

# **Responsys Personalization Language**

---

## **Quick Start Guide**

Update 18A

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.  
Last updated: 3/21/18

Information in this document is subject to change without notice. Data used as examples in this document is fictitious. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission of Oracle Responsys.

Address permission requests, comments, or suggestions about Oracle Responsys documentation by creating a MOS Service Request at <https://support.oracle.com>.

# Chapter 1. Introduction

This guide is an introduction to getting started with the new Responsys Personalization Language (RPL).

Although RPL has many programming capabilities, it is not a full-blown programming language like PHP or Java. It is implemented to work only within the Interact suite. Other than that, you can accomplish many programming tasks using this language.

## What is RPL?

RPL is the language used to create highly personalized messages across various channels, starting with email.

RPL markup can appear anywhere in any text-based file. The engine behind the language reads a text-based file and, when it recognizes certain pieces of text as markup, executes the code in the markup and inserts it into the output file. Other text is copied into the output file as is.

For a summary of RPL, see *Chapter 5: Quick RPL Reference*.

## What happened to built-in functions?

For a long time, personalization in Interact was based on what is commonly known as built-in functions. Built-in functions were developed through customer needs over the years and are a Responsys proprietary language. This language is limited as a programming language in its ability to do sophisticated personalization in an efficient manner. Over time, the next generation of the language was envisioned and it was decided to create a language more akin to a programming language to expand the power of personalization capabilities of Interact.

When the new Email Message Designer is turned on for an account, existing campaigns using the built-in functions will continue to work. Customers can create new campaigns that leverage the new Email Message Designer and RPL.

## Glossary

RPL introduces a few new terms listed below.

### Comment

Content that will not be included in the output. RPL comments are similar to HTML comments. RPL comments begin with `<#--` and end with `-->`. Everything between these tags will be excluded from the output.

### Directive

Instructions to RPL used in templates. Directives begin with `<#` or `<@`.

### Hash

A variable that acts as a container for other variables (known as *sub-variables*). Sub-variables of a hash are accessed by name.

### Interpolation

An instruction to convert an expression to text and to insert that text into the output. Interpolations begin with `${` and end with `}`. Note that interpolations do not obtain a value, they only execute an expression.

### Method

Calculates something based on given parameters and returns the result.

### Namespace

A set of available directives, methods, RPL built-ins, and additional structures/properties that you can use.

## Scalar

A variable that stores a single value. A scalar is of a specific type: string, number, date/time, or boolean.

## Sequence

A structure that stores sub-variables sequentially. Sub-variables in a sequence are accessed using a numerical index.

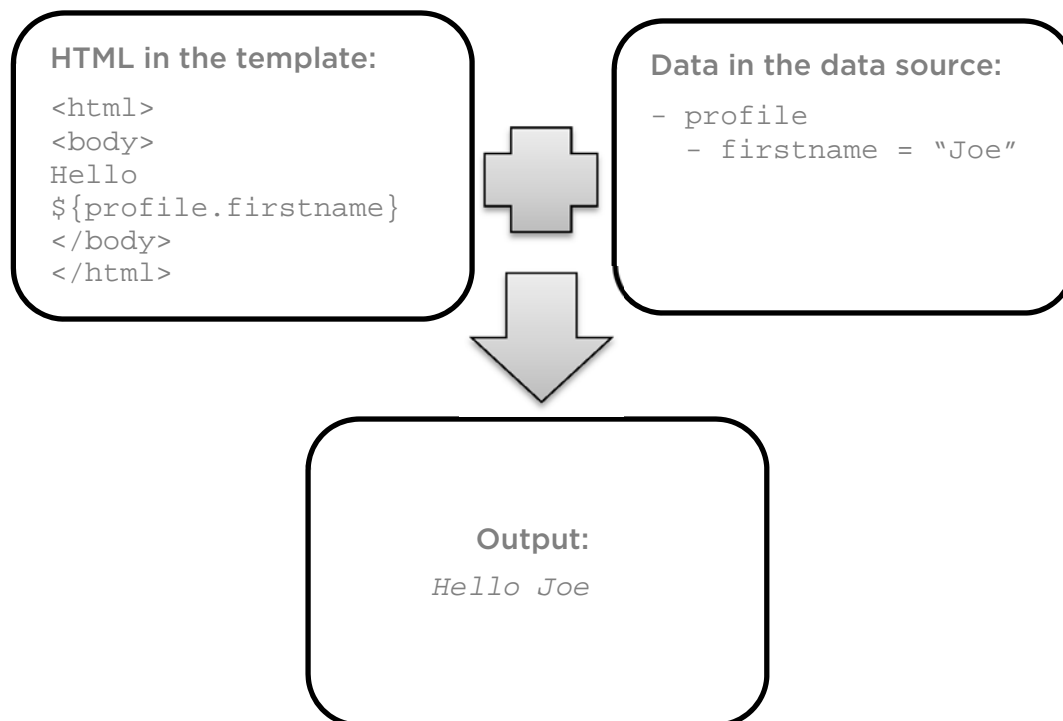
# How do I use RPL instead of built-in functions?

