("<node-name>").all(branchIndex?: number, runIndex?: number)

This gives access to all the items of the current or parent nodes. If you don't supply any parameters, it returns all the items of the current node.

Getting items

JavaScript

```
// Returns all the items of the given node and current run
let allItems = $("<node-name>").all();

// Returns all items the node "IF" outputs (index: 0 which
is Output "true" of its most recent run)
let allItems = $("IF").all();

// Returns all items the node "IF" outputs (index: 0 which
is Output "true" of the same run as current node)
let allItems = $("IF").all(0, $runIndex);

// Returns all items the node "IF" outputs (index: 1 which
is Output "false" of run 0 which is the first run)
let allItems = $("IF").all(1, 0);
```

Python

```
# Returns all the items of the given node and current run
allItems = _("<node-name>").all();

# Returns all items the node "IF" outputs (index: 0 which
is Output "true" of its most recent run)
```

```
allItems = _("IF").all();

# Returns all items the node "IF" outputs (index: 0 which is Output "true" of the same run as current node)
allItems = _("IF").all(0, _runIndex);

# Returns all items the node "IF" outputs (index: 1 which is Output "false" of run 0 which is the first run)
allItems = _("IF").all(1, 0);
```

Accessing item data

Get all items output by a previous node, and log out the data they contain:

JavaScript

```
previousNodeData = $("<node-name>").all();
for(let i=0; i<previousNodeData.length; i++) {
  console.log(previousNodeData[i].json);
}
```

Python

```
previousNodeData = _("<node-name>").all();
for item in previousNodeData:
  # item is of type <class 'pyodide.ffi.JsProxy'>
  # You need to convert it to a Dict
  itemDict = item.json.to_py()
  print(itemDict)
```

AI coding with GPT

Experimental feature with limited availability

As an experimental feature, n8n is gradually rolling this out on Cloud from version 1.3.0. If you don't see the feature when you first upgrade to 1.3.0, please be patient: it's coming soon.

Not available on self-hosted.

Python isn't supported.

Use AI in the Code node

Feature availability

AI assistance in the Code node is available to Cloud users. It isn't available in self-hosted n8n.

AI generated code overwrites your code

If you've already written some code on the **Code** tab, the AI generated code will replace it. n8n recommends using AI as a starting point to create your initial code, then editing it as needed.

To use ChatGPT to generate code in the Code node:

1. In the Code node, set **Language** to **JavaScript**.
2. Select the **Ask AI** tab.
3. Write your query.
4. Select **Generate Code**. n8n sends your query to ChatGPT, then displays the result in the **Code** tab.

Usage limits

During the trial phase there are no usage limits. If n8n makes the feature permanent, there may be usage limits as part of your pricing tier.

Feature limits

The ChatGPT implementation in n8n has the following limitations:

- The AI writes code that manipulates data from the n8n workflow. You can't ask it to pull in data from other sources.
- The AI doesn't know your data, just the schema, so you need to tell it things like how to find the data you want to extract, or how to check for null.
- Nodes before the Code node must execute and deliver data to the Code node before you run your AI query.
- Doesn't work with large incoming data schemas.
- May have issues if there are a lot of nodes before the code node.

Writing good prompts

Writing good prompts increases the chance of getting useful code back.

Some general tips:

- Provide examples: if possible, give a sample expected output. This helps the AI to better understand the transformation or logic you’re aiming for.
- Describe the processing steps: if there are specific processing steps or logic that should apply to the data, list them in sequence. For example: "First, filter out all users under 18. Then, sort the remaining users by

their last name."

- Avoid ambiguities: while the AI understands various instructions, being clear and direct ensures you get the most accurate code. Instead of saying "Get the older users," you might say "Filter users who are 60 years and above."
- Be clear about what you expect as the output. Do you want the data transformed, filtered, aggregated, or sorted? Provide as much detail as possible.

And some n8n-specific guidance:

- Think about the input data: make sure ChatGPT knows which pieces of the data you want to access, and what the incoming data represents. You may need to tell ChatGPT about the availability of n8n's built-in methods and variables.
- Declare interactions between nodes: if your logic involves data from multiple nodes, specify how they should interact. "Merge the output of 'Node A' with 'Node B' based on the 'userID' property". if you prefer data to come from certain nodes or to ignore others, be clear: "Only consider data from the 'Purchases' node and ignore the 'Refunds' node."
- Ensure the output is compatible with n8n. Refer to Data structure for more information on the data structure n8n requires.

Example prompts

These examples show a range of possible prompts and tasks.

Example 1: Find a piece of data inside a second dataset

To try the example yourself, download the example workflow and import it into n8n.

In the third Code node, enter this prompt:

> The slack data contains only one item. The input data represents all Notion users. Sometimes the person property that holds the email can be null. I want to find the notionId of the Slack user and return it.

Take a look at the code the AI generates.

This is the JavaScript you need:

```
const slackUser = $("Mock Slack").all()[0];
const notionUsers = $input.all();
const slackUserEmail = slackUser.json.email;

const notionUser = notionUsers.find(
  (user) => user.json.person && user.json.person.email === slackUserEmail
);

return notionUser ? [{ json: { notionId: notionUser.json.id } }] : [];
```

Example 2: Data transformation

To try the example yourself, download the example workflow and import it into n8n.

In the **Join items** Code node, enter this prompt:

> Return a single line of text that has all usernames listed with a comma. Each username should be enquoted with a double quotation mark.

Take a look at the code the AI generates.

This is the JavaScript you need:

```
const items = $input.all();
const usernames = items.map((item) => `"${item.json.username}"`);
const result = usernames.join(", ");
return [{ json: { usernames: result } }];
```

Example 3: Summarize data and create a Slack message

To try the example yourself, download the example workflow and import it into n8n.

In the **Summarize** Code node, enter this prompt:

> Create a markdown text for Slack that counts how many ideas, features and bugs have been submitted. The type of submission is saved in the property_type field. A feature has the property "Feature", a bug has the property "Bug" and an idea has the property "Bug". Also, list the five top submissions by vote in that message. Use "" as markdown for links.

Take a look at the code the AI generates.

This is the JavaScript you need:

```
const submissions = $input.all();

// Count the number of ideas, features, and bugs
let ideaCount = 0;
let featureCount = 0;
let bugCount = 0;

submissions.forEach((submission) => {
  switch (submission.json.property_type[0]) {
    case "Idea":
      ideaCount++;
      break;
    case "Feature":
      featureCount++;
      break;
    case "Bug":
      bugCount++;
```

```
      break;
    }
});

// Sort submissions by votes and take the top 5
const topSubmissions = submissions
  .sort((a, b) => b.json.property_votes - a.json.property_votes)
  .slice(0, 5);

let topSubmissionText = "";
topSubmissions.forEach((submission) => {
  topSubmissionText += `<${submission.json.url}|${submission.json.name}> with
${submission.json.property_votes} votes\n`;
});

// Construct the Slack message
const slackMessage = `*Summary of Submissions*\n
Ideas: ${ideaCount}\n
Features: ${featureCount}\n
Bugs: ${bugCount}\n
Top 5 Submissions:\n
${topSubmissionText}`;

return [{ json: { slackMessage } }];
```

Reference incoming node data explicitly

If your incoming data contains nested fields, using dot notation to reference them can help the AI understand what data you want.

To try the example yourself, download the example workflow and import it into n8n.

In the second Code node, enter this prompt:

> The data in "Mock data" represents a list of people. For each person, return a new item containing personal_info.first_name and work_info.job_title.

This is the JavaScript you need:

```
const items = $input.all();
const newItems = items.map((item) => {
  const firstName = item.json.personal_info.first_name;
  const jobTitle = item.json.work_info.job_title;
  return {
    json: {
      firstName,
      jobTitle,
    },
  };
});
return newItems;
```

Related resources

Pluralsight offer a short guide on How to use ChatGPT to write code, which includes example prompts.

Fixing the code

The AI-generated code may work without any changes, but you may have to edit it. You need to be aware of n8n's Data structure. You may also find n8n's built-in methods and variables useful.

