

EXPRESSIONS

A DIFFERENT WAY TO PROCESS THREATCONNECT™ DATA
v1.0.9

Introduction

The Expressions application allows any ThreatConnect playbook author the ability to transform data in a single playbook application. The actions provided are:

Evaluate Evaluate a single expression. The output of the expression is always named `expression.result.0` and `expression.result.array`, for the first scalar result, and the array result, respectively.

Evaluate Many

Evaluates multiple expressions. There is a group of *variables* that may be defined (which are not output), and a multitude of *output expressions*. Each variable or output is named (as the key) and contains the expression to evaluate (as the value). The *Many* form allows multiple different output types, depending on need. The default `outputs` are String outputs.

Evaluate in Loop

Evaluates a single expression in a loop, with multiple *loop variables*, which are used to set up and iterate the loop. Loops make processing parallel data very easy, since loop variables with the same length are incremented at the same time. The output of the expression is always named `expression.result.0` and `expression.result.array`, for the first scalar result, and the array result, respectively. The prior iteration of the loop expression is available with the name `_output`, and is initialized to `None`.

Evaluate Many With Loop

Evaluates multiple loop expressions, with *variables* defined before the loops execute, *loop variables* which control the loops, *loop expressions* which calculate data in the loops, and *additional outputs* which allow outputs after the loops complete. The prior iteration of each loop expression is available with the loop expression name prefixed with an underscore, and is initialized to `None`.

Loop expressions which result in lists are used to extend the output array, rather than create nested lists, e.g. two successive outputs of `[1, 2, 3]` and `[4, 5, 6]` would result in `[1, 2, 3, 4, 5, 6]` not `[[1, 2, 3], [4, 5, 6]]`. Tuple outputs are nested, so `(1, 2, 3)` and `(4, 5, 6)` would result in `[(1, 2, 3), (4, 5, 6)]`.

Mini-Programs

Expressions applications are miniature programs, with *initialization* assignments, *loop* variables, *loop outputs*, and *additional outputs*.

The simplest program has one expression and one output; this is the **Evaluate** form of execution, and the output name is set as `expression.result.0` for the scalar output, or `expression.result.array` for the array output.

```
expression.result.0 = {'foo': 'bla'}.foo
```

```
bla
```

Here, the expression is `{'foo': 'bla'}.foo` and the result is `bla`. This is because the dot operator looks up the corresponding key in the origin dictionary. In most cases, the origin dictionary would not be directly in the expression, for example:

```
data = {'foo': 'bla'}
```

```
result = data.foo
```

```
bla
```

This is an example of the **Evaluate Many** form; two variables are defined, and the second refers to the first. There are actually two outputs, but the one we care about is named `result`.

Most programs loop or iterate over data in some way, sometimes with nested loops. This is where the **Evaluate in Loop** form is used:

```
for name in ('Matt', 'Chris', 'Jon'):
```

```
    expression.result.array = 'Hello ' + name
```

```
['Hello Matt', 'Hello Chris', 'Hello Jon']
```

Evaluate in Loop doesn't have any extra setup expressions, but you may have nested loops. This example sets a loop variable called `name` and the expression is a concatenation of a

Expressions

greeting with the name. The output is an array; so the output variable is `expression.result.array`.

A more complicated example uses multiple inputs and loop expressions, and then post-processes the loop data to get a scalar output back (rather than an array output). This is **Evaluate Many in Loop**:

```
names = ('Matt', 'Chris', 'Jon')
greetings = ('Hello', 'Goodbye')

for name in names:
    for greeting in greetings:
        salutation = greeting + ' ' + name

        text = join(',', salutation)

Hello Matt, Goodbye Matt, Hello Chris, Goodbye Chris, Hello
Jon, Goodbye Jon
```

In the ThreatConnect interface, this would appear as:

Expressions

Action *

Evaluate Many With Loop

Configure

Variables

Key	Value	
names	('Matt', 'Chris', 'Jon')	
greetings	('Hello', 'Goodbye')	

Loop Variables *

Key	Value	
name	names	
greeting	greetings	

Loop Expressions *

Key	Value	
salutation	greeting + ' ' + name	

Additional Outputs

Key	Value	
text	join(', ', salutation)	

Note: the order of evaluation of loop variables is *longest list* to *shortest list*. Variables of the same length step at the same time. The order the variables are declared does not control how the loops are iterated.

Variables

Both of the **Many** forms allow multiple expressions to set variables, or outputs. Each variable or output is available to subsequent expressions. However, *loop expressions* are only available to subsequent expressions *after* the looping is completed, not during the loop processing. It is possible to redefine a variable during processing, although caution should be exercised in this case as inadvertent re-ordering, such as editing the list by deleting the old entry and re-adding a new one, can invalidate the predicate definition.

