Script Functions Library

this is some text saved to the page

Document properties

```
<#assign dumy = doc.setTitle("this is the new title")>
<#assign dumy = doc.setDescription("this is the new description")>
<#assign dumy = doc.setFeedItemUrl("https://not-a-real-domain.com")>
```

Document custom properties

Custom document properties travel with a document and are preserved through all workflows just like title, content, and several other standard properties. Custom properties are limited to 20,000 characters or less. There are two groups of custom document properties. The first group, legacy custom properties, is widely used in our standard automations, especially the Rich Summary process, so you use these only if you understand where else they are used. They are named – very imaginatively – custom1, custom2, …, custom10. To access one of these values in a script do:

```
custom1 = ${doc.custom1}
```

To assign a value to one of these variables do:

```
<#assign dumy = doc.setCustom1("uno")>
```

Note a small detail: The values you set on custom variables currently “take effect” at the end of script execution. so if you assign a value to one of these properties in a script and then access its value, you won’t get the updated value. This is a (minor) bug scheduled for the next update.

The upgrade to the properties just described is far more open-ended and the one that you should normally be using. These properties may have arbitrary names and their string values can be **very** large. Each property must also be assigned to a named "aspect" that is used purely organize related properties - the property names must still be unique across your ResultFlow application. Both aspects and properties are created when first accessed or set. To understand this more fully, let's consider some examples.

Creating a custom property and accessing its value

```
<#assign dumy = doc.setProperty('myaspect.property1', 'my value')>
${doc.getProperty('property1')}
```

Note the use of the aspect when first setting (creating) the property. Subsequently, you can use the aspect or just the property name alone, since property names must be unique. For example, both of these statements do the same thing

```
<#assign dumy = doc.setProperty('myaspect.property1','my new value')>
<#assign dumy = doc.setProperty('property1','my new value')>
```

Likewise, getting a property value may use the aspect name or not, so these two calls are equivalent.

```
${doc.getProperty('myaspect.property1')}
${doc.getProperty('property1')}
```

In all of the foregoing, I hope it's clear that property names must not include a period (".") since that would make separating aspects and property names very difficult.

You can also create, edit, and delete aspects and properties using the meta-data functions in the upper right corner of the document edit page.

jSoup

This oddly named package provides easy yet powerful ways to both query and edit HTML documents. The entire package is exposed so go here for the full documentation. Here is a simple example of it’s use.

```
<#assign dom = doc.soup>
<#assign p1 = dom.select("p").first()>
<#assign dumy = p1.attr("style","font-weight:bold")>
<#assign dumy = p1.append("<p>an inserted paragraph</p>")>
<#assign special = dom.select("p#specialp").html("this is the special text we inserted")>
${dom.html()}
```

The code above gets the jSoup structure into 'dom'. We then find the first paragraph, add some styling to it, and then add a new paragraph immediately following. Finally, we find we replace the contents of the paragraph with id=specialp with the text shown. The dom variable now contains the original document with content changed and added. We probably want to keep those changes for use in later workflows, so the final line takes the dom and converts it back to html as the output of this script.

Persona properties

All workflows operate within some Persona. The properties of that Persona are now available to scripts. These are currently of somewhat limited use, but in future releases additional properties will be included and supported in the same way as shown in the sample script below. Note that no updating of these properties is supported – they are read-only.

```
tone=${persona.tone}<br/>
style=${persona.style}<br/>
viewpoint=${persona.viewpoint}<br/>
mission=${persona.mission}<br/>
```

Random

Scripts have not previously had a sound random value function. We now have one. The randomInt call will provide a value between 0 and the specified limit. The randomBoolean call returns true/false values.

```
<#assign n1 = LIB.randomInt(10)>
<#assign b1 = LIB.randomBoolean()>
random num = ${n1}
<#if b1 >
random bool = true
<#else>
random bool = false
</#if>
```

Similarity

Turkers that copy/paste to create an abstract are a problem! Likewise those that produce a 'quote' that is not really a quote. This function allows us to detect these bad actors. The textSimilarity function uses Levenshtein distance to compare two strings. Their degree of similarity is returned as an integer percentage between 0 and 100 where 100 is an exact match.