Check incoming data

At times, you may want to check the incoming data. If the incoming data doesn't match a condition, you may want to return a different value. For example, you want to check if a variable from the previous node is empty and return a string if it's empty. Use the following code snippet to return `not found` if the variable is empty.

```
{{$json["variable_name"]? $json["variable_name"] :"not found"}}
```

The above expression uses the ternary operator. You can learn more about the ternary operator here.

Convenience methods

n8n provides these methods to make it easier to perform common tasks in expressions.

Python support

You can use Python in the Code node. It isn't available in expressions.

JavaScript

Method	Description	Available in Code node?
`$evaluateExpression(expression: string, itemIndex?: number)`	Evaluates a string as an expression. If you don't provide `itemIndex`, n8n uses the data from item 0 in the Code node.	✅
`$ifEmpty(value, defaultValue)`	The `$ifEmpty()` function takes two parameters, tests the first to check if it's empty, then returns either the parameter (if not empty) or the second parameter (if the first is empty). The first parameter is empty if it's: • `undefined` • `null` • An empty string `''` • An array where `value.length` returns `false` • An object where `Object.keys(value).length` returns `false`	✅

Method	Description	Available in Code node?
`$if()`	The `$if()` function takes three parameters: a condition, the value to return if true, and the value to return if false.	❌
`$max()`	Returns the highest of the provided numbers.	❌
`$min()`	Returns the lowest of the provided numbers.	❌

Python

Method	Description
`_evaluateExpression(expression: string, itemIndex?: number)`	Evaluates a string as an expression. If you don't provide `itemIndex`, n8n uses the data from item 0 in the Code node.
`_ifEmpty(value, defaultValue)`	The `_ifEmpty()` function takes two parameters, tests the first to check if it's empty, then returns either the parameter (if not empty) or the second parameter (if the first is empty). The first parameter is empty if it's: • `undefined` • `null` • An empty string `''` • An array where `value.length` returns `false` • An object where `Object.keys(value).length` returns `false`

Custom variables

Feature availability

- Available on Self-hosted Enterprise and Pro Cloud plans.
- You need access to the n8n instance owner account to create and edit variables. All users can use existing variables.

Available in version 0.225.0 and above.

Custom variables are read-only variables that you can use to store and reuse values in n8n workflows.

Variables are shared

When you create a variable, it's available to everyone on your n8n instance.

Create variables

To create a new variable:

1. On the **Variables** page, select **Add Variable**.
2. Enter a **Key** and **Value**. The maximum key length is 50 characters, and the maximum value length is 220 characters. n8n limits the characters you can use in the key and value to lowercase and uppercase letters, numbers, and underscores (`A-Z`, `a-z`, `0-9`, `_`).
3. Select **Save**. The variable is now available for use in all workflows in the n8n instance.

Edit and delete variables

To edit or delete a variable:

1. On the **Variables** page, hover over the variable you want to change.
2. Select **Edit** or **Delete**.

Use variables in workflows

You can access variables in the Code node and in expressions:

```
// Access a variable
$vars.<variable-name>
```

All variables are strings.

During workflow execution, n8n replaces the variables with the variable value. If the variable has no value, n8n treats its value as `undefined`. Workflows don't automatically fail in this case.

Variables are read-only. You must use the UI to change the values. If you need to set and access custom data within your workflow, use Workflow static data.

Current node input

Methods for working with the input of the current node. Some methods and variables aren't available in the Code node.

Python support

You can use Python in the Code node. It isn't available in expressions.

JavaScript

Method	Description	Available in Code node?
`$binary`	Shorthand for `$input.item.binary`. Incoming binary data from a node	❌
`$input.item`	The input item of the current node that's being processed. Refer to Item linking for more information on paired items and item linking.	✅
`$input.all()`	All input items in current node.	✅
`$input.first()`	First input item in current node.	✅
`$input.last()`	Last input item in current node.	✅
`$input.params`	Object containing the query settings of the previous node. This includes data such as the operation it ran, result limits, and so on.	✅
`$json`	Shorthand for `$input.item.json`. Incoming JSON data from a node. Refer to Data structure for information on item structure.	✅ (when running once for each item)

Method	Description	Available in Code node?
`$input.context.noItemsLeft`	Boolean. Only available when working with the Loop Over Items node. Provides information about what's happening in the node. Use this to determine whether the node is still processing items.	✅

Python

Method	Description
`_input.item`	The input item of the current node that's being processed. Refer to Item linking for more information on paired items and item linking.
`_input.all()`	All input items in current node.
`_input.first()`	First input item in current node.
`_input.last()`	Last input item in current node.
`_input.params`	Object containing the query settings of the previous node. This includes data such as the operation it ran, result limits, and so on.
`_json`	Shorthand for `_input.item.json`. Incoming JSON data from a node. Refer to Data structure for information on item structure. Available when you set **Mode** to **Run Once for Each Item**.
`_input.context.noItemsLeft`	Boolean. Only available when working with the Loop Over Items node. Provides information about what's happening in the node. Use this to determine whether the node is still processing items.

Arrays

A reference document listing built-in convenience functions to support data transformation in expressions for arrays.

JavaScript in expressions

You can use any JavaScript in expressions. Refer to Expressions for more information.

`average(): Number`

Returns the value of elements in an array

`chunk(size: Number): Array`

Splits arrays into chunks with a length of size

Function parameters

`size` REQUIRED NUMBER

The size of each chunk.

`compact(): Array`

Removes empty values from the array.

`difference(arr: Array): Array`

Compares two arrays. Returns all elements in the base array that aren't present in arr.

Function parameters

`arr` REQUIRED ARRAY
The array to compare to the base array.

`intersection(arr: Array): Array`

Compares two arrays. Returns all elements in the base array that are present in arr.

Function parameters

`arr` REQUIRED ARRAY
The array to compare to the base array.

`first(): Array item`

Returns the first element of the array.

`isEmpty(): Boolean`

Checks if the array doesn't have any elements.

`isNotEmpty(): Boolean`

Checks if the array has elements.

`last(): Array item`

Returns the last element of the array.

`max(): Number`

Returns the highest value in an array.

`merge(arr: Array): Array`

Merges two Object-arrays into one array by merging the key-value pairs of each element.

Function parameters

`arr` REQUIRED ARRAY
The array to merge into the base array.

`min(): Number`

Gets the minimum value from a number-only array.

`pluck(fieldName?: String): Array`

Returns an array of Objects where keys equal the given field names.

Function parameters

`fieldname` OPTIONAL STRING
The key(s) you want to retrieve. You can enter as many keys as you want, as comma-separated strings.

`randomItem(): Array item`

Returns a random element from an array.

`removeDuplicates(key?: String): Array`

Removes duplicates from an array.

Function parameters

`key` OPTIONAL STRING
A key, or comma-separated list of keys, to check for duplicates.

`renameKeys(from: String, to: String): Array`

Renames all matching keys in the array. You can rename more than one key by entering a series of comma separated strings, in the pattern oldKeyName, newKeyName.

Function parameters

`from` REQUIRED STRING

The key you want to rename.

`to` REQUIRED STRING

The new name.

`smartJoin(keyField: String, nameField: String): Array`

Operates on an array of objects where each object contains key-value pairs. Creates a new object containing key-value pairs, where the key is the value of the first pair, and the value is the value of the second pair. Removes non-matching and empty values and trims any whitespace before joining.

Function parameters

keyfield REQUIRED STRING

The key to join.

namefield REQUIRED STRING

The value to join.

Example

BASIC USAGE

```
// Input
{{ [{"type":"fruit", "name":"apple"},{"type":"vegetable",
"name":"carrot"} ].smartJoin("type","name") }}
// Output
[Object: {"fruit":"apple","vegetable":"carrot"}]
```

---

sum(): Number

Returns the total sum all the values in an array of parsable numbers.

---

toJsonString(): String

Convert an array to a JSON string. Equivalent of `JSON.stringify`.

