Examples

Table of Contents

▶ Expand

Append
Apply
CharacterArray
Convert
Copy
Enumerate
Equivalent
Exists
Fill
Filter
Flatten
Homogeneous
Intersection
Map
Match
Maximum
Merge
Minimum
Ones
Outersection
Pop
PostFixed
PreFixed
Range
Remove
Resolve
Reverse
Transpose
Zip

Notes

▶ Expand

i.e.,

```
# The elements stored by a VBA array will be of the form:
[1, 2, 3, 4, 5]

# The elements stored by a VBA collection will be of the form:
```

```
('Apple', 'Orange', 'Banana', 'Kiwi', 'Mango')

# The {key: item} pairs stored by a VBA Scripting.Dictionary will be of the form:
{'Apple': 20, 'Orange': 3, 'Banana': 5, 'Kiwi': 14, 'Mango': 11}
```

More detail on the analogue in both **Python** & **VBA** for the above can be found in the appendix.

Methods

Append

> Adds an element to the end of any of the supported data structures

Syntax

> *DS*.Append(*body*, *appendix*)

Example

▶ Categorizing a to-do list by adding items to dictionaries (Expand)

```
Dim todo_list As Variant
Dim task As Variant, task_category As String, task_description As String

Dim housework As Collection
Set housework = New Collection

Dim errands As Scripting.Dictionary
Set errands = New Scripting.Dictionary

todo_list = Array("Clean:Living Room", _
                  "Clean:Kitchen", _
                  "Clean:Room", _
                  "Repair:Leaking Faucet", _
                  "Buy:Shoe Rack;1", _
                  "Clean:Window Sills", _
                  "Cook:Lasagna", _
                  "Buy:Detergent;1 Bottle", _
                  "Buy:Milk;2 Cartons", _
                  "Buy:Wool Socks;4 Pairs", _
                  "Return:Library Books;3", _
                  "Return:Faulty Speakers;1")
For Each task In todo_list
    DS.Map(Split(task, ":"), task_category, task_description) ' See Method: Map
    Select Case task_category
        Case "Clean", "Cook", "Repair"
            DS.Append housework, task_description
        Case "Buy", "Return"
            DS.Append errands, Split(task_description, ";")
    End Select
Next task
```

housework

```
('Living Room', 'Kitchen', 'Room', 'Leaking Faucet', 'Window Sills', 'Lasagna')
```

errands

```
{'Shoe Rack': '1', 'Detergent': '1 Bottle', 'Milk': '2 Cartons', 'Wool Socks': '4
Pairs', 'Library Books': '3', 'Faulty Speakers': '1'}
```

Apply

Parameters:

Variable	Data Type(s)	Description
DataStructure_Arr	Variant()	The data structure containing elements to be modified
func_name	String	The name of the function being applied
arg_pos	Integer	The index position of the argument being supplied on each loop through the data structure
other_args	(ParamArray) Variant()	Any of the other arguments required by the function in order

Returns:

Example 1: [Vowel Shifting]

> Return
>
> > Given the following user defined function:

```
' This function shifts vowels by 1, relative to their ASC representation
Function shift_vowels(ByVal char As String) As String
    shift_vowels = IIf(InStr("aeiou", LCase(char)) > 0, _
                        Chr(Asc(char) + 1), _
                        char)
End Function
```

```
Dim arr As Variant
Dim obfuscated_text As Variant
Dim quote As String
quote = "All that is gold does not glitter, " & _
        "Not all those who wander are lost"
arr = DS.CharacterArray(quote)       '['A', 'l', 'l', ' ', 't', 'h', 'a', 't', ...
, 'l', 'o', 's', 't']
obfuscated_text = Join(DS.Apply(arr, "letter_shift", 0), "")
```

> All vowels shifted:
>
> > Bll thbt js gpld dpfs npt gljttfr, Npt bll thpsf whp wbndfr brf lpst

Example 2: [Simple Math]

```
Dim numbers As Variant
Dim output As Variant
arr = Array(2, 4, 6, 8)
output = DS.Apply(arr, "2*", 0)
```

> All elements doubled:

```
[4, 8, 12, 16]
```

Please note that the feature is fairly limited at the moment since it makes use of the restrictive Eval (Access) / Application.Evaluate (Excel) function

- The mathematical operator and any numbers must come before the element in the array
- The example above creates an expression of the form:

```
Application.Evaluate("2*" & "(" & element & ")") 'Where [element] is the control
in the for-each-loop governing the output array sourced from the input data
structure
```