A few things to keep in mind before starting to use RPL:

- You need to assign an RPL-compatible alias for both the data source name and field/column names.
- You cannot use both RPL and built-in functions in the same campaign.
- A campaign that has the Email Message Designer feature enabled can use RPL.
- Campaigns that existed in an account before the new Email Message Designer was turned on will continue to use built-in functions.

# How does RPL work with templates?

Email Message Designer documents come from the Content Library. The following diagram shows how RPL executes templates.



## Template structure

A template comprises the following parts:

- **Text**  
Text is printed in the output as is.
- **Comments**  
Comments are ignored and are excluded from output.
- **Interpolations**  
Interpolations are instructions that RPL will replace in the output with calculated values.
- **RPL Directives**  
Directives are instructions to RPL. They are similar to HTML elements. Directives execute statements, text, interpolations and other directives.

## About data types

Every RPL element is of a specific type. RPL supports the following data types:

- **String**
- **Numeric**
- **Boolean**
- **Date**
- **Hash**  
A hash stores sub-variables by name. The sub-variables in a hash are not ordered and are accessed by name. All sub-variables must be of the same type.
- **Sequence**  
A sequence stores sub-variables by a number known as the *index*. The sub-variables in a sequence are ordered and are accessed by index. The index

starts with a 0; this means that the index of the first sub-variable is 0. The sub-variables can be of different types.

- **Node**  
Nodes represent a node in a tree structure, and are used mostly with XML processing.

These data types are described in more detail in the following sections.

## About interpolations

*Interpolations* are instructions to convert an expression to text and to insert that text into the output. An interpolation begins with `${` and ends with `}`.

You can use interpolations only inside template text. You can not use them inside expressions or to obtain a value. For example:

**Incorrect:**

```
${profile.firstname + ' ' + ${profile.lastname}}  
<#if ${profile.age} lt 50>
```

Correct:

```
    ${profile.age + 3}
    <#if profile.age lt 50>
    ${3+5}
```

## About string expressions

To specify a string value, enclose the string in double quotation marks ("" ) or single quotation marks (' '). Enclosing a string in quotation marks is known as *quoting*. If the string contains the character used for quoting (either " or ' ) or a backslash (\), you must precede the symbol with a backslash (\); this is called *escaping*.

For example, if you use a " for quoting and want to print *It's "quoted" and this is a backslash: \*, use:

```
    ${"It's \"quoted\" and
    this is a backslash: \\"} }
```

If you use ' for quoting and want to print *It's "quoted" and this is a backslash: \*, use:

```
    ${'It\'s "quoted" and
    this is a backslash: \\'} }
```

You can concatenate two strings with the + operator, for example:

```
    ${profile.firstname + ' ' + profile.lastname}
```

Remember that there is a distinction between text in the template and strings. Strings appear where an expression would appear, inside an interpolation or a directive.

## About arithmetic expressions

Arithmetic expressions result in a number. RPL supports the standard arithmetic operations:

- + — addition
- — subtraction



\* – multiplication

/ – division

% – modulus

## About boolean expressions

Boolean expressions result in a value of either *true* or *false*. The constants *true* and *false* are the only two possible values.

You can use the following comparison operators:

**==** – equal

**!=** – not equal

**gt** – is greater than

**lt** – is less than

**le** – is less than or equal to

**ge** – is greater than or equal to

You can also use the following logical operators:

**&&** – boolean logical operation AND. This operation results in true if both its two operands are true, false otherwise.

**||** – boolean logical operation OR. This operation results in true if either one of its two operands is true, false when both operands are false.