`union(arr: Array): Array`

Concatenates two arrays and then removes duplicate.

Function parameters

`arr` REQUIRED ARRAY

The array to compare to the base array.

`unique(key?: String): Array`

Remove duplicates from an array.

Function parameters

`key` OPTIONAL STRING

A key, or comma-separated list of keys, to check for duplicates.

Booleans

A reference document listing built-in convenience functions to support data transformation in expressions for arrays.

JavaScript in expressions

You can use any JavaScript in expressions. Refer to Expressions for more information.

`toInt(): Number`

Convert a boolean to a number. `false` converts to `0`, `true` converts to `1`.

Dates

A reference document listing built-in convenience functions to support data transformation in expressions for dates.

JavaScript in expressions

You can use any JavaScript in expressions. Refer to Expressions for more information.

`beginningOf(unit?: DurationUnit): Date`

Transforms a Date to the start of the given time period. Returns either a JavaScript Date or Luxon Date, depending on input.

Function parameters

`unit` OPTIONAL STRING ENUM

A valid string specifying the time unit.
Default: `week`
One of: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`

`endOfMonth(): Date`

Transforms a Date to the end of the month.

`extract(datePart?: DurationUnit): Number`

Extracts the part defined in datePart from a Date. Returns either a JavaScript Date or Luxon Date, depending on input.

Function parameters

`datepart` OPTIONAL STRING ENUM
A valid string specifying the time unit.
Default: `week`
One of: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`

`format(fmt: TimeFormat): String`

Formats a Date in the given structure

Function parameters

`fmt` REQUIRED STRING ENUM
A valid string specifying the time format. Refer to Luxon | Table of tokens for formats.

`isBetween(date1: Date | DateTime, date2: Date | DateTime): Boolean`

Checks if a Date is between two given dates.

Function parameters

`date1` REQUIRED DATE OR DATETIME

The first date in the range.

`date2` REQUIRED DATE OR DATETIME

The last date in the range.

`isDst(): Boolean`

Checks if a Date is within Daylight Savings Time.

`isInLast(n?: Number, unit?: DurationUnit): Boolean`

Checks if a Date is within a given time period.

Function parameters

`n` OPTIONAL NUMBER

The number of units. For example, to check if the date is in the last nine weeks, enter 9.
Default: `0`

unit OPTIONAL STRING ENUM

A valid string specifying the time unit.
Default: `minutes`
One of: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`

---

isWeekend(): Boolean

Checks if the Date falls on a Saturday or Sunday.

---

minus(n: Number, unit?: DurationUnit): Date

Subtracts a given time period from a Date. Returns either a JavaScript Date or Luxon Date, depending on input.

Function parameters

n REQUIRED NUMBER

The number of units. For example, to subtract nine seconds, enter 9 here.

unit OPTIONAL STRING ENUM

A valid string specifying the time unit.

Default: `milliseconds`
One of: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`

`plus(n: Number, unit?: DurationUnit): Date`

Adds a given time period to a Date. Returns either a JavaScript Date or Luxon Date, depending on input.

Function parameters

n REQUIRED NUMBER
The number of units. For example, to add nine seconds, enter 9 here.

`unit` OPTIONAL STRING ENUM
A valid string specifying the time unit.
Default: `milliseconds`
One of: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`

`toDateTime(): Date`

Converts a JavaScript date to a Luxon date object.

Numbers

A reference document listing built-in convenience functions to support data transformation in expressions for numbers.

JavaScript in expressions

You can use any JavaScript in expressions. Refer to Expressions for more information.

`ceil(): Number`

Rounds up a number to a whole number.

`floor(): Number`

Rounds down a number to a whole number.

`format(locales?: LanguageCode, options?: FormatOptions): String`

This is a wrapper around Intl.NumberFormat(). Returns a formatted string of a number based on the given LanguageCode and FormatOptions. When no arguments are given, transforms the number in a like format 1.234.

Function parameters

`locales` OPTIONAL STRING

An IETF BCP 47 language tag.
Default: `en-US`

`options` OPTIONAL OBJECT

Configure options for number formatting. Refer to MDN | Intl.NumberFormat() for more information.

`isEven(): Boolean`

Returns true if the number is even. Only works on whole numbers.

`isOdd(): Boolean`

Returns true if the number is odd. Only works on whole numbers.

`round(decimalPlaces?: Number): Number`

Returns the value of a number rounded to the nearest whole number, unless a decimal place is specified.

Function parameters

`decimalplaces` OPTIONAL NUMBER
How many decimal places to round to.
Default: `0`

`toBoolean(): Boolean`

Converts a number to a boolean. `0` converts to `false`. All other values convert to `true`.

`toDateTime(format?: String): Date`

Converts a number to a Luxon date object.

Function parameters

`format` OPTIONAL STRING ENUM
Can be `ms` (milliseconds), `s` (seconds), or `excel` (Excel 1900). Defaults to milliseconds.

Objects

A reference document listing built-in convenience functions to support data transformation in expressions for objects.

JavaScript in expressions

You can use any JavaScript in expressions. Refer to Expressions for more information.

`isEmpty(): Boolean`

Checks if the Object has no key-value pairs.

`merge(object: Object): Object`

Merges two Objects into a single Object using the first as the base Object. If a key exists in both Objects, the key in the base Object takes precedence.

Function parameters

`object` REQUIRED OBJECT
The Object to merge with the base Object.

`hasField(fieldName: String): Boolean`

`fieldname` REQUIRED STRING

The field to search for.

`removeField(key: String): Object`

Removes a given field from the Object

Function parameters

`key` REQUIRED STRING

The field key of the field to remove.

`removeFieldsContaining(value: String): Object`

Removes fields with a given value from the Object.

Function parameters

`value` REQUIRED STRING

The field value of the field to remove.

Function parameters

`value` REQUIRED STRING

The field value of the field to keep.

`compact(): Object`

Removes empty values from an Object.

`toJsonString(): String`

Convert an object to a JSON string. Equivalent of `JSON.stringify`.

`urlEncode(): String`

Transforms an Object into a URL parameter list. Only top-level keys are supported.

Strings

A reference document listing built-in convenience functions to support data transformation in expressions for strings.

JavaScript in expressions

You can use any JavaScript in expressions. Refer to Expressions for more information.

base64Encode(): A base64 encoded string.

Encode a string as base64.

---

base64Decode(): A plain string.

Convert a base64 encoded string to a normal string.

---

extractDomain(): String

Extracts a domain from a string containing a valid URL. Returns undefined if none is found.

`extractEmail(): String`

Extracts an email from a string. Returns undefined if none is found.

`extractUrl(): String`

Extracts a URL from a string. Returns undefined if none is found.

`extractUrlPath(): String`

Extract the path but not the root domain from a URL. For example, `"https://example.com/orders/1/details".extractUrlPath()` returns `"/orders/1/details/"`.

`hash(algo?: Algorithm): String`

Returns a string hashed with the given algorithm.

Function parameters

`algo` OPTIONAL STRING ENUM

Which hashing algorithm to use.

Default: `md5`

One of: `md5`, `base64`, `sha1`, `sha224`, `sha256`, `sha384`, `sha512`, `sha3`, `ripemd160`

`isDomain(): Boolean`

Checks if a string is a domain.

`isEmail(): Boolean`

Checks if a string is an email.

`isEmpty(): Boolean`

Checks if a string is empty.

`isNotEmpty(): Boolean`

Checks if a string has content.

`isNumeric(): Boolean`

Checks if a string only contains digits.

`isUrl(): Boolean`

Checks if a string is a valid URL.

`parseJson(): Object`

Equivalent of `JSON.parse()`. Parses a string as a JSON object.

`quote(mark?: String): String`

Returns a string wrapped in the quotation marks. Default quotation is `"`.

Function parameters

`mark` OPTIONAL STRING
Which quote mark style to use.
Default: `"`

`removeMarkdown(): String`

Removes Markdown formatting from a string.

`replaceSpecialChars(): String`

Replaces non-ASCII characters in a string with an ASCII representation.