CharacterArray

Parameters:

Variable	Data Type(s)	Description
text	String	The text that needs to be split into separate characters
---	---	---
---	---	---
---	---	---

Returns:

> An array with UBound = len(text) - 1, having each character of the provide String as a separate element

```
' Comment
Dim book_title as String
Dim all_characters as Variant
book_title = "La Belle Sauvage"
all_characters = DS.CharacterArray(book_title)
```

```
['L', 'a', ' ', 'B', 'e', 'l', 'l', 'e', ' ', 'S', 'a', 'u', 'v', 'a', 'g', 'e']
```

Convert

Parameters:

Variable	Data Type(s)	Description
DataStructure	Variant(), Collection, Dictionary	---
OutputType	String	The name of the data type
ConversionOptions	Variant	No current implementation
keys	Variant()	For conversion to dictionaries and collections, array of keys with the same number of elements as the DataStructure

Returns:

The data structure converted into whatever format was specified

Example 1: Basic Conversions

```
Dim dict As Scripting.Dictionary
Dim ingredients() As String
Dim keys As Variant
Dim in_stock() As Boolean

Set dict = New Scripting.Dictionary
```

Example 2: Array(s) to dictionary of collections

Napkin-Math, except done in bulk.

A quick cost-performance analysis: maximum point load under fixed cantilever loading conditions of various metals

The following example uses the ***Convert*** *method to associate several pieces of linked data.*

Material property sources:

- https://www.engineeringtoolbox.com/young-modulus-d_417.html
- https://www.mcmaster.com
- https://www.aerospacemetals.com/aluminum-distributor.html

> Length
>
> > L = 3 ft = 36 in

> Cross Section
>
> > a = b = 0.25 in (Square)

> Distance to Neutral Axis
>
> > y = 0.5*a
> >
> > > = 0.125 in

> Area Moment of Inertia
>
> > $I = a^4/12$
> >
> > > $= 3.26e^{-4}$ in^4

> Max Force, F_{max}, on a cantilever before yielding:

> F = σ I / y L

```
Function force_at_yield(ByVal yield_strength As Double, ByVal area_mom_inert As
Double, ByVal neutral_axis_dist As Double, ByVal length As Double) As Double
    force_at_yield = (yield_strength * area_mom_inert) / (neutral_axis_dist *
length)
End Function
```

```
Function deflection_at_yield(ByVal force As Double, ByVal length As Double, ByVal
elast_mod As Double, ByVal area_mom_inert As Double) As Double
    deflection_at_yield = (force * (length ^ 3)) / (3 * elast_mod *
area_mom_inert)
End Function
```

```
Dim l1 As Double, ArMoIn As Double, y_neut As Double, dataset As Variant
Dim mat_name As String, md As Variant, max_force As Double, max_defl As Double
Dim materials As Variant
Dim elastic_moduli As Variant
Dim yield_strengths As Variant
Dim mcm_ids As Variant
Dim prices_per_lineal_ft As Variant
Dim material_properties As Variant
Dim header_keys As Variant
Dim property_set As Variant, props_col As Collection, prop_dictionary As
Scripting.Dictionary

' SHAPE -----------------------------------
'Length
l1 = 1 * 12 ' in.
'Area Moment of Inertia
ArMoIn = 0.000326 'in ^ 4
'Distance to Neutral Axis
y_neut = 0.125 'in
' -----------------------------------------

' MATERIAL PROPERTIES ---------------------
'name
materials = Array("Aluminum:Anodized Multipurpose 6061", "Aluminum:Architectural
6063", "Aluminum:High-Strength 2024", "Aluminum:Easy-to-Machine 2011", "Low-Carbon
Steel Bar 1018", "Ultra-Machinable 12L14 Carbon Steel Bars", "A2 Tool Steel")
'modulus of elasticity
elastic_moduli = Array(10000, 10000, 10600, 10150, 29700, 29000, 27500) 'ksi
elastic_moduli = DS.Apply(elastic_moduli, "1000*", 0) 'ksi -> psi
'yield strength
yield_strengths = Array(35000, 16000, 47000, 38000, 54000, 60000, 51000) 'psi
'McMaster
mcm_ids = Array("6023K35", "89755K69", "86895K81", "3031N2", "9143K13",
"6547K112", "9019K95")
'price
prices_per_lineal_ft = Array(24.99 / 3, 8.39 / 8, 57.29 / 6, 17.94 / 6, 9.59 / 6,
42.97 / 6, 123.21 / 6) '$/ft
' ----------------------------------------

header_keys = Array("name", "modulus of elasticity", "yield strength", "McMaster",
"price")
material_properties = DS.Zip(materials, elastic_moduli, yield_strengths, mcm_ids,
prices_per_lineal_ft)
Set prop_dictionary = New Scripting.Dictionary

For Each property_set In material_properties
    Set props_col = DS.Convert(property_set, "Collection", keys:=header_keys)
    prop_dictionary.Add Key:=props_col("name"), Item:=props_col
Next property_set

Debug.Print "Analysis Results for .25x.25 in^2 square bar, length: " & l1 & "in"
Debug.Print "Loading configuration: Point-load, cantilever"
Debug.Print
For Each dataset In DS.Zip(prop_dictionary)
    mat_name = dataset(0) 'material name
    Set md = dataset(1) 'material dataset
    max_force = force_at_yield(md("yield strength"), ArMoIn, y_neut, l1)
    max_defl = deflection_at_yield(max_force, l1, md("modulus of elasticity"),
ArMoIn)

    Debug.Print "Material: [" & mat_name & "]"
    Debug.Print Tab(10); "Yield occurs under [" & Format(CStr(max_force), "#.##")
& "] lbs after deflecting [" & Format(CStr(max_defl), "#.###") & "] inches."

    Debug.Print Tab(5); "Price: " & Format(CStr(md("price") * (l1 / 12)), "$#.##")
    Debug.Print
Next dataset
```