Expressions

The prior iteration of a loop output is available with an underscore prefix, which will be initialized to None on the first iteration. This previous iteration is available during loop processing.

Grammar

The Expression grammar is very similar to Python grammar

Keywords contain only alphanumeric characters and underscore, and must not start with a number. A keyword followed by a parenthesis is a function, otherwise the keyword is a variable.

Literals are any values in single or double quotes. The grammar does not support the Python triple-quote literals. An enclosed quotation mark may be escaped, i.e. `'foo\'s compliment'`.

Numbers may be integer or floating point. Numbers may be signed, but do *not* support scientific notation — 10E15 is not recognized as a valid number.

Operations are `+`, `-`, `*`, `/`, `%`, `**` for the standard add, subtract, multiply and divide, modulus, and raise to a power. `||` and `or`, `&&` and `and` represent logical `and` and `or` operations. The *result* of a logical `and` will be the second operand if both are true, and the result of a logical `or` will be the first operand if both are true. Operations also include comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, and `not in`. Additionally, the `not` operator negates an expression, e.g. `not a < 2`.

When used with strings, `+` will concatenate strings, i.e. `'a' + 'b'` is `'ab'`.

The `.` operator is a dictionary dereference, e.g. `{ 'foo': 'bla' }.foo == 'bla'`.

This expression is valid, but unusual:

`1 + (1 or 2)`

As it happens, the result is **2**. It becomes much more interesting if you use logical expressions like this:

`name or "Stranger"`

Which would be the value of `name` if it has a value, or "Stranger" if it doesn't.

Subscripts are an array reference, which may be a *key* or *index* position, or a *slice*. Array indexes start with 0, e.g. `[1, 2, 3][0] == 1`. If negative, an index is an offset from the end of the array, e.g. `[1, 2, 3][-1] == 3`. A *slice* contains both a start and a stop location in the array separated by a colon. An unspecified start is the beginning of the array and an unspecified stop is the end. A subscript *key* is a dictionary key lookup, similar to using the `.` operator on the dictionary.

Expressions

Subexpressions are expressions in parenthesis to control precedence, i.e. $5 * (1 + 2)$ is 15, whereas $5 * 1 + 2$ is 7.

Lists are comma separated, and enclosed in square brackets, e.g. `[1, 2, 3]`.

Tuples are comma separated, and enclosed in parenthesis, e.g. `(1, 2, 3)`. A tuple is immutable, whereas a list is not, but in the simple expressions grammar, this not particularly relevant.

Dictionaries are enclosed in braces and contain comma separated key: value pairs, e.g. `{'id': 'Scott', 'type': 'Name'}`.

ThreatConnect variables passed in from other applications or triggers. These variables are speculatively evaluated to see if they are valid expressions, and the expression output is used if the evaluation succeeds, otherwise they are treated as literal values.

Function Calls are a keyword followed by a tuple, which represent the function name and parameters, respectively.

Built-in Values

Built-in values are *case insensitive*, i.e. `True`, `true`, `TRUE` are all the same value.

E 2.718281828459045

FALSE False

NULL None

NONE None

PI 3.141592653589793

TAU 6.283185307179586
Which astute readers may recognize as $2 * \pi$

TRUE True

URLRE A regular expression to match a URL, for convenience.

Functions

There are a large number of built-in functions, including some standard Python functions.

`abs(x)`
Absolute value of X

Expressions

`acos(x)`
Arc Cosine of X

`acosh(x)`
Inverse Hyperbolic Cosine

`alter(dictionary, key, value)`
Set a specific key in a dictionary. Returns the value.

`asin(x)`
Arc Sine of X

`asinh(x)`
Inverse Hyperbolic Sine

`atan(x)`
Arc Tangent of X

`atanh(x)`
Inverse Hyperbolic Tangent

`b64decode(s, altchars=None, validate=False, encoding='utf-8')`
Base 64 decode of string

`b64encode(s, altchars=None, encoding='utf-8')`
Base 64 encode of string

`bin(n, sign=True)`
Return the binary value of int

`binary(s, encoding='utf-8', errors=None)`
Convert object to binary string (bytes)

`build(*lists, keys=())`
Constructs a sequence of dictionaries based on the lists, such that each dictionary contains the corresponding key for each list from the keys value, and value from each list, respectively. Columns without a key are ignored. Columns that are longer than the shortest column are truncated.

