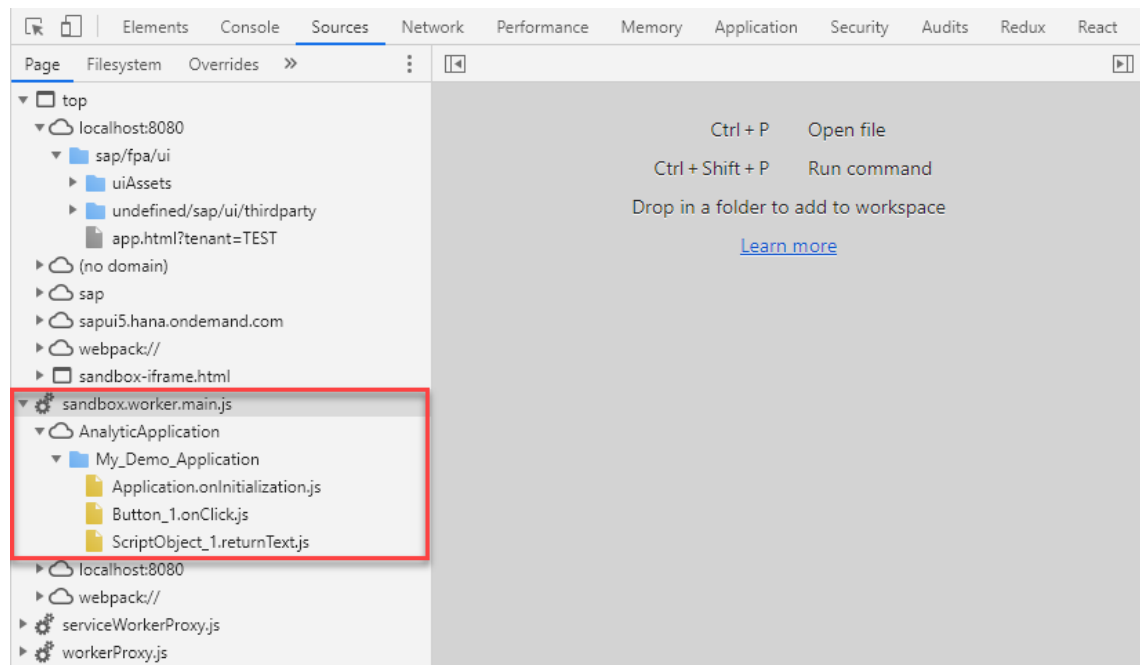


#### Find an Analytics Designer Script by Browsing the File Tree

You can also find a script in the file tree on the left side of the development tools using the following steps:

- Press **F12** to open the browser's development tools.

- Select the tab *Sources*.
- Open the node whose name starts with `sandbox.worker.main`.
- Open the node named `AnalyticApplication`.
- Find the folder with your application's name. The scripts that have already been executed for the current analytic application appear in this folder.

