Introduction

Sidef is a modern, high-level, general-purpose programming language, focusing on simplicity, readability and elegance, taking the best from languages like Ruby, Raku and Julia.

BOOK

The Sidef Programming Language: https://trizen.gitbook.io/sidef-lang/ (legacy)

Installation

Sidef can be installed from the CPAN, by invoking the following command:

```
$ cpan Sidef
```

If the testing takes a long time, add the `-T` flag to build and install Sidef without testing:

```
$ cpan -T Sidef
```

When the `cpan` command is not available, try:

```
$ perl -MCPAN -e "CPAN::Shell->install(q{Sidef})"
```

IMPORTANT: Sidef needs the GMP, MPFR and MPC C libraries.

Installing from git source

To install Sidef manually, download the latest version, unzip it and follow the installation steps:

```
$ perl Build.PL
# ./Build installdeps
# ./Build install
```

When Module::Build is not installed, try:

```
$ perl Makefile.PL
$ make test
# make install
```

Linux installation

Arch Linux

Sidef is available on the AUR and can be installed using an AUR helper, like trizen:

```
$ trizen -S sidef
```

Debian / Ubuntu / Linux Mint

On Debian-based distributions, Sidef can be installed from the CPAN, by executing the following commands:

```
$ sudo apt install libgmp-dev libmpfr-dev libmpc-dev libc-dev cpanminus
$ sudo cpanm -n Sidef
```

Android installation

It's also possible to install Sidef on Android, by installing Termux and executing the following commands:

```
$ pkg install perl make clang libgmp libmpfr libmpc
$ cpan -T Sidef
```

If the installation succeeded, the `sidef` command should be available:

```
$ sidef -h
```

Run Sidef without installing it

It is also possible to run Sidef without having to install it. In a Unix-like environment, the following commands can be executed:

```
$ wget 'https://github.com/trizen/sidef/archive/master.zip' -O 'master.zip'
$ unzip 'master.zip'
$ cd 'sidef-master/bin/'
$ ./sidef -v
```

Those commands will download and unpack the latest version of Sidef and will execute the `bin/sidef` script which will print out the current version of the language.

To execute a Sidef script, run:

```
$ ./sidef ../scripts/sierpinski_carpet.sf
```

It's also possible to add the following alias to `~/.bashrc` or `~/.zshrc`:

```
alias sidef="/path/to/bin/sidef"
```

Packaging

For packaging Sidef, run:

```
$ perl Build.PL --destdir "/my/package/path" --installdirs vendor
$ ./Build test
$ ./Build install --install_path script=/usr/bin
```

Creating the first Sidef script

A Sidef script can be written in any text editor and, by convention, it has the `.sf` extension.

The content of a simple *Hello World* program looks like this:

```
#!/usr/bin/sidef

say "Hello, 世界"
```

If we save the content in a new file called `hello.sf`, we can execute the code by running:

```
$ sidef hello.sf
```

Real code

Before taking a closer look at the syntax of the language, let's take a brief look at how a real program might look like. The following program defines the Bitmap class and generates a PPM file with a color palette:

```
subset Int   < Number {|n| n.is_int  }
subset UInt  < Int    {|n| n >= 0    }
subset UInt8 < Int    {|n| n ~~ ^256 }

struct Pixel {
    R < UInt8,
    G < UInt8,
    B < UInt8
}

class Bitmap(width < UInt, height < UInt) {
    has data = []

    method fill(Pixel p) {
        data = (width*height -> of { Pixel(p.R, p.G, p.B) })
    }

    method setpixel(i < UInt, j < UInt, Pixel p) {

        subset WidthLimit  < UInt { |n| n ~~ ^width  }
        subset HeightLimit < UInt { |n| n ~~ ^height }

        func (w < WidthLimit, h < HeightLimit) {
            data[w*height + h] = p
        }(i, j)
    }

    method p6 {
        <<-EOT + data.map {|p| [p.R, p.G, p.B].pack('C3') }.join
        P6
        #{width} #{height}
        255
        EOT
    }
}

var b = Bitmap(width: 125, height: 125)

for i,j in (^b.height ~X ^b.width) {
    b.setpixel(i, j, Pixel(2*i, 2*j, 255 - 2*i))
}

%f"palette.ppm".write(b.p6, :raw)
```

By executing the code, the following image is produced:

Another example is the implementation of the LCG algorithm, illustrating modules and classes.

```
module LCG {

  # Creates a linear congruential generator with a given seed.
  class Common(seed) {
     has r = seed
  }

  # LCG::Berkeley generates 31-bit integers using the same formula
  # as BSD rand().
  class Berkeley < Common {
    method rand {
      self.r = ((1103515245 * self.r + 12345) & 0x7fff_ffff)
    }
  }

  # LCG::Microsoft generates 15-bit integers using the same formula
  # as rand() from the Microsoft C Runtime.
  class Microsoft < Common {
    method rand {
      self.r = ((214013 * self.r + 2531011) & 0x7fff_ffff)
      self.r >> 16
    }
  }
}

var lcg1 = LCG::Berkeley(1)
say 5.of { lcg1.rand }

var lcg2 = LCG::Microsoft(1)
say 5.of { lcg2.rand }
```

Output:

```
[1103527590, 377401575, 662824084, 1147902781, 2035015474]
[41, 18467, 6334, 26500, 19169]
```

A last example, implementing a simple algorithm that finds the longest common substring from two given strings, illustrating the declaration of functions with typed parameters, the `gather/take` construct and the set-intersection operator (`&`).

```
func createSubstrings(String word) -> Array {
  gather {
    combinations(word.len+1, 2, {|i,j|
        take(word.substr(i, j-i))
    })
  }
}

func findLongestCommon(String first, String second) -> String {
    createSubstrings(first) & createSubstrings(second) -> max_by { .len }
}

say findLongestCommon("thisisatest", "testing123testing")
```

Syntax

The syntax of Sidef looks very much like the syntax of Ruby or JavaScript, but we'll see that there are some important differences regarding the semantics.

All operators have the same precedence, which is controlled by the lack of whitespace between the operands.

```
1+2 * 3+4   # means: (1+2) * (3+4)
```

In the above example, the lack of whitespace between `1`, `+` and `2`, classifies the operation as a distinct expression.

The implications are the following:

```
var n = 1 + 2       # incorrect -- it means: (var n = 1) + 2
var n = 1+2         # correct
var n = (1 + 2)     # correct
```

When no precedence is defined, the order of operations is from left to right:

```
1 + 2 * 3       # means: ((1 + 2) * 3)
```

On the other hand, when too much precedence is defined, the order is from right to left:

```
1+2*3           # means: (1 + (2 * 3))
```

The precedence can also be controlled by backslashing or preceding the operator with a dot.

```
1 + 2 \* 3      # means: (1 + (2 * 3))
1 + 2 .* 3      # =//=
```

The infix backslash (`\`) removes any leading or trailing whitespace at that current position and it's useful for expanding method calls on multiple lines:

```
say "abc".uc      \
         .reverse \
         .chars
```

is equivalent with:

```
say "abc".uc.reverse.chars
```

Keywords

Initially, Sidef was designed without any keywords. However, as it turned out, keywords can simplify the writing of code, at the cost of not having a variable with the same name as that of a keyword.

```
var             # declare a lexical variable
local           # declare a local dynamic variable
func            # declare a function
class           # declare a class
module          # declare a module
subset          # declare a subset
struct          # declare a structure
const           # declare a runtime dynamic constant
static          # declare a runtime static variable
define          # declare and assign a compile-time static constant
enum            # declare and assign a list of constants with distinct numbers

del             # delete an identifier from the lexical scope
eval            # evaluates any arbitrary Sidef code in the current scope
warn            # prints an warning to STDERR followed by the current file name and th
die             # raises an error followed by the current file name and the line numbe
read            # reads from standard input a type of data (e.g.: read(Number))
print           # writes to the standard output
say             # same as print, except that it also writes a newline character at the

assert          # terminate the program if the argument is false
assert_eq       # terminate the program if two arguments are not equal to each other
assert_ne       # terminate the program if two arguments are equal to each other

nil             # the `not initialized` value
true            # boolean representation for a true value
false           # boolean representation for a false value

if              # `if` statement
with            # `with` statement
while           # `while` statement
loop            # infinite loop
for             # `for` loop
try             # `try/catch` statement
given           # `given/when` statement
gather          # `gather/take` statement
continue        # fall-through in a `given/when` statement
return          # stop anything and return a value from a function
break           # break the current loop
next            # go to the next loop iteration

include         # load a Sidef module
import          # import a list of identifiers from a given module
```

Prefix operators

Sidef defines the following list of built-in prefix operators:

```
>               # alias for `say`
>>              # alias for `print`
+               # scalar context
-               # the negative value of an object (calls `neg`)
++              # increments a variable's value (calls `inc`)
--              # decrements a variable's value (calls `dec`)
~               # the 'not' value of an onject (calls `not`)
\               # takes reference to a variable
*               # dereferences a reference
:               # initializes a new hash
!               # the negation of an object in Boolean context
^               # an exclusive range (from 0 to n-1) (calls `range`)
@               # array context (calls `to_a`)
@|              # list context (calls `...`)
√               # square root of a number
```

Postfix operators

Additionally to prefix operators, Sidef also defines a small list of postfix operators:

```
++               # post-increments a variable's value (calls `inc`)
--               # post-decrements a variable's value (calls `dec`)
!                # factorial operation
!!               # double-factorial operation
...              # unpacks an object into a list
```

Built-in types

These classes deal with the standard data types in Sidef and implement the useful methods visible to the user.

```
File
FileHandle
Dir
DirHandle
Arr Array
Vector
Matrix
Pair
Hash
Set
Bag
Str String
Num Number
RangeStr RangeString
RangeNum RangeNumber
Mod
Complex
Fraction
Gauss
Quadratic
Quaternion
Polynomial
Math
Pipe
Ref
Socket
Bool
Sys
Sig
Regex Regexp
Time
Perl
Sidef
Object
Parser
Block
Backtick
LazyMethod
```

Ignoring the performance differences, a string `""` is just the same thing as `String()`.

```
[1,2,3]                  # same as: Array(1,2,3)

true                      # same as: Bool(1)
false                     # same as: Bool(0)

/^some[regex]?/i          # same as: Regex('^some[regex]?', 'i')
:(key => "value")         # same as: Hash("key", "value")
"first":"second"          # same as: Pair("first", "second")
```

Method invocations

A method is a function defined for a specific type of object. For strings, we have a method named `length`, which differs from the method with the same name that is defined for array-type objects.

```
"string".length         #=> 6
[1,2,3].length          #=> 3
```

In many languages, method invocations require parentheses, but in Sidef the parentheses are optional when we call a method without any arguments.

