Raw Notes on the Ruby Language

Marcello Galli, October-2014

These notes are the by-product of a never-held seminar on the Ruby language.

I like the Ruby language: it has a lot of peculiar features, interesting ideas implemented and a lot of features built into the language; it is a complex language, so, while studying the language, I prepared some raw notes for myself, with the idea of a Ruby seminar for my colleagues, but a Python course was more than enough for them, so the seminar was never held.

I ended putting these raw notes on my website, but on the web there is so much material about Ruby, that I doubt I will ever have some casual readers. Nevertheless these notes are here, unfinished, in a raw state, with errors and typos, but free for everyone (for non-commercial use), released under the "Creative Commons License":http://creativecommons.org/licenses/by-nc-sa/4.0/.

Downloaded from http://www.helldragon.eu/

Contents

Introduction 4

References 4

Main Features 6

Usage 8

Syntax 8

Predefined variables and constants 10

Inclusion of External Files 11

Scope of Names 12

Scope of Constants 12

Statements 13

- Conditional Statements 13
- Loop Statements 14
- Exception Handling 16
- Postfix Expressions 18

Operators 19

Classes and Methods 21

- Instance and Class Variables 21
- Methods and Functions 21
- Public, Private and Protected Methods 22
- Function Call 23
- Function Definition 23
- Function Arguments 24
- Blocks Given to Functions 25
- proc and lambda 26
- Class Definition 28
- Inheritance 29
- Class Instances 29
- Class Methods and Instance Methods 29
- Accessor Functions and Instance Variables 30
- Adding Methods to a Class 31
- Singleton Methods 31

The Object Class 33

Logical Classes 34

Numeric Classes 35

Downloaded from http://www.helldragon.eu/

Methods for the Numeric Classes 36
Methods for Integers 37
Methods for Floats 38
Conversion Methods 38
Subscript Operator 38
Precedence for operators 39
String Class 40
String Encoding 40
Double-quoted String 41
Single-quoted Strings 41
String Operators 42
Array Class 45
The Subscript Operator: [] 47
Hash Class 48
Range Class 50
Regular Expressions 51
Other Builtin Classes 54
Iterators 56
The Enumerator Class 56
The Enumerable Module 56
Some iterator for numerics 57
Some Iterators for Strings 58
Some Iterators for Arrays 58
Some Iterators for Hashes 59
Input/Output 60
Modules 63
Builtin Modules 64
Standard Library 64

Downloaded from http://www.helldragon.eu/

Introduction

Ruby is a new language, developed by Yukihiro Matsumoto in 1993, as a simple, very object oriented, interpreted language, influenced by Perl, Python, Lisp. For some years Ruby remained a Japanese only product, developed on Japanese newsgroups and mailing lists. Only in 1999 an English mailing list on Ruby appeared, and only in 2000 the first English book on Ruby was printed; this greatly limited the diffusion of the language, which deserved a wider audience, but, for some time, was known only in Japan. Around 2005, the diffusion of the web framework "*Rails*" gave new emphasis to the diffusion of Ruby.

In many aspects Ruby is different from every other programming language I have ever used; it is an interpreted language, as Python, and has many similarities with Python: harray, hashes, metaprogramming, a polymorphic structure. Many things work in a similar way, but the underline philosophy is very different: Pyhton wants to be simple, with a clear structure, few lexical constructs, each thing being done in only one way (and eventually the correct one); the core is simple and most is in additional modules. Ruby instead gives to the programmer the maximum flexibility; there are many ways to do the same thing, there is an exasperation of the object oriented paradigm, and every possible feature is fitted into the language. Ruby is a very interesting language, but it is not easy.

Ruby versions	
Version 0.95	1995
Version 1.0	1996
Version 1.2	1998
Version 1.4	1999
Version 1.6	2000
Version 1.8	2003
Version 1.9	2007
Version 2.0	2013
Version 2.1	2013

Version 1.8 was accepted as an ISO and JIS standard, but version 1.9 is not backward compatible with version 1.8; version 2 also has minor incompatibilities with version 1.9. Version 2 introduced a better interpreter (YARV). There are also just-in-time Ruby compilers, and Jruby, which produces a Java bytecode.

References

There are many books on Ruby, but most are limited the version 1.8 of the language, new editions being planned for 2014-2015. The evolution of Ruby is fast, and with version 1.9 many things changed, so you have to look for books covering at leas version 1.9. Some books have been translated in Italian, but I suggest to avoid the Italian translations of books on computer science. Often the quality of the translation is bad; it's clearly done by non-programmers: with a wrong choice of Italian technical terms, and errors, so some topics are difficult to understand. Some good books are:

- **Programming Ruby 1.9 & 2.0, by Dave Thomas et al., Pragmatic, 2013.**
  It's the reference book for Ruby, with all class methods listed and explained.
- **The Ruby Programming Language, by David Flanagan, Yukihiro Matsumoto, O'Reilly 2008.**
  A good book to learn the language.
- **Learning Ruby, by Michael Fitzgerald, O'Reilly 2007.**
  A good book but doesn't cover version 1.9.

Downloaded from http://www.helldragon.eu/

- **Ruby Pocket Reference, by Michael J. Fitzgerald, O'Reilly, 2007.**
  An useful booklet, with a summary of the language, but it doesn't cover version 1.9.

The official web site for the language is: https://www.ruby-lang.org

Documentation can be found also on: http://ruby-doc.org/

A bit dated Ruby guide is in: http://www.rubyist.net/~slagell/

Downloaded from http://www.helldragon.eu/

Main Features

A modern, interpreted language:

in may aspect Ruby is similar to Python: variables are reference to objects, and their type is defined at run time, when they are assigned. In this way we have a polymorphic language (the same functions can be used for different data types), but the errors on the type of variables are detected only at run-time.
Ruby allows for reflection (the program knows its internals) and metaprogramming (the program can change itself at run-time).
As an interpreted language Ruby is not very efficient and it is not suited for heavy computations; compiled languages perform better. Ruby uses a bytecode to speed things and has a builtin garbage collection system.

Very, very object oriented:

In Ruby all is an object, the object paradigm is very empathized and this leads to a change in the meaning and usage of some classic elements of computer languages: methods and operators become the same thing; no more distinction between basic types and objects, instance variables are used through methods, common keywords become methods of some basic object, loops are replaced by iterator methods:

- Ruby tries to follow in a strict way the encapsulation paradigm. Class and instance variables are strictly private and accessible only thought special "accessor" methods. These methods, used without parenthesis, seems a direct usage of class variables; *instance variables and methods become two very similar things in practical usage*.
- *Also basic types (as numbers) are objects and have attributes*. Most functions and operators are attributes of objects. Basic operators too are implemented as methods, so *the concepts of operator and class method, that are different things in many languages, are unified in Ruby*. Operator overloading become a very common feature of classes and an operation not allowed for a given type becomes the missing of a method in the corresponding class.
- Many features are implemented as methods of basic classes: every class inherits the class *Object*, which is a common parent for all the Ruby classes. The class Object includes the module "Kernel" with all the major functions of Ruby, and also many statements and operators are implemented as methods of the "*Kernel*" module
- In Smalltalk, one of the first object oriented languages, objects are very well separated entities, and calls to methods are seen as "*messages*" send to objects. In Ruby this idea is not so clearly stated, nevertheless each function is a method of an object which acts as the "**receiver**" for the function. When the program runs you are always in the environment of an instance, acting as a search scope for names and as a receiver for functions; also in the interactive usage one is in the environment of an instance, which is named: "*main*" and has inherited *Object*. This is the way the statements that are methods of the *Kernel* module are made available to the interactive user. The current instance (the receiver) is referred to by the keyword "**self**".
- The class itself (the definition of the class) is built as an instances of the class: *Class*; class methods are methods of this instance;
- Only single inheritance is allowed, but a class can include modules, containing attributes that are mixed into the class (mixins).

Duck Typing:

all is an object; also basic types, as integer an floats, are objects: algebraic operators (as addition, multiplication etc.) become methods of the basic objects.

Downloaded from http://www.helldragon.eu/

In object oriented languages as C++, or Java, there are some basic types and, in addition, objects, which define new types. But here, with the disappearing of basic operators (now class members), the concept of type weakens, and the object interface becomes the essential item;
This way of intending types is called "duck typing": the object interface defines its type. For the origin of the expression "duck typing" see: http://en.wikipedia.org/wiki/Duck_test .

The syntax tries to mimic the human (English) language:

- we have postfix syntax, with condition after statements:
- parenthesis in function calls are optional, so that a function call appears as an actions taken on the arguments;
- iterator functions associated with blocks seem actions on sets of objects
- some functions are attributes of basic types, giving a very compact syntax for some common logical constructs: : Es.: 5.times { ...} ;
- we have functions ending with "=", which mimic an assignment; this choice is related to the need of accessor functions to set variables belonging to classes.
- return statements in functions are optional: each block of statements return something and acts as an expression, if not specified returns the last evaluated statements, and, if has nothing meaningful to return, returns the *nil* object.
- the "then" keyword, used to begin conditional blocks, is optional
- .

Many find this attempt to make a programming language nearer to the human language a great feature, making Ruby programming: "*fun*". Solving an interesting problem can be fun, but the used computer language is only a tool: can be easy, powerful, elegant, but if your job is boring no computer language can change things. For me, as an old programmer, programming means translating from the human logic to the computer logic: humans and computers are different and follow different logic schemes, talking to the computer is worthless and essentially wrong. For example I need parenthesis when calling a function, to have an idea of what the computer is doing; in the Ruby world, without parenthesis as delimiters, finding spare words after a function is deceiving and leads to syntactical ambiguities.

Convention over declaration:

- Constants are capitalized, and also class names.
- variables beginning with "*$*"," @"," @ @" are respectively global, instance and class members.

Extended usage of iterators:

there is an extended usage of iterators, implemented as class methods coupled with block of instructions to be executed at each iteration. It's the preferred way of looping in Ruby, with a very compact syntax and an easy way to implement filters and transformations of sequences of objects. Basic types have tenths of different iterators, which can be used for complex tasks. Iterator methods can easily written also for user defined classes.

A lot of funcionalities are built "*into*" the language:

many features are implemented by means of basic types: common structures, as arrays and associative dictionaries (called *hashes* in Ruby), are also basic types. There are many operators, a complete set of functions to deal with strings, regular expressions, an easy ways to interact with the underling operating system, an integrated help system, builtin modules for common actions and countless additional packages to do everything.

Downloaded from http://www.helldragon.eu/

Usage

Ruby can be used in an interactive way, or as a standalone program. There is also an interactive help shell, invoked with the command: **ri**. Some documentation about a Ruby program can be produced with **rdoc** from specially formatted comments placed into the source code. Ruby can be used in the following three ways:

- The command **irb** invokes an interactive Ruby shell: the Ruby shell has some useful commands as: an "*help*" command; a "*source*" command to load Ruby files; "*conf*" to show the configuration; an "exit"* command and commands to manage subshell, similar to the batch jobs in Unix.
- Using the "**ruby**" command: a file containing a Ruby program can be executed with: "*ruby filename.rb*".

  The ruby command has also optional flags to execute a string of Ruby statements or to apply Ruby statements to an input streams as in the following examples:

  ```
  ruby -e 'a=1 ; print "a=",a,"..OK\n " ;' ==> a=1...OK
  echo "aaaa" | ruby -n -e 'print $_.upcase' ==> AAA
  ```
- In Unix an executable file beginning with the line: "*#!/usr/bin/ruby*" can be executed as a shell command.

Syntax

Ruby is **case sensitive** .

Blank lines are ignored and multiple blanks are condensed;

"**#**" is used for comments, extending from the character: "#" up to the end of the line.

A semicolon: "**;**" , or a newline, is used to separate statements, but incomplete expressions can extend over more lines. The backslash: "****" can be used to escape the final newline character, effectively joining two lines.

Curly braces: "**{ }**" are used to define a block of statements;

Logical blocks are initiated by different keywords (**begin class, def, if, do**), but all are terminated by the keyword: "**end**".

The special keyword: "**__END__**" , on a line by itself, without leading or trailing spaces, can be used to mark the end of the program in a Ruby file; all after this line is ignored.

Special blocks of lines have specific delimiters:

```
=begin

       here an embedded document,
       with comments or documentation
=end


BEGIN { here a block of code executed at the beginning of the program }


END   { here a block of code executed at the end of the program }
```

Arrays, hashes and strings are basic types: arrays are sequence of heterogeneous objects, represented as comma separated values between square brackets; their elements are obtained bu an integer index between square brackets. Hashes, instead of an integer index have elements obtained by a key, which can be any object, but is usually a number or a string. Hashes are represented by a sequence of a key and a value, separated by

Downloaded from http://www.helldragon.eu/

an arrow, between curly brackets. Strings are represented as sequences of characters between single or double quotes. Array and hashes will be described in a subsequent section, but some examples are anticipated here:

```
a=[1,2,3]     # creation of an array of three numbers
              # a[0] is a reference to '1', a[1] to '2' ...


b={"a"=>1,"b"=>2}   # an hash of two values, using strings as keys.
                    # b["a"] is a reference to '1', b["b"] to '2'
```

Variables and method names:

names of variables begin with a lowercase letter and are made by an arbitrary number of letters, digits or underscores;

variables are reference to objects, and assignments for numerics and hashes or dictionaries have different effects:

```
a=1 ; b=a ; a=2         =>  you have a==2 ; b==1 ;
                            the assignment b=a makes: '2':a new number object

a=[1,2,3] ; b=a ; a[1]=0 =>  the assignment b=a makes only a new reference.
                             You have both 'a' and 'b' changed to:[1, 0, 3]
                             To make a new array you have to use: b=a.dup
                             The same happens for hashes or strings.
```

Constants:

constants begin with an uppercase letter, are not really constants, can be changed, but Ruby issues a warning when you change a constant.