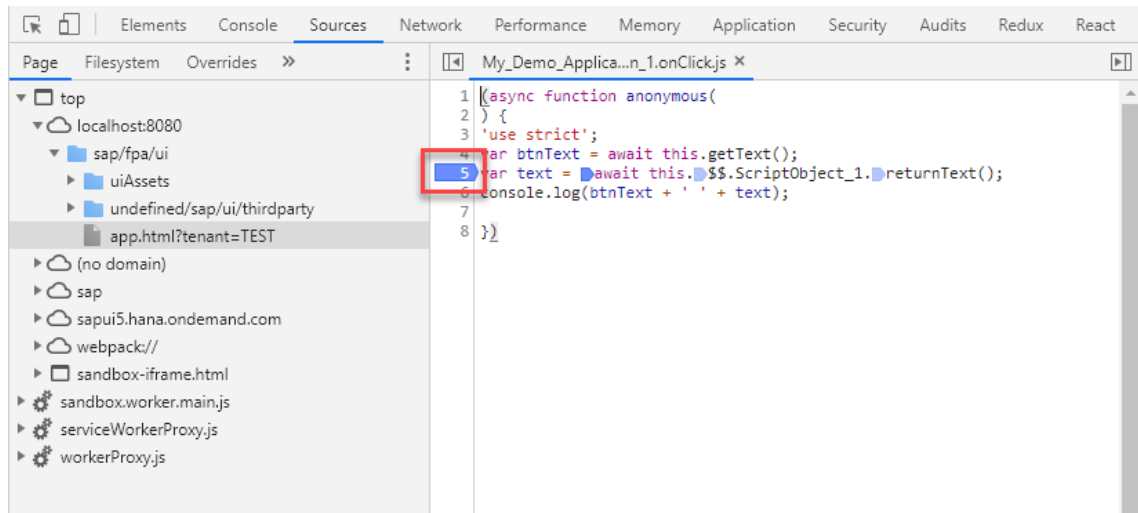


### Setting and Removing Breakpoints in Analytic Designer Scripts

To pause a script while it's being executed, you can set breakpoints at certain locations in the analytic designer script.

To set a breakpoint, open the script you want to pause during its execution and click on the line number on the left side of the opened script in the developer tools.

A blue marker appears, highlighting the clicked line number. It indicates where the script will be paused when it's being executed the next time. You can add several breakpoints in one script to pause its execution at different points in time.



To remove a breakpoint, click on the blue marker. The blue marker disappears, and the script's execution won't stop at this point when the script is run the next time.

### More Debugging Features

Analytic designer supports more debugging features by running an analytic application in debug mode.

You enable the debug mode for an analytic application by appending the string `;debug=true` to the URL of the analytic application in the browser.

Note: The analytic designer script names in the browser change when the analytic application is run in debug mode. In debug mode, the suffix `-dbg` is added to the script name, for example, `Button_1.onClick-dbg.js`.

You enable the debug mode for an analytic application by appending the string `;debug=true` to the URL of the analytic application in the browser.

Note: The analytic designer script names in the browser change when the analytic application is run in debug mode. In debug mode, the suffix `-dbg` is added to the script name, for example, `Button_1.onClick-dbg.js`.

### Enabling the debugger; Statement

When the debug mode is enabled, you can pause an analytics designer script at a specific location while it's being executed by placing a `debugger;` statement at this location of the script. The difference to a regular breakpoint is that you can define the location where the script is paused already while writing the script itself, that's, before running it.



### Preserving Comments in an Analytical Designer Script

When the debug mode is enabled, then comments in a script are preserved in the transformed script that's executed in the browser. This makes it easier to recognize specifically commented locations in a script when its execution in the browser is paused.

### 4.4.14 Using Two-Dimensional Arrays

This section shows you how you can create and work with two-dimensional (2D) arrays in the analytics designer scripting language.

#### Creating 2D-Arrays

Recall that you can easily create a one-dimensional (1D) array in a script. The following snippet, for example, creates a 1D-array of numbers:

```
var arr1D = ArrayUtils.create(Type.number);
```

However, there is no similar method to create a 2D-array. Still, it's possible to create a 2D-array. The trick is to use a 1D-array that in turn contains several 1D-arrays. Together they act as a 2D-array.

*Example:*

In the following example, let's create a 2D-array of 4 columns and 3 rows that contains values of type `number`.

First, let's define the number of rows and columns as constants, this makes the following code clearer:

```
var numCols = 4;
var numRows = 3;
```

As mentioned before, we can't directly create a 2D-array, but we can use a trick: We create a 1D-array `arr2D` containing several 1D-arrays of type `number`. The number of 1D-arrays that `arr2D`

must contain is `numRows`, the number of rows of our array. The completed construct will have the same effect as a 2D-array.

Let's create a 1D-array that's to contain the first row of numbers:

```
var arr1D = ArrayUtils.create(Type.number);
```

Then let's create the 1D-array `arr2D` into which we plug `arr1D`, the array to contain the first row of numbers:

```
var arr2D = [arr1D];
```

At this point you may wonder why we didn't simply define `arr2D` as `var arr2D = []`? Because analytics designer won't let us, as it can't infer the content type of `arr2D` from an empty array.

Then we add the remaining 1D-arrays to `arr2D`. Each of them contains one more row of numbers. Note that the `for`-loop below doesn't start with `0` but with `1` because `arr2D` was already initialized with `arr1D`, the 1D-array to contain the first row of numbers.

```
for (var i = 1; i < numRows; i++) {  
    arr2D.push(ArrayUtils.create(Type.number));  
}
```

We're done!