`bytes(s, encoding='utf-8', errors=None)`
Convert object to binary string (bytes)

`ceil(x)`
Ceiling of X

`center(s, width, fillchar=' ')`
Center string in width columns

`choice(condition, true_result=None, false_result=None)`
Choice of true_result or false_result based on condition

Expressions

`chardet(byteseq)`

Return a dictionary with the guessed character encoding of byteseq, the confidence of the encoding, and the estimated language

`chr(x)`

Return character value of x

`conform(object_list, missing_value=None)`

Conform objects in a list to have the same structure, using missing_value as the value of any missing key

`copysign(x, y)`

Copy sign of X to Y

`cos(x)`

Cosine of X

`cosh(x)`

Hyperbolic Cosine

`csvread(data, header=False, convert=True, delimiter=',', quote='', rows=0, columns=0)`

Process data as a CSV File. Return the data as a list of rows of columns, or if rows=1, return a list of columns). If header is true, the first record is discarded. If rows or columns is nonzero, the row or column count will be truncated to that number of rows or columns. If convert is True, numeric values will be returned as numbers, not strings

`csvwrite(data, delimiter=',', quote='')`

Write data in CSV format. Returns a string

`datetime(datetime, date_format=None, tz=None)`

Format a datetime object according to a format string

`defang(s)`

Return a defanged representation of string, ie, one with textual indicators of compromise converted to the defanged state

`degrees(x)`

Convert X to degrees

`dict(**kwargs)`

Return a dictionary of the arguments

`erf(x)`

Error Function of X

`erfc(x)`

Complimentary Error Function of X

Expressions

`exp(x)`

Math Exp of X

`expm1(x)`

Math Expm1 of X

`extract_indicators(data, ignore=None, dedup=True, fang=False, convert=True)`

Extract IOCs from data, which may be bytes or string. If fang is true, data is re-fanged before processing. This option is ignored if the input is binary. Any entity match on the ignore list will be ignored. If convert is true, bytes mode matches will be converted to utf-8, or the specified conversion e.g. convert='latin-1'. Returns a list of (indicator, value) tuples. If dedup is True, duplicate results are not returned.

`factorial(x)`

Factorial of X

`fang(s)`

Return a fanged representation of string, ie, one with textual indicators of compromise reverted from the defanged state

`fetch_indicators(*search_values, default_type=None)`

Fetches available indicators from ThreatConnect based on search_values. Search values is a list of lists, the inner entries may consist of 1 or 2 items. If 1 item, the value is the search key, if two items, the value is the indicator name and the search key. Returns a list of [(indicator_type, indicator_value, api_entity, indicator), ...], but the api_entity, and result and will be None if that indicator was not found.

`find(ob, value, start=None, stop=None)`

Find index value in ob or return -1

`flatten(ob, prefix='')`

Flatten a possibly nested list of dictionaries to a list, prefixing keys with prefix

`float(s)`

Return floating point value of object

`format(s, *args, default=__notfound__, **kwargs)`

Format string S according to Python string formatting rules. Compound structure elements may be accessed with dot or bracket notation and without quotes around key names, e.g. blob[0][events][0][source][device][ipAddress] or blob[0].events[0].source.device.ipAddress. If default is set, this value will be used for all undefined values in the format, otherwise a KeyError will be raised.

`fuzzydist(hash1, hash2)`

Return the edit distance between two fuzzy hashes

Expressions

`fuzzyhash(data)`
Return the fuzzy hash of data

`fuzzymatch(input1, input2)`
Return a score from 0..100 representing a poor match (0) or a strong match(100) between the two inputs

`gamma(x)`
Return the gamma function at X

`gcd(a, b)`
Greatest Common Denominator of A and B

`hex(n, sign=True)`
Return the hexadecimal value of int

`hypot(x, y)`
Hypotenuse of X,Y

`index(ob, value, start=None, stop=None)`
Index of value in ob

`indicator_patterns()`
Returns a dictionary of regular expression patterns for indicators of compromise, based on ThreatConnect Data.

`indicator_types()`
Return the ThreatConnect Indicator Types

`int(s, radix=None)`
Return integer value of object

`items(ob)`
Items (key, value pairs) of dictionary

`jmespath(path, ob)`
JMESPath search

`join(separator, *elements)`
Join a list with separator

`json_dump(ob, sort_keys=True, indent=2)`
Dump an object to a JSON string

`json_load(ob)`
Load an object from a JSON string

`keys(ob)`
Keys of dictionary

Expressions

`len(container)`
Length of an iterable

`lgamma(x)`
Return the natural logarithm of the absolute value of the gamma function at X

`locale_currency(val, symbol=True, grouping=False, international=False, locale='EN_us')`
Format a currency value according to locale settings

`locale_format(fmt, val, grouping=False, monetary=False, locale='EN_us')`
Format a number according to locale settings

`log(x, base=None)`
Math Logarithm of X to base

`log10(x)`
Math log base 10 of X

`log1p(x)`
Math log1p of x

`log2(x)`
Math log base 2 of X

`lower(s)`
Lowercase string

`lstrip(s, chars=None)`
Strip chars from left of string

`max(*items)`
Return the greatest value of the list

`merge(*iterables, replace=False)`
Merges a list of iterables into a single list. If the iterables are dictionaries, they are updated into a single dictionary per row. If replace is true, subsequent columns overwrite the original values. The result length is constrained to the shortest column.