Methods

For calling a method, we have the following notation:

```
obj.method_name(arg)
```

For convenience, methods can also be invoked using the following prefix notation:

```
method_name(obj, arg)
```

The prefix and postfix notations can be used interchangeably:

```
log("string".length)    # means: "string".length.log
length("string").log    # =//=
log(length("string"))   # =//=
```

This provides very good expressivity, as illustrated with the following 4 statements, which are all equivalent:

```
grep(1..100, {.is_prime})          # calls Range.grep()
1..100 -> grep {.is_prime}         # =//=

grep({.is_prime}, 1..100)          # calls Block.grep()
{.is_prime} -> grep(1..100)        # =//=
```

Using the prefix notation, a method can be invoked even when a function with the same name is declared in the same scope, by preceding the method-name with `::`:

```
func sqrt(n) { "sqrt of #{n} is #{n.sqrt}" }

say   sqrt(42)     # calls the `sqrt` function defined above
say ::sqrt(42)     # calls the `Number.sqrt()` method
```

By placing `::` in front of a method-name, Sidef will parse it as a prefix operator, which allows the parenthesis to be omitted when invoked on only one object:

```
::log 42       # means: log(42)
::sqrt 1+2     # means: sqrt(1+2)
```

There are two method-invocation operators in Sidef: `.` and `->`:

```
20 + 5.sqrt    # means: 20 + sqrt(5)
20 + 5->sqrt   # means: sqrt(20 + 5)
```

`->` will make everything, from the left-most side of it, a single expression and applies the method on the result returned by this expression, while `.` will simply apply the method on the object which precedes the dot.

Additionally, any alphanumeric method name can be used as an infix operator, by surrounding it with backticks:

```
(1 `add` 2)           # means: 1.add(2)
(Math `sum` (1,2,3))  # means: Math.sum(1,2,3)
```

There is also the pipeline operator `|>`, which is defined for any object, except `nil`:

```
25 |> :sqrt |> :say       # means: 25.sqrt.say
25 |> (:pow, 2) |> :say   # means: 25.pow(2).say
```

It can also be used for making function-calls in postfix notation:

```
func increment(n) { n+1 }
func double(n)    { 2*n }
func triple(n)    { 3*n }

say increment(double(triple(42)))                   #=> 253
42 |> triple |> double |> increment |> :say         #=> 253
42 |> {_*3} |> {_*2} |> {_+1} |> :say              #=> 253
42 |> (->(a,b){a*b}, 3) |> {_+_} |> :inc |> :say    #=> 253
```

There is also support for calling a method of which name is not known until at runtime, which can be any expression that evaluates to a string:

```
say ( 50.(["+", "-"].rand)(30) )    # prints 20 or 80
```

If a method is not found for a given object, Sidef will throw a runtime error.

Built-in classes

In the current implementation of the language, we have the following built-in classes:

- Math
  - Math
- Object
  - Enumerator
  - Lazy
  - LazyMethod
  - Object
  - Convert
- Perl
  - Perl
- Sys
  - Sig
  - Sys
- Time
  - Date
  - Time
- Types
  - Array
    - Array
    - Pair
    - Matrix
    - Vector
  - Block
    - Block
    - Fork
    - Try
  - Bool
    - Bool
  - Glob
    - Backtick
    - Dir
    - DirHandle
    - File
    - FileHandle
    - Pipe
    - Socket
    - SocketHandle
    - Stat
  - Hash
    - Hash
  - Null
    - Null
  - Number
    - Complex
    - Number
    - Fraction

        - Gauss
        - Mod
        - Quadratic
        - Quaternion
        - Polynomial
    - Range
        - Range
        - RangeNumber
        - RangeString
    - Regex
        - Match
        - Regex
    - Set
        - Set
        - Bag
    - String
        - String

Variables

Variables are commonly declared using the `var` keyword:

```
var num = 42
var str = "42"
var bool = true
```

Variable types

In Sidef exists four types of variables: **lexical variables**, **static variables**, **global variables** and **local variables**.

LEXICAL VARIABLES

This kind of variables are dynamic, but statically block scoped. This is the usual way of declaring variables in Sidef.

```
var x = 42    # sets the lexical x to 42
say x         # prints the lexical value of x
```

STATIC VARIABLES

This type of variables are static, block-scoped and initialized only once.

```
for k in (1..10) {
    static x = 41+k  # will assign to x only once, setting it to 41+1
    say x            # prints 42
}
```

Multiple static variables can be declared and initialized, using the syntax:

```
static (
    a = 42,
   *b = (1,2,3,4),            # slurpy array
   :c = (x => 1, y => 2),     # slurpy hash
)

say a   #=> 42
say b   #=> [1,2,3,4]
say c   #=> Hash(x => 1, y => 2)
```

GLOBAL VARIABLES

Global variables are declared at the top-level of the current namespace. They can be accessed from everywhere, anytime. However, it's recommended to avoid them, unless there isn't a better alternative.

```
global x = 42     # sets global x to 42
say x             # prints the global value of x
```

LOCAL VARIABLES

Local variables (also known as "dynamically scoped variables") are used to localize array/hash lvalues or global variables to a limited scope.

```
global x = 42    # sets the global x to 42
do {
   local x = 100     # localizes x inside this block to 100
   say x             # prints the local value of x (100)
}
say x            # prints the global value of x (42)
```

Variable scoping

All variables, including functions and classes, are block scoped in the following way:

```
var x = 'o'

do {
    say x              # o
    var x = 'm'
    say x              # m
    do {
        say x          # m
        var x = 'b'
        say x          # b
    }
    say x              # m
}

say x                  # o
```

Declaring multiple variables on the same line works like expected:

```
var (x, y, z) = (3.14, false, "foo")
```

We can, also, declare variables with default values:

```
var (x, y=755, z=777) = (666, 655)

say x        # prints: 666
say y        # prints: 655
say z        # prints: 777
```

It's also possible to omit the second assignment:

```
var (
    a = 42,
   *b = (1,2,3,4),            # slurpy array
   :c = (x => 1, y => 2),     # slurpy hash
)

say a   #=> 42
say b   #=> [1,2,3,4]
say c   #=> Hash(x => 1, y => 2)
```

Additionally, referring to a previously defined variable as a default value is also supported:

```
var (
  a = 42,
  b = 10+a
)

say a   #=> 42
say b   #=> 52
```

Slurpy variables

Slurpy variables are a special type of variables which can be initialized with a list of values, creating automatically a container to hold the data.

```
var *arr = (1,2,3)    # creates an Array
say arr               # prints: [1,2,3]

var :hash = (a => 1, b => 2)   # creates an Hash
say hash                       # prints: Hash(a => 1, b => 2)
```

Working with variables

Any method applied to a variable is applied to the object stored inside the variable.

```
var x = 'sidef'
say x.uc        # prints: `SIDEF`
say x           # prints: `sidef`
```

Special `!` at the end of a method changes the variable in-place (almost like in Ruby):

```
var x = 'sidef'
x.uc!            # notice the `!`
say x            # prints: `SIDEF`
```

Appending the `=` sign at the end of arithmetic operators, the variable will be changed in place:

```
var x = 5
x += 10            # adds 10 to "x"
say x              # prints: 15
```

The special assignment defined-or operator `:=` can be used for changing a variable only when its value is `nil`:

```
var x = nil
x := 42        # sets x to 42
x := 99        # x is already defined
say x          # prints 42
```

This operator is commonly used in creating an hash of arrays:

```
var hash = Hash()
hash{:key} := [] << (1,2)
hash{:key} := [] << 3
say hash{:key}                #=> [1,2,3]
```

Additionally, the `:=` operator returns an lvalue:

```
var hash = Hash()
hash{:key} := 0 -> max!(10)
hash{:key} := 0 -> max!(42)
say hash{:key}                #=> 42
```

In addition to `:=`, there is also the defined-or operator `\\`, which can be used for checking if a value is defined:

```
var x = nil
x \\ say "x is not defined"     # prints "x is not defined"
```

Special variables

Currently, there are only two predefined variables: `ARGV` and `ENV`.

```
ARGV.each { |arg|
   say arg
}

say ENV{:HOME}
```

Deleting variables

Any identifier can be deleted using the `del` keyword followed by the name of the identifier.

```
var foo = 42
del foo
say foo          # parse-time error: attempt to use deleted identifier
```

Topic variable

The special topic variable (`_`) is declared at compile-time in all the block-objects of a program. Its name may not be seen very often because it has been overtaken by the elegant unary dot (`.`) operator:

```
[25,36,49].map {.sqrt} \
          .each{.log.say}
```

`.sqrt` really means `_.sqrt`, and `.log.say` means `_.log.say`.

The `map` method iterates over the anonymous array and calls the block for each element of the array, which gets set to the topic variable `_`. When the iteration is complete, the `map` method will return the new array, on which we call the `each` method with another block as the argument.

The `each` method, as the `map` method, will iterate over the array and will run the block for each element of the array, which will get assigned to the topic variable `_`, on which we call the `log` method, followed by `say`.

Additionally, same as in Raku, we can lookup an element from an array or an hash stored inside the topic variable, using the following syntax:

```
say [[41, 'a'], [42, 'b'], [43, 'c']].map { .[0] }          #=> [41, 42, 43]
```

and

```
say [Hash(a=>41), Hash(a=>42), Hash(a=>43)].map { .{:a} }   #=> [41, 42, 43]
```

Magic variables

Sidef's magic variables are directly bound to Perl's magic variables.

```
local $/ = nil        # changes the input record separator
local $\ = "\n"       # change the output record separator
local $, = "\n"       # changes the field record separator
say $^PERL            # prints the path to perl executable
say $^SIDEF           # prints the path to sidef executable
```

File-handle constants

This constants look like variables, but are actually file-handles.

```
STDERR.say("Some error!")
STDOUT.say("Some output...")
STDIN.readline()   # reads a line from the standard input
```

Another interesting file-handle is the `ARGF` which will read lines from argument-files or from standard-input when no argument has been specified.

Here is the implementation of a very basic `cat`-like program:

```
ARGF.each { |line|
    say line
}
```

Like in Perl, there is also the `DATA` file-handle which will point to the data stored after the `__END__` or `__DATA__` tokens.

```
DATA.each { |line|
    say "=>> #{line.chomp}"
}

__DATA__
here are
some data
lines
```

Constants

Sidef implements three kinds of constants:

const

The common way of declaring constants in Sidef, is by using the `const` keyword:

```
const pi = 3.14
say pi                  # prints: 3.14
#pi = 3                 # runtime error: can't modify non-lvalue constant
```

This kind of constants are created dynamically at runtime and cannot be changed after initialization.