For your reference, this is the complete code:

```
var numCols = 4;  
var numRows = 3;  
var arr1D = ArrayUtils.create(Type.number);  
var arr2D = [arr1D];  
for (var i = 1; i < numRows; i++) {  
    arr2D.push(ArrayUtils.create(Type.number));  
}
```

Note that we didn't need to use the constant `numCols` to define the 2D-array. However, we'll make use of it in the following sections.

## Setting Values to 2D-Arrays

Now we can set a value into the array `arr2D` in the form

```
arr2D[row][col] = value;
```

For example, let's set the increasing value of a counter to each of the elements of the 2D-array `arr2D` with the following script:

```
var count = 0;  
for (var row = 0; row < numRows; row++) {  
    for (var col = 0; col < numCols; col++) {  
        arr2D[row][col] = count;  
        count = count + 1;  
    }  
}
```

## Getting Values from 2D-Arrays

Finally, let's get and output the values from the 2D-array `arr2D` with the following script:

```
for (row = 0; row < numRows; row++) {  
    for (col = 0; col < numCols; col++) {
```

```
        console.log(arr2D[row][col]);
    }
    console.log("");
}
```

The output is:

```
0
1
2
3

4
5
6
7

8
9
10
11
```

## 4.5 Data Sources and Working with Data

You can perform many simple operations on data. Keep in mind there are no standalone data sources, and there is a `getVariables()` function on data sources.

*Example:*

Let's say you want to print the variables on `Chart_1` to the console.

Get the data source of a widget with the widget's `getDataSource()` method. This returns the data source attached to that widget and allows you to perform further operations.

The snippet below prints the data source variables of `Chart_1` to the console:

```
var theDataSource = Chart_1.getDataSource();
var theVariables = theDataSource.getVariables();
console.log(theVariables);
```

### 4.5.1 Setting Range and Exclude Filters

The script API method `DataSource.setDimensionFilter()` supports range filters and exclude filters.

#### 4.5.1.1 Exclude Filters

You can filter out members from the drill-down with exclude filters. The following examples show the use of exclude filters:

*Example:*

The following single filter value is set for dimension `"EMPLOYEE_ID"`. This keeps only the member with employee ID 230 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {value: "230"});
```

*Example:*

The following single but excluding filter value is set for dimension "EMPLOYEE\_ID". This removes the member with employee ID 230 from the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {value: "230", exclude: true});
```

*Example:*

The following multiple filter values are set for dimension "EMPLOYEE\_ID". This keeps the members with employee IDs 230 and 240 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {values: ["230", "240"]});
```

*Example:*

The following multiple but excluding filter values are set for dimension "EMPLOYEE\_ID". This removes the members with employee IDs 230 and 240 from the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {values: ["230", "240"], exclude: true});
```

#### 4.5.1.2 Range Filters

You can filter ranges of members in the drill-down with range filters.

Note the following when using range filters:

- Range filters can only be applied to numeric dimensions.
- A time dimension isn't a numeric dimension.
- SAP BW doesn't support numeric dimensions.

The following examples show the use of range filters:

*Example:*

The following range filter applied to dimension "EMPLOYEE\_ID" keeps the members with employee IDs between 230 and 240 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {from: "230", to: "240"});
```

*Example:*

The following range filter applied to dimension "EMPLOYEE\_ID" keeps the members with employee IDs less than 230 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {less: "230"});
```

*Example:*

The following range filter applied to dimension "EMPLOYEE\_ID" keeps the members with employee IDs less or equal than 230 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {lessOrEqual: "230"});
```

*Example:*

The following range filter applied to dimension "EMPLOYEE\_ID" keeps the members with employee IDs greater or equal than 230 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {greaterOrEqual: "230"});
```

*Example:*

The following range filter applied to dimension "EMPLOYEE\_ID" keeps the members with employee IDs greater than 230 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", {greater: "230"});
```

You can also apply multiple range filters at once.

*Example:*

The following range filter applied to dimension "EMPLOYEE\_ID" keeps the members with employee IDs less than 230 and greater than 240 in the drill-down:

```
DS_1.setDimensionFilter("EMPLOYEE_ID", [{less: "230"}, {greater: "240"}]);
```

## 4.5.2 Getting Dimension Filters

You can get the dimension filters of a dimension with the method `getDimensionFilters()`. It returns an array of all filter values as `FilterValue` objects. The script API definition is as follows:

```
getDimensionFilters(string | DimensionInfo): FilterValue[]
```

*Example:*

In the following example, the dimension filter values of the filter `COUNTRY` are retrieved:

```
var values = Table_1.getDataSource().getDimensionFilters("COUNTRY");
```

Each value in the array is an instance of either `SingleFilterValue`, `MultipleFilterValue`, or `RangeFilterValue`, which all inherit from `FilterValue`. To work with such an instance, that is, to access its type-specific properties, cast the instance to the corresponding type first, using the `type` property. The following example shows how:

```
var value = Table_1.getDataSource().getDimensionFilters("COUNTRY")[0];
switch (value.type) {
  case FilterValueType.Single:
    var singleValue = cast(Type.SingleFilterValue, value);
    console.log(singleValue.value); // you can access the 'value' property now
    break;
  case FilterValueType.Multiple:
    var multipleValue = cast(Type.MultipleFilterValue, value);
    console.log(multipleValue.values); // you can access the 'values' property now
    break;
  case FilterValueType.Range:
    var rangeValue = cast(Type.RangeFilterValue, value);
    console.log(rangeValue.from); // you can access the 'from' property now
    console.log(rangeValue.to); // you can access the 'to' property now
    // further range properties: 'less', 'lessOrEqual', 'greater', 'greaterOrEqual'
    break;
  default:
    break;
}
```

Note: Currently this script API doesn't return time range filters.

Note: In SAP BW backend systems you can create valid filters that aren't yet supported by SAP Analytics Cloud. As this script API implementation is based on SAP Analytics Cloud, it supports the capabilities of SAP Analytics Cloud.

### 4.5.3 Dimension Properties

You can retrieve dimension properties of a data source.

#### getDimensionProperties

```
getDimensionProperties(dimension: string | DimensionInfo): DimensionPropertyInfo[]
```

Returns all available dimension properties of the specified dimension of a data source.

See also section [Dimension Properties](#) on the Dimension Properties script API of the Table.

### 4.5.4 Hierarchies

You can set the drill level of a dimension hierarchy as well as expand or collapse individual hierarchy nodes.

Note: Currently, this is supported only by data sources of Table and Chart widgets.

#### Customize the Display Level of the Hierarchy

Specify the dimension and hierarchy level that you would like to set or retrieve:

```
DataSource.setHierarchyLevel(dimension: string|DimensionInfo, level?: integer): void
DataSource.getHierarchyLevel(dimension: string|DimensionInfo): integer

// Example 1. Chart, set hierarchy level to 2
var ds = Chart_1.getDataSource();
ds.setHierarchy("Location_4nm2e04531", "State_47acc246_4m5x6u3k6s");
ds.setHierarchyLevel("Location_4nm2e04531", 2);

// Example 2. Table, set hierarchy level to 2
var ds = Table_1.getDataSource();
ds.setHierarchy("Location_4nm2e04531", "State_47acc246_4m5x6u3k6s");
ds.setHierarchyLevel("Location_4nm2e04531", 2);
```

#### Expand or Collapse Hierarchy Nodes

Specify the dimension and hierarchy node that you would like to expand or collapse:

```
DataSource.expandNode(dimension: string|DimensionInfo, selection: Selection): void
DataSource.collapseNode(dimension: string|DimensionInfo, selection: Selection): void

// Example 1. Chart has Location in category axis. Hierarchy info is
// "State_47acc246_4m5x6u3k6s", hierarchy level is 1.
// One node (Location="California") is expanded
Chart_1.getDataSource().expandNode("Location_4nm2e04531", {
  "Location_4nm2e04531":
    "[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].[SA1]", // California
  "@MeasureDimension": "[Account_BestRunJ_sold].[parentId].[Discount]"
})
```

```
});