`md5(data)`
Return MD5 hash of data

`min(*items)`
Return the least value of the list

`namevallist(ob, namekey='name', valuekey='value')`
Return a dictionary formatted as a list of name=name, value=value dictionaries

Expressions

`ord(char)`
Return ordinal value of char

`pad(iterable, length, padvalue=None)`
Pad iterable to length

`pformat(ob, indent=1, width=80, compact=False)`
Pretty formatter for displaying hierarchical data

`pivot(list_of_lists, pad=None)`
Pivots a list of lists, such that item[x][y] becomes item[y][x]. If the inner lists are not of even length, they will be padded with the pad value.

`pow(x, y)`
Math X^{**Y}

`printf(fmt, *args)`
Format arguments according to format

`prune(ob, depth=None, prune=(None, '', [], {}), keys=())`
Recursively Prunes entries from the object, with an optional depth limit. The pruned values, and optionally prune keys may be specified. If any dictionary has a key in `keys`, that dictionary element will be removed.

`radians(x)`
Convert X to radians

`range(start_or_stop, stop=None, step=None)`
Return range of values

`refindall(pattern, string, flags='')`
Find all instances of the regular expression in source

`rematch(pattern, string, flags='')`
Regular expression match pattern to source

`replace(s, source, target)`
Replace chars on S

`report(data, columns=None, title=None, header=False, width=None, prolog=None, epilog=None, sort=None, filter=None)`
Generates a text report of data in columnar format. Data is either a list of dictionaries, or a list of lists of columnar data. If a list of lists and `header=True`, then the first row is the header row of the data.

Columns is a list of row specifiers or a single row specifier, which is a list of column definitions. If there are multiple row specifiers, each record takes up multiple output rows.

A row specifier is either an ordered dictionary of `name=column specifier` or a list of `(name, column specifier)` tuples.

Expressions

A column specifier is `width[:height][/option[=value]][/option[=value]]...` If rows are lists of lists (e.g. CSV data) and no column specifiers are used, the widths will be automatically calculated.

Options:

<code>align=left right center</code>	
<code>value=format</code>	format for values e.g. <code>{lineno}</code> . to add a . after <code>lineno</code>
<code>error=string</code>	Replacement value if the <code>value=</code> format fails
<code>notrim</code>	Don't trim leading/trailing space
<code>hang=n</code>	Hanging paragraph by N spaces
<code>indent=n</code>	Indent paragraph by N spaces
<code>split=n</code>	split at n% through the column (default 80) if necessary
<code>label=string</code>	heading label
<code>doublenl</code>	Double newlines (ie, add line after paragraph)
<code>nohyphenate</code>	Don't hyphenate value

If `sort` is specified, it is a column or list of columns to sort by, with the column name optionally prefixed with a '-' to do a descending sort.

If `filter` is specified, it is an expression that must be true for that record to appear in the result, e.g. `filter="salary>70000"`.

`research(pattern, string, flags='')`
Regular expression search pattern to source

`rexpparse(source, template, strip=False, convert=False, **kwargs)`
REXX parse of source using template. If `strip` is True, values are stripped, if `convert` is True, values are converted to float or int if possible. Any other keyword arguments are made available for indirect pattern substitution, in addition to the standard variables.

`rstrip(s, chars=None)`
Strip chars from right of string

`sha1(data)`
Return SHA1 hash of data

`sha256(data)`
Return SHA256 hash of data

`sin(x)`
Sine of X

`sinh(x)`
Hyperbolic Sine

`sort(*elements)`
Sort array

Expressions

`split(string, separator=None, maxsplit=-1)`

Split a string into elements

`sqrt(x)`

Square root of X

`str(s, encoding='utf-8')`

Return string representation of object

`strip(s, chars=None)`

Strip chars from ends of string

`structure(ob)`

Return a reduced structure of the object, useful for comparisons

`sum(*elements)`

Sum a list of elements

`tan(x)`

Tangent of X

`tanh(x)`

Hyperbolic Tangent

`timedelta(datetime_1, datetime_2)`

Return the delta between time 1 and time 2

`title(s)`

Title of string

`trunc(x)`

Math Truncate X

`twoscomplement(n, bits=32)`

Return the twos complement of N with the desired word width

`unique(*args)`

Return the list of unique elements of arguments, which may be a list of arguments, or a single argument that is a list. Inputs are compared by converting them to sorted JSON objects, so dictionaries with the same keys and values but different order will count as duplicates.

`unnest(iterable)`

Reduces nested list to a single flattened list. [A, B, [C, D, [E, F]] turns into [A, B, C, D, E, F].