Class and Module names are references to classes and modules, constant too and capitalized.

Symbols:

symbols are pointers to the internal name table of Ruby; names of symbols have the prefix ":";

they belong to the class "*Symbol*", and, from Ruby 1.9, have some methods similar to the methods of strings (size, uppercase,swapcase etc.). There are also conversion functions between strings and symbols:

```
sym=stringa.to_sym   # change a string into a symbol

stringa=sym.to_s     # change a symbol into a string

stringa=sym.id2name  # change a symbol into a string

Symbol.all_symbols   # shows all symbols
```

Symbols are automatically created by Ruby for nearly every object of a program. The method: **Symbol.all_symbols** return an array of all the defined symbols; **id2name**, or **to_s**, returns a string representing a symbol.

Access to objects through symbols is faster; when an object has been defined, a symbol with the corresponding name can be used to refer to the object. Symbols are often used as keys in hashes.

Some characters have special meaning in names:

Downloaded from http://www.helldragon.eu/

Character	Usage
@	first character for private instance variables
@@	first characters for class variables
$	first character for global variable
?	last character for names of functions returning a bool (not mandatory)
!	last character for names of functions modifying an item in place (not mandatory)
=	last character for names of functions with an usage which mimics an assignment
:	prefix for symbols

Ruby has some reserved words, which can't be used for user-defined names:

```
__ENCODING__        __FILE__ __LINE__ BEGIN    END    alias and   begin
break        case   class    def      defined? do     else  elsif end
ensure       false  for      if       in       module next  nil   not
or           redo   rescue   retry    return   self   super then  true
undef        unless until    when     while    yield
```

Predefined variables and constants

As in Perl, there are many global variables pre-defined, and also pre-defined constants, some are in the following table.

Constant	Meaning
$0	name of the Ruby program (also: $PROGRAM_NAME)
$*	array with command line options (also: ARGV)
$"	array of included modules
$:	path for module searching
$stderr, $stdout, $stdin	standard I/O streams
$_	last read line
$;	default string separator for the split function
$,	separator for printed items (default is nil)
$!	last raised exception
$@	backtrace for last raised exception
$&	for regular expression: matched string
$`	substring before the match
$'	substring after the match
$1...$n	sub-expressions in the match
$?	status of the last sub-process (contains the Status object)
ARGV	array of strings with command line options
ENV	hash of environment variables

Downloaded from http://www.helldragon.eu/

RUBY_DESCRIPTION	description of the Ruby version
RUBY_PLATFORM	the computer type
RUBY_VERSION	Ruby version number

In the ARGV the options are saved as strings, and the first option is in ARGV[0].

Most of these constants can be changed at run-time.

Inclusion of External Files

A ruby program, or the interactive session, can include Ruby statements contained in an external file. The statements are executed in the context of the program; variables, classes, functions and constants defined in the file become available to the program, with the exception of local variables, which are never seen outside of the environment in which they are defined (in this case the external file).

The "**load**" or the "**require**" statements are used to load a file:

```
load 'nomefile.rb'   # can be also used to load compiled binaries

require 'nomefile'   # the ".rb" suffix is assumed

require_relative 'nomefile'   # search path relative to the current directory
                              # not working for the interactive shell.
```

The "*load*" statement can also load compiled libraries into Ruby, and needs the file prefix, which can be "*.so*" or "*dll*" for binary libraries, "*.rb*" for Ruby files. The "*require*" statement loads only Ruby files and doesn't need the file prefix, assuming that it is "*.rb*" .

The "*load*" statement can be executed many times to reload the file; instead the "*require*" statement load the file only once also if called many times; the variable *$"* contains a list of the files loaded with *require*.

Files are searched in the current directory and in system-defined directories. The variable $LOAD_PATH contains an array of directories to search for files, $LOAD_PATH can be changed at run-time.

For example, in a Debian 7 system, we have

```
$LOAD_PATH ==  ["/usr/local/lib/site_ruby/1.9.1",
                "/usr/local/lib/site_ruby/1.9.1/x86_64-linux",
                "/usr/local/lib/site_ruby",
                "/usr/lib/ruby/vendor_ruby/1.9.1",
                "/usr/lib/ruby/vendor_ruby/1.9.1/x86_64-linux",
                "/usr/lib/ruby/vendor_ruby",
                "/usr/lib/ruby/1.9.1",
                "/usr/lib/ruby/1.9.1/x86_64-linux"]

to add the current directory:

  $LOAD_PATH.unshift('./')

or:

  $LOAD_PATH << '.'
```


Downloaded from http://www.helldragon.eu/

The **require_relative** statement doesn't search in the $LOAD_PATH, but only in the path relative the directory from which the statement is called. **require_relative** can't be used in the interactive shell.

Scope of Names

In Ruby the scope for names can be one of: **local, global, instance** and **class**. Names defined out of classes or functions belong to the "*main object*" scope.

Local variables (those beginning with a lowercase letter) are local to the file class, function or block in which are defined and can't be accessed from the outside. Functions don't see local variables defined out of the function itself. A block can see variables defined in the program before the block.

Ruby programs (or the interactive session) can include Ruby files, with the "*load*" or "*require*" statement. These files are executed in the environment of the program, but local variables defined in the file are not seen in the program and vice-versa.

Global variables are accessible in the whole program and the first character of their name is : "**$**" .

Instance variables are private to a class instance; their names begins with: "**@**" and they are seen by the methods of the instance. Special methods are needed to access them from the outside of the object. They have to be initialized by methods of the class, not in the body of the class, outside methods.

In the interactive usage, variables prefixed with "@" are instance variable of the "main" object, and seen in functions.

Class variables, defined in a class, are common to all the instances of the class; if defined in an child class can be seen also in the parent. Their names begin with "**@@**".

The function: "defined?(varname)" return the scope of a defined variable, or "*nil*" if not defined.

Scope of Constants

Names beginning with an uppercase letter (capitalized) are reference to constants.

Constants can't be defined into methods.

Constants defined outside classes are in the global scope.

Constants defined into a class or module belong to the class or module. Can be accessed outside the module with the **scope operator** "**::**" (i.e.: "*Math::PI*"), where the name of the module can be also given by an expression.

Constants defined outside modules, classes or in the main program, can be referred to into classes by using the *scope operator* "**::**", without the module name (Es.: ::Constant).

Downloaded from http://www.helldragon.eu/

Statements

Ruby is a structured language, executes blocks of statements, which are delimited by the words "**begin .. end**" or "**then .. end**". After a conditional instruction the "*then*" statement can be omitted, but not if the statements are on the same line as the condition.

A block returns a value, which is the result of last executed expression.

Conditional Statements

- The **if , then, else** statement:

```
if a==0 then b=77 end    # if on a single line, "then" is mandatory

if boolean-expression then
   statements
elsif boolean-expression then
   statements
else
   statements
end
```

The if statement has a return value: the result of the last executed statement, or nil:

```
name = if  x == 1 then "one"
     elsif x == 2 then "two"
     elsif x == 3 then "three"
     elsif x == 4 then "four"
     else              "many"
     end
```

- **unless**, same as: "*if not ..*"

```
unless a==0 then b=77 end

unless boolean-expression then
  statements
else
  statements
end
```

- The **case** statement, for multiple conditions; the general form is:

```
case target-expr
when comparison , comparison ... then
  statements
when comparison , comparison ... then
  statements
else                 # optional else
  statements
end
```


Downloaded from http://www.helldragon.eu/

The case statement is equivalent to the "*if elsif .. end*" statement:

```
Case statement                     Equivalent if-else statement

name = case                        name = if     x == 1 then "one"
       when x == 1 then "one"             elsif x == 2 then "two"
       when x == 2 then "two"             elsif x == 3 then "three"
       when x == 3 then "three"           elsif x == 4 then "four"
       when x == 4 then "four"            else "many"
       else "many"                        end
       end
```

A more concise syntax; where the x value to test is not repeated at each condition:

```
name = case x
       when 1             # new line as a separator
         "one"
       when 2 then "two"  # then used as a separator
       when 3; "three"    # semicolon as a separator
       else "many"
       end
```

In the case statement, the "**===**" equality is often used. Also ranges are often used in *case* statements:

```
case 74.95
when 1...50 then   puts "low"
when 50...75 then  puts "medium"
when 75...100 then puts "high"
end
```

Loop Statements

The blocks executed many times are between the "**do .. end**" statements, or curly brackets. The "*do*" keyword can be omitted. The "*do*" (or bracket) beginning the loop block must be on the same line as the "*loop*" keyword.

- The infinite **loop**:

```
loop do
  statements
end

loop { a+=1 ; print(a) ; break if a>3 }  # loop on a single line
```

- The **while** loop:

```
while boolean-expression do
  statements
end

x = 0
while x < 10 do puts x = x + 1 end      # loop on a single line
```


Downloaded from http://www.helldragon.eu/

- The **until** loop (same as: while not):

```
until boolean-expression do
  statements
end
```

- *while* and *until* loops executed at least once, with the condition at the end of the block:

```
x = 10
begin                 # Starts the conditional block, executed at least once
  puts x
  x = x - 1
end until x == 0   # condition evaluated at the end

a=10
begin
  a+=1
end while a<3
```

- The **for** loop: this loop returns, at each iteration, an element of a sequence; the sequence is often an array or an hash. The "*do*" keyword is optional if the subsequent statements are on the following lines:

```
for name, name in sequence do
  statements
end

a=[1,2,3,4]
for i in a
     print i
end

for i in a     ; print i ;end
for i in a do    print i ;end

for i in (1..3) do print i ; end      # range usage in the loop


hash = {:a=>1, :b=>2, :c=>3}
for key,value in hash
  puts "#{key} => #{value}"
end
```

- Statements altering the loop flow:

  - **break** : terminates loop immediately; can have an optional argument that is is the loop result
  - **next** : goes to the next iteration.
  - **redo** : repeats the iteration, without testing again the condition.
  - **retry** : repeats the iteration, testing again the condition.

Example:

Downloaded from http://www.helldragon.eu/

```
for i in a
   break if i==3
   next if i==2
   print i
   redo if i==1        # this is an infinite loop
end
```

- **catch, throw** :

  These statements alter the loop behavior; are similar to the trowing of an exception, and can be used to exit from an inner block in a deeply nested structure; it's an involved substitute for a simpler "*goto*" statement; but the "*goto*" statement has not been included in Ruby: it is strictly forbidden by the structured programming dogmas, so we need this construct, which enables us to write true "*spaghetti code*" in structured programming.

  The *catch* statement defines a block, which is interrupted by a *throw* statement. The *catch* block has a label, and the *throw* send to the end of the block with the label:

```
for matrix in data do           # A deeply nested data structure.

  catch :label do               # begin of the catch block
                                # the block has the label: ":label"
    for row in matrix do
      for value in row do
        throw :label unless value       # Break out of two loops at once,
        statements                      # otherwise, executes these statements.
      end
    end

  end                           #  end of the catch block

end
```

Exception Handling

The **raise** and **rescue** statements are used for exception handling; the "*raise*" statement throws an exception, which is an instance of the "*Exception*" class. There are many pre-defined exception classes, the default being "RuntimeError".

The "*rescue*" statements define blocks which are executed if an exception of a specified class is raised. The last raised exception is saved in the global variable: "$!" .

The general form of the "*begin ... rescue .. end*" block is:

```
begin
  ...
  raise exception_name

rescue Exception_class,Exception_class => local_name_for_exception

  ....   (block executed if given the exceptions have been raised )

rescue Exception_class,Exception_class => local_name_for_exception

  .... (block executed if given the exceptions have been raised )
```


Downloaded from http://www.helldragon.eu/

```
else

  ...  block executed if no exception in main block

ensure

  .... block executed in any case
end
```

The *rescue* statement can give to the exception a local name, to be used in the *rescue* block itself.

The *rescue* block can have a "**retry**" statement which re-executes the block after the "*begin*" statement.

The optional block after the last *rescue* statement, defined by an "**else**" statement, is executed if there are no exceptions. If, in the main block, a *return*, *next* or *break* statement is encountered, the "*else*" block is skipped.

The final, optional, "**ensure**" block is executed in every case.

If a new exception is raised in a *rescue* block the old exception is discarded and replaced by the new exception.