> Analysis Results for .25x.25 in^2 square bar, length: 12in

> **Loading configuration**: Point-load, cantilever

> **Material**: [Aluminum:Anodized Multipurpose 6061]
>
> > Yield occurs under [7.61] lbs after deflecting [1.344] inches.

> > Price: $8.33

Material: [Aluminum:Architectural 6063]

> > Yield occurs under [3.48] lbs after deflecting [.614] inches.

> > Price: $1.05

Material: [Aluminum:High-Strength 2024]

> > Yield occurs under [10.21] lbs after deflecting [1.703] inches.

> > Price: $9.55

Material: [Aluminum:Easy-to-Machine 2011]

> > Yield occurs under [8.26] lbs after deflecting [1.438] inches.

> > Price: $2.99

Material: [Low-Carbon Steel Bar 1018]

> > Yield occurs under [11.74] lbs after deflecting [.698] inches.

> > Price: $1.6

Material: [Ultra-Machinable 12L14 Carbon Steel Bars]

> > Yield occurs under [13.04] lbs after deflecting [.794] inches.

> > Price: $7.16

Material: [A2 Tool Steel]

> > Yield occurs under [11.08] lbs after deflecting [.712] inches.

> > Price: $20.54

Copy

Parameters:

N.b.:

- The keys of a collection must be supplied as an optional argument *(implementation in progress)*; on their own, the keys of a collection object are irretrievable.
- Nested objects will only be copied by reference.

Variable	Data Type(s)	Description
DataStructure	Variant(), Collection, Dictionary	The data structure to copy
failOnNestedObjects	Boolean	Pass parameter as True in order to cause the method to return an empty Variant/Collection/Dictionary instead of returning nested references; this will prevent the nested objects from being mutated
args	(optional) Variant	*Implementation in progress* - keys if attempting to copy a collection
---	---	---

Returns:

> A newly created array/collection/dictionary containing the same data from the supplied data structure. Collection keys will not be preserved.

```
' Comment
Dim arr As Variant
Dim col As Collection
Dim dict As Scripting.Dictionary

Dim new_arr As Variant
Dim new_col As Collection
Dim new_dict As Scripting.Dictionary

arr = Array(False, 1, 2, 3, "four", "five")
new_arr = DS.Copy(arr)

Debug.Print Join(DS.Apply(arr, "CStr", 0), ", ")
'
Debug.Print Join(DS.Apply(new_arr, "CStr", 0), ", ")


Set col = New Collection
col.add item:="Apple", key:="A"
col.add item:="Pear", key:="P"
col.add item:="Guava", key:="G"
Set new_col = DS.Copy(col)
Debug.Print "The original collection has elements: " & Join(DS.Convert(col,
"Variant()"), ";")
' Original collection has elements: Apple;Pear;Guava
Debug.Print "The new collection has elements: " & Join(DS.Convert(new_col,
"Variant()"), ";")
' The new collection has elements: Apple;Pear;Guava
```

```
Debug.Print col("A")
' Apple
```

```
Debug.Print new_col("A") 'Please note that the collection's keys were not
transferred
```