`update(target, source, replace=True)`

Updates one dictionary with keys from the other. If the target is a list of dictionaries, each dictionary will be updated. If replace is false, existing values will not be replaced.

Expressions

`upper(s)`

Uppercase string

`urlparse(urlstring, scheme='', allow_fragments=True)`

Parse a URL into a six component named tuple

`urlparse_qs(qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None)`

Parse a URL query string into a dictionary. Each value is a list.

`uuid3(namespace, name)`

Generate a UUID based on the MD5 hash of a namespace and a name. The namespace may be a UUID or one of 'dns', 'url', 'oid', or 'x500'.

`uuid4()`

Generate a random UUID

`uuid5(namespace, name)`

Generate a UUID based on the MD5 hash of a namespace and a name. The namespace may be a UUID or one of 'dns', 'url', 'oid', or 'x500'.

`values(ob)`

Values of dictionary

`xmlread(xmldata, namespace=False, strip=True, convert=True, compact=True)`

Constructs an object from XML data. The XML data should have a single root node. If namespace is True, the resolved namespace will be prefixed to tag names in braces, i.e. {namespace}tag. If strip is True, values will be stripped of leading and trailing whitespace. If convert is True, numeric values will be converted to their numeric equivalents. If compact is true, the object will be compacted to a more condensed form if possible.

`xmlwrite(obj, namespace=False, indent=0)`

Converts an object to XML. If namespace is True or a dictionary, namespace prefixed values will be converted to a derived or specified namespace value. The namespace dictionary should be in the form {key: namespace} and will be used to turn the namespace back into the key. If indent is nonzero, an indented XML tree with newlines will be generated. The namespace option will *not* generate an xml tag with the namespace definition.

Examples

String Formatting

Formats can use variables that are already defined, or pass them as parameters to the format function. Here, `variable` is a previously defined variable.

```
format('{variable}*5 = {value}', value=variable*5)
```

Formats can be any valid Python format string, but not an `f`-string format.

```
format('{pct:5.3f}%', pct=4.182964)
4.183%
```

When using formats, dictionary expressions must be resolved using the subscript notation, but quotations around the keys is not necessary.

```
format('>>{timedelta[total_days]}<<', timedelta=timedelta('now',
    'yesterday'))
>>1<<
```

Array Padding

To make loop variables the same length if they aren't already, use the `pad` function. Note: the consequences of dealing with the nulls added to the arrays are up to you! You could also set up an initial variable which contained the results of the `max` expression to avoid repeating it.

```
pad(variable1, max(len(variable1), len(variable2), len(variable3)))
```

Pro Tip: The `pivot` function will pad arrays to an equal length, so pivoting *twice* will return the original list of arrays, padded out, albeit as a single array of arrays, but you can use subscript notation to refer to each column.

```
pivot(pivot([array1, array2, array3], pad=''))
```

Reconstructing Objects from Parallel Arrays

Let's assume that a prior app provided three StringArrays, `#app.id`, `#app.type`, and `#app.value`, each of the same length. We *could* use a **Loop** to turn them back into dictionaries, but instead, let's use `build`:

```
build(#app.id, #app.type, #app.value, keys=('id', 'type', 'value'))
```

Adding Values to a List of Objects

Sometimes, we have a list of objects (dictionaries, of course!) and we want to add a value or values to each object. Lets assume `#app.list` is a list of objects, and we want to add a source key to each object *if* it doesn't already exist. The `update` function will let us do that:

```
update(#app.list, {'source': 'Spaceman Spiff'}, replace=False)
```

As always, expressions return their result, so in the original list is not modified in place, but the updated list is returned.

Merging Two Lists of Objects

Occasionally we'll get two or more lists of objects that we want to merge into one list of objects. The `merge` function will merge multiple lists down to one list, and will allow control over whether or not subsequent column keys overwrite prior column keys.

```
merge(#app1.list, #app2.list, #app3.list, replace=True)
```

No key matching is done to line up the lists, so beware; this is a simple row merge. If the lists are not dictionaries, the result is equivalent to Python's `zip` function; e.g.

```
merge(['a', 'b', 'c'], [1, 2, 3])  
[[ 'a', 1], [ 'b', 2], [ 'c', 3]]
```

Date/Time Calculations

The `timedelta` function will calculate the time between two date time values. Relative time expressions can be used.

```
timedelta('now', 'yesterday')
```

```
{'datetime_1': '2021-04-10T19:02:12', 'datetime_2':  
'2021-04-09T09:00:00', 'years': 0, 'months': 0, 'weeks': 0,  
'days': 1, 'hours': 10, 'minutes': 2, 'seconds': 12,  
'microseconds': 0, 'total_months': 0, 'total_weeks': 0,  
'total_days': 1, 'total_hours': 34, 'total_minutes': 2042,  
'total_seconds': 122532, 'total_microseconds': 122532000}
```