Here is a simple example use, but see the Abstract and Author Quote steps in the Reference Persona workflows for real applications.

```
<#assign dom = doc.soup>
<#assign source = dom.text()>
<#assign abstract = "blah blah blah">
<#assign sim = LIB.textSimilarity(source, abstract)>
similarity=${sim}
```

Spelling

The same spell checker that is available as a 'predefined' filter is also available to scripts. The value returned is the integer error rate from 0 to 100 where 100 would mean every single word is misspelled! Note that the spelling dictionary is (currently) hard-coded to American English … hmm, how is that not an oxymoron? … so be careful how tightly to filter for spelling errors if you are expecting non-American spellings.

```
<#assign s0 ="this is a string that has no spelling errors of any kind that I know of">
<#assign s1 = "this is a strig that has summ spelling errors of teh sortt yu might ecpect">
<#assign n0 = LIB.getSpellingErrorRate(s0)>
<#assign n1 = LIB.getSpellingErrorRate(s1)>
n0 = ${n0}<br/>
n1 = ${n1}<br/>
```

URL Util

This object provides standards-compliant methods for editing URL query parameters. The most common usage of this class will be (1) removing existing query parameters, especially Google analytics; (2) adding Google analytics parameters; and (3) extracting the source URL from Google news links.

In all three use cases we start with getting the UrlUtil object like so:

```
<#assign u=doc.url>
<#assign uu=LIB.getUrlUtil(u)>
```

Following that, you can remove GA parameters with a single call:

```
<#assign dumy = uu.removeGA()>
```

or set one or more query parameters like so:

```
<#assign dumy=uu.set('utm_content','value1').set('rel','0')>
```

See the promotion scripts in the Reference Persona for real world usage.

If you want to use Google news sources in feeds, you will need to strip the real source URL out of the Google news link. This 6 line script fixes Google URLs and leaves the rest unchanged:

```
<#assign u=doc.url>
<#if u.starts_with("https://www.google.com/url")>
  <#assign uu=LIB.getUrlUtil(u)>
  <#assign url=uu.get("url")>
  <#assign dumy = doc.setFeedItemUrl(url)>
</#if>
```

Word Count

The same word count function that is available as a pre-defined filter is available to scripts as well. Usage is pretty simple. The getWords function returns an array of all the individual words in the input string. Call ?size to get the count. This is now used in the rejection filters for the MTurk workflows in the Reference Persona.

```
<#assign s = "this is a string. it's semi-interesting, if you [sic] like this sort of thing!much">
<#assign n = LIB.getWords(s)?size>
${n}
```

Sentences

Text can be broken up into an array of sentences using standard English delimiters as shown in the sample code below. This code snippet pulls the text from a paragraph in the source document, extracts the array of sentences from that paragraph, and returns the count. See the FTL documentation to see how to use the the array of sentences to do other things as well.

```
<#assign dom = doc.soup>
<#assign text = dom.select("p#abstract").text()>
<#assign sentences = LIB.getSentences(text)>
${sentences?size}
```

Summarizer

There is a “lightweight” language processing utility available to compute a summary of a block of text. The example below uses it to pull one sentence summaries from all paragraphs of a document that are 3 sentences or longer. This is just an example of the use of the function, it’s likely that using this code as-is would not return a usable result in most cases. The parameters to *textSummary()* are the text to summarize and the length of the summary in number of sentences.

```
```

```
<#assign dom=doc.soup>
<#assign ps = dom.select("p")>
<#list ps as p>
  <#if p.hasText() >
    <#assign sentences = LIB.getSentences(p.text())>
    <#if (sentences?size > 3) >
       <#assign summary = LIB.textSummary(p.text(),1)>
       <p>${summary}</p>
    </#if>
  </#if>
</#list>
```

OpenGraph

This is (as of release 4.5.11) a first version of this feature. It allows scripts to fetch and parse into a name/value array the OpenGraph data for a provided URL. All scripting is “an expert feature” and this one is no different, but if you feel the desire, have a go! Provide here are two example scripts to get you started.

A good place to start is to simply dump the values of OG data for a provided URL:

```
<#assign og = LIB.getOpenGraph(doc.url)>
<#if og??>
   <#assign props = og.getProperties()>
    <ul>
    <#list props as prop>
        <li>${prop.getProperty()} = ${prop.getContent()}</li>
    </#list>
    </ul>
<#else>
    og is null
</#if>
```

A more common usage will be fetching (and possibly checking) the og:image value. Here’s some code to get the image URL:

```
<#assign og = LIB.getOpenGraph(doc.url)>
<#if og??>
    <#assign imageUrl = og.getContent("image")>
    <img src="${imageUrl}" />
</#if>
```