Below some examples of the *raise* statement:

```
raise "message error" if n < 1

raise RuntimeError.new("message error") if n < 1

raise RuntimeError.exception("message error") if n < 1
```

An example of a simple "*begin ... rescue .. end*" block:

```
begin

  raise "negative value" if x<0     # raise the exceptions
  y=Math::sqrt(x)

rescue => ex                         # Stores the exception in the local variable ex

   puts "#{ex.class}: #{ex.message}" # Handle exception

end
```

In the following example more *rescue* blocks catch different exceptions; a single rescue block can catch more types of exceptions:

```
begin
  y=Math::sqrt(x)
rescue DomainError => ex
  puts "Try again with a value >= 1"
rescue TypeError,ArgumentError => ex
  puts "Try again with an integer"
end
```

The *rescue* statement can be also used to give an alternate value to a variable:

Downloaded from http://www.helldragon.eu/

```
y = Math::sqrt(x) rescue 0
```

Postfix Expressions

In "*postfix expressions*" a condition is after the statements to be executed; postfix expressions are typical of Ruby, which tries to mimic a natural language with its syntax. The general form of a "*postfix expressions*" is:

```
     expr if  bool-expr      # the same as: "if expr then ... end" )

     expr unless bool-expr   # the same as "unless bool-expr then ... end")

Examples::

    exit if not str or not str[0]

    a=1 if a.nil?
```

There are postfix expressions also for the "*while*" and "*until*" loops, with condition and loop statements to be executed on the same line. In these cases the condition is tested before the first iteration:

```
x = 0
puts x = x + 1 while x < 10    # The while condition is at the end

a = [1,2,3]
puts a.pop until a.empty?      # The until condition is at the end
```


Downloaded from http://www.helldragon.eu/

Operators

In Ruby nearly all operators are class methods: types, operators and functions are all joined in an strong object oriented paradigm.

Only the assignment operators and some logical operators are not implemented as class methods. These operators are:

Operator	Meaning
=	assignment : a=1
+= -= *=	operation and re-assignment, implemented for all numerical operators
:?	ternary operator
&&	logical and
||	logical or
not	logical negation
or	logical operator "or" with lower precedence
and	logical operator "and"with lower precedence
.. ...	range operators (create range objects)
defined?	nil if not defined, otherwise the type: Es.: a=1; defined?(a) => "local-variable"

The assignment operator "**=**" defines a reference to an object, and, for some basic objects, creates an instance of the object:

```
anum=1        #  creates an instance of the Integer class (the number one)
              #  and a reference (the name "anum") to the instance


bnum=anum     # creates a second reference to the same integer
```

In the following example, when a new instance of Integer (the number six) is created, the *anum* reference is redefined, but *bnum* continues to reference to the number one:

```
anum=1
bnum=anum
anum=6
```

If instead of numbers we had a composite object, like an array, we could change the object using any of the references (*anum* or *bnum*) and both see the modified object; this can confuse inexperienced users:

```
anum=["a","b","c"]   # anum is a reference to an object array.
bnum=anum            # I make a copy of the reference,

anum[0]="k"          # Now I change the first element,
;                    # and both bnum and anum point to: ["k", "b", "c"]
```

Multiple assignments are possible:

Downloaded from http://www.helldragon.eu/

```
a=b=c=3        # a,b,c , all reference to '3'
a,b=1,2        # a refers to one, b to two
a,b=1,2,3,4    # extra elements are ignored
a,b,c=1,2      # c is unassigned (c==nil)
```

Many values can be assigned to a single array, or an array can be expanded to single values in an assignment; in this case the "*" operator is used: it is called "*splat*" operator:

```
a,b=[1,2]     # the assignment can be used to expand an array

x,*y=[1,2,3]  # extra elements are assigned to an array:  x => 1 ; y => [2, 3]

*y,x=[2,3,1]       # since Ruby 1.9 the array can be in the first position
a,*y,b=1,2,3,4,5   # and can also be in the middle: a => 1 ; y => [2, 3, 4] ; b => 5

x,y,z=1,*[2,3]     # assignment can be used to expand an array: x=>1 ; y => 2 ; z => 3
```

The operators of the form: "+=,-=,*=" etc. are shorthands for operation and assignment.

The "++" operator doesn't exists in Ruby.

```
a=a+1    # same as: a+=1
a=a-1    # same as: a-=1
```

The logical operators: "and" ,"or" ,"&&" , "||" , evaluate the second operand only if the first doesn't define the relation:

```
a && b   #  evaluates 'a' ;
         #  if 'a' is false (or nil), returns 'a' (false),
         #  otherwise evaluates and returns 'b'


a || b   # evaluates 'a' ;
         # if true returns 'a',
         # otherwise evaluates and returns 'b'
```

This can be used in conditional expressions to execute a routine, as in the Bash shell, or to define a variable only if not yet defined:

```
condition && func("aaa")  # 'func' is executed only if the condition is true

var ||  =77               # an undefined variable is false; the assignment is executed.
```

The ternary operators evaluates a condition, if true returns the first expression, otherwise the second:

```
1==1 ? 'Yes' : 'No'   # the condition is true, the operator returns: 'yes'

1==2 ? 'yes' : 'No'   # the condition is false, the operator returns: 'No'
```


Downloaded from http://www.helldragon.eu/

Classes and Methods

Ruby is a *very* object oriented language, with a rigid implementation of encapsulation, all is an object, common operators also are class methods and each action is done in the environment of an object; also in the interactive shell, or in the main program, one acts in the scope of an object.

The current object is referred to by the keyword **self**, which, in the main program and in the interactive usage, refers to an instance named **main** which acts as a default instance for method calls (it is a default receiver).

Ruby has only single inheritance, but classes can include modules (this is called "**mixin**"): a module defines a space for names and is a collections of methods, classes and constants, usually placed in an external file. When a module is mixed into a class, the methods defined in the module are inserted into the namespace of the class. This is a way to reuse function definitions and define common constants.

A Ruby class can have only one parent, but many modules can be mixed into a class.

All classes have, as a parent, the class "**Object**", which, since Ruby 1.9, inherits "**BasicObject**": a nearly empty class, with only some very basic functions implemented, as the equality operator "==".

The class Object includes (*mixins*) the **Kernel** module; this module contains a lot of methods. Are also implemented as methods of the *Kernel* module the loop statement, the exit method, the statements used to deal with exceptions, methods to open and read files and other functions that in many language are basic statements. The *Kernel* module implements also some equality operators.

Definition of classes are object too: instance of the class "*Class*" which inherits the class "*Module*" (which inherits "*Object*"); the class *Module* has some methods for access rules, mixins and reflection. When executing the statements in the class definition *self* is a reference to this instance of *Class*.

Instance and Class Variables

Variables defined into a class can belong to an instance or to the class; **class variables** are prefixed with "@@", and are unique to all the instances; can be defined in the class body or in a class method; **instance variables** are prefixed with "@" and each instance has its personal value of these variables. Also the class has a personal copy of the instance variables.

The instance variables must be defined and initialized into methods, if in the class body they are seen as instance variables of the class definition, seen as an instance of the class Class.

Both classes and instance variables are encapsulated into the class and can be seen from the outside only by means of special methods, named: *accessors*. Local variables are never seen outside their class or method, and class method don't see local variables defined outside the method itself; local variables are really local.

In spite of the rigid encapsulation instance variables can be obtained by the method: "*instancename.instance_variable_get(:@name_of_var)*" , and class variables with: "*Classname.class_variable_get(:@name_of_var)*". There are also the methods "*class_variable_set*" and "*instance_variable_get*", to set instance and class variables.

Methods and Functions

In Ruby all functions are methods; also if defined in the main program or in the interactive session, they belong to a class or an instance.

The class or instance whose method is called is the **receiver** for the method, the environment in which the method operates: the method sees all the variables of its receiver.

In the main program or in the interactive usage there is a default receiver instance named "**main**", which can be omitted in function calls; otherwise functions are referred to by prefixing their names with a class or instance name, separated from the function name by a dot.

Downloaded from http://www.helldragon.eu/

To be called, every function must be associated to a *receiver*, which can be an instance or a class. There is a distinction between **class methods** and **instance methods**. Instance methods have an instance as the receiver, *class methods* have a class as the receiver, they are two different kind of methods.

Methods are not objects in Ruby, but there is a class, the class: **Method** whose instances store instance methods. These methods are still bound to their original instances, with access to instance methods and instance variables.

An instance of Method can be detached from its *receiver*, becoming an "*UnboundMethod*" object (which can't be called) and then re-associated to another instance. The function **unbind** detaches a method (as instance of Method) from its receiver; the function **bind** associated an *UnboundMethod* to a new instance.

The class Method, has the functions **call** to execute the method, and functions to return some method features, as: **name**, **parameters**, **receiver** (returns the instance), **owner** (returns the class), **source_location** (source filename).

Methods are associated to symbols with the same name.

In a Ruby file every function must be defined before its usage.

Public, Private and Protected Methods

In Ruby there are different access rules for methods: methods can be **public, private or protected**.

Private methods can be called only into their class, by other methods of the class.

Protected methods can be called in the class, in derived classes, or as instance attributes by object of the same class or a derived one.

By default methods are public, visible everywhere; methods can be made protected or private in the class definition, by putting them in the appropriate section of the class body; or with the statement: "*protected*" put after the class definition.

A derived class can redefine also the access properties of a method defined in a parent class.

The statements defining the access properties of a method: *public*, *private* and *protected*, are implemented as methods of the class: "*Module*".

Downloaded from http://www.helldragon.eu/

Function Call

The general form of a function call consists of: an optional receiver name, the function name, zero or more function arguments and an optional block of statements which can be given to the function. The block is executed inside the function by the **yield** statement. When the receiver name is omitted, the current instance is used as a default receiver:

```
receiver_name.function_name( argument, argument, argument) {statement block}
```

There are some conventions on method names:

- method names begin with a lowercase letter.
- Parentheses around arguments are optional.
- Methods ending with "**?**" returns true or false; methods ending with "**!**" changes the internal status of an object.
- Methods ending with "**=**" mimic the setting of values; when these functions are called, if we omit the brackets, instead of "*func=(x)*", we can write: "*func=x*", which seems an assignment. A typical usage of these functions is to give a value to an instance variable. A function giving value to an instance variable is named: "*setter function*", and can be written as:

  ```
  def b=(a)    # '@b' is an instance variable; 'b=' an instance method
      @b=a
  end

  self.b=333  # seems an assignment, but calls a function
  @b          # => 333
  ```

Arguments are passed by reference to the functions and arrays and hashes passed to functions can be changed into the function.

Function Definition

Functions are defined with the statement "**def**" , followed by the function name, and arguments between round brackets; then there is the body of the function, which is a block of statements, terminated by the keyword: "*end*".

The name of the function begins with a lowercase letter.

Parentheses around arguments are optional both in method definition and invocations. As most of the Ruby statements, the "*def*" statement is an expression, and returns a value that is: "*nil*".

The function name can be prefixed by the name of the receiver, separated from the function name by a dot. **Methods can be defined outside their instances or classes, after the instance or class creation**.

Functions return the values given by the "**return**" statement or the last evaluated statement, if *return* is omitted. Multiple values can be returned and they are automatically placed into an array:

```
def func(a,b)
  c=1
  return a+b,c,"xx"
end

func(1,2)  => [3, 1, "xx"] # returns an array
```


Downloaded from http://www.helldragon.eu/

Methods can be deleted with the "**undef**" statement: "*undef func*" deletes the method named: *func*.

Method names can have aliases:

"*alias newname old_name*"

A function can have exception handling blocks at the end, which catch the exceptions raised in the function:

```
def func(..)
      ...
      raise Exception_class
      ...
rescue Exception_class => local_name
   ....
rescue Exception_class => local_name
   ....
else
   ...
ensure
   ....
end
```

Function Arguments

There are many different ways to specify the arguments:

- Arguments can have default values:

```
def iniziostringa(s, caratteri=1)
  s[0,caratteri]        #  a substring of a given length
end

iniziostringa("Abcdef")    => "A"
iniziostringa("Abcdef",3)  => "Abc"
```

- Arguments can be expressions, and also default values can be expressions and methods of the preceding arguments:

```
def iniziostringa(s, caratteri=s.size-2)
  s[0,caratteri]
end

iniziostringa("Abcdef")   => "Abcd"
```

- The number of arguments can be variable, with some arguments going into an array:

```
def stampa(a,*b)
  print(a," ",b)
end

stampa("a")           => a []
stampa("a","b")       => a ["b"]
stampa("a","b","c")   => a ["b", "c"]
```


Downloaded from http://www.helldragon.eu/

- Array arguments can be expanded to multiple values into the function, by using the "*splat*" operator:

```
def stampa(a,b,c)
     print("a=#{a} ","b=#{b} ","c=#{c} ")
end

stampa(*[1,2,3]) =>  a=1 b=2 c=3  # there are 3 values into the function
```

Blocks Given to Functions

A function can receive, in addition to the usual arguments, a block of code, between curly braces. This block is itself similar to a function and have arguments. Arguments are at the beginning of the block , between bars: "||",

Into the function the statement **yield** executes the block, giving arguments to the block. The block is executed in the environment of the calling program and sees the variables of the calling program.

Using the block a function can return many values during its execution and not only a final result.

Inside the function the yield statement returns the value of the block which is available to the function statements. In this way the use of a block can establish a sort of communication channel between the function and the calling environment.

In the function the statement: **block_given?** can test if the block has been given to the function; otherwise, if the block is missing, the statement *yield* raises a "*LocalJumpError*".

The call syntax is:

```
funcname(a,b,c,d) { |x,y,z| statements and function of x,y,z ..  }
```

The function definition syntax is:

```
def funcname(a,b=3,*e)
   ....
   ....
   if block_given?
         yield x1,y2,z1  # here the block is executed, with arguments x1,y1,z1
   end
   ...
   ...
end
```

If the *yield* function is inside a loop it returns a sequence of values, and the block can be executed many times, with the element of the sequence as arguments. In this way Ruby implements iterators: methods returning elements of a sequence of objects. Iterators are the preferred way to implement loops on sequences as arrays and hashes.