Structural Analysis of Objects

The `structure` function will analyze the structure of an object. Using the previous example's data:

```
structure(timedelta('now', 'yesterday'))
```

```
{'datetime_1': 'iso8601', 'datetime_2': 'iso8601', 'years':  
'int', 'months': 'int', 'weeks': 'int', 'days': 'int',  
'hours': 'int', 'minutes': 'int', 'seconds': 'int',  
'microseconds': 'int', 'total_months': 'int', 'total_weeks':  
'int', 'total_days': 'int', 'total_hours': 'int',  
'total_minutes': 'int', 'total_seconds': 'int',  
'total_microseconds': 'int'}
```

Structure descriptions can include more than one component identified in the object.

```
structure('January 10, 2020 11:15 AM  
4c9c0cab51196f093eb49672de64ff05')
```

```
date time md5
```

Flattening Objects

The `flatten` function will turn nested dictionaries into a single dictionary, by creating new keys based on the key path.

```
flatten({'time': timedelta('now', 'yesterday'), 'time2':
        timedelta('now', 'January 1, 1970')})
```

```
{'time1.datetime_1': '2021-04-10T19:38:21',
 'time1.datetime_2': '2021-04-09T09:00:00',
 'time1.days': '1',
 'time1.hours': '10',
 'time1.microseconds': '0',
 'time1.minutes': '38',
 'time1.months': '0',
 'time1.seconds': '21',
 'time1.total_days': '1',
 'time1.total_hours': '34',
 'time1.total_microseconds': '124701000',
 'time1.total_minutes': '2078',
 'time1.total_months': '0',
 'time1.total_seconds': '124701',
 'time1.total_weeks': '0',
 'time1.weeks': '0',
 'time1.years': '0',
 'time2.datetime_1': '2021-04-10T19:38:21',
 'time2.datetime_2': '1970-01-01T00:00:00',
 'time2.days': '9',
 'time2.hours': '19',
 'time2.microseconds': '0',
 'time2.minutes': '38',
 'time2.months': '3',
 'time2.seconds': '21',
 'time2.total_days': '18727',
 'time2.total_hours': '449467',
 'time2.total_microseconds': '1618083501000',
 'time2.total_minutes': '26968058',
 'time2.total_months': '615',
 'time2.total_seconds': '1618083501',
 'time2.total_weeks': '5113',
 'time2.weeks': '1',
 'time2.years': '51'}
```

For what its worth, the `timedelta` calculations of total months and total weeks for `time2` look wrong on their face, but that's coming out of the underlying library. This example is to demonstrate the `flatten` function.

REXX Parsing

For full details of REXX parsing, consult an IBM guide. This function introduces a subset of REXX parsing, equivalent to `PARSE VAR source template`. In general, a template consists of:

- Quoted literals which match their exact string in the source
- Variables which assign a substring to a variable, or “.” when there is no need for assignment
- Numbers which may be signed to indicate a relative offset, or an absolute column position
- Indirect References in parenthesis to allow variable substitution into the template

Substrings are split on spaces if there is more than one variable to be assigned from the substring.

```
rexpparse('Mary had a little lamb', 'name .  
"had a " size animal')
```

```
{'name': 'Mary', 'size': 'little',  
 'animal': 'lamb'}
```

```
rexpparse('127.0.0.1 - frank [10/Oct/  
2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/  
1.0" 200 2326', "ip id user ' [' date ' ] \"  
method url protocol \" ' status size")
```

```
{'ip': '127.0.0.1', 'id': '-', 'user': 'frank',  
'date': '10/Oct/2000:13:55:36 -0700', 'method':  
'GET', 'url': '/apache_pb.gif', 'protocol':  
'HTTP/1.0', 'status': '200', 'size': '2326'}
```

A dot or variable will consume more than one word if there are no subsequent assignments before the next pattern or column position.

Column positions may be relative or absolute. Column positions can move backwards, to re-parse the source with a different template portion.

Expressions

```
rexpparse('Mary had a little lamb', 'name . 1  
song')  
{'name': 'Mary', 'song': 'Mary had a little  
lamb'}
```

After a pattern match, relative position offsets can back up to data before the pattern! Relative matches start at the pattern start.

```
rexpparse('01234FFFFAB0C', '"FFFF" -4 last4  
+4')  
{'last4': '1234'}
```

REXX style parsing is very easy to use as long as your data is not structured data, in which case regular expressions are more suitable.

XML Parsing

The `xmlread` function will parse XML data, optionally compacting (in a non-reversible way) the resulting object structure. XML attributes are added as keys with @ prefixes.

