SLiM

Workshop Series

#4: Eidos Overview

Benjamin C. Haller

Messer Lab, Cornell University

Data types

- `NULL`: no explicit value

Data types

- `NULL`: no explicit value
- `logical`: a Boolean true/false value (`T`/`F`)

Data types

- `NULL`: no explicit value
- `logical`: a Boolean true/false value (`T`/`F`)
- `integer`: a 64-bit signed integer (`10`, `–27`)

Data types

- `NULL`: no explicit value
- `logical`: a Boolean true/false value (`T`/`F`)
- `integer`: a 64-bit signed integer (`10`, `−27`)
- `float`: a floating-point number (`10.0`, `−2.7`)

Data types

- `NULL`: no explicit value
- `logical`: a Boolean true/false value (`T`/`F`)
- `integer`: a 64-bit signed integer (`10`, `–27`)
- `float`: a floating-point number (`10.0`, `–2.7`)
- `string`: a sequence of characters (`"foo"`)

Data types

- `NULL`: no explicit value
- `logical`: a Boolean true/false value (`T`/`F`)
- `integer`: a 64-bit signed integer (`10`, `–27`)
- `float`: a floating-point number (`10.0`, `–2.7`)
- `string`: a sequence of characters (`"foo"`)
- `object`: an instance of a class (`Individual`)

Data types

- `NULL`: no explicit value
- `logical`: a Boolean true/false value (`T`/`F`)
- `integer`: a 64-bit signed integer (`10`, `–27`)
- `float`: a floating-point number (`10.0`, `–2.7`)
- `string`: a sequence of characters (`"foo"`)
- `object`: an instance of a class (`Individual`)
- EVERYTHING IS A VECTOR!

Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`, `^`, `:`   `6 + 2*7`
- Logical: `&`, `|`, `!`   `T & !F`
- Comparison: `<`, `>`, `<=`, `>=`, `==`, `!=`   `2+2 == 4`
- Assignment: `=`   `x = 8`
- Precedence and function call: `()`   `(6+2) * 7`
- Subset: `[]`   `x[5]`
- Property access and method call: `.`   `foo.bar`
- Ternary conditional: `?else`

Objects, properties, methods

- Objects represent **entities**:
  - e.g., individuals, mutations, subpopulations

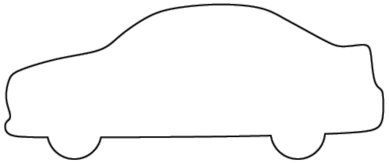
Objects, properties, methods

- Objects represent **entities**:
  - e.g., individuals, mutations, subpopulations
- Objects have **properties**:
  - attributes like `age`, `sex`, `spatialPosition`
  - `individual.age` returns the age of `individual`
  - `individual.age = 10;` changes its age

Objects, properties, methods

- Objects represent **entities**:
  - e.g., individuals, mutations, subpopulations
- Objects have **properties**:
  - attributes like `age`, `sex`, `spatialPosition`
  - `individual.age` returns the age of `individual`
  - `individual.age = 10;` changes its age
- Objects have **methods**:
  - methods perform complex operations
  - `individual.containsMutations(muts)`

Objects, properties, methods



object class: `Car`

object: `my_car`

properties:

```
my_car.make == "Honda"
my_car.model == "CR-V"
my_car.color == "red"
```

method calls:

```
my_car.driveTo(the_mall)
```

object: `i_wish`

properties:

```
i_wish.make == "Ferrari"
i_wish.model == "F12"
i_wish.color == "yellow"
```

method calls:

```
i_wish.driveTo(California)
```

Everything is a vector

- All values in Eidos are vectors
- A single integer, `10`, is really a vector, `10`
- A vector of exactly one value is a *singleton*

Everything is a vector

- All values in Eidos are vectors
- A single integer, `10`, is really a vector, `10`
- A vector of exactly one value is a *singleton*
- Most operations in Eidos are vectorized:
  - operators like `+`, `-`, `*`, `/`
  - property access on objects, like `foo.bar`
  - method calls on objects, like `foo.bar()`
- This allows Eidos to be fast!