> Run-time error '5':

> Invalid procedure call or argument

```
Set dict = New Scripting.Dictionary
dict.add key:="1", item:="odd"
dict.add key:="2", item:="even"
dict.add key:="3", item:="odd"
dict.add key:="4", item:="even"
' There has got to be a faster way to check this

Set new_dict = DS.Copy(dict)

Debug.Print "The number 1 is " & dict("1")                   'odd
Debug.Print "The number 2 is " & dict("2")                   'even
Debug.Print "The number 1 is " & new_dict("1")               'odd
Debug.Print "The number 2 is " & new_dict("2")               'even
```

Enumerate

Parameters:

Variable	Data Type(s)	Default	Description
enumerable	Variant(), Collection, Dictionary	-	The data structure to enumerate
starting_idx	(optional) Variant	0	The starting number for the output list
increment	(optional) Variant	1	The amount that the index is changed after each subsequent item (not yet implemented - ip)
---	---	---	---

Returns:

> A "Zipped" array with the numbers at index 0 and the elements of the data structure at index 1 for each of the original data structure's elements

```
' Enumerate an array
Dim arr As Variant
Dim dict As Scripting.Dictionary
Dim pair As Variant, output As Variant

arr = Array("e", "f", "g", "h", "i", "j")
output = DS.Enumerate(arr, 5, 1)
For Each pair In output
    Debug.Print CStr(pair(0)) & ". " & pair(1)
Next pair
```

> 5. e
> 6. f

> 7. g
> 8. h
> 9. i
> 10. j

```vba
' Enumerate a collection
Dim col As Collection
Dim pair As Variant, output As Variant
Set col = New Collection
col.add "10K Ohm Resistor"
col.add "0.22uF Capacitor"
col.add "RGB LED"
col.add "100W PSU"
col.add "16bit ADC"

output = DS.Enumerate(col, 1, 1)
For Each pair in output
    Debug.Print pair(0) & " - " & pair(1)
Next pair
```

> 1 - 10K Ohm Resistor
> 2 - 0.22uF Capacitor
> 3 - RGB LED
> 4 - 100W PSU
> 5 - 16bit ADC

```vba
' Enumerate a collection
Dim dict As Scripting.Dictionary
Dim pair As Variant, output As Variant
Dim comparative_descriptors As Variant, description As String
Set dict = New Scripting.Dictionary
dict.add key:="key 1 will be lost", item:="Python"
dict.add key:="key 2 will be lost", item:="TypeScript"
dict.add key:="key 3 will be lost", item:="PHP"
dict.add key:="key 4 will be lost", item:="Rust"
dict.add key:="key 5 will be lost", item:="Julia"
dict.add key:="key 6 will be lost", item:="Haskell"
comparative_descriptors = Array("more intuitive", "more versatile", "better",
"easier to read", "more challenging", "more fun", "cooler", "equipped with a wider
array of feature rich IDE", "more future-proof")
output = DS.Enumerate(dict, 1, 1)
For Each pair in output
    desc_idx = int(rnd * (ubound(comparative_descriptors) + 1))
    description = DS.Pop(comparative_descriptors, desc_idx)
    Debug.Print pair(0) & ": It's not that VBA isn't fun, I just think that " &
pair(1) & " is " & description
Next pair
```

> **Randomly generated statements**:
> 1: It's not that VBA isn't fun, I just think that Python is more fun.
> 2: It's not that VBA isn't fun, I just think that TypeScript is easier to read.
> 3: It's not that VBA isn't fun, I just think that PHP is better.
> 4: It's not that VBA isn't fun, I just think that Rust is cooler.
> 5: It's not that VBA isn't fun, I just think that Julia is equipped with a wider array of feature rich IDE.
> 6: It's not that VBA isn't fun, I just think that Haskell is more versatile.

