EXPRESSIONS

A New Way To Process ThreatConnect™ Data
v1.0.6

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)]`.

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.

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.

> 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.

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'`.

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.

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

PI 3.141592653589793

TAU 6.283185307179586
Which astute readers may recognize as 2 * `pi`

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

`acos(x)`
Arc Cosine of X

`acosh(x)`
: Inverse Hyperbolic Cosine

`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)

`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

`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

`degrees(x)`
: Convert X to degrees

`erf(x)`
: Error Function of X

`erfc(x)`
: Complimentary Error Function of X

`exp(x)`
: Math Exp of X

`expm1(x)`
: Math Expm1 of X

`factorial(x)`
: Factorial of X

`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, **kwargs)`
: Format string S according to Python string formatting rules. Compound structure elements are accessed with bracket notation and without quotes around key names, e.g. `blob[0][events][0][source][device][ipAddress]`

`fuzzydist(hash1, hash2)`
: Return the edit distance between two fuzzy hashes

`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

`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

`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

```
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

```
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

```
pow(x, y)
```
Math X ** Y

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

```
prune(ob, depth=None, prune=(None, '', [], {}))
```
Recursively Prunes entries from the object, with an optional depth limit

```
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

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

```
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

```
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

`twoscompliment(n, bits=32)`
: Return the twos compliment 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)`
: Updates one dictionary with keys from the other

`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.

`values(ob)`
: Values of dictionary

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)))
```

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.

Sample Recipes

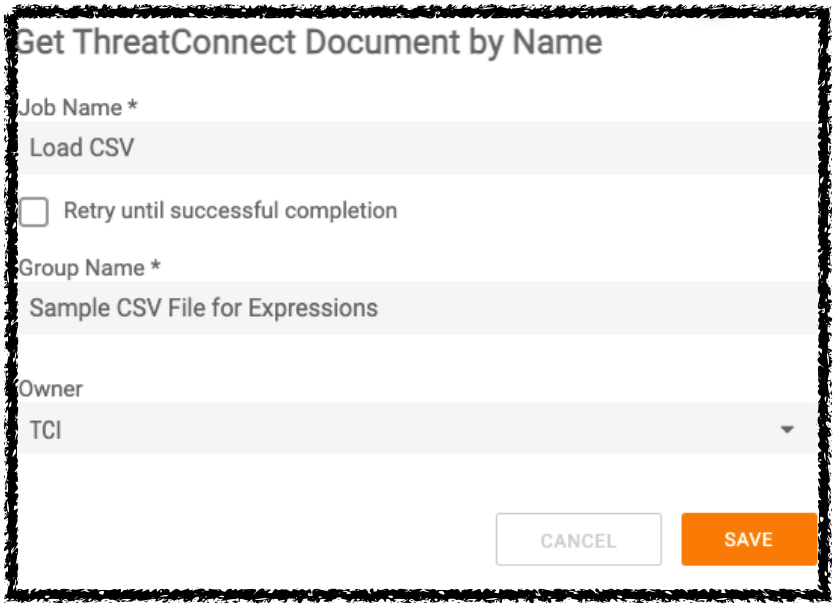
Turn a CSV file into a TCEntityArray

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:

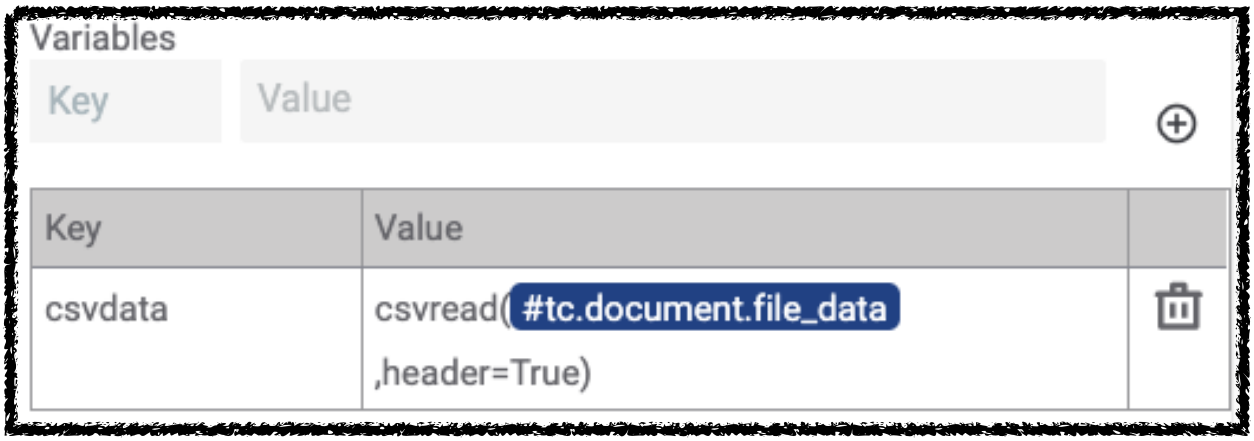
Hostname	IP Address	Conditions
google.com	172.217.0.14	
	120.79.128.109	C&C
	204.48.23.94	C&C

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



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

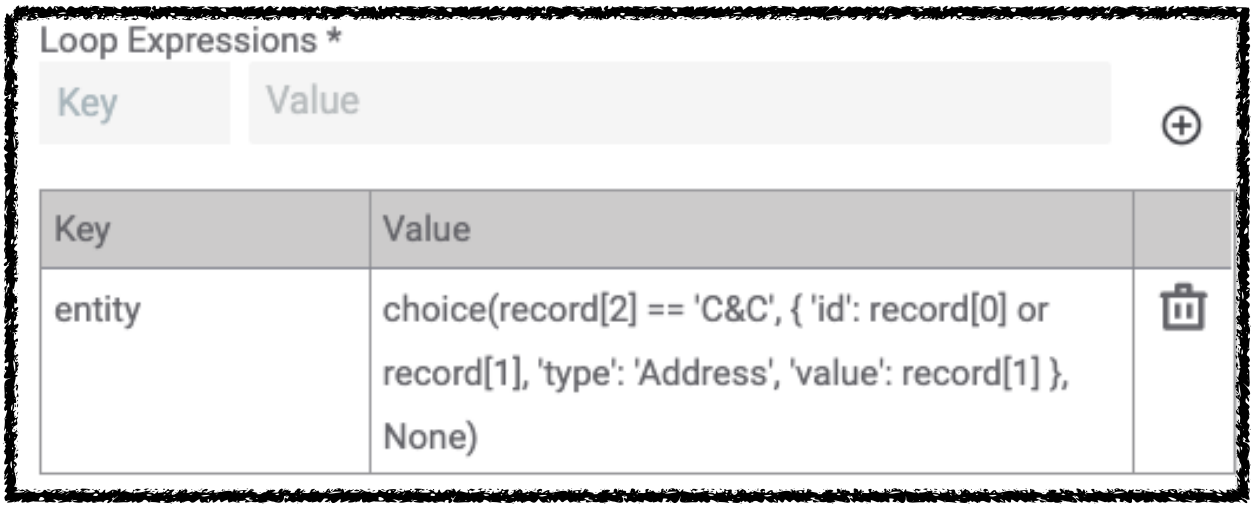
Starting out, we want to import the data from the Document, so under *Variables*, add the key `csvdata` with the values `csvread(#tc.document.file_data, header=True)` and click the circled plus to add the key. It has a header record we need to throw away, so `header=True` is specified for `csvread`.



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.

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.



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.

Voila! Let's see what it output:

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"
	}
	]