`removeTags(): String`

Remove tags, such as HTML or XML, from a string.

`toBoolean(): Boolean`

Convert a string to a boolean. `"false"`, `"0"`, `""`, and `"no"` convert to `false`.

`toDateTime(): Date`

Converts a string to a Luxon date object.

`toDecimalNumber(): Number`

See toFloat

`toFloat(): Number`

Converts a string to a decimal number.

`toInt(): Number`

Converts a string to an integer.

`toSentenceCase(): String`

Formats a string to sentence case.

`toSnakeCase(): String`

Formats a string to snake case.

`toTitleCase(): String`

Formats a string to title case. Will not change already uppercase letters to prevent losing information from acronyms and trademarks such as iPhone or FAANG.

`toWholeNumber(): Number`

Converts a string to a whole number.

`urlDecode(entireString?: Boolean): String`

Decodes a URL-encoded string. It decodes any percent-encoded characters in the input string, and replaces them with their original characters.

Function parameters

`entirestring` OPTIONAL BOOLEAN
Whether to decode characters that are part of the URI syntax (true) or not (false).

`urlEncode(entireString?: Boolean): String`

Encodes a string to be used/included in a URL.

Function parameters

`entirestring` OPTIONAL BOOLEAN
Whether to encode characters that are part of the URI syntax (true) or not (false).

Methods for working with date and time.

Python support

You can use Python in the Code node. It isn't available in expressions.

JavaScript

Method	Description	Available in Code node?
`$now`	A Luxon object containing the current timestamp. Equivalent to `DateTime.now()`.	✅
`$today`	A Luxon object containing the current timestamp, rounded down to the day. Equivalent to `DateTime.now().set({ hour: 0, minute: 0, second: 0, millisecond: 0 })`.	✅

Python

Method	Description
`_now`	A Luxon object containing the current timestamp. Equivalent to `DateTime.now()`.
`_today`	A Luxon object containing the current timestamp, rounded down to the day. Equivalent to `DateTime.now().set({ hour: 0, minute: 0, second: 0, millisecond: 0 })`.

n8n passes dates between nodes as strings, so you need to parse them. Luxon helps you do this. Refer to Date and time with Luxon for more information.

n8n provides built-in convenience functions to support data transformation in expressions for dates. Refer to Data transformation functions | Dates for more information.

execution

execution.id

Contains the unique ID of the current workflow execution.

JavaScript

```
let executionId = $execution.id;
```

Python

```
executionId = _execution.id
```

execution.resumeUrl

The webhook URL to call to resume a waiting workflow.

See the Wait > On webhook call documentation to learn more.

execution.customData

This is only available in the Code node.

JavaScript

```
// Set a single piece of custom execution data
$execution.customData.set("key", "value");

// Set the custom execution data object
$execution.customData.setAll({"key1": "value1", "key2":
```

```
"value2"})

// Access the current state of the object during the
execution
var customData = $execution.customData.getAll()

// Access a specific value set during this execution
var customData = $execution.customData.get("key")
```

Python

```
# Set a single piece of custom execution data
_execution.customData.set("key", "value");

# Set the custom execution data object
_execution.customData.setAll({"key1": "value1", "key2":
"value2"})

# Access the current state of the object during the
execution
customData = _execution.customData.getAll()

# Access a specific value set during this execution
customData = _execution.customData.get("key")
```

Refer to Custom executions data for more information.

Expressions

In version 1.9.0 n8n changed the templating language for expressions

If you have issues with expressions in 1.9.0:

- Please report the issue on the forums.
- Self-hosted users can switch back to RiotTmpl: set `N8N_EXPRESSION_EVALUATOR` to `tmpl`. Refer to Environment variables for more information on configuring self-hosted n8n.

Use expressions to set node parameters dynamically based on data from:

- Previous nodes
- The workflow
- Your n8n environment

You can execute JavaScript within an expression.

n8n created and uses a templating language called Tournament, and extends it with custom methods and variables and data transformation functions that help with common tasks, such as retrieving data from other

nodes, or accessing metadata.

n8n supports two libraries:

- Luxon, for working with data and time.
- JMESPath, for querying JSON.

No Python support

Expressions must use JavaScript.

Data in n8n

When writing expressions, it's helpful to understand data structure and behavior in n8n. Refer to Data for more information on working with data in your workflows.

Writing expressions

To use an expression to set a parameter value:

1. Hover over the parameter where you want to use an expression.
2. Select **Expressions** in the **Fixed/Expression** toggle.

3. Write your expression in the parameter, or select **Open expression editor** to open the expressions editor. If you use the expressions editor, you can browse the available data in the **Variable selector**. All expressions have the format `{{ your expression here }}`.

Example: Get data from webhook body

Consider the following scenario: you have a webhook trigger that receives data through the webhook body. You want to extract some of that data for use in the workflow.

Your webhook data looks similar to this:

```
[
  {
    "headers": {
      "host": "n8n.instance.address",
      ...
    },
    "params": {},
    "query": {},
    "body": {
      "name": "Jim",
      "age": 30,
      "city": "New York"
    }
```

```
    }
]
```

In the next node in the workflow, you want to get just the value of `city`. You can use the following expression:

```
{{$json.body.city}}
```

This expression:

1. Accesses the incoming JSON-formatted data using n8n's custom `$json` variable.
2. Finds the value of `city` (in this example, "New York"). Note that this example uses JMESPath syntax to query the JSON data. You can also write this expression as `{{$json['body']['city']}}`.

Example: Writing longer JavaScript

An expression contains one line of JavaScript. This means you can do things like variable assignments or multiple standalone operations.

To understand the limitations of JavaScript in expressions, and start thinking about workarounds, look at the following two pieces of code. Both code examples use the Luxon date and time library to find the time between two dates in months, and encloses the code in handlebar brackets, like an expression.

However, the first example isn't a valid n8n expression:

```
// This example is split over multiple lines for readability
// It's still invalid when formatted as a single line
{{
  function example() {
    let end = DateTime.fromISO('2017-03-13');
    let start = DateTime.fromISO('2017-02-13');
    let diffInMonths = end.diff(start, 'months');
    return diffInMonths.toObject();
  }
  example();
}}
```

While the second example is valid:

```
{{DateTime.fromISO('2017-03-13').diff(DateTime.fromISO('2017-02-13'), 'months').toObject()}}
```

Get number of items returned by the previous node

To get the number of items returned by the previous node:

JavaScript

```javascript
if (Object.keys(items[0].json).length === 0) {
return [
  {
    json: {
      results: 0,
    }
  }
]
}
return [
  {
    json: {
      results: items.length,
    }
  }
];
```

The output will be similar to the following.

```
[
  {
    "results": 8
  }
]
```

Python

```python
if len(items[0].json) == 0:
  return [
    {
      "json": {
        "results": 0,
      }
    }
  ]
else:
  return [
    {
      "json": {
        "results": items.length,
      }
    }
  ]
```

The output will be similar to the following.

```
[
  {
    "results": 8
  }
]
```

Get the binary data buffer

The binary data buffer contains all the binary file data processed by a workflow. You need to access it if you want to perform operations on the binary data, such as:

- Manipulating the data: for example, adding column headers to a CSV file.
- Using the data in calculations: for example, calculating a hash value based on it.
- Complex HTTP requests: for example, combining file upload with sending other data formats.

Not available in Python

`getBinaryDataBuffer()` isn't supported when using Python.

You can access the buffer using n8n's `getBinaryDataBuffer()` function:

```
/*
* itemIndex: number. The index of the item in the input data.
* binaryPropertyName: string. The name of the binary property.
* The default in the Read/Write File From Disk node is 'data'.
*/
let binaryDataBufferItem = await this.helpers.getBinaryDataBuffer(itemIndex, binaryPropertyName);
```

For example:

```
let binaryDataBufferItem = await this.helpers.getBinaryDataBuffer(0, 'data');
// Returns the data in the binary buffer for the first input item
```

You should always use the `getBinaryDataBuffer()` function, and avoid using older methods of directly accessing the buffer, such as targeting it with expressions like `items[0].binary.data.data`.