Control flow

- **if-else**: conditional execution
  - `if (condition) statement; else statement;`

Control flow

- **if-else**: conditional execution
  - `if (condition) statement; else statement;`
- **while**: loop on a condition, 0+ times
  - `while (condition) statement;`

Control flow

- **if-else**: conditional execution
  - `if (condition) statement; else statement;`
- **while**: loop on a condition, 0+ times
  - `while (condition) statement;`
- **do-while**: loop on a condition, 1+ times
  - `do statement; while (condition);`

Control flow

- **if-else**: conditional execution
  - `if (condition) statement; else statement;`
- **while**: loop on a condition, 0+ times
  - `while (condition) statement;`
- **do-while**: loop on a condition, 1+ times
  - `do statement; while (condition);`
- **for**: loop over the values in a vector
  - `for (i in vector) statement;`

Compound statements

- Compound statements use braces, `{ ... }`

```
if (condition)
{
   a;
   b;
   c;
}
else
{
   d;
   e;
}
```

```
while (condition)
{
   a;
   b;
}
```

```
for (i in 1:100)
{
   a;
   b;
}
```

Built-in functions

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …
- **Distributions**: `rnorm()`, `rpois()`, `runif()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …
- **Distributions**: `rnorm()`, `rpois()`, `runif()`, …
- **Vectors**: `c()`, `rep()`, `seq()`, `sample()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …
- **Distributions**: `rnorm()`, `rpois()`, `runif()`, …
- **Vectors**: `c()`, `rep()`, `seq()`, `sample()`, …
- **Values**: `all()`, `any()`, `identical()`, `sort()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …
- **Distributions**: `rnorm()`, `rpois()`, `runif()`, …
- **Vectors**: `c()`, `rep()`, `seq()`, `sample()`, …
- **Values**: `all()`, `any()`, `identical()`, `sort()`, …
- **Output**: `cat()`, `print()`, `paste()`, `str()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …
- **Distributions**: `rnorm()`, `rpois()`, `runif()`, …
- **Vectors**: `c()`, `rep()`, `seq()`, `sample()`, …
- **Values**: `all()`, `any()`, `identical()`, `sort()`, …
- **Output**: `cat()`, `print()`, `paste()`, `str()`, …
- **Types**: `isFloat()`, `asFloat()`, …

Built-in functions

- **Math**: `abs()`, `ceil()`, `log()`, `setUnion()`, …
- **Statistics**: `max()`, `mean()`, `sd()`, `cov()`, …
- **Distributions**: `rnorm()`, `rpois()`, `runif()`, …
- **Vectors**: `c()`, `rep()`, `seq()`, `sample()`, …
- **Values**: `all()`, `any()`, `identical()`, `sort()`, …
- **Output**: `cat()`, `print()`, `paste()`, `str()`, …
- **Types**: `isFloat()`, `asFloat()`, …
- **Filesystem**: `readFile()`, `writeFile()`, …

Essential built-in functions

Essential built-in functions

- `c()`: Create a vector by concatenation
  - `c(2:5, 7, 9:11)` produces `2 3 4 5 7 9 10 11`

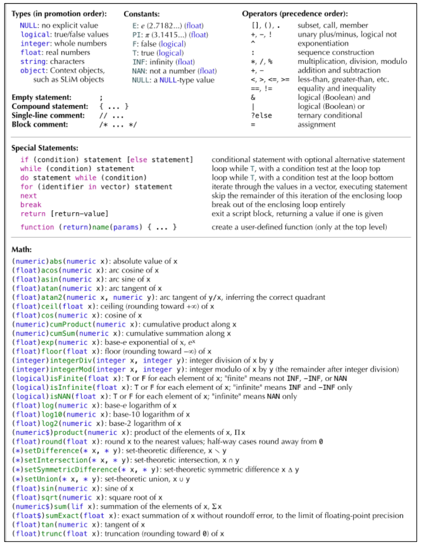
Essential built-in functions

- `c()`: Create a vector by concatenation
  - `c(2:5, 7, 9:11)` produces `2 3 4 5 7 9 10 11`
- `print()`, `cat()`, `catn()`: produce output
  - for printing one value / variable, `print()` is better
  - for formatted output, `cat()` / `catn()` is better