Examples:

```
def iterfun(k)
   a=[1,2,3,4]
   for ia in a
      yield ia*k if block_given?
   end
end
```


Downloaded from http://www.helldragon.eu/

```
iterfun(1)        # returns => [1, 2, 3, 4] :
                  # the last evaluated object (there is no return statement)

iterfun(1){ |k| print k } => prints 1234
iterfun(2){ |k| print k } => prints 2468
```

In the following example two argument are passed to the block only once:

```
def yfunc(a)
  b=0
  a=a+1
  yield a,b
end

yfunc(1){|k,j| print(k,j)} # => 20
```

In the following example the "*block_given?*" statement is used to obtain different results depending on the block presence:

```
def a_method
  return yield if block_given?       # testing if the block exists
  'block missing'
end

a_method                            # => "block missing"
a_method { "block found!" }         # => "block found!"
```

In the following example the value of the block, returned by the statement "*yield*" into the function, is used by the function computation.

```
def a_method(a, b)
   a + yield(a, b)
end
a_method(1, 2) {|x, y| (x + y) * 3 } # => 10
```

proc and lambda

Proc objects are instances of the **Proc class** and hold a block of code that is executable; the attribute "**call**" can be used to execute the proc:

```
myproc = Proc.new {|wine| puts "I love #{wine}!"}
myproc.call("sangiovese")

=>     I love sangiovese!
```

A "*proc*" can be passed as an argument to a function, it must be the last argument, with an ampersand :"*&*" before the proc name:

Downloaded from http://www.helldragon.eu/

```
def func(arg,arg2,arg3=default,  &block)
  ..
  the name: block contains the proc


func(a,b,c,d,&procname)   # ampersand also in the call
```

A block can also be seen as a *proc* into a function, it can be executed, inside the function, as a *proc* object, with the *call* method, or with *yield*:

```
def stampa(&block)
  block.call                # here the block is called as it where a *proc*
  yield                     # here the block is called by Yield
end

stampa { print "--AAAA " }
=> --AAAA --AAAA            # the block is execute twice
```

Another way to create a Proc object is to use the **lambda** method; using this method is essentially **equivalent to calling Proc.new.**

```
myproc = lambda {|x| puts "Argument: #{x}"}

myproc.call("ABCDE!")

=>  Argument: ABCDE!
```

A difference between lambda and proc is that the lambda tests the number of arguments and raise an error if arguments are not given, *proc* doesn't test anything

Downloaded from http://www.helldragon.eu/

Class Definition

The definition of a class by the user is done with the "*class*" statement, followed by a block of statements containing the class body, with the class definitions. The class name is a constant, and must begin with an uppercase letter. The "*class*" statement is an expression and returns the last evaluated statement, usually a "*def*", returning: "*nil*".

When the statements defining the class are executed by the Ruby interpreter, the receiver, pointed by the keyword: "*self*" is the class itself, as an instance of "Class", but into the definition of instance methods "*self*" is the instance being created.

```
class  Classname < Superclasse        # class definition (inherits Superclasse)

      include Modulename,Othername         # the class includes these modules
      include(OtherModulename)

      attr_accessor :x,:y          # creation of accessor methods
      attr_reader   :u,:w
      attr_reader   "v","z"

      @@n=0                        # a class variable

      def initialize(x,y=3)        # class constructor

         @x,@y=x,y                 # instance variables
                                   # must initialized in the constructor

         super( a, b, c )          # calls the constructor of the parent
      end

      def self.classmethodname     # definition of a class method
          ...
      end
      class << self                # block containing class methods
         def nome(..)
         ...
         end
      end

      def metodo                   # definition of an instance method
           .....
      end

      to_s                         # override of a function of a parent class
        "description class"
      end

      protected                    # section for protected methods

         here protected methods

      private                      # section for private methods

         here private methods
```


Downloaded from http://www.helldragon.eu/

```
end
```

The method named: "*initialize*" is the instance constructor; it is a special method, called in an automated way at the instance creation. In this method the variables needed by the instance are initialized.

In the example above the method "*super*" is used by "*initialize*" to call the constructor of the parent class, which is not automatically called at the instance creation.

Inheritance

The "<" symbol is for inheritance. Ruby has only single inheritance, but classes can include in their namespace functions and classes from modules (mixins). Each class inherits the class Object in an automated way, there is no need to specify this dependency.

The method: "**super**" can be used into a class function, to call the method of the parent with the same name of the function.

There are some methods to look at the structure of a class hierarchy:

```
Classname.superclass    # show the inherited class
Classname.ancestors     # show all the inherited classes (ancestors)
Classname.methods       # show all the methods
Classname.constants     # show the constants defined in the class

Classname.included_modules      # list of the included modules
Classname.include?(Nomemodulo)  # tests if a module is included

Classname.respond_to?("string")  # tests if a method exists
Classname.respond_to?(:symbol)   # tests if a method exists, using a symbol
```

Class Instances

To instantiate a class the "*new*" method must be called; the "*new*" method has arguments that are passed to the class constructor: the "*initialize*" function.

```
instancename=Classname.new(argument, arg, arg2)

instancename.instance_of? Classname     # to test if instance of a class
instancename.kind_of? Classname         # to test if a class is among ancestors
```

each instance has an unique identifier, which can be obtained by the method: "**.object_id**".

Class Methods and Instance Methods

In Ruby there are class methods and instance methods: class methods can only be called on classes and instance methods can only be called on an instance of a class.

But indeed, class methods also are instance methods: instance methods of that instance of *Class* which is the class definition.

The syntax for calling class and instance methods is:

Downloaded from http://www.helldragon.eu/

```
Classname.classmethodname(..) # for class methods

instance.methodname(..)    # for instance methods
```

By default the methods are instance methods; in the class definition, class methods are prefixed with the keyword "*self*", or by the class name, but can also be defined in the class block, into a special block delimited by: "*class << self ... end*"

```
class Classe

     def Classe.func
        print("class method")
     end
     def self.func2
        print("class method")
     end

     def func3
        print("Instance method")
      end
end


class Classe2

    class << self       # other way to specify class methods
       def func
           class method
       end
    end
 end
```

Accessor Functions and Instance Variables

The rule is that all variables defined in a class are private to the class and are not visible outside.

To make an instance variable accessible outside the class there are special function, (accessor functions) defined in a way similar to:

```
class A
   def var_a            # read access ids done with: *A.var_a*
     return @var_a      # using the function as the var_a name
   end
   def var_a=(v)        # write access is done with: *A.var_a=123*
     @var_a=v           # using the function as the var_a name,
   end                  # with omitted parenthesis
end
```

The accessor statements automatically make these functions for the specified variables, allowing for a syntax like: "*instancename.variable*" , where "*variable*" is an accessor function which mimic the use of a instance attribute; but **only functions and constants are visible outside a class instance**, all variables are hidden.

Example of the accessor statement:

Downloaded from http://www.helldragon.eu/

```
class Nomeclasse
    attr_accessor :x,:y   # makes accessors for @x,@y
    attr_reader   :z      # makes a read accessor for @z

    def initialize(x,y)   # constructor, called at object creation
       @x=x
       @y=y               # initialization of instance variables
    end

    def to_s             # used by print functions to obtain a
      "#@x,#@y"          # string representing the instance
    end
 end # of class
```

Instance variables should be initialized in the constructor. The class itself (the class definition) is an instance of the object Class, and variables defined out of the constructor are instance variables of the class as an instance of the Class object.

To print an instance of a class one have to define a *to_s* method, that returns a textual representation of the instance. There are also a modules (PP, PrettyPrint) to print a class instance:

```
instancename= Nomeclasse.new(1,2)

require 'pp'
pp instancename   # => 1,2
```

Adding Methods to a Class

Methods can be added to a class outside the class block; to define a class method outside the class block the method has to be prefixed with the class name:

```
def Classname.classmethodname(..)
   ....
end
```

Singleton Methods

A method belonging to a single instance of a class is named "**singleton method**, "*numeric*" and "*symbols*" can't have singleton methods. A singleton method can be added to a specific instance in the following way:

```
oggetto=Classname.new
def oggetto.funzione(..)
  ...
end
```

Also the following syntax can be used:

```
class << oggetto

    def func1(..)
    ...
```


Downloaded from http://www.helldragon.eu/

```
    end

    def func2( ..)
    ...
    end
end
```

In the following a simple example of definition and usage of a singleton method:

```
a="1"
def a.nome
   print "uno"
end

a.nome => uno
```

The term *singleton class* is used in Ruby for class instances with singleton methods, which is not the same as the singleton classes of languages like C++, which are classes with a single instance ... here the term "singleton" is a bit confusing.

In Ruby, classes with a single instance are built with the "*singleton*" module, which, mixed into a class, changes the class method "*new*", which become *private* and impossible to call; the methods "*instance*" is used, instead of "*new*", to make, or refer to, the single instance of the class.

Downloaded from http://www.helldragon.eu/

The Object Class

The class "*Object*" is the main class in the Ruby language. The "*Kernel*" module, containing most of the basic functions, is mixed into the *Object* class, and the *Object* class is inherited by all the Ruby classes, making all its methods available to every instance of every class.

Among the methods in the Object class, some are used to find the class of an object, to test methods of an object , to interact with the operating system. Some of these methods are listed in the following table.

Method	Function	Examples
nil?	test if nil (only a nil value returns true) Es,: nil.nil? => true	
is_a? ; kind_of?	tests if class or ancestor	1.is_a? Numeric => true
instance_of?	tests if instance of a class	1.instance_of? Fixnum => true
respond_to?	test if a class method	1.respond_to?('+') => true
methods	array with public methods	Object.methods => [:allocate, :new,....]
method	returns a Method object	1.method("+") => #<Method: Fixnum#+>
send	calls a method on an instance	1.send("+",2) => 3
clone	clones an object	s2=s1.clone
dup	makes a shallow copy of an object (copies only the references, not the data)	
abort	stops the program	abort("messaggio")
exit	exits from the program, raising SystemExit. Es.: exit(1)	
exit!	exits, skipping then exception handling mechanism	
fork	creates a sub-process	
exec	executes a process, in place of the current one.	exec("ls")
spawn	executes a command in a subshell.	spawn("ls")
sleep(s)	pause the program for some seconds	sleep(5)
system	executes a system command.	system("ls")
command	command for the system	*ls* => lists current directory
%x{command}	command for the system	as ``, but using a different separator
warn	writes a message on stderr.	warn("attention")
eval	evaluates a ruby expression.	a=1;b=2; eval("a+b") =>3
load	loads and execute a ruby program.	load("nomefile.rb)
freeze	makes an object immutable	
frozen?	tests if an object is immutable	

The class Object has also methods to build basic objects: **Integer**, **Float**, **Complex**, **Rational**, **Array** and **String**.

The operator: "**===**" is used to test if an object is an instance of a class or an instance of its descendants, the descendants redefine the operator to allow for comparison. This operator is mainly used in the "*case*" statement.

In Ruby there aren't immutable objects, as in Python. All objects can be changed, but there is a method: *freeze*, which prevents every change in a single object, making that object immutable. A freezed object can't be unfrozen.

Downloaded from http://www.helldragon.eu/

Logical Classes

There is no boolean class in Ruby, but two special classes: **TrueClass**, **FalseClass**. These classes have an unique instance: **true** and **false** representing the results of logical expressions.

There is also: "**nil**", the unique instance of **NilClass**, meaning a missing or undefined value; nil evaluates to false, but all valid numbers, zero included, evaluated to true, and also void strings, void arrays or hashes evaluates to true.

"**nil?** (a method of Object) return true if an object evaluates to *nil*,

The operator: "**defined?**" returns *nil* if an object is not defined; there are no *true?* or *false?* operators, it's easy understand why: in Ruby everything is true.