Equivalent

Parameters:

Variable	Data Type(s)	Description
DataStructure1	Variant(), Collection, Dictionary	The first data structure
DataStructure2	Variant(), Collection, Dictionary	The second data structure
---	---	---
---	---	---

Returns:

> Boolean: True if the elements within each data structure are the same, otherwise False

Example: Alert the chef in case of customer allergies

```
' Given some function that returns an array of ingredients that the customer
cannot consume
Public Function get_requested_dish(some_web_connection_interface As
MythicalFeature) As Variant
'...
End Function
```

```
' Given some function that returns an array of ingredients that the customer
cannot consume
Public Function get_allergies(data_from_web_form As CopyAndPasteFromTheIntern) As
Variant
'...
End Function
```

```
' And another function which returns an array with the required ingredients for a
recipe
```

```
Public Function get_recipe(recipe_name As String) As Variant
'...
End Function
```

```
Dim recipe As Variant
Dim requested_dish As String
Dim allergy_filtered_recipe As Variant
Dim customer_allergies As Variant
Dim requires_custom_recipe As Boolean

requested_dish = get_requested_dish(global_connection)      ' Chocolate Chip
Cookies
recipe = get_recipe(requested_dish)
```

```
['Flour', 'Milk', 'Granulated Sugar', 'Chocolate', 'Egg', 'Butter', 'Cinnamon',
'Vanilla Extract', 'Baking Powder']
```

```
customer_allergies = get_allergies("json.txt") ' Sent by intern via email,
subject: Please see the attached json file
```

> After formatting:

- Peanut
- Milk
- Butter
- Shellfish

```
Set edit_recipe = DS.Convert(DS.Zip(recipe, DS.Range(0, ubound(recipe))))

For Each allergy In customer_allergies
    If DS.Exists(needle:=allergy, haystack:=edit_recipe.keys) Then
        edit_recipe.Remove(allergy)
    End If
Next allergy

requires_custom_recipe = Not DS.Equivalent(recipe, edit_recipe.keys) 'True
If requires_custom_recipe Then
    alert_message = "Custom recipe for [" & requested_dish & "] is needed." _
        & vbCrLf _
        & "Replacements needed for: " & Join(customer_allergies, ", ") _
        & vbCrLf _
        & "in addition to standard ingredients: " & Join(edit_recipe.Keys, ", ")
    alert_the_cook alert_message
End If
```

```
'to do: difference method
```

After formatting:

> Custom recipe for [Chocolate Chip Cookies] is needed.
> Replacements needed for:
>
> > - Peanut
> > - Milk
> > - Butter
> > - Shellfish
>
> Standard ingredients:
>
> > - Flour
> > - Granulated Sugar
> > - Chocolate
> > - Egg
> > - Cinnamon
> > - Vanilla Extract
> > - Baking Powder

Exists

Parameters:

Variable	Data Type(s)	Description
needle	Variant	The thing to find
haystack	Variant(), Collection, Dictionary	Where to look
wildcard_needle	Boolean	Toggle whether to use the 'like' operator on the [needle] argument
wildcard_haystack	Boolean	Toggle whether to use the 'like' operator on the haystack

Returns:

> Boolean: True if the needle is in the haystack, False otherwise

Example: *Given a set of painting colors (requested supplies) and a list of what's in stock... Check whether any of the the following are true:*

- An exact color-match exists (by comparing color names)

- A potential close-color-match exists (by comparing color name)

```
    Dim supplies As Variant
    Dim needles As Variant, needle As Variant
    Dim results As Collection, results_wc_needle As Collection,
results_wc_supplies As Collection, results_wc_all As Collection 'wc:  wildcard
    Dim zipped_results As Variant

    Set results = New Collection
    Set results_wc_needle = New Collection
    Set results_wc_supplies = New Collection
    Set results_wc_all = New Collection

    needles = Array("red", "green", "blue", "off-white", "magenta")
    supplies = Array("red", "mint green", "black", "yellow", "sunset orange",
"crimson", "light blue", "white")

    For Each needle In needles ' "red", "green", "blue", "off-white", "magenta"
        results.Add Array(needle, DS.Exists(needle, supplies))
        results_wc_needle.Add Array(needle, DS.Exists(needle, supplies,
wildcard_needle:=True, wildcard_supplies:=False))
        results_wc_supplies.Add Array(needle, DS.Exists(needle, supplies,
wildcard_needle:=False, wildcard_supplies:=True))
        results_wc_all.Add Array(needle, DS.Exists(needle, supplies,
wildcard_needle:=True, wildcard_supplies:=True))
    Next needle

    zipped_results = DS.Zip(results, results_wc_needle, results_wc_supplies,
results_wc_all)
```

> Result:

Color	Exists w/o modifier	Exists w/ wildcard "needle"	Exists w/ wildcard "haystack"	Exists w/ both wildcards applied
red	True	True	True	True
green	False	True	False	True
blue	False	True	False	True
off-white	False	False	True	True
magenta	False	False	False	False