`getWorkflowStaticData(type)`

This gives access to the static workflow data.

Experimental feature

Static data isn't available when testing workflows. The workflow must be active and called by a trigger or webhook to save static data.

You can save data directly in the workflow. This data should be small.

As an example: you can save a timestamp of the last item processed from an RSS feed or database. It will always return an object. Properties can then read, delete or set on that object. When the workflow execution succeeds, n8n checks automatically if the data has changed and saves it, if necessary.

There are two types of static data, global and node. Global static data is the same in the whole workflow. Every node in the workflow can access it. The node static data is unique to the node. Only the node that set it can retrieve it again.

Example with global data:

JavaScript

```
// Get the global workflow static data
const workflowStaticData =
$getWorkflowStaticData('global');

// Access its data
const lastExecution = workflowStaticData.lastExecution;
```

```
// Update its data
workflowStaticData.lastExecution = new Date().getTime();

// Delete data
delete workflowStaticData.lastExecution;
```

Python

```
# Get the global workflow static data
workflowStaticData = _getWorkflowStaticData('global')

# Access its data
lastExecution = workflowStaticData.lastExecution

# Update its data
workflowStaticData.lastExecution = new Date().getTime()

# Delete data
delete workflowStaticData.lastExecution
```

Example with node data:

JavaScript

```
// Get the static data of the node
const nodeStaticData = $getWorkflowStaticData('node');

// Access its data
const lastExecution = nodeStaticData.lastExecution;

// Update its data
nodeStaticData.lastExecution = new Date().getTime();

// Delete data
delete nodeStaticData.lastExecution;
```

Python

```
# Get the static data of the node
nodeStaticData = _getWorkflowStaticData('node')

```

```
# Access its data
lastExecution = nodeStaticData.lastExecution

# Update its data
nodeStaticData.lastExecution = new Date().getTime()

# Delete data
delete nodeStaticData.lastExecution
```

Date and time with Luxon

Luxon is a JavaScript library that makes it easier to work with date and time. For full details of how to use Luxon, refer to Luxon's documentation.

n8n passes dates between nodes as strings, so you need to parse them. Luxon makes this easier.

Python support

Luxon is a JavaScript library. The two convenience variables created by n8n are available when using Python in the Code node, but their functionality is limited:

- You can't perform Luxon operations on these variables. For example, there is no Python equivalent for `$today.minus(...)`.
- The generic Luxon functionality, such as Convert date string to Luxon, isn't available for Python users.

Variables

n8n uses Luxon to provide two custom variables:

- `now`: a Luxon object containing the current timestamp. Equivalent to `DateTime.now()`.
- `today`: a Luxon object containing the current timestamp, rounded down to the day. Equivalent to `DateTime.now().set({ hour: 0, minute: 0, second: 0, millisecond: 0 })`.

Note that these variables can return different time formats when cast as a string. This is the same behavior as Luxon's `DateTime.now()`.

Expressions (JavaScript)

```
{{$now}}
// n8n displays the ISO formatted timestamp
// For example 2022-03-09T14:02:37.065+00:00
{{"Today's date is " + $now}}
// n8n displays "Today's date is <unix timestamp>"
// For example "Today's date is 1646834498755"
```

Code node (JavaScript)

```
$now
// n8n displays <ISO formatted timestamp>
// For example 2022-03-09T14:00:25.058+00:00
let rightNow = "Today's date is " + $now
// n8n displays "Today's date is <unix timestamp>"
// For example "Today's date is 1646834498755"
```

Code node (Python)

```
_now
# n8n displays <ISO formatted timestamp>
# For example 2022-03-09T14:00:25.058+00:00
rightNow = "Today's date is " + str(_now)
# n8n displays "Today's date is <unix timestamp>"
# For example "Today's date is 1646834498755"
```

n8n provides built-in convenience functions to support data transformation in expressions for dates. Refer to Data transformation functions | Dates for more information.

Date and time behavior in n8n

Be aware of the following:

- In a workflow, n8n converts dates and times to strings between nodes. Keep this in mind when doing arithmetic on dates and times from other nodes.
- With vanilla JavaScript, you can convert a string to a date with `new Date('2019-06-23')`. In Luxon, you must use a function explicitly stating the format, such as `DateTime.fromISO('2019-06-23')` or `DateTime.fromFormat("23-06-2019", "dd-MM-yyyy")`.

Setting the timezone in n8n

Luxon uses the n8n timezone. This value is either:

- Default: `America/New York`
- A custom timezone for your n8n instance, set using the `GENERIC_TIMEZONE` environment variable.
- A custom timezone for an individual workflow, configured in workflow settings.

Common tasks

This section provides examples for some common operations. More examples, and detailed guidance, are available in Luxon's own documentation.

Convert date string to Luxon

You can convert date strings and other date formats to a Luxon DateTime object. You can convert from standard formats and from arbitrary strings.

A difference between Luxon DateTime and JavaScript Date

With vanilla JavaScript, you can convert a string to a date with `new Date('2019-06-23')`. In Luxon, you must use a function explicitly stating the format, such as `DateTime.fromISO('2019-06-23')` or `DateTime.fromFormat("23-06-2019", "dd-MM-yyyy")`.

If you have a date in a supported standard technical format:

Most dates use `fromISO()`. This creates a Luxon DateTime from an ISO 8601 string. For example:

Expressions (JavaScript)

```
{{DateTime.fromISO('2019-06-23T00:00:00.00')}}
```

Code node (JavaScript)

```
let luxonDateTime = DateTime.fromISO('2019-06-23T00:00:00.00')
```

Luxon's API documentation has more information on fromISO.

Luxon provides functions to handle conversions for a range of formats. Refer to Luxon's guide to Parsing technical formats for details.

If you have a date as a string that doesn't use a standard format:

Use Luxon's Ad-hoc parsing. To do this, use the `fromFormat()` function, providing the string and a set of tokens that describe the format.

For example, you have n8n's founding date, 23rd June 2019, formatted as `23-06-2019`. You want to turn this into a Luxon object:

Expressions (JavaScript)

```
{{DateTime.fromFormat("23-06-2019", "dd-MM-yyyy")}}
```

Code node (JavaScript)

```
let newFormat = DateTime.fromFormat("23-06-2019", "dd-MM-yyyy")
```

When using ad-hoc parsing, note Luxon's warning about Limitations. If you see unexpected results, try their Debugging guide.

Get n days from today

Get a number of days before or after today.

Expressions (JavaScript)

For example, you want to set a field to always show the date seven days before the current date.

In the expressions editor, enter:

```
{{$today.minus({days: 7})}}
```

On the 23rd June 2019, this returns `[Object: "2019-06-16T00:00:00.000+00:00"]`.

This example uses n8n's custom variable `$today` for convenience. It's the equivalent of `DateTime.now().set({ hour: 0, minute: 0, second: 0, millisecond: 0 }).minus({days: 7})`.

Code node (JavaScript)

For example, you want a variable containing the date seven days before the current date.

In the code editor, enter:

```
let sevenDaysAgo = $today.minus({days: 7})
```

On the 23rd June 2019, this returns `[Object: "2019-06-16T00:00:00.000+00:00"]`.

This example uses n8n's custom variable `$today` for convenience. It's the equivalent of `DateTime.now().set({ hour: 0, minute: 0, second: 0, millisecond: 0 }).minus({days: 7})`.

For more detailed information and examples, refer to:

- Luxon's guide to math
- Their API documentation on DateTime plus and DateTime minus

Create human-readable dates

In Get n days from today, the example gets the date seven days before the current date, and returns it as `[Object: "yyyy-mm-dd-T00:00:00.000+00:00"]` (for expressions) or `yyyy-mm-dd-T00:00:00.000+00:00` (in the Code node). To make this more readable, you can use Luxon's formatting functions.

For example, you want the field containing the date to be formatted as DD/MM/YYYY, so that on the 23rd June 2019, it returns `23/06/2019`.