Multiple constant values can be declared and initialized, using the syntax:

```
const (
    a = 42,
   *b = (1,2,3,4),            # slurpy array
   :c = (x => 1, y => 2),     # slurpy hash
)

say a   #=> 42
say b   #=> [1,2,3,4]
say c   #=> Hash(x => 1, y => 2)
```

When declared inside a function or a class, the constant is created and initialized dynamically, as illustrated in the following example:

```
func f(a) {
    const x = a          # created dynamically at each function call
    return (x + 2)
}

say f(40)       #=> 42
say f(50)       #=> 52
```

define

This keyword will define a compile-time evaluated constant and will point directly to the object at which it

evaluated to.

```
define PHI =   (1.25.sqrt + 0.5)
define IHP =  -(1.25.sqrt - 0.5)

say (PHI**12 - IHP**12 / PHI-IHP)   #=> 144
```

The value of a `define` constant must be a stand-alone constant expression that can be computed at compile-time.

Multiple constant values can be declared and initialized, using the syntax:

```
define (
    a = 42,
   *b = (1,2,3,4),            # slurpy array
   :c = (x => 1, y => 2),     # slurpy hash
)

say a   #=> 42
say b   #=> [1,2,3,4]
say c   #=> Hash(x => 1, y => 2)
```

Attempting to change a `define` constant, will result in a compile-time error:

```
define pi = 3.14
say pi         # prints: 3.14
#pi = 3        # compile-time error: Can't modify constant item
```

This type of constants are the most efficient ones.

enum

`enum` will automatically declare and assign a list of constants with ascending numeric values (starting with 0):

```
enum |Black, White|
say Black             # prints: 0
say White             # prints: 1
```

Alternatively, we have the possibility for specifying an initial value, which will get incremented after each declaration, by calling the method `inc`.

```
enum |α="a", β|
say α             # prints: 'a'
say β             # prints: 'b'
```

Variable references

Like other programming languages, Sidef is capable of taking references to variables.

```
var name = "sidef"
var ref = \name
var original = *ref
```

Variable references are useful when passing them to functions (or methods) for assigning values.

```
func assign2ref (ref, value) {
    *ref = value
}

var x = 10
assign2ref(\x, 20)
say x                # prints: 20
```

The `Ref` special type can be used for representing a variable reference.

Blocks

In Sidef, a block of code is an object which encapsulates one or more expressions. The block is delimited by a pair of curly braces (`{}`).

```
var block = {
    say "Hello, World!"
}
```

Block callbacks

Blocks are also used as arguments to many built-in methods as callback blocks:

```
{ print "Sidef! " } * 3        # prints "Sidef! Sidef! Sidef! "
5.times {|x| print x }         # prints "01234"
[1,2,3].sort {|a,b| b <=> a }  # returns a new array: [3,2,1]
```

Additionally, the following Block methods are also available:

```
say {|n| n**2 }.map(1..5)        #=> [1, 4, 9, 16, 25]
say { .is_odd }.grep([1,2,3,4])  #=> [1, 3]
```

...and the `Block.each` method, which is also aliased as `<<` :

```
{|n| say n**2 }.each(1..5)
{|n| say n**2 } << 1..5
```

Each of those methods accept more than one argument, which can be any object that accepts the `.iter()` method, as in the following example:

```
{|n| say n**2 } << (1..3, 101..103, 1001..1003)
```

Block parameters

For declaring block parameters, Sidef borrows Ruby's way of doing this, by using the `|arg1, arg2, ...|` special syntax.

```
{  |a, b|

   say a            # prints: 1
   say b            # prints: 2

}(1, 2)
```

Default block parameter values

We can also specify default values for block parameters. This can be done by using the syntax: `arg=value`.

```
say { | a=3, b=4 | a + b }(9)        # prints the result of: 9 + 4
```

The default value can be any expression:

```
say { | a=1/2, b=(a**2) | a + b }(5)      # prints the result of: 5 + 5**2
```

Lazy evaluation

Lazy evaluation is a very common feature in functional programming languages and provides an way to delay the evaluation of an expression until the result is actually needed.

By default, Sidef is an eagerly evaluated language, but it still supports a form of lazy evaluation, which is provided by the method `Object.lazy()`:

```
say (^Inf -> lazy.grep{ .is_prime }.first(10))     # first 10 primes
```

The `.lazy` method returns a `Lazy` object, which behaves almost like an `Array`, except that it executes the methods in a pipeline fashion and dynamically stops when no more elements are required, without creating any temporary arrays in memory:

```
for line in (DATA.lazy.grep{ _ < 'd' }.map{ print ">> "; .uc }) {
   print line
}

__DATA__
a
b
c
d
```

Output (which shows that `.map{}` is really lazy):

```
>> A
>> B
>> C
```

This mechanism is generic enough to support any kind of object that implements the `.iter()` method, which returns a Block that gives back one element at a time when it's called with no arguments. When the iteration ends, it must return `nil`.

```
class Example(data) {
    method iter {
        var p = 0
        {
            data[p++]
        }
    }
}

var obj = Example([1,2,3,4,5])
say obj.lazy.grep{.is_prime}.to_a       # filters all the primes lazily
```

Currently, the `.iter()` method is defined in the following built-in classes: Array, String, FileHandle, DirHandle, RangeString, RangeNumber and Lazy.

Lazy methods

A lazy method is an interesting concept of partially applying a method on a given object, delaying the evaluation until the result is needed.

```
var lz = 42.method('>')        #=> LazyMethod
say lz(41)                      #=> true (i.e.: 42 > 41)
```

`Object.method()` returns a `LazyMethod` object which can be called just like any function, producing the result by calling the method which is stored inside the `LazyMethod` object.

This allow us to store or pass around partial expressions, which can be evaluated multiple times at any point in the future:

```
var str = "a-b-c"
var lzsplit = str.method(:uc).method(:split)

say lzsplit('')         #=> ["A", "-", "B", "-", "C"]
say lzsplit('-')        #=> ["A", "B", "C"]
```

This concept can also be used to create partial virtual functions:

```
var add1 = 1.method('+')
var eqv2 = 2.method('==')

say (1..100 -> map{ add1(_) }.grep{ eqv2(_) })    #=> [2]
```

Method introspection

`Object.methods()` provides a simple way to find the methods implemented in the class of the self object. It returns a `Hash` with the method names as keys and `LazyMethod` objects as values.

Example:

```
var methods = (1..10 -> methods)
say methods.keys                    #=> ["call", "dump", "iter", "new", "prod", "sum"]
say methods{:sum}()                 #=> 55
```

Functions

Like mathematical functions, Sidef's functions can be recursive, take arguments and return values.

```
func hello (name) {
   say "Hello, #{name}!"
}

hello("Sidef")
```

Unlike other programming languages, Sidef requires the arguments to be enclosed in `()` when calling a function:

```
func f(g) {
    g()
}

f({ say "foo" })        # prints "foo"
f { say "bar" }         # this does nothing
```

By executing the above script with the `-r` argument, we will see how the code is parsed:

```
$ sidef -r script.sf
```

which outputs:

```
func f(g) { (g->call()) };
f({|_| (say("foo")) });
f;
{|_| (say("bar")) };
```

Recursive functions:

```
func factorial (n) {
   if (n > 1) {
       return (n * factorial(n - 1))
   }
   return 1
}

say factorial(5)     # prints: 120
```

Anonymous recursion can be achieved by using the `__FUNC__` keyword, which refers to the current function:

```
func recmap(repeat, seed, transform, callback) {
    func (repeat, seed) {
        callback(seed)
        repeat > 0 && __FUNC__(repeat-1, transform(seed))
    }(repeat, seed)
}

recmap(6, "0", func(s) { s + s.tr('01', '10') }, func(s) { say s })
```

Additionally, the `__BLOCK__` keyword can be used for referring to the current block:

```
func fib(n) {
    return NaN if (n < 0)

    {|n|
        n < 2 ? n
              : (__BLOCK__(n-1) + __BLOCK__(n-2))
    }(n)
}

say fib(12)   #=> 144
```

Storing functions in variables:

```
var f = func (name) {
     say "#{name} says 'Hello!'"
}
f('Sidef')
```

Function re-declaration:

```
func f(name) {
   say "Hello, #{name}"
}
f("Sidef")

f = func (name, age) {
   say "Hello, #{name}! You claim to be #{age} years old."
}
f("Sidef", 100)
```

Closures

In Sidef, all functions are first-class objects which can be passed around like any other object. Additionally, all functions and methods are lexical closures.

```
func curry(f, *args1) {
    func (*args2) {
        f(args1..., args2...)
    }
}

func add(a, b) {
    a + b
}

var adder = curry(add, 1)
say adder(3)                    #=> 4
```

Automatically cached functions

By specifying the `cached` trait to a function or a method, Sidef will automatically cache it.

```
func fib(n) is cached {
    return n if (n <= 1)
    fib(n-1) + fib(n-2)
}
say fib(100)              # prints: 354224848179261915075
```

Additionally, the method `Block.cache()` enables memoization on the self-block, while `Block.uncache()` disables the memoization and also cleans-up the cache.

```
func fib(n) {
   n <= 1 ? n : fib(n-1)+fib(n-2)
}

fib.cache       # enables memoization
say fib(100)    #=> 354224848179261915075
fib.uncache     # disables memoization
```

Function parameters

The parameters of a function can be defined to have a default value when the function is called with a lower number of arguments than required.

```
func hello (name="Sidef") {
    say "Hello, #{name}!"
}

hello()           # prints: "Hello, Sidef!"
hello("World")    # prints: "Hello, World!"
```

The default value of a parameter is evaluated only when an argument is not provided for that particular parameter, and it can be any expression:

```
func foo (a = 1.25.sqrt, b = 1/2) {
     a + b
}

say foo()           # prints the result of: sqrt(1.25) + 1/2
say foo(21, 21)     # prints: 42
```

An interesting feature is the possibility of referring to a previously defined parameter as a default value:

```
func foo(a=10, b=a+1) {
    a + b
}

say foo()              # prints: 21 (the result of: 10 + 10+1)
say foo(1)             # prints: 3  (the result of: 1 + 1+1)
say foo(21,21)         # prints: 42 (the result of: 21 + 21)
```

Named parameters (a.k.a. keyword arguments)

This is a very nice feature which allows a function to be called with named parameters, giving us the flexibility to put the arguments in no specific order:

```
func div(a, b) {
   a / b
}

say div(b: 5, a: 35)  # prints: 7
```

Variadic functions

A slurpy variable in the form of `*name` can be used as a function parameter to collect the remaining arguments inside an array:

```
func f(*args) {
   say args       #=> [1, 2, 3]
}

f(1, 2, 3)
```

Alternatively, by using a named variable in the form of `:name`, the arguments are collected inside an hash:

```
func f(:pairs) {
   say pairs      #=> Hash(a => 1, b => 2)
}

f(a => 1, b => 2)
```

Typed parameters

A function can be declared with typed parameters, which are checked at runtime.

```
func concat(String a, String b) {
    a + b
}
```

Now, the function can only be called with strings as arguments:

```
say concat("o", "k")  # ok
say concat(1, 2)      # runtime error
```

The typed parameters require a specific type of object, but they do not default to anything when no value is provided.

