# Groups

Groups are powerful objects. With them you can do things that would be much harder in other languages. A group is a collection of other objects. Any object whatsoever can be in a group: a number, a string, the value true, a cash-register, or any object you create. Even a group can be part of another group. The items in a group are its elements.

There are many different ways of working with groups, because there are many different ways of working with groups of objects. Sometimes you want to treat them simply as a collection of related objects. Sometimes the objects are kept in a specific order. In other languages there are stacks, queues, lists, and arrays. In Glyphic Script, there is only group, which serves them all.

## Basic Operations

g add e
add e to the group g
g remove e
remove the first occurrence of e, if it's there, from g
g.size
return the number of objects in g

These are the basic messages for handling a group. Use them when you don't particularly care about the order of items in a group. You can follow the effect of the following expressions by opening up a workspace, executing the first expression, then opening up the workspace variable a to view it. Then, as you execute each expression in turn, you can see the effect on the group:

| Expression | Contents of a | Comments |
|---|---|---|
| a := new group. | empty | creates a new, empty group |
| a add "cat". | cat | adds a string |
| a add "bird". | cat, bird | adds another |
| a add 123. | cat, bird, 123 | adds a number |
| a remove "bird". | cat, 123 | removes a string, gap closes |
| a add"cat". | cat, 123, cat | |

adds a second copy of a string
a remove "cat".
123, cat
removes the first one it finds
a remove 456.
123, cat
no effect, as 456 isn't there
a.size.

resulting size is 2

## List Operations

A group can be treated as an ordered list. Each element in the list has an index. The index runs from one to the size of the group. If you insert or remove a given item, all the items shift over.

g insert e after i
g insert e before i
insert e into the group after or before index i
g delete i

delete i-th item of the group
g place e after f
g place e before f
insert e into the group after or before the first occurrence of f
g @ i

returns the i-th item of the group
g @ i := e

replaces the i-th item of the group with e
g index of e
returns the index of the first occurrence of e, if there aren't any then ???

Expression

Contents of a

a := new group.
empty
a insert "cat" before 1.
cat
a insert "bird" after 1.
cat, bird
a place 123 before "bird".
cat, 123, bird
bird is now item 3
a delete 1.
123, bird
bird is back to item 2

a place "cat" after 123.
123, cat, bird
a delete 3.
123, cat
a @ 2 := "dog".
123, dog
replaces cat with dog
a @ 1.
123, dog
results in 123
a index of "dog"
123, dog
results in 2
a index of "cat"
123, dog
results in ???

## Queue Operations

Many tasks involve keeping track of objects in a queue. A queue is a computer science term for lists where items are added or removed only from the ends. For example: a cash-register tape is a kind of queue: new entries are only added to the end. A movie ticket line is a kind of queue: new people get on the end of the queue, and as tickets are sold, people leave the front.

While you could use the list operations above, there is a set of queue messages to make queue management easier. Queues have two ends, the front and back. The object at the front of the queue is first, and the object at the back is last. Pushing an object onto a queue means to add it from an end. Popping it means to remove it from an end. When you pop an object, the result of the operation is the object just removed.

g push e first add e to the front of the queue
g push e last add e to the end of the queue
g pop first remove and return the front of the queue
g pop last remove and return the end of the queue
g.first
return the front of the queue (without removing it)
g.last
return the end of the queue (without removing it)

Example Sequence:

Expression

Contents of a

```
a := new group.
                empty

a push "sue" last.
        sue
add sue

a push "bob".

                  sue, bob
last is the default

a push "jill" first.
      jill, sue, bob
jill gets ahead in line

a.first

jill, sue, bob
fi jill

a.last

                                  jill, sue, bob
fi bob

a pop last.

                     jill, sue
fi bob, take from end

a pop.

jill
 fi sue, last is default

a pop first.

                empty  fi jill
```

## Set operations

In all the above operations, a group could hold an object more than once. If you add "cat" to a group three times, there will be three "cats" in the group. Similarly, if there were two numbers in the group both 100, then removing 100, would only remove one of them. Sometimes it is desirable to only add an object to a group if it is not there, and remove every copy of it if it is. These messages do this:

g include e
add e to the group if it isn't there
g exclude e
remove e from the group, multiple times if needed

If a group doesn't have any element more than once, then the group can be thought of as a set. You can perform set operations on two groups, producing a third group from them:

g union f
a new group with every element in g and f
g intersect f
a new group with only elements that are in both g and f
g difference f
a new group with every element in g that is not in f

Example Sequence:

| Expression | Contents of a/b/result |
|---|---|
| a := new group. | empty |
| a add "sue". | sue |
| start off with a group | |
| a add "bob". | |
| sue, bob | |
| a add "sue". | sue, bob, sue |
| a include "bob". | |
| sue, bob, sue | |
| no change, already has bob | |
| a exclude "sue". | |
| bob | |
| removes both sues | |
| a include "jill". | |
| bob, jill | |
| adds jill | |
| b := new group. | empty |
| b include "bob". | bob |
| b include "sue". | |
| bob, sue | |
| a union b | fi bob, jill, sue |
| a new group | |
| a intersect b | fi bob |

only item in both
a difference b
fi jill

item in a, not in b

## General Operations

These messages are applicable to groups in almost all contexts.

g.empty
returns true if g is empty
g from i
g from i to j
g from i for j
returns a new group that contains the elements of g starting at index i. With no extra arguments, it takes all the elements to the end. With the to argument, it takes elements up to index j. With the for argument, it takes the next j elements.
g & f
returns a new group which has all the elements of g followed by all the elements of f
g has e
returns true if g has the element e
g count e
returns the number of times e is in g
choose g
returns a random element of g

These messages take blocks of code to control their operation.

g transform by [ $ element e, index i. … ]

Returns a new group that contains the results of executing the block for each element in g.

g reject by [ $ - e. … ]

Executes the block for each element in g and removes the element if the result of the block is true.

g sort
g sort by [ $ - a, - b. … ]
g sort by-value [ $ - a. … ]

These messages sort the group. In the first form, the elements are compared with the < operator. This works for numbers and strings, but most other objects don't understand the < operator. In these cases, you have two choices: the first form, with the by argument, you supply a block that compares the two block arguments and returns true if the first should sort before the second. The second form, with the by-value argument, you supply a block that returns for each element a value that can be compared with the < operator. The first form is more efficient than the second.

```
for g do [ $ element e, index i. … ]
for g reverse do [ $ element e, index i. …]
```

This is just the for statement discussed earlier in this manual.