Essential built-in functions

- `c()`: Create a vector by concatenation
  - `c(2:5, 7, 9:11)` produces `2 3 4 5 7 9 10 11`
- `print()`, `cat()`, `catn()`: produce output
  - for printing one value / variable, `print()` is better
  - for formatted output, `cat()` / `catn()` is better
- `defineConstant()`: define a new constant
  - `defineConstant("X", 1:10)` defines `X` to be `1:10`
  - `X` will be defined globally and permanently

Eidos reference sheet



Types (in promotion order):

NULL: no explicit value
logical: true/false values
integer: whole numbers
float: real numbers
string: characters
object: Context objects,
such as SLiM objects

Constants:

E: *e* (2.7182...) (float)
PI: π (3.1415...) (float)
F: false (logical)
T: true (logical)
INF: infinity (float)
NAN: not a number (float)
NULL: a NULL-type value

Operators (precedence order):

[], (), .	subset, call, member
+, −, !	unary plus/minus, logical not
^	exponentiation
:	sequence construction
*, /, %	multiplication, division, modulo
+, −	addition and subtraction
<, >, <=, >=	less-than, greater-than, etc.
==, !=	equality and inequality
&	logical (Boolean) and
|	logical (Boolean) or
?else	ternary conditional
=	assignment

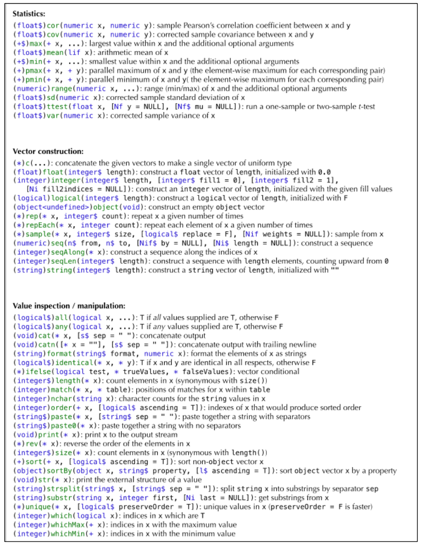
Empty statement:	;
Compound statement:	{ ... }
Single-line comment:	// ...
Block comment:	/* ... */

Special Statements:

if (condition) statement [else statement]	conditional statement with optional alternative statement
while (condition) statement	loop while T, with a condition test at the loop top
do statement while (condition)	loop while T, with a condition test at the loop bottom
for (identifier in vector) statement	iterate through the values in a vector, executing statement
next	skip the remainder of this iteration of the enclosing loop
break	break out of the enclosing loop entirely
return [return-value]	exit a script block, returning a value if one is given
function (return)name(params) { ... }	create a user-defined function (only at the top level)

Math:

(numeric)abs(numeric x): absolute value of x
(float)acos(numeric x): arc cosine of x
(float)asin(numeric x): arc sine of x
(float)atan(numeric x): arc tangent of x
(float)atan2(numeric x, numeric y): arc tangent of y/x, inferring the correct quadrant
(float)ceil(float x): ceiling (rounding toward +∞) of x
(float)cos(numeric x): cosine of x
(numeric)cumProduct(numeric x): cumulative product along x
(numeric)cumSum(numeric x): cumulative summation along x
(float)exp(numeric x): base-e exponential of x, e^x
(float)floor(float x): floor (rounding toward −∞) of x
(integer)integerDiv(integer x, integer y): integer division of x by y
(integer)integerMod(integer x, integer y): integer modulo of x by y (the remainder after integer division)
(logical)isFinite(float x): T or F for each element of x; "finite" means not INF, −INF, or NAN
(logical)isInfinite(float x): T or F for each element of x; "infinite" means INF and −INF only
(logical)isNAN(float x): T or F for each element of x; "infinite" means NAN only
(float)log(numeric x): base-*e* logarithm of x
(float)log10(numeric x): base-10 logarithm of x
(float)log2(numeric x): base-2 logarithm of x
(numeric$)product(numeric x): product of the elements of x, Π x
(float)round(float x): round x to the nearest values; half-way cases round away from 0
(*)setDifference(* x, * y): set-theoretic difference, x ∖ y
(*)setIntersection(* x, * y): set-theoretic intersection, x ∩ y
(*)setSymmetricDifference(* x, * y): set-theoretic symmetric difference x ∆ y
(*)setUnion(* x, * y): set-theoretic union, x ∪ y
(float)sin(numeric x): sine of x
(float)sqrt(numeric x): square root of x
(numeric$)sum(lif x): summation of the elements of x, Σ x
(float$)sumExact(float x): exact summation of x without roundoff error, to the limit of floating-point precision
(float)tan(numeric x): tangent of x
(float)trunc(float x): truncation (rounding toward 0) of x