This means that all the typed-parameters are mandatory, unless a default value is provided:

```
func concat(String a="foo", String b="bar") {
   a + b
}

say concat()          # prints: "foobar"
say concat("mini")    # prints: "minibar"
say concat(1, 2)      # this is still a runtime error
```

Subsets

A subset is a definition which specifies the upper limit of inheritance, with optional argument validation.

```
subset Integer     < Number  { |n| n.is_int }
subset Natural     < Integer { |n| n.is_pos }
subset EvenNatural < Natural { |n| n.is_even }

func foo(n < EvenNatural) {
    say n
}

foo(42)         # ok
foo(43)         # failed assertion at runtime
```

In some sense, a subset is the opposite of a type. For example, let's consider the following class hierarchy:

```
class Hello(name) {
    method greet { say "Hello, #{self.name}!" }
}

class Hi < Hello {
    method greet { say "Hi, #{self.name}!" }
}

class Hey < Hi {
    method greet { say "Hey, #{self.name}!" }
}
```

If we declare a function that accepts a subset of `Hi`, it will accept `Hello`, but it cannot accept `Hey`:

```
func greet(obj < Hi) { obj.greet }       # `Hi` is the upper limit

greet(Hi("Foo"))        # ok
greet(Hello("Bar"))     # ok
greet(Hey("Baz"))       # fail: `Hey` is too evolved
```

On the other hand, if we use `Hi` as a type assertion, it will accept `Hey`, but not `Hello`:

```
func greet(Hi obj) { obj.greet }        # `Hi` is the lower limit

greet(Hi("Foo"))        # ok
greet(Hey("Baz"))       # ok
greet(Hello("Bar"))     # fail: `Hello` is too primitive
```

Subsets can also be used for combining multiple types into one type, creating an union type:

```
subset StrNum < String, Number

func concat(a < StrNum, b < StrNum) {
    a + b
}

say concat("o", "k")       # ok
say concat(13, 29)         # 42
say concat([41], [42])     # runtime error
```

Multiple dispatch

Sidef also includes multiple dispatch for functions and methods, based on the number of arguments and their types:

```
func test(String a){
    say "Got a string: #{a}"
}

func test(Number n) {
    say "Got a number: #{n}"
}

func test(Number n, Array m) {
    say "Got a number: #{n} and an array: #{m.dump}"
}

func test(String s, Number p) {
    say "Got a string: #{s} and a number: #{p}"
}

test("hello", 21)
test("sidef")
test(12, [1,1])
test(42)
```

Output:

```
Got a string: hello and a number: 21
Got a string: sidef
Got a number: 12 and an array: [1, 1]
Got a number: 42
```

Functional pattern matching

This feature looks like this:

```
func fib ((0)) { 0 }
func fib ((1)) { 1 }
func fib  (n)  { fib(n-1) + fib(n-2) }

say fib(12)      # prints: 144
```

We have three functions, where the first two, each have an expression as a parameter. Sidef will check each expression for equality with a given argument and will call the corresponding function when it passes the test. Otherwise, it will default to the third function.

Alternatively, we can specify a block instead of an expression:

```
func fib (Number n { _ <= 1} = 0) {
    return n
}

func fib (Number n) {
    fib(n-1) + fib(n-2)
}

say fib(12)      # prints: 144
```

When a block is specified, the type and the name of the parameter must come before the block, while an optional default value goes after the block. The types and the default values can be omitted.

The name of the parameters can be omitted as well:

```
func fib({.is_neg})  { NaN }
func fib({.is_zero}) { 0 }
func fib({.is_one})  { 1 }
func fib(n)          { fib(n-1) + fib(n-2) }

say fib(12)      # prints: 144
```

Combining the power of multiple dispatch with subsets and pattern matching, we can achieve impressive results, as illustrated in the following example, which implements the arithmetic derivative recursively for all integers and fractions:

```
subset Integer  < Number   { .is_int   }
subset Positive < Integer  { .is_pos   }
subset Negative < Integer  { .is_neg   }
subset Prime    < Positive { .is_prime }

func arithmetic_derivative((0)) { 0 }
func arithmetic_derivative((1)) { 1 }

func arithmetic_derivative(_ < Prime) { 1 }

func arithmetic_derivative(n < Negative) {
    -arithmetic_derivative(-n)
}

func arithmetic_derivative(n < Positive) is cached {

    var a = n.lpf
    var b = n/a

    arithmetic_derivative(a)*b + a*arithmetic_derivative(b)
}

func arithmetic_derivative(Number n) {
    var (a, b) = n.nude
    (arithmetic_derivative(a)*b - arithmetic_derivative(b)*a) / b**2
}

printf("(42!)' = %s\n", arithmetic_derivative(42!))
```

Returned type

Sidef also has the capability to check the return type of a function and stop the execution of the program if the returned type doesn't match the type defined in the function declaration.

```
func ieq(a, b) -> Bool {
   a.lc == b.lc
}

say ieq("Test", "tEsT")    # true
```

On the other hand:

```
func concat(a, b) -> Array {
    a + b
}

say concat([1,2,3], [4,5,6])   # ok
say concat("123", "456")       # runtime error
```

Multiple return-type checks are supported as well:

```
func foo() -> (Number, String) {
   (42, "foo")
}
var (a, b) = foo()
```

Alternative function declaration

A function can also be declared by using the fancy unary operator `->`, which is synonym with the `func` or `method` keywords, depending on the context where it is used.

```
-> my_add(a,b) { a + b }
```

If the function is declared inside a class, it will be defined as a method belonging to that class. However, if the function is declared inside another method, it will be defined as a lexical function in the scope of that method, as expected.

The name of the function is optional. If the name is omitted, an anonymous function will be created.

```
[1,2,3].sort(->(a,b) { b <=> a })
```

This notation is close to a lambda function, as illustrated in the following declaration of the Y-combinator:

```
var y = ->(f) {->(g) {g(g)}(->(g) { f(->(*args) {g(g)(args...)})})}

var fib = ->(f) { ->(n) { n < 2 ? n : (f(n-2) + f(n-1)) } }
say 10.of { |i| y(fib)(i) }
```

Output:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Loops

Creating loops in Sidef is as simple as it can get. Any loop can be stopped with the `break` keyword.

Infinite looping with the `loop` keyword:

```
loop {
  say "Sidef is looping!"
}
```

`while` loop:

```
var x = 123456789
while (x > 0) {
   var (div, mod) = x.divmod(10)
   say mod
   x = div
}
```

`do-while` loop:

```
var n = 1
do {
   say n
} while (++n <= 10)
```

`for` loop:

```
for (var i = 0; i <= 10; i++) {
   say i
}
```

`for-in` loop:

```
for item in (1..5) {
    say item
}
```

`for` each loop:

```
for (1..5) { |item|
    say item
}
```

`.each` method:

```
(1..5).each { |i|
    say i**2
}
```

Repetition block:

```
{ |n|
  say "Hello there! (#{n})"
} * 10
```

or:

```
n.times { ... }
```

Collecting `n` items can be done by using the `n.of {...}` method, which calls the given block with the values `0` to `n-1` and returns an array with the block-values from each call:

```
say 5.of {|n| 2**n }     #=> [1, 2, 4, 8, 16]
```

Similarly, there is the `n.by { ... }` method, which returns the first `n` integers (`>= 0`) for which the block evaluates to a true value:

```
say 5.by { .is_prime }   #=> [2, 3, 5, 7, 11]
```

Recursive block:

```
{
  say "Hello, World!"
  __BLOCK__.run
}.run
```

Things to remember: 1. `|x,y,z|` is used to capture the block arguments in variables. 2. The keyword `next` will skip one interation of the loop. 3. The keyword `break` will break a loop.

gather/take

The `gather/take` construct, borrowed from Raku:

```
var arr = gather {
   take(1)
   take(2)
   take(3)
}

say arr  #=> [1, 2, 3]
```

try/catch

The `try/catch` construct:

```
try {
   # unsafe code here
}
catch { |msg|
   say "try/catch failed with message: #{msg}"
}
```

The value of `msg` is a String containing the error text produced in the `try` branch.

Additionally, the `catch` branch is optional and can be omitted. If omitted and the `try` branch fails, the `nil` value is returned.

Strings

```
"I am a string"
```

A string is a group of characters which exists inside the same object.

```
var new_str = ("a" + "b")
say new_str   # prints: "ab"
```

A Sidef-string have many methods which will really help working with strings in a new exciting way.

Most used methods are:

```
.uc                    # uppercase the string
.lc                    # lowercase the string
.wc                    # capitalize each word
.tc                    # capitalize the string
.reverse               # reverse the string
.trim                  # removes leading and trailing whitespace

.length                # string size, in characters
.is_empty              # true if string has 0 length

.div(N)                # divide the string into N chunks
.split(N)              # split the string by N characters
.split('str')          # split the string by 'str'
.split(/regex/)        # split the string by a regular expression
.each{|c| ...}         # iterate over string characters
.each_word{|w| ...}    # iterate over words
.each_line{|l| ...}    # iterate over lines

.char(3)               # returns the character at the specified index
.substr(beg, len)      # returns a sub-string from position beg
.begins_with('str')    # true if string begins with 'str'
.ends_with('str')      # true if string ends with 'str'
.contains('str')       # returns true if string contains 'str'
.index('str')          # returns the position where 'str' begins
.match(/regex/)        # returns a Match object

.sub(/regex/, 'str')   # regexp substitution (returns a new string)
.gsub(/regex/, 'str')  # global regexp substitution (returns a new string)
```

For more methods, see: String.pod

String quotes

Being a new programming language, Sidef has built-in support for Unicode quotes:

```
var dstr = „double quoted”
var sstr = ‚single quoted’
```

The difference between double-quoted and single-quoted strings is the following: 1. double-quoted strings can interpolate code. * example: `"code: #{say 'inside string'}"` 2. double-quoted strings understand special escapes. * example: `"\n, \t, \Q...\E"`

while the single-quoted strings can't do any of this.

```
say 'single\tquoted'   # prints the string as it is
say "double\tquoted"   # replaces '\t' with a tab-character

var name = "string"
say 'single quoted #{name}'     # prints the string as it is
say "double quoted #{name}"     # prints: "double quoted string"
```

A single word can also be quoted by placing `:` in front of it. This feature makes things easier when it's used in hash look-ups and hash literal definitions.

```
:word == 'word'
:another_word == 'another_word'
```