To use the standard operators **>**, **>=**, **<** and **<=**, you must enclose them in parenthesis. This is because these operators can conflict with tag annotations. For example:

```
<#if (profile.age > 30 && profile.gender=='F')>
```

## About date expressions

You cannot specify dates directly as constants. You must enter a date string then convert it to a date using the `?date`, `?datetime`, or `?time` built-ins. For an example, see the *Convert a string to a date* example in the *Using RPL built-ins* section.

Dates can be of type `date`, `date-time`, `time`, or `unknown`. Dates can be compared to other dates.

TIP: You can use the `dayadd` method to add or subtract days from the specified date. For more information, see `dayadd` in *Chapter 3: Method Reference*.

## About hashes

A *hash* is a mapping technique used to associate a key with a value. In other programming languages, hashes are called maps or dictionaries.

You specify a hash as:

```
{'key1':'value1', 'key2':'value2'}
```

Keys are usually strings. Values can be of any type.

## Accessing items in a hash

To retrieve a value in a hash, specify the key of the item. For example, to obtain the color of the apple:

```
<#assign fruitColors={'apple':'red', 'orange':'orange'}>  
${fruitColors["apple"]}
```

Alternately, when the key is a valid identifier (letters, followed by letters and/or digits), you can access an item using the dot notation. Using the previous example, you can obtain the color of the apple this way:

```
${fruitColors.apple}
```

## About sequences

You use a sequence to build a list.

You specify a sequence as: ["a", "b", "c"], for example:

```
<#assign list1=['one', 'two', 'three']>
```

The items in a sequence can be of different types, for example:

```
<#assign list1=[1, 'two', 3]>
```

You can concatenate sequences with the plus sign (+), for example:

```
<#assign list3=list1 + list2>
```

You can specify sequences as a range, for example:

```
<#assign list4=1..4>
```

You can add variables to a sequence, for example:

```
<#assign list5=['a', profile.gender, 'b']>
```

You can iterate a sequence with the `<#list>` directive, for example:

```
<#list list5 as element>
  ${element}
</#list>
```

## Accessing items in a sequence

To access a sequence item, specify its index in square brackets ([]), for example:

```
${list5[2]}
```

Indexes start with a 0. This means that the index of the first item is 0. The example above accesses the third item in the list.

# Using RPL instead of built-in functions

The following section shows examples of how to use RPL instead of built-in functions.

## Accessing variables and fields

Using built-in functions	Using RPL
Generally, the syntax for referring to fields/columns from a data source is:	
<code>Datasourcealias.fieldnamealias</code>	
<code>\$Lookup(Field)\$</code>	<code>\${Datasourcealias.Fieldalias}</code>
<code>\$Lookup(Variable)\$</code>	<code>\${Variable}</code>
We also support simple replacements: <code>\$Field\$</code> Dear <code>\$Lookup(FIRST_NAME)\$</code>	Dear <code>\${ContactList.FIRST_NAME}</code>  Assuming all the fields above are coming from the profile list called ContactList.

## Using conditions

Using built-in functions	Using RPL
<code>\$cond(condition, onTrue, onFalse)</code>	<pre>&lt;#if condition&gt;   ...text for true case &lt;#else&gt;   ... text for false case &lt;/#if&gt;</pre>
<code>\$cond(eq(Lookup(Travel), 'Y'), document(Travel, TravelOffer), nothing())\$</code>	<pre>&lt;#if Travel='Y'&gt;   &lt;table&gt;     &lt;tr&gt;...&lt;/tr&gt;   &lt;/table&gt; &lt;/#if&gt;</pre>

## Using operators

<p>Generally, the syntax for referring to fields/columns from a data source is:</p> <p style="text-align: center;">Data source alias.fieldname alias</p>	
Using built-in functions	Using RPL
<p><code>\$add(2,mul(5*Price))\$</code></p> <p><code>\$and(ge(Price, 1000.00), le (Price, 2000.00))\$</code></p>	<p><code>#{2 + 5*Price}</code></p> <p><code>#{(Price &gt;= 1000.00 &amp;&amp; Price &lt;= 2000.00)}</code></p> <p>Parentheses are used to tell RPL not to interpret "&gt;" or "&lt;" as a tag markers</p> <p><code>#{Price ge 1000.00 &amp;&amp; Price le 2000.00}</code></p>
<p>String concatenation:</p> <p><code>\$concat(FIRST_NAME, concat(space(), LAST_NAME))\$</code></p>	<p><code>#{ContactList.FIRST_NAME + " + ContactList.LAST_NAME}</code></p> <p>This example assumes that all fields are coming from the profile list called <i>ContactList</i>.</p>

## About RPL built-ins

RPL includes a set of routines that allow for some common programming tasks such as converting text to uppercase and converting a number to a string. These routines are called *built-ins*. Do not confuse these with Responsys built-in functions as they are different.