Statistics:

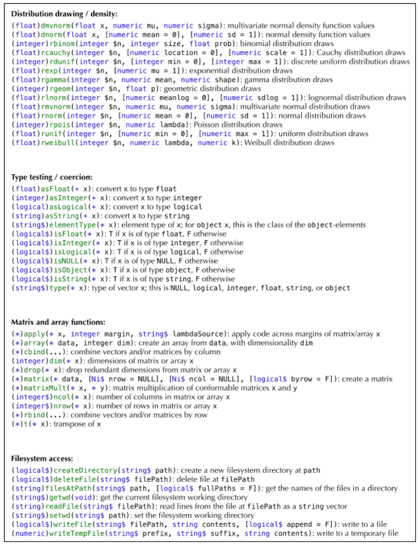
(float$)cor(numeric x, numeric y): sample Pearson's correlation coefficient between x and y
(float$)cov(numeric x, numeric y): corrected sample covariance between x and y
(+$)max(+ x, ...): largest value within x and the additional optional arguments
(float$)mean(lif x): arithmetic mean of x
(+$)min(+ x, ...): smallest value within x and the additional optional arguments
(+)pmax(+ x, + y): parallel maximum of x and y (the element-wise maximum for each corresponding pair)
(+)pmin(+ x, + y): parallel minimum of x and y(the element-wise maximum for each corresponding pair)
(numeric)range(numeric x, ...): range (min/max) of x and the additional optional arguments
(float$)sd(numeric x): corrected sample standard deviation of x
(float$)ttest(float x, [Nf y = NULL], [Nf$ mu = NULL]): run a one-sample or two-sample *t*-test
(float$)var(numeric x): corrected sample variance of x

Vector construction:

(*)c(...): concatenate the given vectors to make a single vector of uniform type
(float)float(integer$ length): construct a float vector of length, initialized with 0.0
(integer)integer(integer$ length, [integer$ fill1 = 0], [integer$ fill2 = 1],
[Ni fill2indices = NULL]): construct an integer vector of length, initialized with the given fill values
(logical)logical(integer$ length): construct a logical vector of length, initialized with F
(object<undefined>)object(void): construct an empty object vector
(*)rep(* x, integer$ count): repeat x a given number of times
(*)repEach(* x, integer count): repeat each element of x a given number of times
(*)sample(* x, integer$ size, [logical$ replace = F], [Nif weights = NULL]): sample from x
(numeric)seq(n$ from, n$ to, [Nif$ by = NULL], [Ni$ length = NULL]): construct a sequence
(integer)seqAlong(* x): construct a sequence along the indices of x
(integer)seqLen(integer$ length): construct a sequence with length elements, counting upward from 0
(string)string(integer$ length): construct a string vector of length, initialized with ""

Value inspection / manipulation:

(logical$)all(logical x, ...): T if *all* values supplied are T, otherwise F
(logical$)any(logical x, ...): T if *any* values supplied are T, otherwise F
(void)cat(* x, [s$ sep = " "): concatenate output
(void)catn([* x = ""], [s$ sep = " "]): concatenate output with trailing newline
(string)format(string$ format, numeric x): format the elements of x as strings
(logical$)identical(* x, * y): T if x and y are identical in all respects, otherwise F
(*)ifelse(logical test, * trueValues, * falseValues): vector conditional
(integer$)length(* x): count elements in x (synonymous with size())
(integer)match(* x, * table): positions of matches for x within table
(integer)nchar(string x): character counts for the string values in x
(integer)order(+ x, [logical$ ascending = T]): indexes of x that would produce sorted order
(string$)paste(* x, [string$ sep = " "): paste together a string with separators
(string$)paste0(* x): paste together a string with no separators
(void)print(* x): print x to the output stream
(*)rev(* x): reverse the order of the elements in x
(integer$)size(* x): count elements in x (synonymous with length())
(+)sort(+ x, [logical$ ascending = T]): sort non-object vector x
(object)sortBy(object x, string$ property, [l$ ascending = T]): sort object vector x by a property
(void)str(* x): print the external structure of a value
(string)strsplit(string$ x, [string$ sep = " "]): split string x into substrings by separator sep
(string)substr(string x, integer first, [Ni last = NULL]): get substrings from x
(*)unique(* x, [logical$ preserveOrder = T]): unique values in x (preserveOrder = F is faster)
(integer)which(logical x): indices in x which are T
(integer)whichMax(+ x): indices in x with the maximum value
(integer)whichMin(+ x): indices in x with the minimum value



Distribution drawing / density:

(float)dmvnorm(float x, numeric mu, numeric sigma): multivariate normal density function values
(float)dnorm(float x, [numeric mean = 0], [numeric sd = 1]): normal density function values
(integer)rbinom(integer $n, integer size, float prob): binomial distribution draws
(float)rcauchy(integer $n, [numeric location = 0], [numeric scale = 1]): Cauchy distribution draws
(integer)rdunif(integer $n, [integer min = 0], [integer max = 1]): discrete uniform distribution draws
(float)rexp(integer $n, [numeric mu = 1]): exponential distribution draws
(float)rgamma(integer $n, numeric mean, numeric shape): gamma distribution draws
(integer)rgeom(integer $n, float p): geometric distribution draws
(float)rlnorm(integer $n, [numeric meanlog = 0], [numeric sdlog = 1]): lognormal distribution draws
(float)rmvnorm(integer $n, numeric mu, numeric sigma): multivariate normal distribution draws
(float)rnorm(integer $n, [numeric mean = 0], [numeric sd = 1]): normal distribution draws
(integer)rpois(integer $n, numeric lambda): Poisson distribution draws
(float)runif(integer $n, [numeric min = 0], [numeric max = 1]): uniform distribution draws
(float)rweibull(integer $n, numeric lambda, numeric k): Weibull distribution draws

Type testing / coercion:

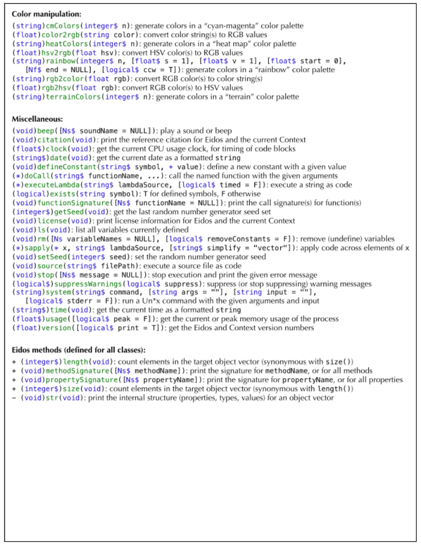
(float)asFloat(+ x): convert x to type float
(integer)asInteger(+ x): convert x to type integer
(logical)asLogical(+ x): convert x to type logical
(string)asString(+ x): convert x to type string
(string$)elementType(* x): element type of x; for object x, this is the class of the object-elements
(logical$)isFloat(* x): T if x is of type float, F otherwise
(logical$)isInteger(* x): T if x is of type integer, F otherwise
(logical$)isLogical(* x): T if x is of type logical, F otherwise
(logical$)isNULL(* x): T if x is of type NULL, F otherwise
(logical$)isObject(* x): T if x is of type object, F otherwise
(logical$)isString(* x): T if x is of type string, F otherwise
(string$)type(* x): type of vector x; this is NULL, logical, integer, float, string, or object