Sidef also borrow from Ruby some operator-like quotes and implements some new ones:

```
var sstr = %q{single {} quoted string}
var dstr = %Q«double «» quoted string»
var arr1 = %w(word1 word2)
var arr2 = %W(double quoted words)
var arr3 = <single <quoted> words>
var arr4 = «double «quoted» words»
var reg  = %r⌈some\h+regxp?⌋
var file = %f„filename.txt”
var dir  = %d‘/my/dir’
var pipe = %p(ls -l)
var tick = %x(uname -r)
```

Simple HERE-doc support:

```
var str = <<'EOF'
some
text
EOF
```

Indented HERE-document (with interpolation):

```
func hello(arg) {
    return <<-"EOT"
    Hello, #{arg}!
    EOT
}

print hello('Sidef')    # prints: 'Hello, Sidef!'
```

By single-quoting the name of an HERE-document, interpolation will be disabled:

```
{
    print <<-'EOT'
    Not interpolated: #{1+2}
    EOT
}.run     # prints: 'Not interpolated: #{1+2}'
```

Nested HERE-docs are supported as well:

```
print(<<'EOF', "- - -\n", <<'EOT' + <<"EOD")
1 2 3
EOF
4 5 6
EOT
7 8 9
EOD
```

Numbers

In Sidef, numbers play an important role and are treated correspondingly. The numerical system is implemented with Math::GMPq, Math::GMPz, Math::MPFR, and Math::MPC, giving us a really good precision in calculations with an astonishing performance.

```
say Number.pi       # prints: 3.14159265358979323846264338327795
say 2**999          # 53575430359313366074742125245300...02149158263121934186028340346888
```

Additionally, the Number class can be used for constructing new Number objects:

```
Number("1234.52")         # new decimal number
Number("10110111", 2)     # new binary number
Number("deadbeef", 16)    # new hexadecimal number
```

Integers

Here is `255` written as integer in different bases:

```
255             # decimal
0xff            # hexadecimal
0377            # octal
0b1111_1111     # binary
```

Decimal literals

In Sidef, decimal literals are always represented in rational form, using Math::GMPz or Math::GMPq.

```
1.234          # 1.234
.1234          # 0.1234
1234e-5        # 0.01234
12.34e5        # 1234000
```

Example:

```
say 1.234.dump           #=> 617/500
say (0.1 + 0.2 == 0.3)   #=> true
```

Floating-point values

Floating-point values are represented by Math::MPFR with a default precision of 192 bits.

A literal floating-point value is usually returned by some mathematical functions (including `sin(x)`, `log(x)`, etc...).

A given number can be explicitly converted to a floating-point value by calling the `.float` method on it:

```
1.234.float      # floating-point value
```

Complex numbers

Complex numbers are represented by Math::MPC in floating-point form, with a default precision of 192 bits for each component.

A complex number can be created by using either one of the following ways:

```
3:4             # 3+4i
3+4i            # 3+4i
3+4.i           # 3+4i
Complex(3,4)    # 3+4i
```

Complex numbers are deeply integrated into the language and can be used in combination with all the other Number types (with implicit propagation):

```
sqrt(-1)        # 1i
log(-1)         # 3.14159265358979323846264338327950288419716939938i
4 + sqrt(-1)    # 4+i
(3+4i)**2       # -7+24i
```

Floating-point precision

The default floating-point precision can be changed with the `-P int` command-line flag passed to `sidef`, which specifies the number of decimals of precision.

Example:

```
$ sidef -P100 script.sf    # executes "script.sf" with 100 decimals of precision
```

It's also possible to dynamically change the floating-point precision at runtime, by modifying the `Num!PREC` class variable:

```
say sqrt(2)                     #=> 1.41421356237309504880168872420969807856967187538
local Num!PREC = 42.numify      # sets the floating-point precision to 42 bits
say sqrt(2)                     #=> 1.414213562
```

Mod

The Mod class represents a modular operation, similar to PARI/GP's built-in `Mod(a,m)` class.

```
Mod(3, 4)       # represents 3 mod 4
```

Example:

```
var a = Mod(13, 19)

a += 15             # Mod(9, 19)
a *= 99             # Mod(17, 19)
a /= 17             # Mod(1, 19)

say a               # Mod(1, 19)
say (a == 1)        # true
say (a == 20)       # true

a -= 43             # Mod(15, 19)

say a**42           # Mod(11, 19)
say a**(-1)         # Mod(14, 19)
say sqrt(a+1)       # Mod(4, 19)

say chinese(Mod(43, 19), Mod(13, 41))   # Mod(423, 779)
```

Fraction

The Fraction class represents a generic fraction:

```
var a = Fraction(3, 4)
var b = Fraction(5, 7)

say a*b     #=> Fraction(15, 28)
say a+b     #=> Fraction(41, 28)
```

Gauss

The Gauss class provides support for rational Gaussian numbers and various operations on these numbers.

```
Gauss(3, 4)     # represents 3+4i
```

Example:

```
say Gauss(3,4)**100
say Mod(Gauss(3,4), 1000001)**100   #=> Mod(Gauss(826585, 77265), 1000001)

var a = Gauss(17,19)
var b = Gauss(43,97)

say a+b     #=> Gauss(60, 116)
say a-b     #=> Gauss(-26, -78)
say a*b     #=> Gauss(-1112, 2466)
say a/b     #=> Gauss(99/433, -32/433)
```

Quadratic

The Quadratic class represents a quadratic integer of the form: `a + b*sqrt(w)`.

```
var x = Quadratic(3, 4, 5)  # represents: 3 + 4*sqrt(5)
var y = Quadratic(6, 1, 2)  # represents: 6 + sqrt(2)

say x**10       #=> Quadratic(29578174649, 13203129720, 5)
say y**10       #=> Quadratic(253025888, 176008128, 2)

say x.powmod(100, 97)   #=> Quadratic(83, 42, 5)
say y.powmod(100, 97)   #=> Quadratic(83, 39, 2)
```

Quaternion

The Quaternion class represents a quaternion number of the form `a + b*i + c*j + d*k`, where `a`,

`b`, `c`, and `d` are real numbers; and `i`, `j`, and `k` are the basic quaternions.

```
var a = Quaternion(1,2,3,4)
var b = Quaternion(5,6,7,8)

say a+b     #=> Quaternion(6, 8, 10, 12)
say a-b     #=> Quaternion(-4, -4, -4, -4)
say a*b     #=> Quaternion(-60, 12, 30, 24)
say b*a     #=> Quaternion(-60, 20, 14, 32)
say a/b     #=> Quaternion(35/87, 4/87, 0, 8/87)

say a**5                #=> Quaternion(3916, 1112, 1668, 2224)
say a.powmod(43, 97)    #=> Quaternion(61, 38, 57, 76)
say a.powmod(-5, 43)    #=> Quaternion(11, 22, 33, 1)
```

Polynomial

The Polynomial class implements support for polynomials.

```
say Polynomial(5)                   # monomial: x^5
say Polynomial([1,2,3,4])           # x^3 + 2*x^2 + 3*x + 4
say Polynomial(5 => 3, 2 => 10)     # 3*x^5 + 10*x^2
```

Also aliased as `Poly()`:

```
var a = Poly([1,2,3])
var b = Poly([4,5,-6,7])

say a+b     #=> 4*x^3 + 6*x^2 - 4*x + 10
say a-b     #=> -4*x^3 - 4*x^2 + 8*x - 4
say a*b     #=> 4*x^5 + 13*x^4 + 16*x^3 + 10*x^2 - 4*x + 21

say 42-a    #=> -x^2 - 2*x + 39
say 42+b    #=> 4*x^3 + 5*x^2 - 6*x + 49
say 42*b    #=> 168*x^3 + 210*x^2 - 252*x + 294

say a/42    #=> 1/42*x^2 + 1/21*x + 1/14
say b/42    #=> 2/21*x^3 + 5/42*x^2 - 1/7*x + 1/6
```

Numerical conversions

A numerical string can be converted into a rational number by using the `Number` class:

```
Number("0.75")          # "0.75" is parsed and stored in rational form as 3/4
Number("fff/aaa", 36)   # parse a base-36 fraction as 4095/2730
```

The Number method `as_rat` can be used for getting the rational form of a number:

```
say 3/4                # 0.75
say as_rat(3/4)        # 3/4
say as_rat(1.234, 36)  # h5/dw   (which is 617/500 in base-10)
```

The `Number.base(b)` method provides conversion from numbers into strings in a given base:

```
1234.base(13)       # to string in base 13
1234.base(36)       # to string in base 36
```

Arrays

```
var array = [1, 2, 3, 4, 5]

array[0] = 6
array[1] = 7

say array
```

Arrays are simple objects which can store other objects, and provide a zero-based indexing. In Sidef, an array can grow as much as the system memory permits it.

Array autovivification:

```
var array = []
array[3][4] = "hei"
say array
```

If you're familiar with Perl, you already know about autovivification. It's the feature responsible for the dynamic creation of data structures.

Array filtering

The `Array` object has many interesting methods for making it safer and easier to work with arrays in a pure OO style.

`grep`ing some elements from an array:

```
var new_arr = arr.grep { _ > 10 }      # returns a new array containing numbers greater
```

`map`ing an array:

```
var new_arr = arr.map {|n| 2*n + n**2}   # returns a new array with the result returned
```

`sort`ing an array:

```
# generic sort
var new_arr = arr.sort

# naive string case-insensitive sort
var new_arr arr.sort {|a,b| a.lc <=> b.lc}

# efficient string case-insensitive sort
var new_arr arr.sort_by { .lc }
```

Unroll operator

It's a nice metaoperator borrowed from Raku, which unrolls two arrays and applies the operator on each two element-wise objects, creating a new array with the results. The operator can be a method or any other valid operator and must be enclosed between `» «` or `>> <<`.

```
[1,2,3] »+« [4,5,6]           # [1+4, 2+5, 3+6]
%w(a b c) >>cmp<< %w(c b a)   # [-1, 0, 1]
```

Internally, the `unroll_operator` method is called, which can, also, be implemented in user-defined classes.

Map operator

The array map operator works exactly like the `Array.map{}` method, but it's slightly more efficient and easier to write. The map operator must be enclosed between `» »` or `>> >>`.

```
[1,2,3] »*» 4      # [1*4, 2*4, 3*4]
```

Internally, the `map_operator` method is called.

Pam operator

The pam operator is kind of a reversed mapping of the array ("pam" is "map" spelled backwards), where the provided argument is used as the first operand to the operator provided. The operator must be enclosed between `« «` or `<< <<`.

```
[1,2,3] «/« 10     # [10/1, 10/2, 10/3]
```

Internally, the `pam_operator` method is called.

Reduce operator

This metaoperator reduces an array to a single element. The operator needs to be enclosed inside `« »` or `<< >>`.

```
[1,2,3]«+»      # 1 + 2 + 3
[1,2,3]«/»      # 1 / 2 / 3
```

Internally, the `reduce_operator` method is called.