This expression gets the date seven days before today, and converts it to the DD/MM/YYYY format.

Expressions (JavaScript)

```
{{$today.minus({days: 7}).toLocaleString()}}
```

Code node (JavaScript)

```
let readableSevenDaysAgo = $today.minus({days: 7}).toLocaleString()
```

You can alter the format. For example:

Expressions (JavaScript)

```
{{$today.minus({days: 7}).toLocaleString({month: 'long', day: 'numeric', year: 'numeric'})}}
```

On 23rd June 2019, this returns "16 June 2019".

Code node (JavaScript)

```
let readableSevenDaysAgo = $today.minus({days: 7}).toLocaleString({month: 'long', day: 'numeric', year: 'numeric'})
```

On 23rd June 2019, this returns "16 June 2019".

Refer to Luxon's guide on toLocaleString (strings for humans) for more information.

Get the time between two dates

To get the time between two dates, use Luxon's diffs feature. This subtracts one date from another and returns a duration.

For example, get the number of months between two dates:

Expressions (JavaScript)

```
{{DateTime.fromISO('2019-06-23').diff(DateTime.fromISO('2019-05-23'),
```

```
    'months').toObject()}}
```

This returns `[Object: {"months":1}]`.

Code node (JavaScript)

```
let monthsBetweenDates = DateTime.fromISO('2019-06-23').diff(DateTime.fromISO('2019-05-23'), 'months').toObject()
```

This returns `{"months":1}`.

Refer to Luxon's Diffs for more information.

A longer example: How many days to Christmas?

This example brings together several Luxon features, uses JMESPath, and does some basic string manipulation.

The scenario: you want a countdown to 25th December. Every day, it should tell you the number of days remaining to Christmas. You don't want to update it for next year - it needs to seamlessly work for every year.

Expressions (JavaScript)

```
{{"There are " + $today.diff(DateTime.fromISO($today.year + '-12-25'), 'days').toObject().days.toString().substring(1) + " days to Christmas!"}}
```

This outputs `"There are <number of days> days to Christmas!"`. For example, on 9th March, it outputs "There are 291 days to Christmas!".

A detailed explanation of what the expression does:

- `{{`: indicates the start of the expression.

- `"There are "`: a string.
- `+`: used to join two strings.
- `$today.diff()`: This is similar to the example in Get the time between two dates, but it uses n8n's custom `$today` variable.
- `DateTime.fromISO($today.year + '-12-25'), 'days'`: this part gets the current year using `$today.year`, turns it into an ISO string along with the month and date, and then takes the whole ISO string and converts it to a Luxon DateTime data structure. It also tells Luxon that you want the duration in days.
- `toObject()` turns the result of diff() into a more usable object. At this point, the expression returns `[Object: {"days":-<number-of-days>}]`. For example, on 9th March, `[Object: {"days":-291}]`.
- `.days` uses JMESPath syntax to retrieve just the number of days from the object. For more information on using JMESPath with n8n, refer to our JMESpath documentation. This gives you the number of days to Christmas, as a negative number.
- `.toString().substring(1)` turns the number into a string and removes the `-`.
- `+ " days to Christmas!"`: another string, with a `+` to join it to the previous string.
- `}}`: indicates the end of the expression.

Code node (JavaScript)

```
let daysToChristmas = "There are " +
$today.diff(DateTime.fromISO($today.year + '-12-25'),
'days').toObject().days.toString().substring(1) + " days to
Christmas!";
```

This outputs `"There are <number of days> days to Christmas!"`. For example, on 9th March, it outputs "There are 291 days to Christmas!".

A detailed explanation of what the code does:

- `"There are "`: a string.
- `+`: used to join two strings.
- `$today.diff()`: This is similar to the example in Get the time between two dates, but it uses n8n's custom `$today` variable.
- `DateTime.fromISO($today.year + '-12-25'), 'days'`: this part gets the current year using `$today.year`, turns it into an ISO string along with the month and date, and then takes the whole ISO string and converts it to a Luxon DateTime data structure. It also tells Luxon that you want the duration in days.
- `toObject()` turns the result of diff() into a more usable object. At this point, the expression returns `[Object: {"days":-<number-of-days>}]`. For example, on 9th March, `[Object: {"days":-291}]`.
- `.days` uses JMESPath syntax to retrieve just the number of days from the object. For more information on using JMESPath with n8n, refer to our JMESpath documentation. This gives you the number of days to Christmas, as a negative number.
- `.toString().substring(1)` turns the number into a string and removes the `-`.
- `+ " days to Christmas!"`: another string, with a `+` to join it to the previous string.

HTTP node variables

Variables for working with HTTP node requests and responses when using pagination.

Refer to HTTP Request for guidance on using the HTTP node, including configuring pagination.

Refer to HTTP Request node cookbook | Pagination for example pagination configurations.

HTTP node only

These variables are for use in expressions in the HTTP node. You can't use them in other nodes.

Variable	Description
`$pageCount`	The pagination count. Tracks how many pages the node has fetched.
`$request`	The request object sent by the HTTP node.
`$response`	The response object from the HTTP call. Includes `$response.body`, `$response.headers`, and `$response.statusCode`. The contents of `body` and `headers` depend on the data sent by the API.

JMESPath method

This is an n8n-provided method for working with the JMESPath library.

Python support

You can use Python in the Code node. It isn't available in expressions.

JavaScript

Method	Description	Available in Code node?
`$jmespath()`	Perform a search on a JSON object using JMESPath.	✅

Python

Method	Description
`_jmespath()`	Perform a search on a JSON object using JMESPath.

LangChain Code node methods

n8n provides these methods to make it easier to perform common tasks in the LangChain Code node.

LangChain Code node only

These variables are for use in expressions in the LangChain Code node. You can't use them in other nodes.

Method	Description
`this.addInputData(inputName, data)`	Populate the data of a specified non-main input. Useful for mocking data. • `inputName` is the input connection type, and must be one of: `ai_agent`, `ai_chain`, `ai_document`, `ai_embedding`, `ai_languageModel`, `ai_memory`, `ai_outputParser`, `ai_retriever`, `ai_textSplitter`, `ai_tool`, `ai_vectorRetriever`, `ai_vectorStore` • `data` contains the data you want to add. Refer to Data structure for information on the data structure expected by n8n.
`this.addOutputData(outputName, data)`	Populate the data of a specified non-main output. Useful for mocking data. • `outputName` is the input connection type, and must be one of: `ai_agent`, `ai_chain`, `ai_document`, `ai_embedding`, `ai_languageModel`, `ai_memory`, `ai_outputParser`, `ai_retriever`, `ai_textSplitter`, `ai_tool`, `ai_vectorRetriever`, `ai_vectorStore`

Method	Description
	• `data` contains the data you want to add. Refer to Data structure for information on the data structure expected by n8n.
`this.getInputConnectionData(inputName, itemIndex, inputIndex?)`	Get data from a specified non-main input. • `inputName` is the input connection type, and must be one of: `ai_agent`, `ai_chain`, `ai_document`, `ai_embedding`, `ai_languageModel`, `ai_memory`, `ai_outputParser`, `ai_retriever`, `ai_textSplitter`, `ai_tool`, `ai_vectorRetriever`, `ai_vectorStore` • `itemIndex` should always be `0` (this parameter will be used in upcoming functionality) • Use `inputIndex` if there is more than one node connected to the specified input.
`this.getInputData(inputIndex?, inputName?)`	Get data from the main input.
`this.getNode()`	Get the current node.
`this.getNodeOutputs()`	Get the outputs of the current node.
`this.getExecutionCancelSignal()`	Use this to stop the execution of a function when the workflow stops. In most cases n8n handles this, but you may need to use it if building your own chains or agents. It replaces the Cancelling a running LLMChain code that you'd use if building a LangChain application normally.

n8n metadata

Methods for working with n8n metadata.

This includes:

- Access to n8n environment variables for self-hosted n8n.
- Metadata about workflows, executions, and nodes.
- Information about instance Variables and External secrets.

Python support

You can use Python in the Code node. It isn't available in expressions.