> Explanation:
>
> > - **Red** matches because there is an exact match of color in existing supplies
> > - **Green** doesn't have an exact match but matches in the statement mint_green (wc applied to supplies) like "*green*" (wc applied to requested)

> > - The same applies to "**blue**", with "light blue"
> > - **off-white** doesn't have an exact match but "*white*" is like "*off-white*"
> > - **Magenta** has no match, nor does it have a wildcard match

---

Fill

Parameters:

Variable	Data Type(s)	Description
container	Variant(), Collection, Dictionary, Integer	The data structure into which elements will be inserted
---	---	---
---	---	---
---	---	---

- **container**: Variant() | Collection | Dictionary | Integer
  - The data structure into which elements will be filled
  - OR: The number of elements in the newly created array
- **stuff**: Variant
  - The "*filling*"
- **extra_serving_size**: Integer (Optional)

Returns:

> Data structure with type corresponding to the provided data structure, defaulting to Variant() if no data structure is given

```
' Create a new array & fill with 5 instances of Integer value 1
Dim arr As Variant
arr = DS.Fill(5, 1)
```

> [1, 1, 1, 1, 1]

```
' Fill a fixed-size array of upper bound 3 with instances of Integer value 5
Redim arr(3)
DS.Fill arr, 5
```

> [5, 5, 5, 5]

Filter

Parameters:

Variable	Data Type(s)	Optional	Default	Description
DataStructure	Variant(), Collection, Dictionary	No	-	The data structure to filter
operator_expression	String	No	-	The operator, as a string, to apply to the elements of the data structure
compare_against	Variant	No	-	the comparison value to apply the operator against
other_paired_expressions	Variant	Yes	-	No current implementation - in progress
FilterMode	eDataStructureFilterMode	Yes	eFilterTrap	Whether to keep or discard the elements in the supplied data structure that meet the criteria

Enum:

```
Public Enum eDataStructureFilterMode
    eFilterTrap
    eFilterOut
End Enum
```

Returns:

> data structure with [eFilterTrap], or without [eFilterOut] elements meeting the conditional derived from the supplied arguments

Quick Examples:

```
Dim arr As Variant
Dim filtered_arr As Variant
```

```
' Filter numbers from an array
arr = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

filtered_arr = DS.Filter(arr, ">", 4) 'FilterMode is unspecified, default
(eFilterTrap)
' 5, 6, 7, 8, 9, 10

filtered_arr = DS.Filter(arr, ">", 4, FilterMode:=eFilterOut)
' 1, 2, 3, 4

filtered_arr = DS.Filter(arr, "between", Array(3, 8))
' 3, 4, 5, 6, 7, 8

filtered_arr = DS.Filter(arr, "=", 1)
' 1
```

```
arr = Array(60, 1, 4.2, 5.8, 20.1, 100, 101, 11.573, 46.642, 174, 200.3, 58,
121.4)
filtered_arr = DS.Filter(arr, "outside", Array(40, 60))
' 1, 4.2, 5.8, 20.1, 100, 101, 11.573, 174, 200.3, 121.4
```

```
' Remove past-tense verbs
Dim verbs As Variant
verbs = Array("Look", "Looked", "Filter", "Filtered", "Attempt", "Attempted",
"Load", "Loaded")
filtered_arr = DS.Filter(verbs, "like", "*ed", FilterMode:=eFilterOut)
' Look, Filter, Attempt, Load

'OR
filtered_arr = DS.Filter(verbs, "not like", "*ed")
' Look, Filter, Attempt, Load
```

Accepted operator expressions

Operator Expression (and aliases)	Expected Comparison	Description
=	Value	Tests for equality
<>	Value	Tests for inequality
is	Value or Object	Tests equality for value variables, tests same reference with objects
is not, isn't	Value or Object	Tests inequality for value variables, tests different reference with objects

Operator Expression (and aliases)	Expected Comparison	Description
>	Number	Tests whether elements are above the comparison value
>=	Number	Tests whether elements are at least the comparison value
<	Number	Tests whether elements are below the comparison value
<=	Number	Tests whether elements are at most the comparison value
like	String	Tests whether string elements resemble the comparison text
not like, liken't	String	Tests whether string elements do not resemble the comparison text
in, is in	Array	Tests whether elements have a match within the comparison array
not in, is not in	Array	Tests whether elements do not have a match within the comparison array
inside, between	Array(low, high)	Tests whether values fall on or within bounds
out, outside, beyond	Array(low, high)	Tests whether values fall out of bounds