You specify built-ins as `expression? built-in name`.

Here are some examples of how you can use RPL built-ins instead of built-in functions.

Generally, the syntax for referring to fields/columns from a data source is:

`Datasourcealias.fieldnamealias`

The examples below assume that all fields are coming from the profile list called *ContactList*.

Using built-in functions	Using RPL built-in
<code>\$uppercase(Region)\$</code>	<code>\${ContactList.Region?upper_case}</code>
<code>\$leadingcapital(City)\$</code>	<code>\${ContactList.City?cap_first}</code>
<code>\$capitalizewords(Location)\$</code>	<code>\${ContactList.Location?capitalize}</code>

## Looking up records and looping

Using built-in functions	Using RPL
<code>\$COND(EMPTY(LOOKUPRECORDS(!Event Tables, Events_All_Next_21_Days, PAIRS(GENRE_CATEGORY_DESCR, MLB Tickets, Region_Name, LOOKUP(Region_Name)), Event_ID)), NOTHING(), ESCAPECOMMAS(DOCUMENT(Modules, MOD_TU_Event_Sports_Baseball_MLB)))\$</code>	<pre>&lt;table&gt; &lt;#data Events_All_Next_21_Days as event&gt;   &lt;#filter description="MLB Tickets"     region_name=Region_Name&gt;   &lt;#fields eventid description city state date_local&gt;   &lt;tr&gt;&lt;td&gt;...\${event.city}...&lt;/td&gt;&lt;/tr&gt; &lt;/#data&gt; &lt;/table&gt;</pre>

## Including a document

Using built-in functions	Using RPL
<code>\$document(folderName, doc1, doc2, ...)\$</code>	<code>&lt;#include "cms://contentlibrary/folderName/doc1.RPL"&gt;</code>
<code>\$document(folderName, doc1, pairs(name1, value1, name2, value2, ...))\$</code>	<code>&lt;#include "cms://contentlibrary/folderName</code>

<code>\$documentnoobr(folderName, doc1, doc2, ...)\$</code>	<pre> /doc2.html" parse="false"&gt; &lt;#list docList as doc&gt;   &lt;#assign random=rand(0,2000)&gt; &lt;!-- available in included doc - -&gt;   &lt;#include "cms://contentlibrary/folderName /" + doc&gt; &lt;/#list&gt; </pre>
---	---

## Using RPL built-ins

This section shows how to use some common RPL built-ins.

To do this	Specify this
Convert a string to a date	<code>"2013-03-27"?date("yyyy-MM-dd")</code>
Convert a number to a string in currency format	<code>7326847?string.currency</code>
Protect a string for html output	<code>pet.description?html</code>
Convert a number to hex format	<code>123?hex</code>
Convert a string to hex format (Responsys built-in function equivalent)	<code>"abc"?hex</code>
Skip a user's record if a string is empty	<code>profile.firstname?skip</code>
Sort a sequence	<code>list5?sort</code>
Get all the keys of a hash	<code>fruitColors?keys</code>

## Chapter 2: Most Commonly Used Directives

This chapter describes three of the most commonly used RPL directives.

### if

#### Usage

```
<#if condition>  
  ...  
</#if>
```

#### Description

This directive checks whether a condition is true. For example:

```
Welcome, ${user}<#if user == " Joe">Joe</#if>!
```

checks whether the value of the variable `user` is `Joe`. If so, the line will read *Welcome, Joe!*.

In this example, " Joe" will appear only if the value of the variable `user` is equal to the string `Joe`. With this directive, everything between the `<#if >` and `</#if>` tags is skipped if the condition specified within the tags is false (in this example, not `Joe`).

`${user}` is an interpolation that provides instructions which RPL will replace with the calculated values.

`user` is the name of a variable to test. Generally, unquoted words inside directives or interpolations are treated as references to variables, and strings within double quotes are treated as literal strings.

`==` is the operator that tests whether the value at right of the operator is true. An equal value results in `true`.



## list

### Usage

```
<#list sequence as loopVariable>repeatThis</#list>
```

### Description

This directive creates a list. The code within the `<#list >` and `</#list>` tags will be processed for each variable.

`loopVariable` will hold the value of the current item in all repetitions. This variable exists only between the `<#list ...>` and `</#list>` tags.