JavaScript

Method	Description	Available in Code node?
`$env`	Contains n8n instance configuration environment variables.	✅
`$execution.customData`	Set and get custom execution data. Refer to Custom executions data for more information.	✅
`$execution.id`	The unique ID of the current workflow execution.	✅

Method	Description	Available in Code node?
`$execution.mode`	Whether the execution was triggered automatically, or by manually running the workflow. Possible values are `test` and `production`.	✅
`$execution.resumeUrl`	The webhook URL to call to resume a workflow waiting at a Wait node.	✅
`$getWorkflowStaticData(type)`	View an example. Static data doesn't persist when testing workflows. The workflow must be active and called by a trigger or webhook to save static data. This gives access to the static workflow data.	✅
`$("<node-name>").isExecuted`	Check whether a node has already executed.	✅
`$itemIndex`	The index of an item in a list of items.	❌
`$nodeVersion`	Get the version of the current node.	✅
`$prevNode.name`	The name of the node that the current input came from. When using the Merge node, note that `$prevNode` always uses the first input connector.	✅
`$prevNode.outputIndex`	The index of the output connector that the current input came from. Use this when the previous node had multiple outputs (such as an If or Switch node). When using the Merge	✅

Method	Description	Available in Code node?
	node, note that `$prevNode` always uses the first input connector.	
`$prevNode.runIndex`	The run of the previous node that generated the current input. When using the Merge node, note that `$prevNode` always uses the first input connector.	✅
`$runIndex`	How many times n8n has executed the current node. Zero-based (the first run is 0, the second is 1, and so on).	✅
`$secrets`	Contains information about your External secrets setup.	✅
`$vars`	Contains the Variables available in the active environment.	✅
`$version`	The node version.	❌
`$workflow.active`	Whether the workflow is active (true) or not (false).	✅
`$workflow.id`	The workflow ID.	✅
`$workflow.name`	The workflow name.	✅

Python

Method	Description
`_env`	Contains n8n instance configuration environment variables.
`_execution.customData`	Set and get custom execution data. Refer to Custom executions data for more information.
`_execution.id`	The unique ID of the current workflow execution.
`_execution.mode`	Whether the execution was triggered automatically, or by manually running the workflow. Possible values are `test` and `production`.
`_execution.resumeUrl`	The webhook URL to call to resume a workflow waiting at a Wait node.
`_getWorkflowStaticData(type)`	View an example. Static data doesn't persist when testing workflows. The workflow must be active and called by a trigger or webhook to save static data. This gives access to the static workflow data.
`_("<node-name>").isExecuted`	Check whether a node has already executed.
`_nodeVersion`	Get the version of the current node.
`_prevNode.name`	The name of the node that the current input came from. When using the Merge node, note that `_prevNode` always uses the first input connector.

Method	Description
`_prevNode.outputIndex`	The index of the output connector that the current input came from. Use this when the previous node had multiple outputs (such as an If or Switch node). When using the Merge node, note that `_prevNode` always uses the first input connector.
`_prevNode.runIndex`	The run of the previous node that generated the current input. When using the Merge node, note that `_prevNode` always uses the first input connector.
`_runIndex`	How many times n8n has executed the current node. Zero-based (the first run is 0, the second is 1, and so on).
`_secrets`	Contains information about your External secrets setup.
`_vars`	Contains the Variables available in the active environment.
`_workflow.active`	Whether the workflow is active (true) or not (false).
`_workflow.id`	The workflow ID.
`_workflow.name`	The workflow name.

Methods for working with the output of other nodes. Some methods and variables aren't available in the Code node.

Python support

You can use Python in the Code node. It isn't available in expressions.

JavaScript

Method	Description	Available in Code node?
`$("<node-name>").all(branchIndex?, runIndex?)`	Returns all items from a given node. If `branchIndex` isn't given it will default to the output that connects `node-name` with the node where you use the expression or code.	✅
`$("<node-name>").first(branchIndex?, runIndex?)`	The first item output by the given node. If `branchIndex` isn't given it will default to the output that connects `node-name` with the node where you use the expression or code.	✅
`$("<node-name>").last(branchIndex?, runIndex?)`	The last item output by the given node. If `branchIndex` isn't given it will default to the output that connects `node-name` with the node where you use the expression or code.	✅
`$("<node-name>").item`	The linked item. This is the item in the specified node used to produce the current item. Refer to Item linking for more information on item linking.	❌
`$("<node-name>").params`	Object containing the query settings of the given node. This includes data such as the operation it ran, result limits, and so on.	✅
`$("<node-name>").context`	Boolean. Only available when working with the Loop Over Items node. Provides information about what's	✅

`name>").itemMatching(currentNodeInputIndex)`	if you need to trace back from an input item.

Python

Method	Description	Available in Code node?
`_("<node-name>").all(branchIndex?, runIndex?)`	Returns all items from a given node. If `branchIndex` isn't given it will default to the output that connects `node-name` with the node where you use the expression or code.	✅
`_("<node-name>").first(branchIndex?, runIndex?)`	The first item output by the given node. If `branchIndex` isn't given it will default to the output that connects `node-name` with the node where you use the expression or code.	✅
`_("<node-name>").last(branchIndex?, runIndex?)`	The last item output by the given node. If `branchIndex` isn't given it will default to the output that connects `node-name` with the node where you use the expression or code.	✅
`_("<node-name>").item`	The linked item. This is the item in the specified node used to produce the current item. Refer to Item linking for more information on item linking.	❌
`_("<node-name>").params`	Object containing the query settings of the given node. This includes data such as the operation it ran, result limits, and so on.	✅
`_("<node-name>").context`	Boolean. Only available when working with the Loop Over Items node. Provides information about what's happening in the node. Use this to determine whether the node is still processing items.	✅

Output to the browser console with `console.log()` or `print()` in the Code node

You can use `console.log()` or `print()` in the Code node to help when writing and debugging your code.

For help opening your browser console, refer to this guide by Balsamiq.

console.log (JavaScript)

For technical information on `console.log()`, refer to the MDN developer docs.

For example, copy the following code into a Code node, then open your console and run the node:

```
let a = "apple";
console.log(a);
```

print (Python)

For technical information on `print()`, refer to the Real Python's guide.

For example, set your Code node **Language** to **Python**, copy the following code into the node, then open your console and run the node:

```
a = "apple"
print(a)
```

Handling an output of `[object Object]`

If the console displays `[object Object]` when you print, check the data type, then convert it as needed.

To check the data type:

```
print(type(myData))
```

JsProxy

If `type()` outputs `<class 'pyodide.ffi.JsProxy'>`, you need to convert the JsProxy to a native Python object using `to_py()`. This occurs when working with data in the n8n node data structure, such as node inputs and outputs. For example, if you want to print the data from a previous node in the workflow:

```
previousNodeData = _("<node-name>").all();
for item in previousNodeData:
  # item is of type <class 'pyodide.ffi.JsProxy'>
  # You need to convert it to a Dict
  itemDict = item.json.to_py()
  print(itemDict)
```

Refer to the Pyodide documentation on JsProxy for more information on this class.

Query JSON with JMESPath

JMESPath is a query language for JSON that you can use to extract and transform elements from a JSON document. For full details of how to use JMESPath, refer to the JMESPath documentation.

The `jmespath()` method

n8n provides a custom method, `jmespath()`. Use this method to perform a search on a JSON object using the JMESPath query language.

The basic syntax is:

JavaScript

```
$jmespath(object, searchString)
```

Python

```
_jmespath(object, searchString)
```

To help understand what the method does, here is the equivalent longer JavaScript:

```
var jmespath = require('jmespath');
jmespath.search(object, searchString);
```

Expressions must be single-line

The longer code example doesn't work in Expressions, as they must be single-line.

`object` is a JSON object, such as the output of a previous node. `searchString` is an expression written in the JMESPath query language. The JMESPath Specification provides a list of supported expressions, while their Tutorial and Examples provide interactive examples.