Assume that `xmldata` is

```
<people>
  <person class="Peon">
    <name>Bob</name>
    <job>Developer</job>
  </person>
  <person class="Pooh-Bah">
    <name>Alice</name>
    <job>Manager</job>
  </person>
</people>
```

`xmlread(xmldata, compact=False)`

```
{'people': [{ 'person': [{ 'name': 'Bob'}, { 'job':
'Developer'}], '@class': 'Peon'}, { 'person': [{ 'name':
'Alice'}, { 'job': 'Manager'}], '@class': 'Pooh-Bah'}]}
```

`xmlread(xmldata, compact=True)`

```
{'people': [{ 'person': { 'Bob': 'Developer'}, '@class':
'Peon'}, { 'person': { 'Alice': 'Manager'}, '@class':
'Pooh-Bah'}]}
```

The `compact=True` option applies two transforms to the resulting output structures:

1. If a list of dictionaries each have only a single key, the list is collapsed to a single dictionary, i.e. `[{'key1': 'value1'}, {'key2': 'value2'}] -> {'key1': 'value1', 'key2': 'value2'}`
2. If a dictionary has exactly two keys, one of which has a key name containing “name” in any case, that dictionary is compacted to a single name=value dictionary, and compaction is reattempted, i.e. `{'data': [{ 'name': 'Hamburgler'}, { 'value': 'RobbleRobble'}]} -> {'data': { 'Hamburgler': 'RobbleRobble'}}`

The corresponding `xmlwrite` is the inverse of `xmlread`, although it cannot undo the compaction effects of `compact=True`.

The `namevallist` function can produce results similar to the inverse of compaction step 2.

```
xmlwrite({'people': {'person': namevallist({'name':  
'Matt', 'job': 'Developer'})}}, indent=2)
```

```
<people>  
  <person>  
    <name>name</name>  
    <value>Matt</value>  
    <name>job</name>  
    <value>Developer</value>  
  </person>  
</people>
```

```
xmlwrite({'people': {'person': namevallist({'name':  
'Matt', 'job': 'Developer'}, namekey='@name')}},  
indent=2)
```

```
<people>  
  <person>  
    <value name="name">Matt</value>  
    <value name="job">Developer</value>  
  </person>  
</people>
```

Here, the `namekey` being `@name` makes `xmlwrite` write out the values with a `name` attribute.

Reports

Reports can largely be inferred entirely from the columnar data they are passed; however, if necessary column specifications can be used to control report formatting.

A report is made up of a list of row values, each of which is a list of column values for data, e.g.

Expressions

```
report(data=(
    ('Column A', 'Column B', 'Column C'),
    (1, 2, 3),
    ('a', 'b', 'c')))
```

Column A	Column B	Column C
1	2	3
a	b	c

Reports can have titles, prologs and epilogs which are printed along with the report value.

Report values may be formatted to replace the value displayed in the column using the `value` option.

Data may either be a list of lists (e.g.1 CSV format) or a list of dictionaries. When data is a list of dictionaries, the key of the item is the column name, which when title cased is the row label.

Column specifications in the *simple* form are a list of headings for the column values, in the tuple form is a list of (heading, specification) values, and in the row dictionary form is a list of dictionaries of specifications. There is also a multi-row form with lists of rows of column specifiers. Multi-row reports never generate header rows for the supplemental rows.

If column specifiers are not provided a default column specification is generated using the values of the first row (or the first row's keys).

An example of a multi-row report with dictionary specifications.

```
report(({ 'A': 1, 'B': 2 }, { 'B': 3, 'A': 4 }),
columns=(({'A': 20}), {'B': '20/indent=5'}))
```

A
1
2
4
3

A column can be constrained to a specific width, and optionally a specific height.

Expressions

An example of a width and height constrained column:

```
report({'stanza': 'Mary had a little lamb'},
columns=({'stanza': '10:2'}))
```

```
Stanza
-----
Mary had
a little
```

For ease of use, data values that are *not* lists are turned into one, allowing a single dictionary instead of a list of dictionaries.

Columns with values too long for the column will be hyphenated unless the `nohyphenate` option is specified.

An example of hyphenation in a column.

```
report(({ 'name': 'Rumplestiltskin' }),
columns=(('name', '12'), ('nohyname', '12/
nohyphenate/value={name}'))))
```

```
Name                Nohyname
-----
Rumplestilt- Rumplestilts
skin          kin
```

Using the `indent` or `hang` options the value can be indented on the first line, or on all successive lines, respectively.