Cross operator

The metaoperator `~X` or `~Xop` crosses two arrays and returns a new one.

```
[1,2] ~X+ [3,4]    # [1+3, 1+4, 2+3, 2+4]
[1,2] ~X  [3,4]    # [[1,3], [1,4], [2,3], [2,4]]
```

Internally, the `cross_operator` method is called.

Zip operator

The metaoperator `~Z` or `~Zop` zips two arrays and returns a new one.

```
[1,2] ~Z+ [3,4]    # [1+3, 2+4]
[1,2] ~Z  [3,4]    # [[1,3], [2,4]]
```

Internally, the `zip_operator` method is called.

Wise operator

Almost equivalent with the zip metaoperator, it does element-wise folding on two arbitrary nested arrays, where both arrays must have the same structure.

```
[1,2]       ~W  [3,4]         # [[1,3], [2,4]]
[1,2]       ~W+ [3,4]         # [1+3, 2+4]
[[[1]],[2]] ~W+ [[[3]],[4]]   # [[[1+3]], [2+4]]
```

Internally, the `wise_operator` method is called.

Scalar operator

The scalar operator applies a given operator to the elements of an arbitrary nested array, where the provided scalar is used as the second operand to the given operator.

```
[1,2,3]       ~S  5  # [[1,5], [2,5], [3,5]]
[1,2,3]       ~S* 5  # [1*5, 2*5, 3*5]
[1,[[2,[3]]]] ~S+ 5  # [1+5, [[2+5, [3+5]]]]
```

Internally, the `scalar_operator` method is called.

Reverse scalar operator

The reverse scalar operator uses the given scalar as a first operand to the given operator and is also defined for arbitrary nested arrays.

```
[3,4,5]       ~RS  1  # [[1,3], [1,4], [1,5]]
[3,4,5]       ~RS/ 1  # [1/3, 1/4, 1/5]
[3,[[4,[5]]]] ~RS/ 1  # [1/3, [[1/4, [1/5]]]]
```

Internally, the `rscalar_operator` method is called.

Entrywise operations

The `Array.wise_op()` method takes two arbitrary nested arrays and an operator, folding each element (entrywise) with the provided operator, which is also available as `a ~Wop b`:

```
say ([1,2,[3,[4]]] ~W+ [42,43,[44,[45]]])       #=> [43, 45, [47, [49]]]
```

Alternatively:

```
say wise_op([1,2,[3,[4]]], '+', [42,43,[44,[45]]])   #=> [43, 45, [47, [49]]]
```

When the provided operator is an empty string (`''`), the pairwise elements are combined together in a new array:

```
say wise_op([1,2,3], '', [4,5,6])                  #=> [[1, 4], [2, 5], [3, 6]]
say wise_op([1,2,[3,[4]]], '', [42,43,[44,[45]]])  #=> [[1, 42], [2, 43], [[3, 44], [[4
```

Multiple arbitrary nested arrays can be combined together using the `Array.combine{...}` method, added in Sidef 3.50:

```
var a = [[6, 6], [4, 4]]
var b = [[1, 2], [3, 4]]
var c = [[9, 5], [7, 2]]

say [a,b,c].combine{|x,y,z|
    x + y + z
}
```

Output:

```
[[16, 13], [14, 10]]
```

Scalar operations

```
A `scalar_add` 42   # scalar addition       (aliased as `sadd`)
A `scalar_sub` 42   # scalar subtraction    (aliased as `ssub`)
A `scalar_mul` 42   # scalar multiplication (aliased as `smul`)
A `scalar_div` 42   # scalar division       (aliased as `sdiv`)
```

This methods are provided by `Array.scalar_op()`, which, just like `Array.wise_op()`, also supports arbitrary nested arrays:

```
say ([1,2,[3,[4]]] ~S+ 42)   #=> [43, 44, [45,  [46]]]
say ([1,2,[3,[4]]] ~S* 42)   #=> [42, 84, [126, [168]]]
```

which is equivalent with:

```
say scalar_op([1,2,[3,[4]]], '+', 42)  #=> [43, 44, [45,  [46]]]
say scalar_op([1,2,[3,[4]]], '*', 42)  #=> [42, 84, [126, [168]]]
```

Pipeline array operators

The Array operators `|>>`, `|Z>` and `|X>` can be used in a pipeline-fashion to map the elements of an array, given a list of method names or function objects.

Map pipeline operator

The map pipeline operator `|>>` maps the array to a given method or function, with optional arguments:

```
[1,2,3] |>> (:pow, 2) |>> (:mul, 5) |> :say                  #=> [5, 20, 45]
[1,2,3] |>> { _**2 } |>> { _*5 } |> :say                     #=> [5, 20, 45]
[1,2,3] |>> ({|a,b| a**b }, 2) |>> ({|a,b| a*b }, 5) |> :say  #=> [5, 20, 45]
```

Zip pipeline operator

The zip pipeline operator, `|Z>`, zips the array over the callback argument list, `n`-elements at a type, where `n` is the number of callbacks:

```
[1,2,3,4,5,6] |Z> (:exp2, :exp10) |> :say   #=> [2, 100, 8, 10000, 32, 1000000]
```

Each callback that contains arguments, must be enclosed inside an array:

```
[42, 100, 99, 49] |Z> ([{|a,b| a + b }, 3], :sqrt) |> :say     #=> [45, 10, 102, 7]
```

Cross-product pipeline operator

The cross-product pipeline operator, `|X>`, maps each element of the array over each element of the argument list, which is a list of callbacks. Each callback that contains arguments, must be enclosed inside

an array.

```
[25, 36, 49] |X> (:sqrt, [{|a,b| a + b }, 3]) |> :say     #=> [5, 28, 6, 39, 7, 52]
```

The pipeline operators can be combined freely in any order:

```
[25, 36, 49] |X> (:sqrt, [{|a,b| a+b }, 3]) |Z> ({_*_}, [{|a,b| a-b }, 3]) |> :say
```

Matrix

The built-in `Matrix` class (child of the `Array` class) provides support for defining and working with matrices:

```
Matrix(
    [1, 2],
    [3, 4],
)
```

Operations

A subset of `Matrix` operations are included in the following example:

```
var A = Matrix(
    [2, -3,  1],
    [1, -2, -2],
    [3, -4,  1],
)

var B = Matrix(
    [9, -3, -2],
    [3, -1,  7],
    [2, -4, -8],
)

say (A + B)     # matrix addition
say (A - B)     # matrix subtraction
say (A * B)     # matrix multiplication
say (A / B)     # matrix division

say (A + 42)    # matrix-scalar addition
say (A - 42)    # matrix-scalar subtraction
say (A * 42)    # matrix-scalar multiplication
say (A / 42)    # matrix-scalar division

say A**20       # matrix exponentation
say A**-1       # matrix inverse: A^-1
say A**-2       # (A^2)^-1

say B.det             # matrix determinant
say B.solve([1,2,3])  # solve a system of linear equations
```

Matrix iteration

The extended `for-in` loop provides built-in iteration over a 2D-array, which is useful in combination with the cross or zip metaoperators:

```
for a,b in ([1,2] ~X [3,4]) {
    say "#{a} #{b}"
}
```

This is equivalent with:

```
[[1,2], [3,4]].cartesian {|a,b|
    say "#{a} #{b}"
}
```

and outputs:

```
1 3
1 4
2 3
2 4
```

The same functionality is also provided by the `.each_2d {|a,b,...| ... }` Array method.

Lists

An array can be converted into a list using the following notations:

```
var arr = ["a", 1, "b", 2]
say Hash(arr...)            # creates an Hash by passing the array as a list of values
say Hash(@|arr)             # ==//==
```

The difference between postfix `...` and prefix `@|` consists in the fact that `@|` invokes the `...` method only when its argument can respond to this method, while in the first case, the `...` method is invoked unconditionally.

Slices

A slice is a sub-array, just like a sub-string is for a string.

```
var arr = ["foo", "bar", "baz"]

var *slice1 = arr[0, 1]         # fetches the first two values

var indices = [-2, -1]
var *slice2 = arr[indices]      # automatically unpacks the `indices` and fetches the l

say slice1                      # prints: ["foo", "bar"]
say slice2                      # prints: ["bar", "baz"]
```

Using slices, it's also possible to change multiple values inside an array:

```
var arr = ["foo", "bar", "baz"]
arr[0, 1] = ("a", "b")          # changes the first two values
say arr                         # prints: ["a", "b", "baz"]
```

Alternatively, using indices stored inside an array:

```
var arr = ["foo", "bar", "baz"]
var indices = [0, 1]
arr[indices] = ("a", "b")       # changes the first two values
say arr                         # prints: ["a", "b", "baz"]
```

An empty `[]`, will return the entire array as a list of lvalues:

```
var arr = ['a', 'b', 'c']
arr[] = ('foo', 'bar')         # replaces the entire array
say arr                        # prints: ['foo', 'bar']
```

Ranges

A range is a definition of some consecutive values, either in increasing or decreasing order. The main advantage of a range over an array is that a range can be infinite, while an array can't, and this is because ranges are lazy.

A range can be created with one of the following operators: `..`, `^..` or `..^`;

Mnemonics: * `..` go from left to right (inclusive) * `..^` begin from down and go up (exclusive) * `^..` begin from up (exclusive) and go down

Count from 1 to 10:

```
for i in (1 .. 10) {
   say i
}
```

Count from 1 to 9:

```
for i in (1 ..^ 10) {
   say i
}
```

Count from 9 to 1:

```
for i in (10 ^.. 1) {
   say i
}
```

A range can be shifted (`+`, `-`), stretched (`*`) and shrank (`/`):

```
(1..10) + 2       #=>   3 .. 12
(1..10) - 2       #=>  -1 ..  8
(1..10) * 2       #=>   2 .. 20
(1..10) / 2       #=> 0.5 ..  5
```

Also, ranges can be reversed (`.reverse`) and granularized (`.by`):

```
(1..10).reverse        # a new reversed range from 10 down to 1
(1..10).by(0.5)        # a range from 1 up to 10, counting by 0.5
```

Alternatively, ranges can be created with the `RangeNum` data-type:

```
var evens = RangeNum(0, Inf, 2)      # range of all even numbers
say evens.lazy.first(10)             # the first 10 even numbers
```

A negative third argument will create a descending range:

```
for i in RangeNum(10, 5, -1) {    # count from 10 down to 5
    say i
}
```

The `RangeNumber` class inherits methods from the `Range` class, but it also implements some useful methods for working with numerical ranges, such as:

```
say  sum(1..10, {|n| n**3 })     # sum of the first 10 cubes
say prod(1..10, {|n| n**2 })     # product of the first 10 squares
```

Hashes

Hashes are used to quickly locate a data record (e.g., a dictionary definition) given its search key.

```
var hash = Hash(
    name => 'foo',
    age  => 42,
)
```

Working with hashes is almost the same as working with arrays, but instead of specifying a position index in square brackets, we now lookup with a string specified in curly brackets.

```
say hash{:name}    # prints the value associated with the "name" key
say hash.{"name"}  # ==//==
```

Changing a hash value:

```
hash{:age} = 99    # sets the key "age" to value 99
hash.{"age"} = 99  # ==//==
```

Multiple values

Just like arrays, hashes also support the retrieving of multiple values at once.

```
var hash = Hash(
    a => 1,
    b => 2,
    c => 3,
)

# Returns a list of values
var *vals = hash{:a, :b, :c}

# Print the values
say vals                        #=> [1,2,3]

# Using the keys defined inside an array
var keys = %w(a b c)
var *vals = hash{keys...}

# Print the values
say vals                        #=> [1,2,3]
```

An empty `{}` will return the entire hash as a list of lvalue-pairs:

```
var hash = Hash(a => 1, b => 2)
hash{} = (c => 3, d => 4)         # replaces the entire hash
say hash                           # prints: Hash(c => 3, d => 4)
```

Working with hashes

Hashes, like everything else, are objects which have many methods built-in, helping in dealing with hash tables.