// Example 2. Chart has Location and Product in category axis.
// Hierarchy level of both location and product is 1.
// All nodes (Product="Alcohol") are expanded
Chart_1.getDataSource().expandNode("Product_3e315003an", {
  "Product_3e315003an": "[Product_3e315003an].[Product_Catego_3o3x5e06y2].&[PC4]",
  // Alcohol
  "@MeasureDimension": "[Account_BestRunJ_sold].[parentId].&[Discount]"
});

// Example 3. Table has Location and Product in row.
// Hierarchy level of both location and product is 1.
// One node (Location="California" and Product="Alcohol") is expanded
Table_1.getDataSource().expandNode("Location_4nm2e04531", {
  "Product_3e315003an": "[Product_3e315003an].[Product_Catego_3o3x5e06y2].&[PC4]",
  // Alcohol
  "Location_4nm2e04531":
  "[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].&[SA1]", // California
  "@MeasureDimension": "[Account_BestRunJ_sold].[parentId].&[Discount]"
});
```

## 4.5.5 Getting Members

### 4.5.5.1 Getting a Single Member

You can retrieve information about a single member with the following script API:

```
DataSource.getMember(dimension: string | DimensionInfo, memberId: string,
  hierarchy?: string | HierarchyInfo): MemberInfo
```

This returns the member info object of a member specified by its member ID and the member's dimension. Optionally, you can specify a hierarchy of the dimension. If no hierarchy is specified, then the current active hierarchy of the dimension is used in identifying the member.

*Example:*

Let's assume the current active hierarchy of dimension "Location\_4nm2e04531" is set to a flat presentation, then

```
Table_1.getDataSource().getMember("Location_4nm2e04531", "CT1");
```

returns the member info object

```
{id: 'CT1', description: 'Los Angeles', dimensionId: 'Location_4nm2e04531',
  displayId: 'CT1'}
```

Note: The required member ID of a member may differ, depending on the current active hierarchy of the member's dimension. For example, for a HANA system, the member ID of the same member may be "CT1" for the dimension "Location\_4nm2e04531" with a flat presentation hierarchy and "[Location\_4nm2e04531].[State\_47acc246\_4m5x6u3k6s].&[CT1]" for an actual hierarchy. This influences the results of method `getMember()`:

*Example:*

Let's assume that the current active hierarchy of dimension "Location\_4nm2e04531" is set to "State\_47acc246\_4m5x6u3k6s", a hierarchy of US-American states. Then

```
Table_1.getDataSource().getMember("Location_4nm2e04531", "CT1");
```

returns `undefined`, whereas

```
Table_1.getDataSource().getMember("Location_4nm2e04531",
"[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].&[CT1]");
```

returns the member info object

```
{id: '[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].&[CT1]', description: 'Los Angeles', dimensionId: 'Location_4nm2e04531', ...}
```

*Example:*

Let's assume that the current active hierarchy of "Location\_4nm2e04531" is set to a flat presentation. Then

```
Table_1.getDataSource().getMember("Location_4nm2e04531", "CT1");
```

returns the member info object

```
{id: 'CT1', description: 'Los Angeles', dimensionId: 'Location_4nm2e04531', ...}
```

whereas

```
Table_1.getDataSource().getMember("Location_4nm2e04531",
"[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].&[CT1]");
```

returns `undefined`.

Note: If the specified hierarchy doesn't exist, then `undefined` is returned.

Note: If the data source is associated with an R visualization and you specify a hierarchy other than `Alias.FlatHierarchy` (flat presentation), then `undefined` is returned.

#### 4.5.5.2 Getting Members

You can retrieve information about the members of a dimension with the following script API:

```
DataSource.getMembers(dimension: string | DimensionInfo, options?: integer | MembersOptions JSON): MemberInfo[]
```

This returns the members of the specified dimension. If you additionally specify a number, then at most this many members are returned (default: 200). If you specify members options instead of a number, then you can control the returned set of members even finer yet.

The member options are defined as

```
{
  // Type of members to retrieve (default: MemberAccessMode.MasterData)
  accessMode: MemberAccessMode
  // Hierarchy ID (default: currently active hierarchy)
  hierarchyId: string
  // Maximum number of returned members, which must be zero or a positive number.
  limit: integer
}
```

You can specify one or more member option parameters, in any sequence.

*Examples:*

Let's assume that the current active hierarchy of "Location\_4nm2e04531" is set to a flat presentation. Then