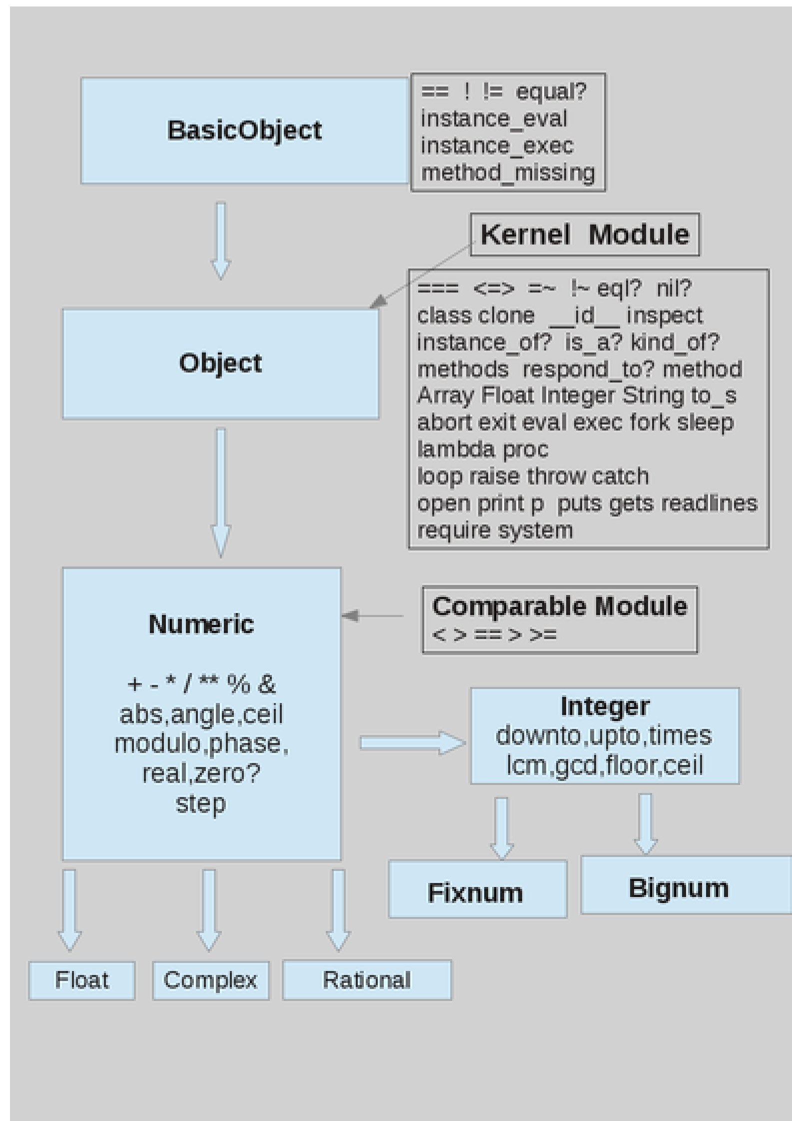
TrueClass, *FalseClass* and *NilClass* implement the logical operators: "& | ^". The *NilClass* implements also some conversion operators, to change *nil* objects into void objects (but void objects don't evaluate to *nil*):

```
nil.to_a  => []        : empty array
nil.to_c  => (0+0i)    : complex zero
nil_to_s  => ""        : empty string
nil.to_f  => 0.0       : zero
nil.to_i  => 0
nil.to_r  => (0/1)     : rational zero
```


Downloaded from http://www.helldragon.eu/

Numeric Classes

In the figure the class hierarchy of the numeric classes; some methods of the classes are listed in the figure.



The classes for numbers are: **Float**, **Complex**, **Integer**, **Rational**; there are two subclass of *Integer*: **Bignum** and **Fixnum**; Bignum are numbers with an arbitrary number of digits; Fixnum are 8 bytes long integers, (but different platform can have different lengths); conversion between Bignum and Fixnum is automatically managed by Ruby: when a number is less than about 4E9 it's a Fixnum, otherwise it is a Bignum. Integer numbers can be built with the **Integer method**, converting its argument to an integer. Integers can also be expressed in hexadecimal, octal or binary form:

> **0x** is the prefix for hexadecimal representations
>
> **0b** is the prefix prefix for binary representation
>
> **0** is the prefix prefix for octal representation

Floats are double precision , **always a digit is needed before and after the point**: ".1" or "1." are not valid float numbers. "e" or "E" is before the (optional) exponent. Float numbers can be built with the **Float method**, converting its argument to a float number.

Downloaded from http://www.helldragon.eu/

There are two special values for floats: "*infinite*" and "*nan*", to deal with float division by zero, but integer division by zero raises the exception: ZeroDivisionError

```
0.0/0.0  => NaN
1.0/0.0  => Infinity

0/0      => ZeroDivisionError:  ....
```

Rational numbers are the ratio of two integers, used to express periodic numbers without a loss of precision. Complex numbers are printed as sum of a real and an imaginary part. Es.: *(2+5i)* .

Rational and complex have some specific methods and can be built with the "**Rational** and **Complex methods**:

```
 Complex(1)           => (1+0i)
 Complex(2, 3)        => (2+3i)
 Complex(0.3)         => (0.3+0i)
 Complex('0.3-0.5i')  => (0.3-0.5i)

 Complex(2, 3).real  => 2            # real part
 Complex(2, 3).imag  => 3            # imaginary part)
 Complex(2, 3).conj  => (2-3i)

 Complex(0,1).phase  => 1.5707963267948966       # radians
 Complex(0,1).polar  => [1, 1.5707963267948966]  # modulus, phase
 Complex(0,1).rect   => [0, 1]                   # Array with real and imaginary parts

 a=Rational("1/2")
 a=1.quo(5)            # division between integers that gives a rational number

 Rational(1)      => (1/1)
 Rational(2, 3)   => (2/3)
 Rational(4, -6)  => (-2/3)
 Rational('0.3')  => (3/10)
 Rational('2/3')  => (2/3)
 Rational('0.5')  => (1/2)

(Rational(3,2)).numerator     => 3
(Rational(3,2)).denominator   => 2
(Rational(3,2)).truncate      => 1
(Rational(3,2)).round         => 2
```

Underscores can be used inside numbers to separate digits:

```
a=1_000_000 =>  1000000
b=2_000.0_1 =>  2000.01
```

Methods for the Numeric Classes

All numerical operators are implemented as methods of the Numeric class: for this reason a function-like notation is legal for operators: "*1.+(2)*" can be used, and returns the number 3, as: "*1+2*".

The comparison operators: < *;* > *;* <= *;* >=, == *; between?* are implemented in the **Comparable module**; each numeric class implements in a different way only the operator: <=> (sometimes named: *spaceship* operator); the Comparable operators use the <=> operator of the specific class to give results. This mechanism is similar to the use virtual functions of the C++ language.

Downloaded from http://www.helldragon.eu/

In the following table the most used methods of Numerics are included, also those which apply only to some subclasses of Numerics.

Operator	Function	Examples
** ; pow(x,y)	power	a**3
* /	Multiplication,division	a*b ; 3*2 => 6 ; 3/2=1
%	Remainder	5/2. => 1.0
+ -	Addition, subtraction	a+b ; 2+5 => 7
<< >>	bitwise shift	4<<1 => 8
& | ^	bitwise and, or, xor	
<=>	comparisons, returns: -1, 0 , 1 ; if less, equal greater	
< > <= >=	comparisons	4<1=> false
between?	inclusion in a range	2.between?(1,4) => true
== !=	equal, not equal	
zero?	true if zero	3.2.zero? => false
eql?	same value AND type	1.0.eql? 1 => false
equal?	reference equality: if a is the same object as b: a.equal?(b) => true	
abs ; magnitude	Absolute value	-1.abs => 1
abs2	square modulus	-2.abs2 => 4
ceil	minimum greater integer	1.2.ceil => 2 ; -1.2.ceil => -1
floor	maximum lower integer	1.7.floor =>1 ; -1.2.floor =>-2
round	rounds a number	1.4.round => 1 ; 1.5.round => 2
div	integer division	5.2.div(2.0) => 2
divmod	quotient and remainder	5.divmod(2) => Array :[2, 1]
fdiv	float division	3.fdiv(2) => 1.5
quo	division with maximum precision (Rational numbers if integer operands)	

Methods for Integers

even?	true if even	2.even? => true
odd?	true if odd	3.odd? => true
next ; succ	next integer	1.next => 2
pred	previous integer	2.pred =>1
gcd	greatest common denominator	10.gcd(15) => 5
lcm	lowest common multiple	10.lcm 15 => 30

Downloaded from http://www.helldragon.eu/

Methods for Floats

finite?	true if finite	(1.0/0.0).finite? => false
infinite?	test if infinite	(-1.0/0.0).infinite? => -1
nan?	test if not a number	(0.0/0.0).nan? => true

Conversion Methods

```
**to_s**     # converts to string, for fixmum the argument is the base, es.:

             16.to_s     => "16" ;
             16.to_s(2)  => "10000" ;
             16.to_s(16) => "10"

**to_i**     # converts to integer;

**to_f**     # converts to float;

**to_r**     # converts to rational
```

Subscript Operator

The subscript operator "**[]**" can be used for Fixnum, and returns single bits of a number:

```
2[0] => 0
2[1] => 1

3[0] => 1
3[1] => 1
3[2] => 0
```


Downloaded from http://www.helldragon.eu/

Precedence for operators

In the following some operators are listed, ordered by precedence (high to low precedence):

```
[ ] [ ]=
**
! ~ + -       (unary operators)
* / %
+ -
>> <<
&
^ |
<= < > >=
<=> == === != =~ !~
&&
||
.. ...
?:
= %= /= -= += |= &= >>=
  <<= *= &&= ||= **= ^=
not
or and
if unless while until
begin/end
```


Downloaded from http://www.helldragon.eu/

String Class

In Ruby strings are sequences of characters, but can also be used to store binary data as sequences of bytes. A string can be created from a sequence of characters:

a=String.new("characters")

Ruby has a complete set of functions to deal with strings. Strings are basic objects in Ruby, inherit Objects and include the **Comparable module**, which implements the basic operator for comparison: < *;* > *;* <= *;* >=, == *; between?*.

String comparison is based on the order of characters in in the ASCII sequence, where there are: first numbers, then uppercase letters, then lowercase ones. If the first characters of two strings are the same, then the longer string is considered greater:

```
'0'<'A' => true
'A'<'a' => true
'a'<'b' => true

'azzz'<'baaa' => true # the first different characters matters!
'aab'<'abb'   => true
'abc'<'abc0'  => true # the second string is longer
```

String Encoding

In Ruby each string has it's own encoding, which can be obtained by the method: **encoding**; the default encoding is UTF-8 for Ruby version 2, US-ASCII in Ruby 1.9; all strings where ASCII in Ruby 1.8.

The encodings are described by the **Encode class**. The class method *Encode.list* return an array with the list of all the available encodings.

In a source file the encoding of the file can be specified in the first lines:

```
#!/usr/bin/ruby
# coding: utf-8
```

Some operations between strings can't be done if their encoding is not compatible; the **encode** method can be used to change the encoding of a string; this method has options to deal with undefined or invalid characters in the new encoding and to change the final newline character; there is also a **force_encodig** method that sets the encoding property of a string:

```
'a'.encoding => #<Encoding:UTF-8>

b='a'.encode("ISO-8859-1")

b.encoding   => #<Encoding:ISO-8859-1>

b.encode!("UTF-8")             # encode! changes the string on-place

b.encoding   => #<Encoding:UTF-8>

"abcd".encode("UTF-8", undef:   :replace, replace: "X") # "X" replaces undefined characters
"abcd".encode("UTF-8", invalid: :replace, replace: "X") # "X" replaces invalid characters
```


Downloaded from http://www.helldragon.eu/

```
b='a'.encode("ISO-8859-1")
b.force_encoding("UTF-8")          # tells to Ruby that b is an UTF-8 string,
b.encoding    => #<Encoding:UTF-8>    # but 'b' is not changed
```

Double-quoted String

Strings can be created from characters between double quotes;

Es.: a="stringa"

An alternative syntax is: **%Q(string)** ; where the string inside parentheses can contain double quotes, but not the parentheses, which are used as a string delimiter. You can use the delimiters you like instead of parentheses, but if you use parentheses the initial and final delimiters must match: {} [] () <>

```
a="stringa\n"
a=%Q(abcd\n)       # () are used as delimiters
a=%QZ 12 ef(), Z   # 'Z' is used as a delimiter
```

double quoted strings can extend over many lines, preserving the newline character. The backslash can be used to escape the final newline, to effectively join two lines.

Between double quotes all the usual backslash substitutions are performed:

\n	end of line
\b	backspace
\e	escape
\s	space
\t \v	tabs
\f \r	form-feed, return
\hhh	ottale
\xhh	exadecimal
\uxxxx	unicode
\C-x	control-x sequence
\M-x	meta-char sequence

Single-quoted Strings

Strings can be created from characters between single quotes, when single quotes are used only some backslash substitutions are performed: "\\" and "\`" .

Es.: a='stringa'

an alternative syntax is: **%q(string)** ; the string inside parentheses can contain single quotes, but not parentheses You can use the delimiter you like instead of parentheses, but if you use parentheses the initial and final delimiters must match: {} [] () <>

```
a='stringa'
a=%q(stringa)             # () are used as delimiters
a=%qA xyx string aad A    # The letter 'A' is used as a delimiter
```


Downloaded from http://www.helldragon.eu/

A string between single quotes can extend over many lines , the end of line is not escaped, but inserted into the string as "\n"

String Operators

In the following table a list of some methods of the String class.