Sorting by value:

```
# Sorting in ascending order
var asc_array = hash.sort_by{|_,v| v }

# Sorting in descending order
var des_array = hash.sort_by{|_,v| v }.reverse
```

For lower-level comparisons, the `.sort{}` method can be used:

```
# Sorting in ascending order
var keys_array = hash.keys.sort{|a,b| hash{a} <=> hash{b} }

# Sorting in descending order
var keys_array = hash.keys.sort{|a,b| hash{b} <=> hash{a} }
```

For more methods, see: Hash.pod

Sets

A set is an unordered collection of objects, with no duplicates.

```
Set('foo', 'bar', 'baz')
```

Operations

All the set operators, such as intersection, difference, symmetric difference, union and concatenation, are supported.

```
var A = Set('foo', 'bar', 'baz', 'foo')
var B = Set('bar', 'foo', 'qux')

# Intersection
say (A & B)         #=> Set("foo", "bar")

# Union
say (A | B)         #=> Set("foo", "qux", "bar", "baz")

# Difference
say (A - B)         #=> Set("baz")
say (B - A)         #=> Set("qux")

# Symmetric difference
say (A ^ B)         #=> Set("baz", "qux")

# Concatenation
say (A + B)         #=> Set("baz", "bar", "qux", "foo")
```

Updating

The method `set.delete(obj)` can be used for removing a given object from the set.

Bags

A bag (also known as a multi-set) is a unordered collection of objects, similar to a hash table, where each object has a count number, which represents the number of times it exists in the bag.

```
Bag('foo', 'bar', 'baz')
```

Operations

The Bag class supports all the set operators, such as intersection, difference, symmetric difference, union and concatenation.

```
var A = Bag('foo', 'bar', 'baz', 'foo')
var B = Bag('bar', 'foo', 'qux')

# Count how many times is 'foo' present in the bag A
say A.count('foo')  #=> 2

# Intersection
say (A & B)         #=> Bag("foo", "bar")

# Union
say (A | B)         #=> Bag("qux", "bar", "baz", "foo", "foo")

# Difference
say (A - B)         #=> Bag("baz", "foo")
say (B - A)         #=> Bag("qux")

# Symmetric difference
say (A ^ B)         #=> Bag("foo", "qux", "baz")

# Concatenation
say (A + B)         #=> Bag("foo", "foo", "foo", "bar", "bar", "baz", "qux")
```

Updating

The methods `bag.add_pair(obj, count)` and `bag.update_pair(obj, count)` can be used for efficiently updating a bag in-place.

```
var A = Bag('foo', 'foo', 'bar')

# Add 'bar' with count=2
A.add_pair('baz', 2)

say A   #=> Bag("baz", "baz", "bar", "foo", "foo")

# Update the count of 'foo'
A.update_pair('foo', 1)

say A   #=> Bag("baz", "baz", "bar", "foo")
```

Furthermore, the method `bag.delete(obj)` can be used for removing one occurrence of object `obj` from the bag, while the `.delete_all(obj)` can be used for removing all the occurrences of `obj` from the bag.

Pairs

Creating a long chained list in Sidef is as simple as it can get:

```
var tree = 'root':'child':'grandchild':'end'
```

The above code creates a pair of pairs, which looks like this:

```
Pair('root',
    Pair('child',
        Pair('grandchild', 'end')
    )
)
```

For traversing the list, we can use:

```
loop {
    say tree.first
    tree.second.is_a(Pair) || break
    tree = tree.second
}
```

This is also useful in creating an array of pairs:

```
var array_of_pairs = [
    "red":   9,
    "blue":  4,
    "green": 0,
]
```

Now, each element of the array is a `Pair` and can be accessed by using the `first` and `second` methods:

```
array_of_pairs.each { |pair|
    say "#{pair.first} == #{pair.second}"
}
```

Structs

A structure is very similar with a class without methods and can be used as a stricter alternative to hashes.

```
struct Person {
    String name,
    Number age,
}

# Create a new person
var john = Person(name: "John Smith", age: 42)

# Change a value
john.name = "Dr. #{john.name}"

# Increment a value
john.age++

say john.name       #=> Dr. John Smith
say john.age        #=> 43
```

Files

A `File` is a built-in type in Sidef, in the same way as a `String` or an `Array` is.

Declaring a `File` object:

```
var file1 = File('/tmp/abc.txt')
var file2 = %f(/tmp/abc.txt)            # same thing
```

Being an object, it can have some interesting methods, and it does. The following code will simply edit a file in place:

```
File('/tmp/abc.txt').edit { |line|
    line.gsub(/this/, 'that')              # replaces 'this' with 'that' anywhere ins
}
```

File info

Here is a list with the most important methods which verifies some attributes of the file.

```
file.exists       # true if file exists
file.size         # size of the file
file.is_emtpy     # true if size is zero
file.is_dir       # true if file is a directory
file.is_readable  # true if file is readable
file.is_link      # true if file is a link
file.is_text      # true if file is a text-file
```

Some more information can be achieved by using the `stat` or `lstat` method:

```
var info = file.stat    # or 'lstat'
say info.atime
say info.mtime
say info.ctime
say info.size
```

Information related to the self object file:

```
file.name        # the original name of the file
file.base        # the base name of the file
file.abs         # the absolute path the file
file.dir         # the parent directory of the file
```

Manipulating files

We can also delete and rename files and do other things to files.

```
file.rename("new-name.ext")       # rename file
file.move("new-name.ext")         # rename file safer
file.copy("new-name.ext")         # copy file
file.unlink                       # delete file
file.touch                        # create file if it doesn't exists
file.chmod(0666)                  # change the permissions
file.utime(atime, mtime)          # change the access and modification times
```

Open files

A `File` object has a main `open` method which is directly bound to Perl's `open` function.

```
file.open('<:utf8', \var fh, \var err) \
     || die "Can't open file #{file}: #{err}\n"
```

A simpler interface is provided by the `open_*` methods:

```
var fh   = file.open_r             # open the file for reading
var bool = file.open_r(\var fh)    # same thing, but returns a Boolean value
```

File handles

For reading the content of a file into a string, we can use the `FileHandle.slurp()` method:

```
var str = fh.slurp         # reads the content of the file into a string
```

Alternatively, the `FileHandle.lines()` method will return an array will each line from the file:

```
var arr = fh.lines         # reads the content of the file into an array
```

For reading one line at a time, we can use the `FileHandle.each{}` method:

```
fh.each { |line|
     say line
}
```

Conditional expressions

Conditional expression are almost the same in most programming languages and *Sidef* will not make an exception, so we'll have the classic `if`, `while` and `for` conditional expressions.

The `if` statement is one of the most basic conditional constructs.

```
if (bool) {

}
elsif (bool) {

}
else {

}
```

The `with` statement behaves almost like the `if` statement, but instead of testing for trueness, it checks to see if the given argument is not a `nil` value.

```
with (obj) {

}
orwith (obj) {

}
else {

}
```

In addition to the `if` statement, it also supports capturing of a defined value in a block variable:

```
with (some_function()) { |value|
    say value
}
```

The `while` statement is almost like the `if` construct, except that it will keep executing its block as long the given expression evaluates to a true value.

```
while (bool) {

}
```

The `for` statement it's usually used for iteration over collections and for counting.

```
for (var i = 0; i <= 10; i++) {

}
```

Also, we have the ternary operator and the *case and switch statements*:

```
bool ? (true) : (false)
```

Given/when is used to compare two values using the rules of the smartmatch operator (`~~`):

```
given ('b') {
  when ('a') { say "a" }
  when ('b') { say "b" }
  else       { say "Unknown!" }
}
```

Additionally, to test expressions for trueness, Sidef has the `case` statement:

```
given (-1) {
  case (.is_zero) {
       say "Null value!"
  }
  case (.is_neg) {
       say "Negative value!"
  }
  else {
       say "Positive value!"
  }
}
```

`case` and `when` statements can be mixed together:

```
given (42) { |value|
  case (value < 0) {
       say "Negative value!"
  }
  when (0) {
       say "Null value!"
  }
  case (value > 1) {
       say "Positive value!"
  }
}
```

When a value is found, the construct breaks automatically, but we have the `continue` keyword which will prevent this.

```
given (1) {
   when (1) {
      say "true once"
      continue        # will fall through
   }
   when (1) {
      say "true twice"
   }
}
```

Exceptions

An exception is thrown by the `die` keyword, which, if not caught, it terminates the program with an appropriate exit code.

```
try  {
    die "I'm dead!"        # throws an exception
}
catch { |msg|
    say "msg: #{msg}"      # msg: I'm dead! at test.sf line 2.
}

say "I'm alive..."
die "Now I'm dead!"        # this line terminates the program
say "Or am I?"             # Yes, you are!
```

Regular expressions

Sidef borrows the regular expressions from Perl. Any regular expression is analyzed and compiled by Perl's regexp engine, so we have all the good stuff we are already familiar with.

```
var regex = /^my+[regex]?\z/ixmsu
```

Matching against regular expressions:

```
var string = 'something'

if (string =~ regex) {
   say "Matches!"
}
```

Storing and using the captured matches:

```
var string = "Sidef <3 Perl"
var match = string.match(/(\w+)\h+<3\h+(\w+)/)

if (match) {
   var captures = match.captures
   say captures[0]                # prints: Sidef
   say captures[1]                # prints: Perl
}
```

The returned match object, it's a special object which accepts array indexing of values.

```
var m = "hello world".match(/^(\w+) (\w+)/)
say m[0]                                  # prints: hello
say m[1]                                  # prints: world
```