`repeatThis` will be repeated for each item in the sequence, one after the other, starting from the first item.

## include

### Description

This directive inserts the content of another file into the template.

For example: you have to include the same copyright notice in several templates. The file `/contentlibrary/common/copyright_footer.htm` contains the copyright notice. The following code in the template will insert the `copyright_footer.htm` file into the template:

```
<#include "cms://contentlibrary/common/copyright_footer.html">
```

The output will include the content of the copyright file. If you change `copyright_footer.htm` at any time, the updated copyright notice will appear on all pages.

## data

### Description

This directive obtains data from pre-declared data sources in the Email Message Designer. Before using this directive, you must set up the data sources in the system and include them in the campaign that you are going to launch, with proper aliases.

When you include a data source in a campaign, you need to specify aliases for the data source name and its columns, and to identify which columns are used as lookup keys.

RPL checks the data source to verify that:

- The field specified as the lookup key in the directive is identified as such in the campaign. If that field is not identified as a lookup key, an error will occur.
- The field used as a returned field is declared in the campaign with the given alias.

To perform the data check, you must specify a data declaration section (identified by the `<#data></#data>` tags) with three parts:

- **data**  
The alias of the data source to use
- **filter**  
The lookup key
- **fields**  
The returned fields

## Example

```
These items are on sale right now!  
<table>  
<tr><th>Item</th><th>Discount</th>  
<#data sales as offer>  
  <#filter region="NY">  
    <fields sales_id description discount>  
      <tr id="sale${offer.sales_id}">  
        <td>${offer.description}</td>  
        <td>${offer.discount}</td>  
      </tr>  
    </#data>  
</table>
```

The `data` declaration section specifies which data source to query, the lookup key to use, and the fields to return.

The looping variable `offer` specifies the hash being created, with the requested fields, as the source for each record returned from the inquiry.

The looping section is repeated for each record. The `offer` hash is updated on each iteration with the data obtained from the data source record.

## Chapter 3: Method Reference

### avg

#### Usage

```
avg(number1, number2, number3, ...)  
or  
avg(numeric-list-expr)
```

#### Description

This method computes the average of the given numbers.

#### Parameters

<code>number1,</code> <code>number2,</code> <code>number3, ...</code>	The numbers from which the average is computed.
<code>numeric-list-expr</code>	A sequence expression containing the numbers to average.

#### Example

```
<#assign list=[1,73,22]>  
${avg(list)}  
${avg(1,73,22)}
```

Produces this output:

```
32  
32
```

## bazaarvoiceauthstring

### Usage

```
bazaarvoiceauthstring(key, query-string-expr)  
or  
bazaarvoiceauthstring(key)
```

### Description

This method helps create links to the Bazaarvoice service.

To create a proper URL, you need the base URL and the key/passphrase. These will be provided to you in a document that encrypts the required key and query parameters.

Please consult technical services for information about obtaining these two elements, and for further details of how to utilize the Bazaarvoice service.

### Parameters

key	A key/passphrase value known to both the sending and receiving parties.
query-string-expr	The decoded and unencrypted query string. This parameter is optional.

## dayadd

### Usage

```
dayadd(date-expr, days-expr)
```

### Description

This method adds or subtracts the number of days specified by `days-expr` to a base date specified by `date-expr`. To subtract days, specify a negative number in `days-expr`. Note decimal numbers will be truncated.

**NOTE:** Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?date` and `?time` to specify the date type. The following examples show the different possibilities.

## Parameters

date-expr	The base date.
days-expr	The number of days to add or subtract.

### Example 1: Move one day forward and back

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${dayadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${dayadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

Produces this output:

```
2012-12-28 13:25:03
2012-12-26 13:25:03
```

### Example 2: Advance one day from a date without time

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${dayadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

Produces this output:

```
2012-12-28 00:00:00
```

### Example 3: Using a time-only date will cause an error

```
<#assign date="2012-12-27 13:25:03"?time("yyyy-MM-dd HH:mm:ss")>
${dayadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

Produces an error because there are no days in time-only dates.

## emaildomain

### Usage

```
emaildomain(email-expr)
```

### Description

This method extracts the string that represents the domain from the email address specified by `email-expr`.

### Parameters

email-expr	The email address.
------------	--------------------

## Example

```
${emaildomain("jamesbond@m5.com")}
```

Produces this output:

```
m5.com
```

## facebooklike

### Usage


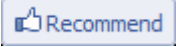
```
facebooklike(button-type, link-name, button-verb, button-style, description, thumbnail)
```

### Description

This method adds a Facebook Like or Facebook Recommend button.

### Standard Facebook images

The standard Facebook Like button and Facebook Recommend button images are available in Interact via the following image SRC paths.

Image Path	Image
<pre><code>/interact/ui/styles/images/likeonfacebook.png</code></pre> <pre><code>&lt;img src="/interact/ui/styles/images/likeonfacebook.png "&gt;</code></pre>	
<pre><code>/interact/ui/styles/images/recommendonfacebook.png</code></pre> <pre><code>&lt;img src="/interact/ui/styles/images/recommendonfacebook.png"&gt;</code></pre>	

When you use these paths, Interact automatically updates the SRC path to the proper Akamai URL for your Interact account.

## Parameters

button-type	<p>The type of Like button:</p> <p><b>0</b> – Like an email</p> <p><b>1</b> – Like an offer</p>
link-name	<p>The name of the link for tracking.</p> <p>This parameter is ignored if <code>button-type</code> is <b>0</b>.</p> <p>This is the name of the link as configured in the link table.</p> <p>The link name must be a string expression without commas.</p> <p>For Liking an offer, the link name and target URL (<code>LINK_URL</code>) of the offer in the link table is required.</p>
button-verb	<p>The verb to show on the Like button:</p> <p><b>0</b> – Like</p> <p><b>1</b> – Recommend</p> <p>The verb also appears in the individual's Facebook news feed, for example:</p> <p><i>Richard likes GiftCo's Sale</i></p> <p><i>John recommends GiftCo's Sale</i></p>
button-style	<p>The format of the Like button on the Like landing page:</p> <p><b>0</b> – Standard</p> <p>One line, the Like button followed by a text string: "X likes."</p> <p><b>1</b> – Button Count</p> <p>A horizontal presentation of the button and the number of Likes for the item.</p> <p><b>2</b> – Box Count</p> <p>Stacked presentation of the button and number of Likes for the item.</p>
description	<p>The string shown on the Like landing page. This should be a tag line string of what is liked.</p> <p>This parameter is ignored if <code>button-type</code> is <b>0</b>.</p> <p>The dollar sign symbols (\$) must be escaped.</p>
thumbnail	<p>The image used for the Like landing page.</p> <p>We recommend the image size of 200x200 pixels.</p>



## firstname

### Usage

```
firstname(string-expr)
```

### Description

This method obtains the first name with a small set of formats. If you need additional formats, we recommend you create a new function in a library.

### Parameters

string-expr	The expression that contains the individual's full name. Valid formats are: <b>First Last</b> <b>Last, First</b> <b>First M. Last</b> <b>Last, First M.</b>
-------------	---

## max

### Usage

```
max(number1, number2, number3, ...)  
or  
max(numeric-list-expr)
```

### Description

This method determines which number is the largest of the given numbers.

### Parameters

number1, number2, number3, ...	The numbers to compare.
numeric-list-expr	A sequence expression containing the numbers to compare.

## Example

```
<#assign list=[1,73,22]>
${max(list)}
${max(1,73,22)}
```

Produces this output:

```
73
73
```

## messedigest

### Usage

```
messedigest(string-expr)
or
messedigest(string-expr, algorithm)
```

### Description

This method generates a one-way digest by using two standard hashing algorithms. The hashing algorithm converts a string to a unique signature that identifies the message.

You can use this method to deliver encrypted information anywhere in a campaign or form message. For example, you might use it to pass encrypted promotion codes in the query string of the link URL so that the destination website can compare the code to a list of authorized codes.

### Parameters

string-expr	The string to which to apply the algorithm.
algorithm	Either “MD5”, “SHA”, or “SHA-256”, “SHA-384”, “SHA-512”. The default is “SHA”.

## Example

```
<#assign text="This is a test">
${messagedigest(text, "SHA")}
```

Produces this output:

```
a54d88e06612d820bc3be72877c74f257b561b19
```

## min

### Usage

```
min(number1, number2, number3, ...)
or
min(numeric-list-expr)
```

### Description

This method determines which number is the smallest of the given numbers.

### Parameters

number1, number2, number3, ...	The numbers to compare.
numeric-list	A sequence expression containing the numbers to compare.

## Example

```
<#assign list=[1,73,22]>
${min(list)}
${min(1,73,22)}
```

Produces this output:

```
1
1
```

## nonemptyfields

### Usage

```
nonemptyfields(field1, field2, field3, etc.)  
or  
nonemptyfields(sequence-expr)
```

### Description

In some cases, the personalization record is set up so that some fields might be empty. This is done so that only the fields that actually contain values are used.

This method allows you to create dynamic content by specifying a complete set of potential values, but retrieving only the values appropriate for the current record.

To extract the field names, use the hash built-in `?keys`. To extract only the values, use the hash built-in `?values`.

This method returns a hash with the proper field names and values associated with it.

### Parameters

<code>field1, field2, field3, ...</code>	The names of the fields to be examined.
<code>sequence-expr</code>	A sequence or expression that produces a sequence. This is useful, for instance, for use with the <code>randomsubset</code> method.

### Example

In this example, the personalization record for Mary has values in the fields *Boots* and *Backpacks*, with the values *2 pair* and *1*, respectively.

```
<#assign populated=nonemptyfields("Hats", "Shirts", "Shorts",  
"Boots", "Backpacks", "Tents")>  
<#list populated?keys as fieldName>  
  - ${fieldName} - ${populated[fieldName]}  
</#list>
```

Produces this output:

```
Boots - 2 pair
Backpacks - 1
```

## parsexml

### Usage

```
parsexml(string-expr)
```

### Description

This method converts a text string into a set of nodes.

### Parameters

string-expr	The text that contains XML.
-------------	-----------------------------

### Example

This example assumes that `xmlField` contains an XML string.

```
<#assign doc=parsexml(xmlField)>
```

## rand

### Usage

```
rand(value)
```

### Description

This method returns a random numeric value between 0 and the given number.

## randomsubset

### Usage

```
randomsubset(list-expr, on-empty-expr, max-size-expr)
```

### Description

This method returns a subset of list elements. The `max-size-expr` parameter specifies the maximum number of elements to return.

## Parameters

<code>list-expr</code>	The list from which to get the random subset.
<code>on-empty-expr</code>	The default value to return if the list has fewer elements than the size specified by the <code>max-size-expr</code> parameter.
<code>max-size-expr</code>	The maximum number of elements to return. If the list contains fewer elements, the value specified by the <code>on-empty-expr</code> parameter will be used.

## Example

```
<#assign fruits=["bananas", "oranges", "apples", "strawberries",  
"pears"]>  
<#list randomsubset(fruits, [], 3) as fruit>${fruit}</#list>  
<#assign morefruits=["bananas", "oranges" ]>  
<#list randomsubset(morefruits, fruits[0..2], 3) as  
fruit>${fruit}</#list>  
<#list randomsubset(fruits, fruits, 6) as fruit>${fruit}</#list>
```

Produces this output:

```
oranges strawberries pears  
bananas oranges apples  
bananas oranges apples strawberries pears
```

## Chapter 4: Namespace Reference

A *namespace* is a set of available directives, methods, RPL built-ins and additional structures and properties that you can use.

This chapter describes the RPL predefined namespaces.

### Campaign

The variables in the Campaign namespace are defined in the campaign definition. The following table lists all campaign variables.

<code>campaign.id</code>	The unique identifier of the campaign.
<code>campaign.name</code>	The campaign name.
<code>campaign.marketingprogram</code>	The specified marketing program in the campaign definition.
<code>campaign.marketingstrategy</code>	The specified marketing strategy in the campaign definition.
<code>campaign.externalcode</code>	The specified external campaign code as defined in the campaign.

### Data source hashes

Data source namespaces, or hashes, contain the fields that are defined in the data sources user interface. The data source appears as a top level hash, and its fields appear under that hash. Both the hash and its fields use the respective aliases for registration into the namespace.

For example, if you declared a namespace to provide an email address (from `EMAIL_ADDRESS_`) with the alias *profile*, you can specify the following interpolation to obtain the email address of an individual record:

```
${profile.email}
```

## Launch

The Launch namespace contains only one value, `launch.id`, which defines the current unique identifier of the given launch.

## Message

The Message namespace describes attributes of the message campaign.

<code>message.format</code>	Defines the format being used to personalize the current message: “H” - if the format is HTML “T” - if the format is text
-----------------------------	---

**NOTE:** In some situations, two personalization data sources include a field with the same name but different values. Internally, only the first value is available, based on the order in which the data sources were specified. Trying to access the second value using the namespace expression will result in undefined `values.Special variables reference`.

*Special variables* are variables defined by RPL. To access special variables, use the `.variable_name` syntax (notice the dot).

RPL defines the following special variables:

<code>.data_model</code>	A hash that you can use to access the data model directly. That is, variables created with the <code>global</code> directive are not visible here.
<code>.error</code>	This variable is accessible in the body of the <code>recover</code> directive, where it stores the error message of the error being recovered.
<code>.globals</code>	A hash that you can use to access global variables (data model variables and the variables created with the <code>global</code> directive). Note that variables created with the <code>assign</code> or <code>macro</code> directives are not global variables, thus they never hide the variables when you use <code>globals</code> .



<code>.lang</code>	Returns the language part of the current value of the locale setting. For example if <code>.locale</code> is <code>en_US</code> , then <code>.lang</code> is <code>en</code> .
<code>.locale</code>	Returns the current value of the locale setting. This is a string, for example <code>en_US</code> .
<code>.locals</code>	A hash that you can use to access local variables (variables created with the <code>local</code> directive and parameters of a macro).
<code>.main</code>	A hash that you can use to access the main namespace. Note that global variables such as the data model variables are not visible through this hash.
<code>.namespace</code>	A hash that you can use to access the current namespace. Note that global variables such as data model variables are not visible through this hash.
<code>.node</code>	The node you are currently processing with the visitor pattern (i.e. with the <code>visit</code> , <code>recurse</code> , ...etc. directives).
<code>.now</code>	Returns the current date and time. Usage examples: "Page generated: <code>\${.now}</code> ", "Today is <code>\${.now?date}</code> ", "The current time is <code>\${.now?time}</code> ".
<code>.output_encoding</code>	Returns the name of the current output charset. This special variable does not exist unless the framework that encapsulates RPL specifies the output charset for RPL.
<code>.template_name</code>	The name of the current template.
<code>.url_escaping_charset</code>	If the variable exists, it stores the name of the charset that should be used for URL escaping. If this variable does not exist, it means the charset to use for URL encoding has not been specified. In this case, the <code>url</code> built-in uses the charset specified by the <code>output_encoding</code> special variable for URL encoding.

<code>.vars</code>	<p>The expression <code>.vars.foo</code> returns the same variable as expression <code>foo</code>.</p> <p>This variable useful if you have to use the square bracket syntax: since that works only for hash sub-variables, you need an artificial parent hash. For example, to read a top-level variable with a name that might confuse RPL, you can write <code>.vars["Confusing Name!"]</code>. Or, to access a top-level variable with dynamic name given with variable <code>varName</code> you can write <code>.vars[varName]</code>. Note that the hash returned by <code>.vars</code> does not support <code>?keys</code> and <code>?values</code> built-ins.</p>
--------------------	--

## Chapter 5: Quick RPL Reference

If you already know RPL or are an experienced programmer, you can use this chapter as a quick reference.

### Specifying values directly

Variable Type	Example
String	"Foo" or 'Foo' or "It's \"quoted\"" or r"C:\raw\string"
Number	123.45
Boolean	true, false
Sequence	["foo", "bar", 123.45], 1..100
Hash	{"name": "green mouse", "price": 150}

### Retrieving variables

Variable Type	Example
Top-level	user
Hash	user.name, user["name"]
Sequence	products[5]
Special variable	main

### String operations

Operation	Example
Interpolation (or concatenation)	"Hello \${user}!" (or "Inter" + "act")
Getting a character	name[0]

## Sequence operations

Operation	Example
Concatenation	<code>users + ["guest"]</code>
Sequence slice	<code>products[10..19]</code> or <code>products[5..]</code>

## Hash operations

Operation	Example
Concatenation	<code>passwords + {"joe":"secret42"}</code>

## Numeric/boolean expressions

Expression Type	Example
Arithmetical calculations	<code>(x * 1.5 + 10) / 2 - y % 100</code>
Comparison	<code>x == y, x != y, x &lt; y, x &gt; y, x &gt;= y, x &lt;= y, x &amp;lt; y, ...etc.</code>
Logical operations	<code>!registered &amp;&amp; (firstVisit    fromEurope)</code>
Built-ins	<code>name?upper_case</code>
Method calls	<code>repeat("What", 3)</code>

## Missing value handler operators

Action	Example
Define a default value	<code>name!"unknown"</code> or <code>(user.name)!"unknown"</code> or <code>name!</code> or <code>(user.name)!</code>
Test for a missing value	<code>name??</code> or <code>(user.name)??</code>