+	concatenates strings: "a"+"b" => "ab"
*	repeats strings: "abc" * 2 => "abcabc"
<<	concatenates strings: "a"<<"b" => "ab"
ascii_only?	true if only ascii characters
empty?	true if empty
end_with?("string")	true if end with the given string
include?("substring")	test if substring included: 'abc'.include?(b)=> true
index("substring")	index of a given substring: 'abc'.index('b') => 1
rindex("substring")	index of a given substring starting from the end
insert(index,string)	substring insertion: "abc".insert(1,"xx")=>"axxbc"
split(pattern)	splits into an array, default pattern is a space
capitalize ; capitalize!	makes the first character uppercase
upcase ; upcase!	to upper cases; upcase! changes string in place
downcase ; downcase!	to lowercase
swpacase ;swapcase!	upper case to lower and lower to uppercase
sub(pattern,replacement)	first occurrence substring replacement
gsub(pattern,replacement)	all occurrence substring replacement
tr('old char','new') .tr!	change characters, as the "tr" Unix command
center(n," ")	centers in n characters, specifying the padding character
ljust(n," ")	shifted to left, in n characters, padded with space
rjust(n," ")	shifted to right
lstrip ; lstrip!	strip leading spaces
rstrip ; rstrip!	strip final spaces
strip ; strip!	strip final and leading spaces
squeeze(characters) ;squeeze	eliminates duplicates for the given characters
reverse ; reverse!	reverse the string
clear	empties the string
replace(newstring)	replaces the string with a new one
chomp ; chomp!	strips the final end of line, if present
encoding	returns the string encoding
valid_encoding?	if a valid encoding

Downloaded from http://www.helldragon.eu/

encode("iso-8859-1") ; encode!	re-encode the string in the given encoding
force_encoding("utf-8")	tell the encoding to Ruby
to_i ; to_f	conversion to numbers
length ; bytesize	length in characters or bytes
getbyte(num)	get a single byte at a given position
setbyte(num)	set a single byte at a given position
bytes.to_a	byte contents: "ab".bytes.to_a =>[97, 98]
count("substring")	counts how many times the substring is found
count("a-c")	count characters
delete("chars") delete!("b")	delete characters
crypt	crypt the string using the operating system function
sum	computes a simple checksum for the string
next ; succ	next in the ascii sequence: "a".next => "b"
ord	encoding number of first character : "ab".ord => 97

If the argument of the "**<<**" operator is a number it is intended as the the numeric code of a character in the encoding of the string; the corresponding character is appended to the string:

```
"a"<<"b" => "ab"
"a"<<98  => "ab"
```

- **The function "chars"**

  It is used to separate a string into an array of characters (produces an Enumerator object):

  "string".chars.to_a => ["s", "t", "r", "i", "n", "g", "a"]

- **count and delete**

  these functions are very versatile: can count or delete ranges of characters, characters out of a range etc.:

  "abccd".count("a-c") => 4 ; "abccd".count("^a-c") => 1 ;

  "abcdeff".delete("a-cf") => "de"

- **sub and gsub**

  these functions can have a regular expression or a string as the pattern argument, and also the subsequences of the match can be used:

```
"abcdcde".sub("cd","xy") => "abxycde"

"abcdcde".gsub("cd","xy") => "abxyxye"
```

- **slice**

  can be used to extract substrings, as the **[]** operator; the version: **slice!** changes the string in place:

```
"abcde".slice(2..4)  =>  "cde"
"abcde".slice(1,3)   =>  "bcd"
```


Downloaded from http://www.helldragon.eu/

```
"abcde".slice("bcd") =>  "bcd"
```

- **The "?" operator**

  this operator gives the representation of a single character in a string:
  ?a => a ; ?C-d => "u0004" # this is the unicode for Cntrl/d

- **The [] operator**

  this operator can be used to extract characters from strings, it will be described in the section about Arrays

- **String interpolation:**

  A Ruby expression can be inserted into a string, and its result is computed and used into the string:

  "stringa #{ruby statements } ... "

- **The format operator "%" :**

  this operator acts the same as in the printf routine of the C language:
  " string with %s %d " % ['abc',123] => " string with abc 123 "

- **The plus operator "+"**

  this operator is used to concatenate strings.
  "abc"+"def" => "abcdef"
  Strings following strings are automatically concatenated:

  a='asd' 'asd' => "asdasd"

- **The operator "*"**

  this operator is used to repeat strings.

  "a"*3 => "aaa"

- **Here documents:**

  very long string can be inserted in the following way:

```
nomestringa = <<HERE
    here a long text

HERE
```

HERE is an arbitrary word, used as a delimiter, the final delimiter is alone, on a single line. No space is allowed between "<<" and the first delimiter, or after the last delimiter.

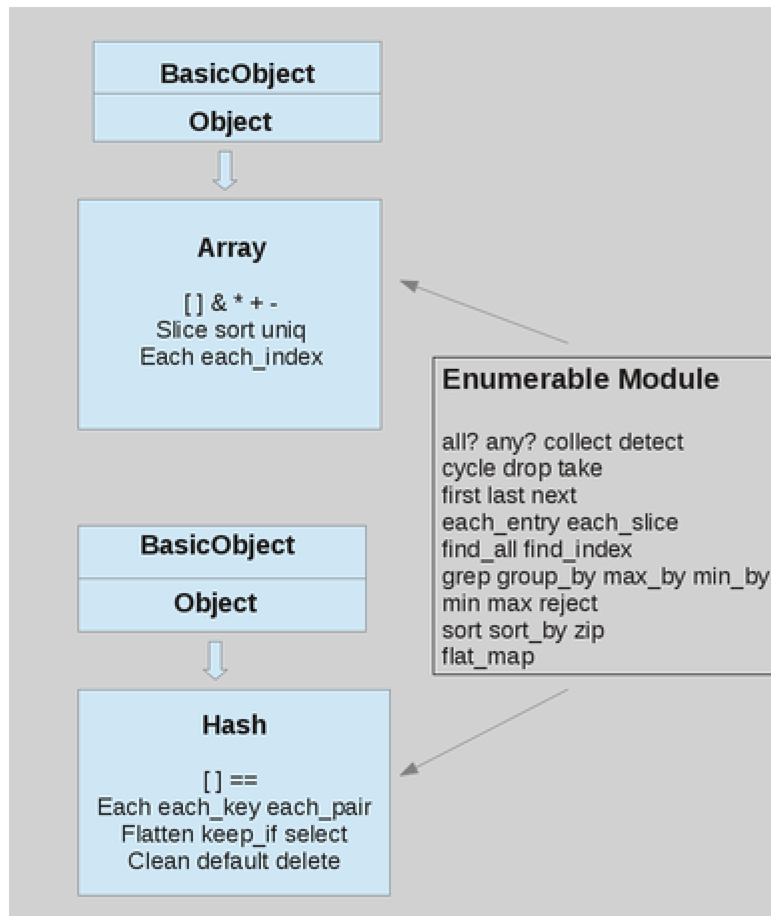
Downloaded from http://www.helldragon.eu/

Array Class

Arrays are sequences of heterogeneous elements, are represented as a comma separate list between square brackets; an integer index, between square brackets is used to refer to single elements; We can have arrays of arrays, simulating multi-dimensional structures.

The class mixes in the **Enumerable module** , with a lot of iterators.

In the following figure the hierarchy of the Array and Hash classes; some class methods are listed in the square box.



Arrays can be created in the following ways:

```
a=Array.new            : an empty array

a=Array.new(3)         : to make an array of 3 elements containing "nil"

a=Array.new(3,"a")     : to make an array of 3 elements containing the object: "a"


a=[12,"asd",3.5E3,1,2,3]    : another way to make an array
```

To make an array from words in a string:

Downloaded from http://www.helldragon.eu/

```
a= %w{ a b c d \n  }  => ["a", "b", "c", "d", "\\n"]

a= %W{ a b c d \\n  } => ["a", "b", "c", "d", "\n"]
```

if "**%w**" is used, backslash symbols are not interpreted, as in single-quoted strings. If "**%W**" is used, backslash are interpreted as in double-quoted strings

The "**split**" function can obtain an array from a string, the "**join**" function obtains a string from an array.

The '*' operator can act as a join for arrays, if the second operator is a string:

```
"1-2-3".split("-") => ["1", "2", "3"]
[1,2,3].join("-")  => "1-2-3"

[1,2]*"a"  => "1a2"
```

Some operators and functions for arrays are listed in the following table:

+	concatenates: "a"+"b" => "ab"
-	difference: [1,1,2,3] - [1,2,4] => [3]
*	array repetition: [1]*2 => [1, 1]
concat	concatenates: [1].concat([2]) => [1, 1]
&	common elements: [1,2] & [2,3] => [2]
|	add without duplicates: [2,2,3] | [1,2,4] => [2, 3, 1, 4]
<<	append elements: [1]<<2 => [1, 2]
include?(value)	true if the value is included: [1,2].include?(1) => true
empty?	true if empty
length	number of elements
count	counts elements: [1,2,1].count => 3 ; [1,2,1].count(1) => 2
compact ; compact!	removes nil elements
uniq ; uniq!	remove duplicates: [1,2,2].uniq => [1, 2]
delete(value)	fetches and deletes an item, given its value
delete_at(n)	fetches and deletes an item, given the position
insert(index,value)	insert at the given position: [1,2].insert(1,5) => [1,5,2]
fill(value)	change each element to the given value
first(n)	first elements: [1,2,3].first(2) => [1, 2]
last(n)	last elements: [1,2,3].last(2) => [2, 3]
max	maximum element: [1,3,2].max => 3
min	minimum element: [1,3,2].min => 1
flatten	makes uni-dimensional: [[1,2],[3,4]].flatten => [1,2,3,4]
transpose	in 2-dimensional transposes row and columns
join(separator)	joins elements into a string: [1,2].join("-") => "1-2"

Downloaded from http://www.helldragon.eu/

combination(n).to_a	array with the all combinations of n elements of the array
permutations(n).to_a	all permutations of n elements
repeated_combination(n)	all combinations with repetitions
repeated_permutations	all permutations with repetitions
replace(newarray)	replaces with an other array
reverse ; reverse!	reverses the order o f elements
rotate(n) ; rotate!(n)	circular shift: [1,2,3].rotate(1) => [2, 3, 1]
sample(n)	extracts n random elements
unshift(1)	add a first element [1].unshift(2) => [2, 1]
shift(n)	extracts and deletes first n elements
push(value)	add an element at the end: [1].push(2) => [1, 2]
pop(n)	extract and deletes elements from the end
shuffle ; shuffle!	random reordering
sort ; sort!	sorts elements:[2,1].sort=>[1, 2]; ["b","a"].sort=>["a","b"]
to_s	a string representing the array

flatten has an optional argument: the number of dimension to eliminate:

```
a=[1,2,[13,14],[15,[26,[37,38]]]]

a.flatten(1) => [1, 2, 13, 14, 15, [26, [37, 38]]]

a.flatten(2) => [1, 2, 13, 14, 15, 26, [37, 38]]

a.flatten(3) => [1, 2, 13, 14, 15, 26, 37, 38]

a.flatten    => [1, 2, 13, 14, 15, 26, 37, 38]
```

to_a, a member Enumerable, creates an array from a sequence.

zip combines arrays element by element, producing an array of arrays:

```
(1..3).zip [ "a","b","c"]                  => [[1, "a"], [2, "b"], [3, "c"]]

[1,2,3].zip([10,20,30],["a","b","c"]) => [[1, 10, "a"], [2, 20, "b"], [3, 30, "c"]]

(1..3).zip                                 => [[1], [2], [3]]
```

The Subscript Operator: []

"**[]**" returns single bits for numbers, characters for strings, elements of arrays; inside the square brackets we can have numbers, ranges or also regular expressions:

```
a=[1,2,3,4] ;

a[0] => 1                # index begins from zero
```


Downloaded from http://www.helldragon.eu/

```
a[-1]=> 4                 # negative indexes from the end
a[a.size-1]               # last element

a[0..2]   => [1, 2, 3]    # extracts using a range (last element included)
a[0...2]  => [1, 2]       # extracts using a range (last element NOT included)
a[-4..-2] => [1, 2]

a[0,2]  => [1, 2]         # a sub-array (2 elements from 0)
a[1,2]  => [2, 3]         # 2 elements , from 1
a[-3,2] => [2, 3]         # 2 elements , from -3 ( a[-3]=> 2 )
a[-4,2]    => [1, 2]      # from element -4 (the first) takes 2 elements

a[/regular expr/]         # regular expressions can be used too

a.first(2) => [1, 2]      # first elements
a.last(2)  => [3, 4]      # last elements

a=[1,2,3,4] ; a=['a','b']=> ["a", "b", 2, 3, 4] # reassigns a range of elements
```

An element is changed or added with the "**[]=**" operator; *nil* **elements are created to void the gaps**:

```
a=[0] ; a[3]=3    => [0, nil, nil, 3]

a=[1,2,3,4]
a[0,2]=0      => [0, 3, 4]     # starting from position 0, sets 2 elements

a=[1,2,3,4]
a[0,2]=5,6    => [5, 6, 3, 4]  # multiple assignments are possible
a[0,2]=[5,6]  => [5, 6, 3, 4]

a=[1,2,3,4]
a[-1]=9       => [1, 2, 3, 9]  # negative indexes counts from the end
```

Hash Class

Hashes, also named maps, or associative arrays, or dictionaries, are sequences of objects, not retrieved by an integer index, but by a key, which can be any object. Hashes are written as key/value pairs, separated by an arrow, in curly braces, Es.: *{1=>"a",2=>"b"}*

Hashes mixes in the **Enumerable module**, with a lot of useful iterators.

The elements are in the order in which they have been inserted in the hash; the last element being the last inserted.

The attempt to access to a non-existing elements returns *nil* or a default value which can be set at the hash creation or with the *default=* method.