Search parameter order

The examples in the JMESPath Specification follow the pattern `search(searchString, object)`. The JMESPath JavaScript library, which n8n uses, supports `search(object, searchString)` instead. This means that when using examples from the JMESPath documentation, you may need to change the order of the search function parameters.

Common tasks

This section provides examples for some common operations. More examples, and detailed guidance, are available in JMESPath's own documentation.

When trying out these examples, you need to set the Code node **Mode** to **Run Once for Each Item**.

Apply a JMESPath expression to a collection of elements with projections

From the JMESPath projections documentation:

> Projections are one of the key features of JMESPath. Use it to apply an expression to a collection of elements. JMESPath supports five kinds of projections:

> - List Projections
> - Slice Projections
> - Object Projections
> - Flatten Projections
> - Filter Projections

The following example shows basic usage of list, slice, and object projections. Refer to the JMESPath projections documentation for detailed explanations of each projection type, and more examples.

Given this JSON from a webhook node:

```
[
  {
    "headers": {
      "host": "n8n.instance.address",
      ...
    },
    "params": {},
    "query": {},
    "body": {
      "people": [
        {
          "first": "James",
          "last": "Green"
        },
        {
          "first": "Jacob",
          "last": "Jones"
        },
        {
          "first": "Jayden",
          "last": "Smith"
        }
      ],
      "dogs": {
        "Fido": {
          "color": "brown",
```

```
            "age": 7
          },
          "Spot": {
            "color": "black and white",
            "age": 5
          }
        }
      }
    }
  ]
```

Retrieve a list of all the people's first names:

Expressions (JavaScript)

```
{{$jmespath($json.body.people, "[*].first" )}}
// Returns ["James", "Jacob", "Jayden"]
```

Code node (JavaScript)

```
let firstNames = $jmespath($json.body.people, "[*].first"
)
return {firstNames};
/* Returns:
[
  {
    "firstNames": [
      "James",
      "Jacob",
      "Jayden"
    ]
  }
]
*/
```

Code node (Python)

```
firstNames = _jmespath(_json.body.people, "[*].first" )
return {"firstNames":firstNames}
"""
Returns:
[
```

```
  {
    "firstNames": [
      "James",
      "Jacob",
      "Jayden"
    ]
  }
]
"""
```

Get a slice of the first names:

Expressions (JavaScript)

```
{{$jmespath($json.body.people, "[:2].first")}}
// Returns ["James", "Jacob"]
```

Code node (JavaScript)

```
let firstTwoNames = $jmespath($json.body.people, "
[:2].first");
return {firstTwoNames};
/* Returns:
[
  {
    "firstNames": [
      "James",
      "Jacob",
      "Jayden"
    ]
  }
]
*/
```

Code node (Python)

```
firstTwoNames = _jmespath(_json.body.people, "[:2].first"
)
return {"firstTwoNames":firstTwoNames}
"""
Returns:
[
```

```
  {
    "firstTwoNames": [
    "James",
    "Jacob"
    ]
  }
]
"""
```

Get a list of the dogs' ages using object projections:

Expressions (JavaScript)

```
{{$jmespath($json.body.dogs, "*.age")}}
// Returns [7,5]
```

Code node (JavaScript)

```
let dogsAges = $jmespath($json.body.dogs, "*.age");
return {dogsAges};
/* Returns:
[
  {
    "dogsAges": [
      7,
      5
    ]
  }
]
*/
```

Code node (Python)

```
dogsAges = _jmespath(_json.body.dogs, "*.age")
return {"dogsAges": dogsAges}
"""
Returns:
[
  {
    "dogsAges": [
      7,
      5
```

```
    ]
  }
]
"""
```

Select multiple elements and create a new list or object

Use Multiselect to select elements from a JSON object and combine them into a new list or object.

Given this JSON from a webhook node:

```
[
  {
    "headers": {
      "host": "n8n.instance.address",
      ...
    },
    "params": {},
    "query": {},
    "body": {
      "people": [
        {
          "first": "James",
          "last": "Green"
        },
        {
          "first": "Jacob",
          "last": "Jones"
        },
        {
          "first": "Jayden",
          "last": "Smith"
        }
      ],
      "dogs": {
        "Fido": {
          "color": "brown",
          "age": 7
        },
```

```
        "Spot": {
          "color": "black and white",
          "age": 5
        }
      }
    }
  }
]
```

Use multiselect list to get the first and last names and create new lists containing both names:

Expressions (JavaScript)

```
{{$jmespath($json.body.people, "[].[first, last]")}}
// Returns [["James","Green"],["Jacob","Jones"],
["Jayden","Smith"]]
```

Code node (JavaScript)

```
let newList = $jmespath($json.body.people, "[].[first,
last]");
return {newList};
/* Returns:
[
  {
    "newList": [
      [
        "James",
        "Green"
      ],
      [
        "Jacob",
        "Jones"
      ],
      [
        "Jayden",
        "Smith"
      ]
    ]
  }

```

```
]
*/
```

Code node (Python)

```python
newList = _jmespath(_json.body.people, "[].[first, last]")
return {"newList":newList}
"""
Returns:
[
  {
    "newList": [
      [
        "James",
        "Green"
      ],
      [
        "Jacob",
        "Jones"
      ],
      [
        "Jayden",
        "Smith"
      ]
    ]
  }
]
"""
```

An alternative to arrow functions in expressions

For example, generate some input data by returning the below code from the Code node:

```
return[
  {
    "json": {
      "num_categories": "0",
      "num_products": "45",
      "category_id": 5529735,
```

```
        "parent_id": 1407340,
        "pos_enabled": 1,
        "pos_favorite": 0,
        "name": "HP",
        "description": "",
        "image": ""
      }
    },
    {
      "json": {
        "num_categories": "0",
        "num_products": "86",
        "category_id": 5529740,
        "parent_id": 1407340,
        "pos_enabled": 1,
        "pos_favorite": 0,
        "name": "Lenovo",
        "description": "",
        "image": ""
      }
    }
  ]
```

You could do a search like "find the item with the name Lenovo and tell me their category ID."

```
{{ $jmespath($("Code").all(), "[?json.name=='Lenovo'].json.category_id") }}
```

").itemMatching(currentNodeinputIndex)`">

Retrieve linked items from earlier in the workflow

Every item in a node's input data links back to the items used in previous nodes to generate it. This is useful if you need to retrieve linked items from further back than the immediate previous node.

To access the linked items from earlier in the workflow, use `("<node-name>").itemMatching(currentNodeinputIndex)`.

For example, consider a workflow that does the following:

1. The Customer Datastore node generates example data:


```
[
  {
    "id": "23423532",
    "name": "Jay Gatsby",
    "email": "gatsby@west-egg.com",
    "notes": "Keeps asking about a green light??",
    "country": "US",
    "created": "1925-04-10"
  },
  {
    "id": "23423533",
    "name": "José Arcadio Buendía",
    "email": "jab@macondo.co",
    "notes": "Lots of people named after him. Very
confusing",
    "country": "CO",
    "created": "1967-05-05"
  },

```

```
    ...
  ]
```

2. The Edit Fields node simplifies this data:

```
[
  {
    "name": "Jay Gatsby"
  },
  {
    "name": "José Arcadio Buendía"
  },
    ...
]
```

3. The Code node restore the email address to the correct person:

```
[
  {
    "name": "Jay Gatsby",
    "restoreEmail": "gatsby@west-egg.com"
  },
  {
    "name": "José Arcadio Buendía",
    "restoreEmail": "jab@macondo.co"
  },
  ...
]
```

The Code node does this using the following code:

JavaScript

```
for(let i=0; i<$input.all().length; i++) {
  $input.all()[i].json.restoreEmail = $('Customer Datastore (n8n training)').itemMatching(i).json.email;
}
return $input.all();
```

Python

```
for i,item in enumerate(_input.all()):
  _input.all()[i].json.restoreEmail = _('Customer Datastore (n8n training)').itemMatching(i).json.email

return _input.all();
```

You can view and download the example workflow from n8n website | itemMatchin usage example .