Matrix and array functions:

(*)apply(* x, integer margin, string$ lambdaSource): apply code across margins of matrix/array x
(*)array(* data, integer dim): create an array from data, with dimensionality dim
(*)cbind(...): combine vectors and/or matrices by column
(integer)dim(* x): dimensions of matrix or array x
(*)drop(* x): drop redundant dimensions from matrix or array x
(*)matrix(* data, [Ni$ nrow = NULL], [Ni$ ncol = NULL], [logical$ byrow = F]): create a matrix
(*)matrixMult(* x, * y): matrix multiplication of conformable matrices x and y
(integer$)ncol(* x): number of columns in matrix or array x
(integer$)nrow(* x): number of rows in matrix or array x
(*)rbind(...): combine vectors and/or matrices by row
(*)t(* x): transpose of x

Filesystem access:

(logical$)createDirectory(string$ path): create a new filesystem directory at path
(logical$)deleteFile(string$ filePath): delete file at filePath
(string)filesAtPath(string$ path, [logical$ fullPaths = F]): get the names of the files in a directory
(string$)getwd(void): get the current filesystem working directory
(string)readFile(string$ filePath): read lines from the file at filePath as a string vector
(string$)setwd(string$ path): set the filesystem working directory
(logical$)writeFile(string$ filePath, string contents, [logical$ append = F]): write to a file
(numeric)writeTempFile(string$ prefix, string$ suffix, string contents): write to a temporary file



Color manipulation:

(string)cmColors(integer$ n): generate colors in a "cyan-magenta" color palette
(float)color2rgb(string color): convert color string(s) to RGB values
(string)heatColors(integer$ n): generate colors in a "heat map" color palette
(float)hsv2rgb(float hsv): convert HSV color(s) to RGB values
(string)rainbow(integer$ n, [float$ s = 1], [float$ v = 1], [float$ start = 0],
[Nf$ end = NULL], [logical$ ccw = T]): generate colors in a "rainbow" color palette
(string)rgb2color(float rgb): convert RGB color(s) to color string(s)
(float)rgb2hsv(float rgb): convert RGB color(s) to HSV values
(string)terrainColors(integer$ n): generate colors in a "terrain" color palette

Miscellaneous:

(void)beep([Ns$ soundName = NULL]): play a sound or beep
(void)citation(void): print the reference citation for Eidos and the current Context
(float$)clock(void): get the current CPU usage clock, for timing of code blocks
(string$)date(void): get the current date as a formatted string
(void)defineConstant(string$ symbol, + value): define a new constant with a given value
(*)doCall(string$ functionName, ...): call the named function with the given arguments
(*)executeLambda(string$ lambdaSource, [logical$ timed = F]): execute a string as code
(logical)exists(string symbol): T for defined symbols, F otherwise
(void)functionSignature([Ns$ functionName = NULL]): print the call signature(s) for function(s)
(integer$)getSeed(void): get the last random number generator seed set
(void)license(void): print license information for Eidos and the current Context
(void)ls(void): list all variables currently defined
(void)rm([Ns variableNames = NULL], [logical$ removeConstants = F]): remove (undefine) variables
(*)sapply(* x, string$ lambdaSource, [string$ simplify = "vector"]): apply code across elements of x
(void)setSeed(integer$ seed): set the random number generator seed
(void)source(string$ filePath): execute a source file as code
(void)stop([Ns$ message = NULL]): stop execution and print the given error message
(logical$)suppressWarnings(logical$ suppress): suppress (or stop suppressing) warning messages
(string)system(string$ command, [string args = ""], [string input = ""],
[logical$ stderr = F]): run a Un*x command with the given arguments and input
(string$)time(void): get the current time as a formatted string
(float$)usage([logical$ peak = F]): get the current or peak memory usage of the process
(float)version([logical$ print = T]): get the Eidos and Context version numbers

Eidos methods (defined for all classes):