For hashes the operator "**[]**" returns values, based on keys; symbols can also be used as keys, for a faster access::

```
a={"one" => 1, "two" =>2 ,3 => "a"}

a["one"] => 1
```


Downloaded from http://www.helldragon.eu/

```
a[3]     => "a"

h["a"]=723                 # adds, or replaces, en element

h={ :one => 1, :two =>2 } # symbols can be used as keys

h={one: 1 , two: 2 }       # alternate syntax, possible  when keys are symbols
```

Operators and methods for hashes are listed in the following table:

a=Hash.new	makes an empty hash
a={}	makes an empty hash
a=Hash.new("default")	makes an empty hash defining the default value
default	returns the default value
default=	sets the default value
delete	deletes an element by key Es.: h.delete("one")
fetch	fetches element by key: h.fetch(key)
has_key? ;include? : key?	True if a key is present: h.has_key?(k)
has_value?	true if a value is present: h.has_value?(val)
merge ; merge!	merges hashes, duplicate keys are overwritten: h.merge(h2)
sort	sorted array of pairs: [[key1,value],[key2,value]..]
flatten(n)	array with keys and values, with n dimensions flattened
invert	keys become values and values keys
flatten(n)	array with keys and values, with n dimensions flattened
empty?	true if the hash is empty: {}.empty? => true
length ; size	number of elements
values	array with values
h.keys	array with keys
h[key]	a value, given the key (the default value if key not found)
h[key]=value	adds, or changes, the value for a given key
key(value)	a key, given a value
clear	voids an hash
sort	an ordered array of [key,value] ,ordered by keys
shift	extracts and remove the first [key,value] pair
to_a	array of [key,value] pairs
to_s	a string representing the hash

Downloaded from http://www.helldragon.eu/

Range Class

Range objects represent an interval in a sequence, are mainly used in loops and case statements, but also as indexes of arrays and strings. Ranges mixes in the module Enumerable, with a lot of iterators.

Ranges are written as two numbers or characters separated by two or three dots. When three dots are used the last object is not included, when two are used the last element is included.

Not only numbers or characters can be used to make a Range object, but every object which can be treated as a sequence, responding to the comparison method: "<=>" and the "*succ*" method (which gives the next item of the sequence).

The Range method can be used to built ranges:

```
a=Range.new(1,3)
1..3              => 1,2,3   # last element is INCLUDED

a=Range.new(1,3,true)
1...3             => 1,2   ; range with last element NOT INCLUDED

(1...3).exclude_end? : true

-3..-1          => -3,-2,-1

'a'..'c'     # range of characters

 1.2..3.5    # range of float (but you can't iterate on these)

 a=[0,1,2,3,4]
 a[1..3]        => [1, 2, 3]
 a[1...3]       => [1, 2]
```

We can have array of ranges, in this case ranges are not expanded to the sequence they represent, there is nothing as the list comprehension of Python, and, to built an array of values from the range, the "*to_a*" method must be used.

a=[1..3,5...9] => [1..3, 5...9]

Some methods of ranges are in the following table:

exclude_end?	tests if last value is included: (1...3).exclude_end? => true
cover?	tests if between bounds: (1...3).cover?(2.5) => true
include? ; member?	tests if member of the range: (1...3).include?(3) => false
begin	the range begin: (1..3).begin => 1
end	the range end: (1...3).end => 3
first(n)	first n elements: (1..3).first(2) => [1, 2]
last(n)	last n elements (1..3).last(2) => [2, 3]
min	minimum value (1...3).min => 1
max	maximum value: (1...3).max => 2
to_a	converts to array: (1..3).to_a => [1, 2, 3]
to_s	string representation (1..3).to_s => "1..3"

Downloaded from http://www.helldragon.eu/

Regular Expressions

Regular expressions are special string patters of characters, used with strings, for substring search and substring substitutions. It's a powerful, but complicated tool; for an introduction to regular expressions see: http://en.wikipedia.org/wiki/Regular_expression .

In Ruby regular expressions are build into the language, as the "**Regexp**" class. The matching with a a string is done using the match operator: "**=~**", or by the method: **match**. The **to_s** method give a string representation of a Regexp object.

Regular expressions are created by the Regexp constructor method, from a string pattern between "/ /" or with the "**%r**" operator: **r=%r{pattern}**:

```
r=/pattern /

r=/pattern /options

r=%r{pattern}          # every character can be used in place of "{}"

r=%r{pattern}options   # a regular expression with options

r=Regexp.new(pattern,options)
```

A regular expression can have options, some options are:

```
i : case insensitive search
o : a substitution is performed only once
m : the dot match also \n
x : extended syntax
    (pattern can contain comments and other constricts)
```

In regular expression patterns the backslash is used as an escape characters and some special constructs are interpreted according to the following table:

.	the dot matches any character
+	one ore more of the preceding substring
*	zero or more of the preceding substring
?	one or zero of the preceding substring
[abc123]	one of these characters
[^aeiou]	a character not in the list; not a vowel for: [^aeiou]
[a-c]	a range of characters
[^a-c]	a character not in the range
a|b	logical or for characters ("a" or "b")
{m}	m times the preceding substring
{n,m}	min m and max m occurrence of the preceding substring
{,m}	at least m times
{,n}	at most n

Downloaded from http://www.helldragon.eu/

^	the beginning of a line
$	the ending of a line
\Z	ending of a string
\A	beginning of a string
\b	word boundaries
\B	non word boundaries
\s	space characters (and also \n \t \r \f)
\S	NON space characters
\d	a digit, same as [0-9]
\w	a word character, same as [A-Z,a-z,0-9_]
\W	a non word character
( )	to group characters into a sub-pattern
\1 \2	substrings matched by a preceding sub-pattern

The operator: "**~=**" returns the index of the first match or *nil* if the match didn't occur; the **match method** returns instead a **MatchData object** with members containing the results of the match.

If sub-patterns are used, the substrings matched by the sub-patterns are also saved. String matched by sub-patterns can be used also into the match pattern itself.

The match methods have the following syntax:

```
string =~ /pattern/         # returns a number or nil:
                              n= string =~ /pattern/

/pattern/.match(string)     # returns a MatchData object:
                              matchdata = /pattern/.match(string)

string !~ /pattern/         # is the same as !(string =~ /pattern/)
```

After a match some global variables, and a Matchdata object, are defined, the Matchdata object is saved in the global variable: "**$~**

$&	matchdata[0]	the matched part of the string
$'	matchdata.pre_match	the part preceding the match
$`	matchdata.post_match	the part after the match
$1 $2	matchdata[1]	strings matched by sub-patterns
	matchdata,size	size of the matched string

Regexp objects are also used for string substitution, using the **sub** and **gsub** methods; *sub* makes a single substitutions, *gsub* changes all the occurrences of the match:

```
"string".sub(/pattern/,"replacement string")

"string".gsub(/pattern/,"replacement string")  # gsub for multiple substitutions
```


Downloaded from http://www.helldragon.eu/

```
"string".sub!(/pattern/)       # sub!: for on-place replacement
"string".gsub!(/pattern/)
```

A block of statements can be associated to the *sub* and *gsub* methods, these blocks have, as argument, the matched string; the result of the block is substituted into the string (blocks of statements will be described further):

```
"string".sub(/pattern/)  { |p| statements }   # "p" is the matched string
"string".gsub(/pattern/) { |p| statements }
```

Patterns can be used to subdivide a string, by using the **scan** method:

```
"one word or two".scan(/\w+/)   =>  ["one", "word", "or", "two"]

"one word or two".scan(/\w+/) {|w| statements }  # also passing matches to a block
"string".scan(/(t).*(n)/) { |a,b| print a,b }    # sub-matches given to the block
```

Examples of regular expression usage:

match	position	matched string	meaning
/abc/ =~ "012abc34"	3	"abc"	"abc" substring
/^abc/ =~ "012abc34"	nil	nil	not "a", then "bc"
/d+/ =~ "012abc34"	0	"012"	some digits
/.+s.+/ =~ "123 abc"	0	"123 abc"	first space between characters
/4$/ =~ "4234"	3	"4"	"4" at the end
/(34)$/ =~ "34234"	3	"34"	"34" at the end
/^(34)/ =~ "34234"	0	"34"	"34 at the beginning
/3{2}/ =~ "12343312"	4	"33"	"3" two times
/[0-9]/ =~ "abc3de"	3	"3"	a character in a range
/(dd):(dd)/=~"a12:30"	1	"12:30"	subpattern: $1=>"12"; $2=>"30"

Downloaded from http://www.helldragon.eu/

Other Builtin Classes

There are many other classes builtin into the Ruby interpreter, some of these are:

- **Dir**

  to manage directory and files, with functions as: chdir, pwd, exists?, mkdir, new, delete.
  There are also some iterators on files.
  Dir.pwd is the current directory.

- **Exception**

  to manage exceptions with methods: message , status, success? (true is status zero or nil)
  to_s : a string representation of the message

- **File**

  represents files, with all the unix functions for files and more, as: atime, ctime, dirname, executable?, directory?, file? ,exists?, readable? ,zero?, truncate, delete, new, rename, ftype, lstat ,link, symlink, path, size, stat

- **IO**

  for input/output and functions as: new, pipe, open, read, write, sync, eof?, and many iterators on the file content.

- **Mutex**

  a semaphore for synchronization of accesses to resources

- **Random**

  random numbers

- **Struct**

  A mini-class to contain strings which can be accessed by symbols:

```
Customer = Struct.new(:name, :surname) # => makes the Customer object

c1=Customer.new("giovanni", "bianco")   # makes  instances
c2=Customer.new("giovanna","rossi")

c1.name    => "giovanni"         # using builtin accessors
c1["name"] => "giovanni"

c1.members=> [:name, :surname]   # using symbols as  accessors

c1.to_a    => ["giovanni", "bianco"]

c1.to_h    :  make an hash ( only for Ruby 2 )
```

- **Thread**

  to manage threads , with function as fork, kill , new, stop.

- **Time**

  class for time and date:

```
t = Time.at(0)                       => 1970-01-01 01:00:00 +0100

t1= Time.gm(2000,"jan",1,20,15,1)    => 2000-01-01 20:15:01 UTC
```


Downloaded from http://www.helldragon.eu/

```
t2= Time.local(2000,"jan",1,20,15,1)   => 2000-01-01 20:15:01 -0600

t3= Time.new(2010, 12, 25, 8, 0, 0, "-06:00") # => 2010-12-25 08:00:00 -0600

tn=Time.now         => 2014-08-27 16:04:34 +0200

tn.getgm            => 2014-08-27 14:06:40 UTC    # convert from local to UTC

Time.now.asctime    => "Wed Aug 27 16:05:51 2014" # as an ASCII string

tn.strftime(special string )     # formatted as in the printf function of C
tn.strftime("today is %d/%m/%Y %H:%S")  => "today is 27/08/2014 16:40"

tn.to_a     => [40, 6, 16, 27, 8, 2014, 3, 239, true, "CEST"] # as an array

to_f ; to_i => giulian seconds from  1870

localtime ; gmtime ; =>  convert times  in place
```

Parts of the data can be obtained by:

```
t = Time.at(0)

t.min  ;  t.sec ; t.hour;
t.hour ;  t.day ; t.mon; t.year;  t.zone ;
t.wday ;  t.mday;

t.sec ; t.usec ; t. nsec  => in julian seconds, microseconds, nanoseconds
```


Downloaded from http://www.helldragon.eu/

Iterators

The Enumerator Class

The concept of iterator is implemented, in Ruby, by the **Enumerator class**, most iterator methods return an object of this class.

An Enumerator has the method **to_a** to produce an array, the method **next** to extract (and delete) an element, a method **peek** to fetch the current value (without deletion), a method **rewind** to begin again the sequence and a method **each** that loops over the elements giving them, in turn, to a block:

```
a=Enumerator.new([1,2,3])

a.each {|k| print k}         => 123

a.next  => 1
a.next  => 2    # extract, and  consume the element
a.peek  => 3    # only fetch the current element
a.peek  => 3    # always the same element without a "*next*"
a.next  => 3
a.next  => raise a StopIteration exception

a.rewind        # to begin again the sequence
a.next  => 1
```

The Enumerable Module

Most iterators are implemented in the Enumerable module; this module is mixed into classes which contain sequences: Array, Hashes, Range, Struct, IO and Dir; since Ruby 1.9 the class String doesn't use the Enumerable module but implements all its own iterators.