Flatten

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
Dim nested As Variant, flattened As Variant
nested = Array(1, 2, 3, _
                Array(4, 5, 6), _
                Array( _
                        Array(7, 8), _
                        9, _
```

```
                                          Array(10)))
flattened = DS.Flatten(nested)
```

> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Homogeneous

TypeName checks all elements within a data structure and returns the typename if they're the same type, otherwise returns False

Debating whether to change the return to empty string (?) on mismatch

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Example
Dim thing1 As Variant, thing2 As Variant, thing3 As Variant, thing4 As Variant,
thing5 As Variant, thing6 As Variant

Set thing1 = New Collection
Set thing2 = New Collection
Set thing3 = New Collection
Set thing4 = New Collection
Debug.Print "Type is: " & DS.Homogeneous(thing1, thing2, thing3, thing4)
' Type is: Collection

Set thing5 = New Scripting.Dictionary
Set thing6 = 1
Debug.Print "Type is: " & DS.Homogeneous(thing1, thing2, thing3, thing4, thing5,
thing6)
' Type is: False
```

> Result

Intersection

' IN PROGRESS

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Map

Pseudo-In-line value assignment

Variable	Data Type(s)	Description
source_values	Variant(), Collection	---
variables	ParamArray	variable names to distribute values to
---	---	---
---	---	---

Pending: single value mapped to all variables

Pending #2: Object compatibility

Returns:

> Return

```
' Example
Dim text As String, num As Integer, big_num As Long, small_num As Single, any_num
As Variant
```

```
DS.Map Array("Single statement assignment: ", 100, 2147483647, .0625, 11), text,
num, big_num, small_num, any_num

Debug.Print text & "num: " & num & " big_num: " & big_num & " small_num: " &
small_num & " any_num: " & any_num
' Single statement assignment: num: 100 big_num: 2147483647 small_num: 0.0625
any_num: 11
```

```
' Example: More values than containers
Dim value_source As Variant
Dim n1, n2, n3, n4, n5, n6, n7, n8, n9, n10
value_source = DS.Range(1, 20) '20 values
DS.Map value_source, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10 '10 containers
Debug.Print Join(Array(n1, n2, n3, n4, n5, n6, n7, n8, n9, n10), ", ")
' 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
' Example: More containers than values
Dim value_source As Variant
Dim n1, n2, n3, n4, n5, n6, n7, n8, n9, n10
value_source = DS.Range(1, 5) '5 values
DS.Map value_source, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10 '10 containers
Debug.Print Join(Array(n1, n2, n3, n4, n5, n6, n7, n8, n9, n10), ", ")
' 1, 2, 3, 4, 5, , , , ,
```

> Result

Match

Mix between Select Case statement & switch Statement See [Filter] method for compatible operators

Variable	Data Type(s)	Description
things	Variant	---
operator	String	---
matched_value1	Variant	---
output1	Variant	---
matched_value2	Variant	---
output2	Variant	---
matching_pairs	ParamArray	any other pairs

> This method is roughly equivalent to:

```
foo = Switch(thing <operator> matched_value1, output1, _
            thing <operator> matched_value2, output2, _
            .
            .
            .
            thing <operator> matched_valueN, outputN)
```

Returns:

> some specific output

```
' Example 1: Simple values - Convert Percentage to Letter Grade

Dim grade As String
Dim score As Variant
score = 75.64
grade = DS.Match(score, ">=", _
                    90, "A", _
                    80, "B", _
                    70, "C", _
                    60, "D", _
                    score, "Failure")
Debug.Print "Grade is: " & grade
' Grade is: C
```

```
' Example 2: Categorize the SQL statement
Dim sql1 As String, sql2 As String, sql3 As String
Dim sql_type As String
dim sql_statement As Variant
Dim type_col As Collection
Set type_col = New Collection
```

```
--sql1
SELECT * FROM mytable AS T1 WHERE [column1] = 1;

--sql2
SELECT * FROM mytable AS T1 INNER JOIN yourtable AS T2 ON T1.[column1] = T2.
[column1];

--sql3
DELETE * FROM mytable AS T1 WHERE [column3] = 'delete me';
```

```
For Each sql_statement In Array(sql1, sql2, sql3)
    Debug.Print "Type: [" & DS.Match(sql_statement, "like", _
                                            "*SELECT*FROM*JOIN*ON*", "SELECT
JOIN", _
                                            "*SELECT*FROM*UNION*SELECT*FROM*",
"SELECT UNION", _
                                            "*SELECT*INTO*", "SELECT INTO", _
                                            "*SELECT*FROM*WHERE*", "SELECT", _
                                            "*INSERT*INTO*SELECT*", "INSERT
SELECT", _
                                            "*INSERT*INTO*VALUES*", "INSERT
VALUES", _
                                            "*UPDATE*SET*WHERE*", "UPDATE", _
                                            "*DELETE*FROM*WHERE*", "DELETE") _
                        & "] - " & sql_statement
Next sql_statement
' Output
' Type: [SELECT] - SELECT * FROM mytable AS T1 WHERE [column1] = 1;
' Type: [SELECT JOIN] - SELECT * FROM mytable AS T1 INNER JOIN yourtable AS T2 ON
T1.[column1] = T2.[column1];
' Type: [DELETE] - DELETE * FROM mytable AS T1 WHERE [column3] = 'delete me';
```