Expressions

An example of hanging paragraph indentation.

```
report([[ 'This is a sample paragraph wrapped  
into multiple lines with hanging indentation.  
\nNewlines reset the hang.' ]], header=False,  
columns=(( 'heading', '50/hang=5' )))
```

```
This is a sample paragraph wrapped into multiple  
lines with hanging indentation.  
Newlines reset the hang.
```

The `value` specification option will allow python f-string style subexpressions to calculate values. If an expression contains a slash, escape it with a backslash e.g. `value={units\ / 100:5.2f}%`. Note that the subexpressions can't handle nested braces; escape inner braces with `\` as well.

The title, prolog, and epilog also have value substations performed on them. The expression values (variables and outputs defined previously) are available for all formats and filters.

Sample Recipes

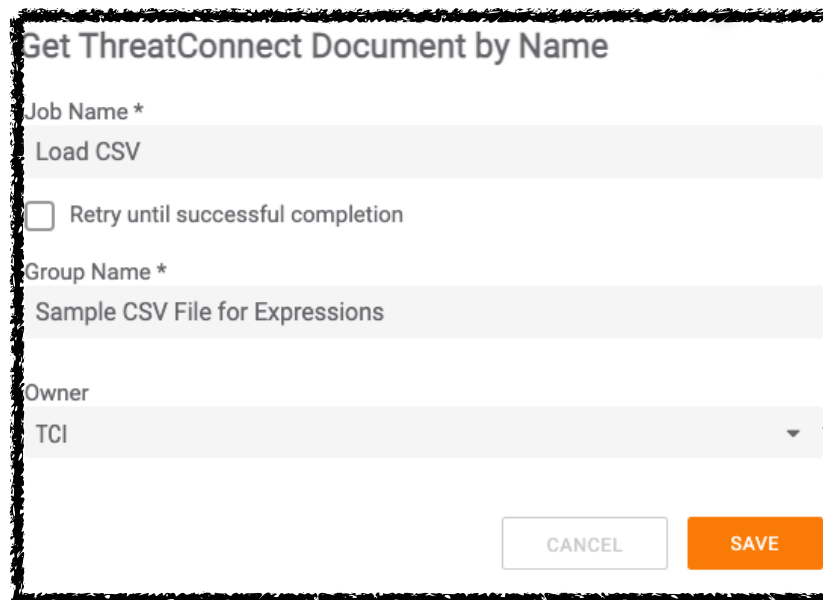
Turn a CSV file into a TCEntityArray

Let's assume that there is a CSV file that has contains multiple columns, and column 2 of the file is an IP address that we want to turn into an array of Address indicators. To spice things up, column 3 contains what kind of thing it is, so we can selectively import.

Here's what the sample data looks like:

Hostname	IP Address	Conditions
google.com	172.217.0.14	
	120.79.128.109	C&C
	204.48.23.94	C&C

Using the app **Get ThreatConnect Document by Name** we load the document in the playbook:



Get ThreatConnect Document by Name

Job Name *

Load CSV

☐ Retry until successful completion

Group Name *

Sample CSV File for Expressions

Owner


TCI

CANCEL SAVE


Then, we connect that to the **Expressions** application with the action **Evaluate Many With Loop**.

Starting out, we want to import the data from the Document, so under *Variables*, add the key `csvdata` with the values `csvread(#tc.document.file_data, header=True)` and click the circled plus to add the key. It has a header record we need to throw away, so `header=True` is specified for `csvread`.

Expressions

Variables		
Key	Value	
csvdata	csvread(#tc.document.file_data ,header=True)	


Then we want to loop through each record at a time, so under *Loop Variables*, add the key `record` with the value `csvdata`, and click the circled plus to add the key. We could have loaded the CSV document here with the `csvread` function directly, as well.

Loop Variables *		
Key	Value	
record	csvdata	

We want to create a TCEntity dictionary *if* the 3rd column is “C&C”, otherwise we will output a null element. To add some complexity, column 1, the name column, isn’t always set, so we’ll use the name if specified, *or* the address if it isn’t.

Under *Loop Expressions* add the key `entity` with the value `choice(record[2] == 'C&C', { 'id': record[0] or record[1], 'type': 'Address', 'value': record[1] }, None)` and click the circled plus to add the key.

Expressions

Loop Expressions *		
Key	Value	
entity	choice(record[2] == 'C&C', { 'id': record[0] or record[1], 'type': 'Address', 'value': record[1] }, None)	

Then finally under the *TCEntity Array Outputs*, we want to output the entity array we created earlier, but *without* any of the Null records in it, so we'll add a key of `indicators` with a value of `prune(entity)` to output the pruned list of entities.

TCEntity Array Outputs		
Key	Value	
indicators	prune(entity)	

Voila! Let's see what it output:

Expressions

indicators (TCEntityArray)

```
[
  {
    "id": "120.79.128.109",
    "type": "Address",
    "value": "120.79.128.109"
  },
  {
    "id": "204.48.23.94",
    "type": "Address",
    "value": "204.48.23.94"
  }
]
```