Some member of the module Enumerable are listed in the following table:

all?	true if all the elements are true. Es.: [0,true,nil].all? => false
any?	true if some elements is true. Es.: [0,true,nil].any? => true
collect ; map	array obtained by the block: `(1..2).collect {|a| a*2} => [2, 4]`
count	counts true values from block: `(1..6).count {|a| a<3 } => 2`
cycle(n)	cycles n time over the sequence: `[1,2].cycle(2) {|a| print a} => 1212`
detect	return the first true element: `(1..6).detect {|a| a>3 } => 4`
drop(n)	array without the first n elements: [1,2,3,4].drop(2) => [3, 4]
drop_wile	drops until block returns true: `[1,2,3,4].drop_while {|a| a < 3 } => [3, 4]`
take_while	take until block returns false: `[1,2,3,4].take_while {|a| a < 3 } => [1, 2]`
each_slice(n)	loops on groups of n values:`(1..4).each_slice(2){|a| print a}=>[1,2][3,4]`
each_with_index	value and index to the loop:`(1..3).each_with_index{|a,i| print a,i}=>102132`
find_all ; select	all elements for which the block is true: `(1..3).find_all {|a| a>1}=>[2, 3]`
reject	all elements for which the block is false: `(1..3).reject {|a| a>1} => [1]`

Downloaded from http://www.helldragon.eu/

find_index	index of first element giving a true block: `(1..3).find_index{|a| a>1} => 1`
first(n);take(n)	first n elements: `(1..4).first(2) => [1, 2]`
grep(pattern)	array of matching elements:`["a","b","c"].grep(/b|c/){|i| i}=>["b","c"]`
include?	true if an element is included: `(1..3).include? => true`
max	maximum value: `(1..3).max => 3`
min	minimum value: `` (1..3).min => 1 ``
max_by	value giving the greater block: `(1..3).max_by {|a| 1.0/a } => 1`
min_by	value giving the smaller block: `(1..3).min_by {|a| 1.0/a } => 3`
minmax	minimum and maximum value `(1..3).minmax => [1, 3]`
one?	true if the block return true only once: `(1..3).one? {|a| a==2 } => true`
none?	true if the block is always false `(1..3).none? {|a| a==2 }=> false`
reverse_each	loops in reverse order: `(1..3).reverse_each {|a| print a } => 321`
sort	sort elements: [3,2,1].sort => [1, 2, 3]
sort_by	sorting based on the block: `(1..3).sort_by {|a| 1.0/a } => [3, 2, 1]`

flat_map can be used to build an array, by appending consecutive results of the block:

```
(1..2).flat_map{|a| [a,a+10,a+20]} => [1, 11, 21, 2, 12, 22]
```

group_by can be used to build an hash, the block gives the keys:

```
(1..3).group_by {  |a| a+100 } => {101=>[1], 102=>[2], 103=>[3]}
```

minmax can have an associated block, giving a comparison expression which utilized the "<=>" operator; also **sort** can have an associated block:

```
(1..3).minmax {|a,b| 1.0/a <=> 1.0/b } => [3, 1]

(1..3).sort {|a,b| 1.0/a <=> 1.0/b }   => [3, 2, 1]
```

Some iterator for numerics

upto	`3.upto(5) {|i| print i, " " } => 3 4 5` (last value is included)
downto	`3.downto(1) {|n| print n, ".. " } => 3.. 2.. 1..`
times	`3.times {|n| print n, ".. " } => 0.. 1.. 2..`
step	`1.step(6, 2) {|i| print i, " " } => 1 3 5`

If the block is missing these methods return an Enumerator object which can be converted to an array with the *to_a* method

```
3.upto(5).to_a     => [3, 4, 5]
```


Downloaded from http://www.helldragon.eu/

```
3.downto(1).to_a   => [3, 2, 1]

3.times.to_a       => [0, 1, 2]

1.step(6, 2).to_a  => [1, 3, 5]

12.2.step(14.0,0.5).to_a  => [12.2, 12.7, 13.2, 13.7]
```

Some Iterators for Strings

each_char	loops on characters: `"abcd".each_char {|k| print k+"z" }azbzczdz=> "abcd"`
each_byte;bytes	loops on bytes: `"abcd".each_byte {|k| print k,"-" } => 97-98-99-100-`
eachy_line	loops on lines (keep the final "n" characters)
upto	loops on a sequence of strings: `"a".upto("d"){|k| print k} => abcd`

Also these methods, if the block is not present, return an iterator which can be converted to an array.

Some Iterators for Arrays

each	gives each item to the block: `[1,2,3].each {|k| print k} => 123`
reverse_each	gives items in reverse order: `[1,2,3].each {|k| print k} => 123`
each_index	gives indexes to the block: `[1,2,3].each_index {|k| print k} => 012`
map	array with block results: `[1,2,3].map {|k| k+1} => [2,3,4]`
collect;collect!	array of block results: `[1,2,3].collect {|i| i*i } => [1, 4, 9]`
delete_if	if block false deletes elements: `[1,2,3].delete_if {|x| x.even?} => [1, 3]`
keep_if	keeps if block true: `[1,2,3].keep_if { |x| x.even? } => [2]`
select;select!	elements with a true block: `[1,2,3].select { |x| x.even? } => [2]`
rindex	index of element of first true block: `[1,2,3].rindex {|k| k==2}=> 1`
uniq ; uniq!	elements with unique block value: `[1,2,3,4].uniq {|k| k.even?}=> [1,2]`

The creator of arrays can also be used as an iterator:

```
a=Array.new(3) { |i| i} => [0, 1, 2]
```


Downloaded from http://www.helldragon.eu/

Some Iterators for Hashes

Hashes has iterators similar to the iterators for arrays, but give different arguments to the block:

each ; each_pair	gives each pair key,value `{1=>"a",2=>"b"}.each {|k,v| print k,v} => 1a2b`
each_key	gives the keys to the block: `{1=>"a",2=>"b"}.each_key {|k| print k} =>12`
each_value	gives the values to the block: `{1=>"a",2=>"b"}.each_value{|k| print k} =>ab`
delete_if	if block false deletes: `{1=>"a",2=>"b"}.delete_if {|k,v| k==1 => {2=>"b"}`
keep_if	keeps if block true: `{1=>"a",2=>"b"}.keep_if {|k,v| k==1 } => {1=>"a"}`
select,select!	hash of elements with true block: `{1=>"a",2=>"b"}.select {|k,v| k==1}{1=>"a"}`

The **merge** method can be used associated with a block: when there are duplicated keys the block is executed and the new value for the key is the the block result

```
{1=>"a",2=>"b"}.merge({1=>"x",3=>"z"}) {|key,v1,v2|  v1+v2 } => {1=>"ax", 2=>"b", 3=>"z"}
```


Downloaded from http://www.helldragon.eu/

Input/Output

For input/output there is a "**File class**" which inherits the "**IO class**". The *File class* has methods to interact with the file system and the *IO class* has methods to read and write files, but general usage statements as "print", "open", "close", are methods of the Object class.

There are also some useful iterators to deal with files.

A file is represented by a "*File object*", an instance of the "*File class*"; Ruby treats input and output as is done by the Unix system: files are simple streams of bytes; there are global variables referring to the default streams for reading, writing and writing error messages: "**$stdin, $stderr, $stdin**. These are instance of the *IO* class. There are also constants referring to these streams: **STDIN, STDOUT, STDERR**.

The function: **open** returns a File object:

```
f=open(filename,"access string")
```

If the filename begins with "|" it is a Unix named pipe; the access string defines the access method: read-only, read-write append etc.

"r:iso-8859-1"	to read-only, for a specific encoding
"w+"	read/write
"w"	write only
"a"	append to file, write only
"a+"	append, read/write
"b"	binary, to be added to the other codes: "wb" , "rb" ..

Some methods useful to read a file are listed in the following table; "f" is the file object:

f.close()	closes the file
a=f.gets	gets the next line
a=f.readline	
a=f.gets(10)	gets 10 characters
a=f.gets("separator")	gets a line, defined by an user-given line separator
a=gets	next line from standard input
a=gets("separator")	
a=f.getc	gets next character
a=f.getbyte	gets next byte
f.lineno	returns the current line number
f.lineno=10	set the initial value of the line counter (file position unchanged)
f.pos	returns the file position (current byte)
f.path	the complete file name
a=f.readlines	returns an array with the file lines
a=f.readlines("s")	lines array, a line separator is given
f.each {|a| ..}	an iterator over the file lines

Downloaded from http://www.helldragon.eu/

f.each_line {|a| ..}	an iterator over the file lines
f.each("separator"){|a| ..}	here a line separator is given
f.each_byte {|b| ..}	an iterator over bytes
f.each_char {|b| ..}	an iterator over characters

In the following table some methods of the *File* class, useful to deal with files:

File.exist?(filename)	true if the file exists
File.file?(filename)	tests if filename is a regular file
File.directory?(filename)	tests if a directory
File.symlink?(filename)	tests if filename is a symbolic link
,	,
File.size(filename)	file size in bytes.
File.size?(filename)	file size in bytes or nil if empty
File.zero?(filename)	true if empty
,	,
File.readable?(filename)	true if readable
File.writable?(filename)	true if writable
File.executable?(filename)	true if executable
File.world_readable?(filename)	true if readable by everybody
File.world_writable?(filename)	true if writable by everybody
,	,
File.ftype("/usr/bin/ruby")	type of the file: "file", "directory", "link"
,	,
File.rename("newname", "oldname")	renames a file
File.symlink("name", "link")	makes a link
File.delete("test2")	deletes a file
File.utime(atime, mtime, filename)	changes access and modification time
File.chmod(0600, f)	sets unix file permissions (octal argument)
.	.
File.read("filename")	reads and return the entire file as a string
File.read("filename", 4, 2)	returns 4 bytes starting at byte 2
File.read("filename", nil, 6)	returns from byte 6 to the end-of-file
linee = File.readlines("filename")	return an array with the file lines
File.foreach("filename") { }	block looping on lines

Downloaded from http://www.helldragon.eu/

The following examples show how the functions **putc**, **puts**, **print**, **write** can be used for output:

```
o = STDOUT

o.putc(65)          # writes the single byte 65 (capital A)
o.putc("B")         # writes the single byte 66 (capital B)

o << x              # print 'x'
o << x << y         # print 'x' and 'y'

o.print s

o.printf fmt,*args # the printf function of C

o.puts              # prints a newline
o.puts x            # prints 'x', then a newline

o.write s           # doesn't print a final newline
```

The following examples show how a specific position in a file can be accessed (random access):

```
f = File.open("test.txt")

f.pos        #  returns the current byte, same as: 'f.tell'
f.pos = 10   #  set the position counter to 10

f.rewind     #  goes to the beginning of the file

f.seek(10)                 # go to byte 10, same as: 'f.seek(10, IO::SEEK_SET)'
f.seek(10, IO::SEEK_CUR)   # skips 10 bytes from current position

f.eof?                     # tests if at end of file
f.closed?                  # tests if the file has been closed
f.tty?                     # tests if the stream is the interactive console
```

Ruby doesn't write immediately to e file when a print statement is issued, but use memory buffers for a temporary storage to optimize the writing times; the operating system uses its own buffers too: the Ruby output is buffered by Ruby, then given to the system and stored again; the writing is not a real-time process; but some Ruby functions can control the buffering:

```
out=File.open('filename')
out.write("abcd")

out.flush          # flush the Ruby output buffer for this stream

  # to set the buffer mode:

out.sync = true    # buffers are flushed after every write
out.sync = false   # output is buffered by Ruby
out.sync           # show the current buffer mode
out.fsync          # flush the system buffers (not possible for some systems)
```


Downloaded from http://www.helldragon.eu/

Modules

A module is a collection of classes and methods, stored in a file and defining a namespace. A module is an instance of the **Module class**, the "*Module*" class is inherited by the class: "*Class*", whose instances are the class definitions.

A module in a file named "*nomefile.rb*" can be included in a Ruby program and executed with the "**load**" or the "**require**" statements.

To use a method or a constant defined into a module the method or the constant name must be prefixed with the module name as: "*Math::sin(90.0) ; Math::PI*" "**::**" being the "*resolution*" operator.

Modules can be nested.

A module is defined by the keyword "*module*" ; the name of a module has the first letter capitalized; the name is a constant associated with the module

```
module Nomemodulo       # module definition

  class ...       # definition of a class in the module

  end

  def ...          # definition of a function in the module

  end


end
```

Modules can be included into a class (mixins); if variables are defined into the module, module accessor functions have to be defined in the module to use the variables as attributes of instances of the class including the module.

The statement "**include**" mixes a module into the namespace of a class, if the module is in its own file, and not in the same file of the class, the file containing the module must be loaded with "*load*" or "*require*" before the inclusion in the class:

```
module Mod             # a simple module

  def func             # with a simple method
    "A function of Mod!"
  end
end

class Classe
    include Mod        # mixed into a class
end

b=Classe.new                          # an instance

b.respond_to? :func     => true        # responds to the module method
b.respond_to? "func"    => true

b.func                  # => "A function of Mod!"
```


Downloaded from http://www.helldragon.eu/

The *extend* method can insert a module into an instance, and the methods of the module become singleton methods of the instance:

```
module Mod
  def func
    "A function of Mod!"
  end
end

a = []
a.extend(Mod)
a.func                    # => "A function of Mod!"
a.singleton_methods  # => ["func"]
```

When, for an object, the **freeze** method is called, the object can't be extended with a module.

Builtin Modules

Many functionality is offered by a number of builtin modules, as Math, for mathematics functions, or Marshal, to transform complex objects into a single string. Some builtin modules are listed in the following table

Name	Content
Kernel	Implements main methods and statements
Comparable	Methods for comparisons
Enumerable	A lot of iterators
Marshal	Data serialization
Math	Mathematical functions
Process	System process management
Signal	System signal processing

Standard Library

What is not implemented into the languages or the builtin modules is into a standard library, structured into modules and distributed along with the language; there are about hundred modules in the standard library, in the following table a list of some modules.

CGI	Common gateway interface
CSV	for CSV files
Curses	To deal with textual interfaces
Digest	compute digests
erb	html template (used also by rails)
find	the unix find utility
gserver	for tcp net servers
ipaddr	for tcp/ip addresses
json	json parser and writer

Downloaded from http://www.helldragon.eu/

logger	messages to the system
matrix	vector and matrixes in Ruby
net/ftp	ftp client
net/http	httpd client
net/imap	imap client
net/pop	pop client
net/smtp	to send mail
net/telnet	remote connection
socket	socket implementation
openssl	ssl sockets connection
rss	parse rss streams
scanf	the scanf function of the C language
set	for the set algebra
shellwords	split a phrase in an array of words
singleton	makes a class with a single instance
socket	BSD sockets for network applications
syslog	writes on the system syslog
tempfile	temporary files management
tk	interface to the tk language
webrick	http server
xmlrpc	xmlrpc implementation
yaml	yaml parser and serializer
zlib	for file compression
marshal	to serialize using binary files
gems	virtual environments and gems management

Downloaded from http://www.helldragon.eu/