+ (integer$)length(void): count elements in the target object vector (synonymous with size())
+ (void)methodSignature([Ns$ methodName]): print the signature for methodName, or for all methods
+ (void)propertySignature([Ns$ propertyName]): print the signature for propertyName, or for all properties
+ (integer$)size(void): count elements in the target object vector (synonymous with length())
− (void)str(void): print the internal structure (properties, types, values) for an object vector

Function signatures

- Function signatures describe calling conventions
  - parameter types and names, return type

Function signatures

- Function signatures describe calling conventions
  - parameter types and names, return type
- For example, for `runif()`:

```
(float)runif(integer$ n, [numeric min = 0], [numeric max = 1])
```

Function signatures

- Function signatures describe calling conventions
  - parameter types and names, return type
- For example, for `runif()`:

```
(float)runif(integer$ n, [numeric min = 0], [numeric max = 1])
```

- Why do we need this cryptic syntax?
  - Understanding the reference sheets
  - Creating user-defined functions

Function signatures

```
(float)runif(integer$ n, [numeric min = 0], [numeric max = 1])
```

- The basic outline is:
  - `(return-type)functionName(parameters)`
- A `$` indicates a required *singleton*
- Brackets, `[ ]`, indicate *optional* parameters
- An `=` indicates a *default value*
- Type `numeric` means `integer` or `float`
- Type `void` means no return value

Function signatures

- A type of `*` means “any type”

```
(integer$)size(* x)
```

Function signatures

- A type of `*` means “any type”

```
(integer$)size(* x)
```

- A type of `+` means “any non-object type”

```
(integer)asInteger(+ x)
```

Function signatures

- A type of `*` means “any type”

```
(integer$)size(* x)
```

- A type of `+` means “any non-object type”

```
(integer)asInteger(+ x)
```

- Complex types are represented with single letters

```
(numeric$)sum(lif x)
```

Function signatures

- A type of `*` means “any type”

```
(integer$)size(* x)
```

- A type of `+` means “any non-object type”

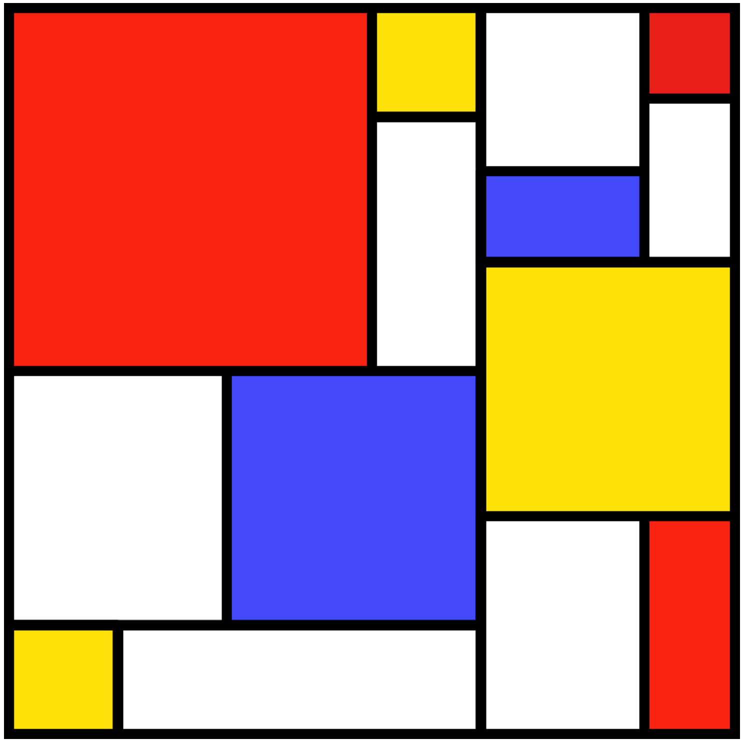
```
(integer)asInteger(+ x)
```

- Complex types are represented with single letters

```
(numeric$)sum(lif x)
```

- An ellipsis, `...`, indicate *variable* parameters

```
(*)c(...)
```



SLiM Workshop Exercise #4