Global matching

A regex can match multiple times inside a given string, therefore Sidef provides support for global matching.

```
var str = "a cat, a dog and a fox"
while (var m = str.match(/\ba\h+(\w+)/g)) {
    say m[0]
}
```

Alternatively, there is the `String.gmatch()` method:

```
var string = 'Sidef <3 regular expressions'
while (var m = string.gmatch(/(\S+)/)) {
    say m[0]
}
```

Smart-matching

The smart-match operator (`~~`) take two operands and compare them based on their type and their order.

```
"hello" ~~ /^h/        # true: string matches regex
"oo" ~~ "foobar"       # false: "oo" doesn't equal "foobar"
"a" ~~ %w(a b c)       # true: item exists in array
/^b/ ~~ %w(foo bar)    # true: regex matches an element from array
/^f/ ~~ Hash(foo => 1) # true: regex matches a key from hash
```

There is also `!~` which simply flips the Boolean value returned by `~~` .

```
/abc/ !~ "abcdef"     # false
```

Modules

In Sidef, a module is the declaration of a new namespace:

```
module Fibonacci {
    func nth(n) {
        n > 1 ? nth(n-2)+nth(n-1) : n
    }
}

say Fibonacci::nth(12)    # prints: 144
```

The default namespace name is `main`. Variables from other namespaces can be used inside a module by either importing them, or by specifying their full name, including the namespace:

```
var foo = 42

module Bar {
    var baz = 99
    say main::foo   #=> 42
}

say Bar::baz        #=> 99
```

Importing an identifier in the current namespace, can be done using the syntax `import namespace::identifier_name`:

```
var foo = 42

module Bar {
    import main::foo
    var baz = 2*foo
}

import Bar::baz
say baz          #=> 84
```

Classes

A class is a collection of methods and attributes. From classes, we can create objects. When a class is *called*, an instance-object of that class is returned, which encapsulates the given data provided at class initialization. Each individual *call* to a class, returns a different instance-object.

```
class Person (name, age, address) {
    method position {
        # GPS.locate(self.address)
    }

    method increment_age(amount=1) {
         self.age += amount
    }
}

var obj = Person(name: "Foo", age: 50, address: "St. Bar")
say obj.age                # prints: 50
say obj.name               # prints: "Foo"
say obj.address            # prints: "St. Bar"

obj.name = "Baz"           # changes name to "Baz"
say obj.name               # prints: "Baz"

obj.increment_age          # increments age by 1
say obj.age                # prints: 51
```

Classes in Sidef are a little bit different than classes from other languages. Instance-variables (also known as attributes) are accessible via a method call which can either get or set a value, as illustrated in the example above.

Class attributes

The attributes of a class can be either specified as parameters, or declared with the `has` keyword.

```
class Example(a, b) {
   has c = 3
   has d = a+c
}

var obj = Example(1, 2)

say obj.a   #=> 1
say obj.b   #=> 2
say obj.c   #=> 3
say obj.d   #=> 4
```

Class initialization

Extra object-initialization setup can be done by defining a method named `init`, which will be called automatically called whenever a new instance-object is created.

```
class Example (a, b) {

   has r = 0

   method init {      # called automatically
      r = a+b
   }

   method foo {
      r
   }
}

var obj = Example(3, 4)
say obj.foo              #=> 7
```

Class inheritance

A class can inherit methods from other classes by using the special operator `<`, followed by the name of the inherited class:

```
class Animal(String name, Number age)  {
    method speak { "..." }
}

class Dog(String color) < Animal {
    method speak { "woof" }
    method ageHumanYears { self.age * 7 }
}

class Cat < Animal {
    method speak { "meow" }
}

var dog = Dog(name: "Sparky", age: 6, color: "white")
var cat = Cat(name: "Mitten", age: 3)

say dog.speak          #=> woof
say cat.speak          #=> meow
say cat.age            #=> 3
say dog.ageHumanYears  #=> 42
say dog.color          #=> white
```

Multiple inheritance is declared with the `<<` operator, followed by two or more class names, separated by commas:

```
class Camera { }
class MobilePhone { }
class CameraPhone << Camera, MobilePhone { }
```

Class variables

The syntax `ClassName!var_name` can be used for defining, accessing or modifying a class variable.

```
class Example {

    Example!hidden = 'secret'    # global class variable

    method concat (str) {
        str + ' ' + Example!hidden
    }
}

var x = Example()
var y = Example()

say x.concat('foo')   #=> 'foo secret'
say y.concat('bar')   #=> 'bar secret'

Example!hidden = 'public'  # changing the class variable

say x.concat('foo')   #=> 'foo public'
say y.concat('bar')   #=> 'bar public'
```

The modification of a class variable can be localized by prefixing the declaration with the `local` keyword:

```
local Example!hidden = 'local value'
```

Metaprogramming

An interesting feature is the definition of methods at runtime:

```
var colors = Hash(
               'black'   => "000",
               'red'     => "f00",
               'green'   => "0f0",
               'yellow'  => "ff0",
               'blue'    => "00f",
               'magenta' => "f0f",
               'cyan'    => "0ff",
               'white'   => "fff",
             )

for color,code in colors {
    String.def_method("in_#{color}", func (self) {
        '<span style="color: #' + code + '">' + self + '</span>'
    })
}

say "blue".in_blue
say "red".in_red
say "white".in_white
```

Output:

```
<span style="color: #00f">blue</span>
<span style="color: #f00">red</span>
<span style="color: #fff">white</span>
```

Methods can have variable-like names (`a` , `hello` , etc...), or operator-like names (`+` , `**` , etc...).

```
class Number {
    method ⊕(arg) {
        self + arg
    }
}

say (21 ⊕ 42)
```

The special method *AUTOLOAD* is called when a given method is missing.

```
class Example {
    method foo {
        say "this is foo"
    }
    method bar {
        say "this is bar"
    }
    method AUTOLOAD(_, name, *args) {
        say ("tried to handle unknown method %s" % name)
        if (args.len > 0) {
            say ("it had arguments: %s" % args.join(', '))
        }
    }
}

var example = Example()

example.foo           # prints “this is foo”
example.bar           # prints “this is bar”
example.grill         # prints “tried to handle unknown method grill”
example.ding("dong")  # prints “tried to handle unknown method ding”
                      # prints “it had arguments: dong”
```

Parallel computation

Sidef has a very basic support for parallel computation, but pretty powerful. There is the `Block.ffork()` method which creates a new system process that executes (in parallel) the content of a given block and returns a new `Fork` object which accepts the `wait` (or `get`) method that waits for the process to finish and returns the computed value.

To take advantage of this mechanism, fork two or more processes and store the returned objects in variables or inside an array and get their values at a later time. The process is executed soon after it has been forked.

Example for the quicksort algorithm in parallel:

```
func quicksort(arr {.len <= 1}) { arr }

func quicksort(arr) {

    say arr
    var p = arr.pop_rand

    var forks = [
        quicksort.ffork(arr.grep { _ <= p }),
        quicksort.ffork(arr.grep { _ >  p }),
    ]

    forks[0].wait + [p] + forks[1].wait
}

say quicksort(@("a".."z") -> shuffle)
```

Alternatively, Sidef provides the `Block.thr` method, which creates a deprecated Perl thread, or a system fork if forks is installed.

Interacting with Perl modules

Sidef can interact with Perl modules in a very easy way. There is the `require` keyword which will try to load an object-oriented Perl module.

```
var lwp  = require('LWP::UserAgent')
var ua   = lwp.new(show_progress => 1)
var resp = ua.get('http://example.net')

if (resp.is_success) {
    say resp.decoded_content.length
}
```

For functional Perl modules, the `frequire` keyword should be used instead to denote that the module is function-oriented.

```
var spec = frequire('File::Spec::Functions')
say spec.rel2abs(spec.curdir)
```

Literal Perl modules

There is also a special syntax for literal names of Perl modules, creating a module-interface object, without `require`-ing the module in the first place.

```
%O<LWP::UserAgent>         # object-oriented module
%S<File::Spec::Functions>  # subroutine/function oriented module
```

This allows us to ignore the return value of `require` and create the module-interface object afterwards.

```
require('ntheory')
var nt = %S<ntheory>   # function-oriented module

if (nt.is_prime(43)) {
    say "43 is prime!"
}

nt.forprimes({|p| say p }, 0, 100)
```

Graphical interface

Due to the fact that we can interact with Perl modules, we can also create graphical interfaces from Sidef, by using the `Gtk3` library.

```
Perl.eval('use Gtk3 -init')

var gtk3   = %O<Gtk3>
var window = %O<Gtk3::Window>.new
var label  = %O<Gtk3::Label>.new('Goodbye, World!')

window.set_title('Goodbye, World!')
window.signal_connect(destroy => { gtk3.main_quit })

window.add(label)
window.show_all

gtk3.main
```

Deparsing

Deparsing is the reverse process of parsing, which translates the AST back into code. Currently, Sidef supports deparsing into two languages with the `-R lang` command-line switch:

- `-R perl`
  - Deparses the AST into valid Perl code.
- `-R sidef`
  - Deparses the AST into valid Sidef code.

Example:

```
$ sidef -Rperl script.sf | perl
```

The `-Rsidef` switch (or simply `-r`) is useful for verifying how the code is parsed.

Example:

```
$ sidef -r -E '1 + 2/3'
```

Outputs:

```
(1)->+((2)->/(3));
```

Creating an executable

By using the PAR::Packer tool, we can create an executable binary from a Sidef script (`script.sf`), by executing the following commands:

```
$ sidef -Rperl script.sf > script.pl
$ pp --execute script.pl
```

Currently, Sidef code that includes `eval()` cannot be compiled into an executable.

Inlining Perl code in Sidef

`Perl.eval()` can evaluate arbitrary Perl code and convert the result into a Sidef data structure which can be used later in the program.

```
var perl_code = <<'CODE'

sub fact {
    my $p = 1;
    $p *= $_ for 2..$_[0];
    $p;
}

my %data = (
    result => fact(10)
);

\%data;   # returned data to Sidef
CODE

var data = Perl.eval(perl_code)
say data{:result}                #=> 3628800
say Sys.ref(data{:result})       #=> Sidef::Types::Number::Number
```

A practical example would be the creation of Sidef blocks which incorporate arbitrary Perl code. Doing so, the returned object will behave exactly like a native Sidef block:

```
var perl_code = <<'CODE'

Sidef::Types::Block::Block->new(
    code => sub {
        my ($n) = @_;
        ($n <= 1) ? $n
                  : (CORE::__SUB__->($n-1) + CORE::__SUB__->($n-2));
    },
    type => 'func',
    name => 'fib',
    table => {'n' => 0},
    vars => [{name => 'n'}],
);

CODE

var block = Perl.eval(perl_code)
say block(28)     # 28-th Fibonacci number
```

More examples

For more examples, see: https://github.com/trizen/sidef-scripts