```
Table_1.getDataSource().getMembers("Location_4nm2e04531", 3);
```

returns the following array of member info objects:

```
[{id: 'CT1', description: 'Los Angeles', dimensionId: 'Location_4nm2e04531', ...},
 {id: 'CT10', description: 'Reno', dimensionId: 'Location_4nm2e04531', ...},
 {id: 'CT11', description: 'Henderson', dimensionId: 'Location_4nm2e04531', ...}]
```

You can achieve the same result with member options, specifying the `limit` parameter:

```
Table_1.getDataSource().getMembers("Location_4nm2e04531", {limit: 3});
```

The following example retrieves the first two members and specifies a hierarchy of US-American states:

```
Table_1.getDataSource().getMembers("Location_4nm2e04531", {limit: 2, hierarchy:
"State_47acc246_4m5x6u3k6s"});
```

This returns the result:

```
[{id: '[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].&[SA1]', description:
'California', dimensionId: 'Location_4nm2e04531', ...},
 {id: '[Location_4nm2e04531].[State_47acc246_4m5x6u3k6s].&[CT1]', description: 'Los
Angeles', dimensionId: 'Location_4nm2e04531', ...}]
```

In addition, you can specify the access mode, that is, whether the returned data is from master data (the default) or booked values.

The following code snippet

```
Table_1.getDataSource().getMembers("Location_4nm2e04531", {accessMode:
MemberAccessMode.MasterData});
```

returns the array

```
[{id: 'CT1', description: 'Los Angeles', dimensionId: 'Location_4nm2e04531', ...},
 {id: 'CT10', description: 'Reno', dimensionId: 'Location_4nm2e04531', ...},
 ...
 {id: 'CT9', description: 'Las Vegas', dimensionId: 'Location_4nm2e04531', ...},
 {id: 'SA1', description: 'California', dimensionId: 'Location_4nm2e04531', ...},
 {id: 'SA2', description: 'Nevada', dimensionId: 'Location_4nm2e04531', ...},
 {id: 'SA3', description: 'Oregon', dimensionId: 'Location_4nm2e04531', ...}]
```

whereas the following code snippet

```
Table_1.getDataSource().getMembers("Location_4nm2e04531", {accessMode:
MemberAccessMode.BookedValues});
```

returns the array

```
[{id: 'CT1', description: 'Los Angeles', dimensionId: 'Location_4nm2e04531', ...}
 {id: 'CT10', description: 'Reno', dimensionId: 'Location_4nm2e04531', ...}
 ...
 {id: 'CT9', description: 'Las Vegas', dimensionId: 'Location_4nm2e04531', ...}]
```

In the second example the members of the three states California, Nevada, and Oregon aren't part of the booked values.

Tip: To find out the booked values of a dimension (in combination with a specific hierarchy), create an analytic application with a table that contains a data source with this dimension. Set the same hierarchy to the dimension. In the *Builder* panel of the table, open the ... ("three dots") menu of the dimension, then deselect the menu entry *Unbooked Values*. The table now displays the booked values of the dimension (in combination with the selected hierarchy).

Note: If the hierarchy specified in the members options doesn't exist, then an empty array is returned.

Note: If the data source is associated with an R visualization and you specify a hierarchy other than `Alias.FlatHierarchy` (flat presentation) in the members options, then an empty array is returned.

#### 4.5.6 Getting Information About a Data Source

You can get information about a data source with the `DataSource.getInfo()` script API method. It returns a `DataSourceInfo` object containing information about the specified data source. The information is contained in a set of properties according to the following definition:

```
class DataSourceInfo {
  modelName: string,
  modelId: string,
  modelDescription: string,
  sourceName: string,
  sourceDescription: string,
  sourceLastChangedBy: string,
  sourceLastRefreshedAt: Date
}
```

Note: The `DataSourceInfo` properties `sourceName`, `sourceDescription`, `sourceLastChangedBy`, and `sourceLastRefreshedAt` are only supported for data sources based on SAP BW models. For all other models they contain the value `undefined`.

##### Example:

With the following sample code snippet, you can print information about a table's data source to the browser console:

```
var dsInfo = Table_1.getDataSource().getInfo();
console.log("Model name: " + dsInfo.modelName);
console.log("Model ID: " + dsInfo.modelId);
console.log("Model description: " + dsInfo.modelDescription);
console.log("Source name: " + dsInfo.sourceName);
console.log("Source description: " + dsInfo.sourceDescription);
console.log("Source last changed by: " + dsInfo.sourceLastChangedBy);
var strLastRefresh = "undefined";
if (dsInfo.sourceLastRefreshedAt !== undefined) {
  strLastRefresh = dsInfo.sourceLastRefreshedAt.toISOString();
}
console.log("Source last refreshed at: " + strLastRefresh);
```

A sample output for a data source based on an SAP BW model is:

```
Model name: HAL_TEST_Scenario_Query
Model ID: t.H:C9gjfpmu5ntxaf3dbfwtyl5wab
Model description: Sample scenario query
Source name: TEST_SCENARIO_QUERY
Source description: Test Query Scenario
Source last changed by: SYSTEM
```

Source last refreshed at: 2021-09-23T22:00:00.000Z

A sample output for a data source based on an SAP HANA model is:

Model name: BestRunJuice\_SampleModel  
Model ID: t.2.CMRCZ9NPY3VAER9A06PT80G12:CMRCZ9NPY3VAER9A06PT8314150G34  
Model description: Sample Model  
Source name: undefined  
Source description: undefined  
Source last changed by: undefined  
Source last refreshed at: undefined

## 4.6 Working with Pattern-Based Functions

With pattern-based functions you can create string-based functions by only providing input and output examples instead of writing script code. For example, you can transform email addresses like `john.doe@sap.com` to names like `John Doe`.

### 4.6.1 Adding a Pattern-Based Function

1. In the *Outline* of your application, add a *ScriptObject*.
2. Choose ... (*More Actions*) beside the script object and select *Add Pattern Based Function*. This opens the *Pattern Based Function* dialog where you can see the generated name of the function and where you can add a description for the function.

### 4.6.2 Creating the Pattern

After you've added a pattern-based function to your application you need to create the pattern through input and output training examples.

1. By clicking the + symbol next to *Create Pattern* you can add an input and output example. Under *Training Example*, you can define that a certain input shall become a certain output (for example, `john.doe@sap.com` shall become `John Doe`). Note: Sometimes one example isn't enough due to potential ambiguity. In this case you can use up to three training examples by clicking the + symbol for each following example.
2. Click *Create* so that a machine learning algorithm starts looking for a pattern that matches every example. Note: As soon as you change a *Training Example*, the pattern will fall back to its default, which is just returning the input. Note: If you want to undo your changes on new or modified training examples, click *Reset*, which sets the pattern-based function to the last working pattern with the according training examples.
3. If the pattern creation has been successful, you can verify the pattern by entering input examples that follow your pattern above; otherwise, it will keep its default pattern, which is just outputting the input. To verify the pattern, click the + symbol next to *Verify Pattern*. Type a test example under *Input* to test the output: In the *Output* field you should see the correct output that follows your output pattern above.
4. When you've finished your work, click *Done*.

### 4.6.3 Scripting

If you've successfully created a pattern-based function and its pattern, you can use it in every script of the application just like any other script object function. The following example shows how the pattern-based function is used in a script of the application:

```
var firstNameandName = ScriptObject_1.myPatternBasedFunction("joe.doe@sap.com");
```

### 4.6.4 More Examples

#### 4.6.4.1 Transforming Dates

Let's assume you've a list of dates in the form *month.day.year*, for example, *10.11.2011* and you want to transform this to *11.10.11*. So, you type *10.11.2011* as input and *11.10.11* as output in the *Training Example* section.

In this example, you want to swap the day with the month and only take the last two digits of the year. But since this is ambiguous (it's not clear if the number 11 at the end is taken from the month or from the last digits of the year) you need to provide a second training example like *09.05.2020* as an input example and *05.09.20* as an output example.

#### 4.6.4.2 Extracting Specific Strings and Adding New Ones

Let's assume you've a list of appointments like this: *John Doe has an appointment on 06.07.20 at 3:00pm*. You want to transform this to *Name: John Doe, Date: 06.07.20, Time: 3:00pm*.

So, you type *John Doe has an appointment on 06.07.20 at 3:00pm* as input and *Name: John Doe, Date: 06.07.20, Time: 3:00pm* as output in the *Training Example* section.

## 4.7 Method Chaining

Typically, scripts are written such that one line of code executes one operation. But some scripts need to be made compact, and this can be done with method chaining. Certain JavaScript libraries support method chaining where the result of a previous operation can immediately be used in the same statement. Analytics designer supports method chaining.

Suppose you were only logging the variables in the above example as a debug aid. You were not re-using them, and the multiple lines were visual clutter. Then you might want to use method chaining. The code below uses method chaining for compactness and does the same thing:

```
console.log(Chart_1.getDataSource().getVariables());
```

## 4.8 Script Runtime

Analytics designer validates the script before execution because running arbitrary JavaScript in the browser is a risk. It ensures that only allowed JavaScript subset can be used. Critical features like sending requests can be prevented or forced to use alternative secured APIs if needed. In addition, the execution is isolated to prevent

- Accessing the DOM
- Accessing global variables
- Modifying globals or prototypes

- Sending requests
- Importing scripts
- Including ActiveX, and so on
- Launching other Web Workers
- Accessing cookies
- Enforcing different domains

### Validation

Validation at runtime follows the same logic as for the script editor. Not all validations have to be performed, for example, validating analytic data like filter values.

## 4.9 The R Widget and JavaScript

You might know the R widget from stories already. It becomes much more powerful in applications. The R widget has two separate runtime environments:

1. The R environment is on the server, in the R engine.
2. The JavaScript environment runs in the normal browser space along with the rest of the widget scripts.

### Execution Order

On Startup, the R script runs only. The `onResultSetChanged` event script (JavaScript) doesn't run because the widget is in its initial view state.

On data change, the R script runs first, then the `onResultChanged` event script (JavaScript) runs.

### Accessing the R Environment from JavaScript

The R environment can be accessed from the JavaScript environment. It can be read from and manipulated. However, the JavaScript environment can't be accessed from the R environment.

### Reading

Suppose you had an R widget that had a very simple script. It just gets the correlation coefficient between two measures on a model and puts that into a number named `gmCorrelation`:

```
grossMargin <- BestRun_Advanced$`Gross Margin`  
grossMarginPlan <- BestRun_Advanced$`Gross Margin Plan`  
gmCorrelation <- cor(grossMargin, grossMarginPlan)
```

Use the `getEnvironmentValues` on the R widget to access its environment and `getNumber` to read a number from the R environment. The following JavaScript code takes the correlation coefficient from the R environment and logs it to the JavaScript console. Note the `this`. This code was taken from the `onResultChanged` event script of a widget with the above R snippet. It means that R widgets can be used as global data science scripts:

```
var nCcor = this.getEnvironmentValues().getNumber("gmCorrelation");  
var sCor = nCcor.toString();  
console.log("Margin Correlation: " + sCor);
```

## Writing

You can also manipulate the R environment from JavaScript. The magic methods are `getInputParameters` and `setNumber`. The following line of JavaScript sets an R environment variable named `userSelection` to 0.

```
RVisualization_1.getInputParameters().setNumber("userSelection", 0);
```

## 4.10 Differences Between SAP Analytics Cloud and Lumira Designer

Design Studio / Lumira Designer and SAP Analytics Cloud, analytics designer have broadly similar scripting environments. Both are JavaScript based, perform similar missions and analytics designer's scripting framework was informed by experience with Design Studio. However, there are some differences that you should keep in mind.

Lumira scripts execute on the server. Analytics designer scripts execute in the browser JavaScript engine. Lumira scripts execute close to the data. Analytics designer scripts execute close to the user.

Analytics designer isn't copy-and-paste compatible with Lumira. This is partially a consequence of the close-to-data vs close-to-user philosophical difference.

Data sources are currently hidden within data-bound widgets and you must access them using `getDataSource()`. When standalone data sources become available, you can access them as global variables, as in Lumira.

Analytics designer isn't supporting automatic type conversion makes scripts more explicit and avoids common mistakes. This includes requiring a strict equality comparison operator, whereas Lumira allowed the use of the double equals comparison operator for expressions of different types.