But why would this be in the data structures class if it couldn't operate on data structures?

```
' Example 3: Operating on a data structure
' Taking the last example - the following syntax will return an array

Dim multiple_sql_statements As Variant
Dim categorized As Variant

multiple_sql_statements = Array(sql1, sql2, sql3)
categorized = DS.Match(multiple_sql_statements, "like", _
                                            "*SELECT*FROM*JOIN*ON*", "SELECT
JOIN", _
                                            "*SELECT*FROM*UNION*SELECT*FROM*",
"SELECT UNION", _
                                            "*SELECT*INTO*", "SELECT INTO", _
                                            "*SELECT*FROM*WHERE*", "SELECT", _
                                            "*INSERT*INTO*SELECT*", "INSERT
SELECT", _
                                            "*INSERT*INTO*VALUES*", "INSERT
VALUES", _
                                            "*UPDATE*SET*WHERE*", "UPDATE", _
                                            "*DELETE*FROM*WHERE*", "DELETE"))
Debug.Print "sql1, sql2, & sql3 are " & Join(categorized, ", ") & " types,
respectively."
' sql1, sql2, & sql3 are SELECT, SELECT JOIN, DELETE types, respectively.
```

Maximum

Returns the maximum value out of all elements in the provided data structure.

Variable	Data Type(s)	Description
DataStructure	Variant(), Collection, Dictionary	The data structure containing the values to check
---	---	---
---	---	---
---	---	---

Returns:

> Variant

```
' Comment
Dim code
```

> Result

Merge

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Minimum

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Ones

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Outersection

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Pop

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

PostFixed

Variable	Data Type(s)	Description

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

PreFixed

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Range

Variable	Data Type(s)	Description

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Remove

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Resolve

Variable	Data Type(s)	Description

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```vb
' Comment
Dim code
```

> Result

Reverse

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```vb
' Comment
Dim code
```

> Result

Transpose

Variable	Data Type(s)	Description

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Zip

Variable	Data Type(s)	Description
---	---	---
---	---	---
---	---	---
---	---	---

Returns:

> Return

```
' Comment
Dim code
```

> Result

Notes

Appendix

Analagous Shorthand Python-VBA

Array ~ List

```
' The elements stored by a VBA array
Array(1, 2, 3, 4, 5)
```

```
# Python Analogue:
[1, 2, 3, 4, 5]
```

Collection ~ Tuple

```
' The items comprising the Collection in variable col after executing the
following:
Dim col as Collection, arr As Variant, fruit As Variant
Set col = New Collection
arr = Array("Apple", "Orange", "Banana", "Kiwi", "Mango")
For Each fruit In arr
    col.add item:=fruit
Next fruit
```

```
# Python Analogue:
("Apple", "Orange", "Banana", "Kiwi", "Mango")
```

Dictionary ~ Dictionary

```
' The {Key:Item} pairs comprising the Scripting.Dictionary in variable dict after
executing the following:
Dim dict as Scripting.Dictionary, fruit As Variant, quantities As Variant, i As
Integer
Set dict = New Scripting.Dictionary
fruit = Array("Apple", "Orange", "Banana", "Kiwi", "Mango")
quantities = Array(20, 3, 5, 14, 11)
For i = 0 To Ubound(fruit)
    dict.add Key:=fruit(i), Item:=quantities(i)
Next fruit
```

```
# Python Analogue:
{'Apple': 20, 'Orange': 3, 'Banana': 5, 'Kiwi': 14, 'Mango': 11}
```